Typed Open Programming

A higher-order, typed approach to dynamic modularity and distribution

Andreas Rossberg

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht Saarbrücken, 5. Januar 2007
(Korrigierte Fassung, 18. März 2008)

Dekan:	Prof. Dr. Thorsten Herfet
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Bernd Finkbeiner
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Prof. Dr. Andreas Zeller
Tag des Kolloquiums:	2007/05/11

Abstract

In this dissertation we develop an approach for reconciling *open programming* – the development of programs that support dynamic exchange of higher-order values with other processes – with strong static *typing* in programming languages.

We present the design of a concrete programming language, *Alice ML*, that consists of a conventional functional language extended with a set of orthogonal features like higher-order modules, dynamic type checking, higher-order serialisation, and concurrency. On top of these a flexible system of dynamic components and a simple but expressive notion of distribution is realised. The central concept in this design is the *package*, a first-class value embedding a module along with its interface type, which is dynamically checked whenever the module is extracted.

Furthermore, we develop a formal model for *abstract types* that is not invalidated by the presence of primitives for dynamic type inspection, as is the case for the standard model based on existential quantification. For that purpose, we present an idealised language in form of an extended λ-calculus, which can express dynamic generation of types. This calculus is the first to combine and explore the interference of sealing and type inspection with higher-order *singleton kinds*, a feature for expressing sharing constraints on abstract types. A novel notion of *abstraction kinds* classifies abstract types. Higher-order type and kind *coercions* allow for modular translucent encapsulation of values at arbitrary type.

Kurzdarstellung

In dieser Dissertation entwickeln wir einen programmiersprachlichen Ansatz zur Verbindung *offener Programmierung* – der Entwicklung von Programmen, die das dynamische Laden und Austauschen höherstufiger Werte mit anderen Prozessen erlauben – mit starker statischer *Typisierung*.

Wir stellen das Design einer konkreten Programmiersprache namens *Alice ML* vor. Sie besteht aus einer konventionellen funktionalen Sprache, die um einen Satz orthogonaler Konzepte wie höherstufige Modularisierung, dynamische Typüberprüfung, höherstufige Serialisierung und Nebenläufigkeit erweitert wurde. Darauf aufbauend ist ein flexibles System dynamischer Komponenten sowie ein einfacher aber expressiver Ansatz für Verteilung verwirklicht. Zentral ist dabei das Konzept eines *Pakets* (*package*), welches ein Modul in Kombination mit seinem Schnittstellentyp in einen Wert einbettet, und bei der Extraktion des Moduls eine dynamische Typüberprüfung vornimmt.

Weiterhin entwickeln wir einen theoretischen Ansatz zur Modellierung von *abstrakten Typen*, welcher im Gegensatz zum herkömmlichen formalen Modell existentieller Quantifizierung auch in Gegenwart dynamischer Typinspektion gültig ist. Zu diesem Zweck definieren wir eine idealisierte Sprache in Form eines erweiterten λ-Kalküls, der dynamische Typgenerierung ausdrücken kann. Der Kalkül kombiniert diese erstmals mit höherstufigen "*Singleton Kinds*, einem Sprachkonstrukt, welches Gleichheit von Typen ausdrücken kann. Zur Klassifizierung abstrakter Typen werden *Abstraktions-Kinds* als verwandtes Konzept entwickelt. Höherstufige Konversionen auf Term- und Typebene erlauben zudem die nachträgliche modulare Enkapsulierung von Werten beliebigen Typs.

Zusammenfassung

Die zunehmende Verbreitung des Internets hat begonnen, die Struktur von Software nachhaltig zu verändern. An die Stelle von in sich geschlossenen Programmen, die nur lokal operieren, treten mehr und mehr *offene* Applikationen, die dynamisch Daten mit anderen Prozessen im Netzwerk austauschen, oder von dort sogar neue Funktionalität beziehen. Das offensichtlichste Beispiel für ein Programm dieser Kategorie ist ein Web-Browser.

Auf programmiersprachlicher Ebene erfordert dieser Paradigmenwechsel eine verbesserte Unterstützung *offener Programmierung*, zu der wir Konzepte wie Modularität, Dynamik, Portabilität, Sicherheit, Verteilung und Nebenläufigkeit zählen. Nur wenige existierende Sprachen sind bisher darauf ausgelegt. Zu ihnen gehören vor allem die zu diesem Zweck entwickelte objektorientierte Sprache Java, die mittlerweile weite industrielle Verbreitung gefunden hat, und die im akademischen Umfeld entwickelte nebenläufige Constraint-Sprache Oz. Diese setzt entsprechende Konzepte noch weitaus konsequenter um, insbesondere durch die einheitliche Repräsentation von Programmkomponenten und externen Daten, so dass beide beliebig gemischt werden können.

Diese Dissertation widmet sich einem spezifischen Aspekt offener Programmierung, der bislang von keinem der Vertreter auf befriedigende Weise gelöst wurde: der Kombination offener Programmierung mit einem expressiven, starken *Typsystem*. Ein Typsystem ist ein in die Programmiersprache integriertes formales Werkzeug zur automatischen Verifikation bestimmter Programmeigenschaften. Es weist jedem Programmkonstrukt einen *Typ* zu, eine logische Formel, die das bei Ausführung des Konstrukts zu erwartende Resultat klassifiziert. Die damit möglichen Konsistenzprüfungen können die Zuverlässigkeit von Software verbessern. Moderne Programmiersprachen bieten zudem die Möglichkeit, die Typstruktur um benutzerdefinierte, sogenannte *abstrakte Typen* zu erweitern, welche die Festlegung gewisser Zugriffsbeschränkungen erlauben. Wenn die Semantik der Programmiersprache verhindert, dass diese Zugriffsbeschränkungen umgangen werden können, so spricht man von *Abstraktionssicherheit*. Diese garantiert Modularitätseigenschaften und steigert damit vor allem die Wartbarkeit von Programmen.

Typüberprüfungen erfolgen naturgemäss *vor* der Ausführung eines Programmes, üblicherweise durch den Übersetzer der verwendeten Programmiersprache. Dadurch entsteht ein inhärenter Konflikt mit offener Programmierung, da in einem offenen Ansatz im Allgemeinen nicht alle Programmteile vorweg bekannt sind und analysiert werden können. Es ist deshalb unausweichlich, bestimmte Typüberprüfungen in die Laufzeit des Programms zu verlagern. Ein seit langem bekannter Ansatz dafür ist die Einbringung eines speziellen universellen Typs *Dynamic*, der Werte der Sprache gepaart mit ihrem jeweiligen Typ beinhaltet. Die Extraktion eines Wertes erfolgt explizit und erfordert die Angabe eines oder mehrerer erwarteter Zieltypen, die dynamisch abgeglichen werden. Leider haben sich *Dynamics* jedoch in der Praxis als zu unhandlich erwiesen. Zudem ergeben sich durch die Möglichkeit des dynamischen Typabgleichs semantische Implikationen, die unter anderem die Abstraktionssicherheit abstrakter Typen beeinträchtigen.

Wir nähern uns diesen Problemen von zwei Seiten an. Zum einen beschreiben wir das Design einer konkreten Sprache names *Alice ML*, welche *typisierte offene Programmierung* ermöglicht. Dabei handelt es sich um einen Dialekt der funktionalen Sprache Standard ML, die durch einen relativ kleinen Satz orthogonaler und hinreichen einfacher Sprachkonstrukte erweitert wurde. Dabei handelt es sich zunächst um *Pickling* zum serialisierten Import und Export höherstufiger

Werte, verschiedene Formen von *Futures* für die Synchronisation nebenläufiger Berechnungen, sowie *Module* höherer Ordnung, welche die Sprache um wichtige Abstraktionsmöglichkeiten ergänzen. Die meisten dieser Konstrukte sind bekannt und für sich gut verstanden, aber bisher nicht in dieser Form und zu diesem Zweck in einem kohärenten Design integriert worden. Neu ist ausserdem das zentrale Konzept von *Paketen* (*packages*), welches das Kernproblem der dynamischen Typisierung löst. Es ähnelt der Idee von *Dynamics*, jedoch werden nicht einzelne Werte, sondern komplette Module eingebettet. Die feinkörnige Typunterscheidung weicht so einem strukturellen Inklusionstest auf Modulschnittstellen, der robust gegenüber Erweiterungen ist und eine Handhabung auf hohem Abstraktionsgrad erlaubt. Auf Grundlage dieser Basiskonzepte definiert die Sprache einen flexiblen, typsicheren Begriff von *Komponenten*, der nicht nur bedarfsgetriebenes dynamisches Laden ermöglicht, sondern Komponenten als Werte erster Klasse verfügbar macht, die dynamisch berechnet und aus einem Prozess exportiert werden können. Mit Hilfe dieser Idee wiederum ist ein vergleichsweise einfacher aber expressiver Ansatz für *verteilte Programmierung* möglich, bei dem Verbindungen zwischen Prozessen durch den initialen Austausch einer dynamisch berechneten Komponente aufgebaut werden. Das Konzept von programmierbaren *Komponentenmanagern* erlaubt es dem Empfängerprozess dabei, gezielte Sicherheitsstrategien durch Einschränkung der Importrechte für die empfangene Komponente zu realisieren. Eine nahezu vollständige Implementation von Alice ML wurde realisiert und steht als offene Software zur Verfügung.

Zum anderen entwickeln wir einen theoretischen Ansatz zur Modellierung von Typabstraktion, der Abstraktionssicherheit auch in Gegenwart dynamischer Typinspektion sicherstellt. Zu diesem Zweck führen wir eine idealisierte Formalisierung der Sprache Alice ML ein, die auf dem polymorphen λ-Kalkül basiert. Sie modelliert zentrale Konzepte des Typ- und Modulsystems: höherstufige Typen spiegeln Polymorphismus und parametrisierte Module wider, *Singleton Kinds* können Gleichheit von abstrakten Typen ausdrücken (*type sharing*), ein Konditional über Typen erlaubt Typinspektion und das Kodieren von Paketen, *Subtyping* und *Subkinding* erfassen Schnittstelleninklusion. Zudem wird *Pickling* als spezielles Konstrukt eingeführt, welches das Hantieren mit potentiell nicht-wohlgeformten Werten ermöglicht. Das wichtigste Merkmal ist jedoch *dynamische Typgenerierung*, welche abstrakte Typen realisiert. Der Kalkül modelliert damit erstmals die Interaktion von höherstufiger dynamischer Typabstraktion mit dynamischem Typ-Sharing, welche zentral ist für die Typisierung von Alice ML. Typgenerierung geht einher mit der neu entwickelten Idee von *Abstraktions-Kinds*, welche zur feinkörnigen Klassifikation abstrakter Typen höherer Ordnung benutzt werden. Auf Termebene erlauben explizite, höherstufige Konversionen (*coercions*) die nachträgliche Enkapsulation beliebig komplexer Objekte in punktuell abstrahierte Typen. Wir beweisen die Entscheidbarkeit des Typsystems, seine Korrektheit in Bezug auf die operationale Semantik des Kalküls, sowie eine einfache Eigenschaft von Abstraktionssicherheit.

Contents

1. Introduction

Computing used to be a simple affair. A user sat down in front of a dedicated machine and fed it a program whose task was to solve a well-defined problem. The computer executed the steps dictated by the program, one after the other. It was not long, and programs became interactive, had to react to the user's input while running. But still, the user worked with a single machine, and usually a single program at a time, and interaction largely followed linear paths.

And then came the Internet...

There no longer is such a thing as "a program", running on "a computer". Computers form a world-wide network of communicating processes. Activities like surfing the Web consist of concurrent interaction with countless processes, running simultaneously, on a large number of distant and diverse machines. These processes have to exchange all kinds of functionality and data to please anonymous users and handle their increasingly demanding requests. Sometimes they may fail to do so, for a variety of challenging reasons. And a few of these processes even turn out to be evil, trying to disturb the work of others on purpose!

The world of computing sure has become complicated.

Of course, we are vastly simplifying matters here, for sheer effect. But the point is: programming in today's world tends to be fundamentally different from what it used to be. Complex issues like concurrency, distribution, failure, and security could largely be ignored in traditional programming. They no longer can be today – programs now have to be *open* to communicate, be extended, move around, adapt.

The primary tool for constructing programs is – and probably will be for a long time to come – a *programming language*. So how have programming languages, and underlying concepts, been adapted to these fundamental changes? The sad truth is, most of them have not. Or have only in patched-up ways, which hardly appear adequate under scrutiny. And those languages that *do* offer serious support for open programming usually pay with substantial compromises in language design and semantics.

In this dissertation, we are trying to address some of the issues raised by an open approach to programming. We present a language design that features a novel combination of programming language concepts to support open programming in a coherent manner. We especially focus on one central but critical feature of programming languages: their *type system*. A type system helps to improve correctness and safety by performing automated consistency checks when a program is translated. Our main objective is to explore ways to encompass the incomplete type information available during translation of open programs, and ideally, to extend the type system's utility to the run-time of a program. Our thesis is as simple as the following:

> *Type systems can be reconciled with open programming without sacrificing their desirable properties.*

The remainder of this chapter is dedicated to explaining, in a little more depth, the purpose of type systems and what their desirable properties are, our understanding of open programming, and the difficulties in combining both. In due course, we will have a brief look at the current state of the art in the programming language mainstream. Finally, we present our take on attempting to improving this state.

1.1. Type Systems

Most modern, high-level programming languages incorporate a *type system*. By that, we understand a formal discipline according to Pierce's definition [Pie02]:

> *A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.*

This definition implies that a type system is inherently *static*. Nevertheless, we will sometimes explicitly talk about *static typing*, to differentiate clearly from dynamically checked languages like Lisp or Oz that are often characterised as *"dynamically typed"*, by slight abuse of terminology. In accordance with standard literature [Car97b] we will classify the latter languages as *untyped* (but *safe*).

The essence of static typing is to classify the objects used and computed in a program with descriptive entities called *types*. Typing thus provides a method for statically stating and asserting certain invariants about program objects. It can also be used to express domain-specific concepts and invariants on a more abstract level. If understood as a tool, the advantages of static typing are hence many-fold:

- **Specification.** A program design largely consists of the specification of data structures and operations. Types provide a precise language to express and communicate essential information about such designs on varying levels of abstraction.

- **Verified Documentation.** Non-trivial programs are not understandable without a certain amount of inline documentation. Type annotations allow expressing a vital part of this documentation within the programming language itself. That documentation can never get "out of sync", because it is checked by the compiler.

- **Error Detection.** A type system identifies and locates a large class of problems before a program is run. The strong invariants thus provided also increase the locality of other errors, reducing the search space for finding them. That becomes the more important the more complex data structures and operators grow.

- **Encapsulation.** Large programs have to be decomposed into smaller entities of restricted responsibility, usually called *modules*. In order to keep module interference manageable, all interaction between modules should be explicitly restricted, so that individual modules can establish internal invariants. *Abstract types* can enforce modular separation and interface compliance.

- **Maintenance.** Modifying existing programs is a difficult task, because local changes might require unanticipated adaption of remote program parts. A type checker often can point out inconsistencies after program changes and extensions.

- **Safety.** A sound type system ensures that a program accesses the machine's resources in valid ways only. For instance, *memory safety* is a property which precludes access to memory in uncontrolled ways that may lead to – potentially fatal – internal inconsistencies or interference with other programs.

- **Security.** A rich enough type system can even ensure absence of certain security violations, like attempting access to delicate resources or restricted parts of the system. Innovative type theories for purposes like this are an active research topic.

- **Efficiency.** A type-correct program is guaranteed not to encounter certain conditions during execution. Some aspects of a runtime system can be simplified and made more efficient by being able to ignore these conditions. Moreover, a compiler can make use of the typing invariants derived for individual programs to generate specialised code.

It should be mentioned that a type system also is a useful tool for the language designer and researcher: it provides a well-founded organisation principle for designing new language features, as well as a formal framework for proving certain properties about language constructs.

Of course, nothing comes without a price, so there are well-known disadvantages to static typing, which have caused even some recently developed languages to stay untyped:

- **Restrictiveness.** Every type system will rule out some useful programs.
- **Inflexibility.** A static type system usually prevents from testing and experimenting with partially correct programs.
- **Verbosity.** Most type systems require some additional declarations on part of the programmer. This is particularly true for all type systems that do not support type inference (which, unfortunately, still is common in the mainstream).
- **Language Complexity.** The requirement to have proper types associated with all language constructs can complicate language design.
- **Lack of Dynamicity.** Some language features are inherently hard to type. In particular, many dynamic concepts cannot be typed by purely static means.

The last item deserves particular attention. A type system crucially depends on enough information being available statically (at compile time) to assert that program execution is well-behaved. If a program may operate on structures about which no information is available in advance, then no exhaustive checking can be performed. Unfortunately, this is precisely the crux of the open programming scenario, as we will see in Section 1.2.

1.1.1. Type Abstraction

Type systems are particularly useful when it comes to organising larger programs. A program that grows beyond trivial size has to be decomposed into *modules*. Proper modularisation demands for the identification of suitable abstractions [Par72]. Types play an important role in this game, which Reynolds [Rey83] defines as follows:

> *Type structure is a syntactic discipline for enforcing levels of abstraction.*

Most prominently, the definition of *abstract (data) types* (*ADTs*) allows a user to explicitly create her own abstractions within a program. Type abstraction defines a 'new' type, along with a set of functions operating on values of that type. The type is considered different from any other previously available type, and the defined operations are the only means to process it. An abstract type is *implemented* by providing a *representation type* that determines how values are actually represented, but this representation is hidden inside the implementation. Only the operations that are part of the ADT implementation, can exploit that information.

The step of taking an implementation, and hiding the representation type via type abstraction, is also called *sealing* in literature [Mor73b, HP05, DCH03].

Cardelli [Car91] coined the catch phrase *typeful programming* for programming with the help of the type system, particularly through conscious use of type abstraction. Type abstraction establishes two important properties [Mor73b]:

- **Authentication.** Only the implementation can construct values of the abstract types. This allows the implementation to maintain representational invariants that would otherwise be difficult to enforce and had to be substituted by dynamic consistency checks. Depending on the problem, such dynamic checks might be very costly, or even impossible (for example, a time stamp generator can only be guaranteed to deliver fresh stamps by authentication, not by dynamic checks on stamp values).

- **Secrecy.** Only the implementation can inspect values of the abstract types. The advantage is that it enforces *loose coupling* between the abstraction and potential client code. Clients cannot rely on internals of the implementation, which allows the implementation to be changed or improved later, independently, with the guarantee not to break anything.

Together, these properties provide for a strong form of *encapsulation*. While encapsulation can also be enforced by other means (e.g. by tagging with unforgeable names), type abstraction is a particularly elegant approach, which, as an additional plus, has no operational overhead.

If the programming language semantics and the type system guarantee that the encapsulation provided by type abstraction can never be breached, then the language is called *abstraction safe*. Note that abstraction safety is a stronger property than mere type safety (but generally cannot exist without the latter).

As a running example that we will discuss more concretely in later chapters, consider computing with complex numbers. For modularity, it is desirable to introduce a type complex of complex numbers, plus a number of arithmetic operations to compute with them. However, there are at least two ways to represent a complex number: either as a cartesian pair of real and imaginary coordinates, or in polar representation, by a pair of magnitude and angle (or *argument*). The particular choice of representation is a local implementation detail, client code should abstract from it and not make any assumptions. Moreover, polar representation will usually work with the invariant that the argument stays within the interval of $[0, 2\pi[$, so that equality is most efficient to check. If the complex type is made abstract (and the language is abstraction safe), both of these properties are trivial to enforce.

1.2. Open Programming

About ten years ago, most computer applications were still developed as *closed* programs. Such a program is basically defined by a set of source files and probably accesses some local libraries. The source files to compile and the libraries to link are all fixed when the application is built, that is, before it is executed the first time. When running, such a program interacts with its environment only in severely limited ways, accessing the operating system interfaces more or less directly. Its primary means of communication is then by reading and writing raw sequences of bytes from or to files and sockets.

Although today's operating systems offer mechanisms for dynamic linking as well as more structured means for exchanging information between applications, these mechanisms tend to be low-level, system-oriented, and heavy-weight. They do not integrate well with programming languages, and few languages do provide suitable abstractions that enable their seamless use.

But software is less and less often delivered as a closed, monolithic whole. As complexity and integration of software grows, it becomes more and more important to allow flexible dynamic acquisition of additional functionality. Also, program execution is no longer restricted to one local machine only. With the Internet having gone mainstream and net-oriented applications being omnipresent, programs become increasingly distributed across local or global networks. As a result, programs need to exchange larger amounts of data, and the exchanged data is of

	Closed programming	**Open programming**
components	fixed at development time	added at runtime
interfaces	known at compile time	may change between runs
locations	accessed at link time, local	resolved at runtime, remote
origin	trusted	untrusted
import/export	trivially structured	arbitrary, higher-order
architecture	homogeneous	heterogeneous
failure potential	small	large

Figure 1.1.: Closed vs. open Programming

growing complexity. In particular, programs need to exchange *behaviour*, that is, data may include code.

We refer to development for the described scenario as *open programming*. Our understanding of open programming includes the following main characteristics:

- **Modularity**, to flexibly combine software blocks that were created separately.
- **Dynamicity**, to import *and* export software blocks in running programs.
- **Safety**, to gracefully deal with erroneous software blocks.
- **Security**, to safely deal with unknown or untrusted software blocks.
- **Portability**, to run programs independent from platform issues.
- **Distribution**, to communicate data and software components over networks.
- **Concurrency**, to deal with asynchronous events and non-sequential tasks.

The table in figure 1.1 summarises a few characteristic differences between traditional closed programming and the situation faced with open programming.

Today, open programming, specifically in the Internet context, is usually dealt with on a low level, by mixing a plethora of ad-hoc languages and technologies that are only loosely integrated. For example, average web pages are steamed by a code conglomerate of HTML, XML, PHP, Perl, Python, Ruby, ECMAScript, SQL, Flash's ActionScript, to name but a few. Significant impedance mismatch, lots of boilerplate code and overhead, and communication based on the lowest common denominator (usually text strings) are the consequences.

1.2.1. Java

Notwithstanding important precursors like CLU [LAB+79] and Modula-3 [CDG+91], whose contributions we will value later, the programming language *Java* [GJS96] was the first general-purpose language taking a consistent open-world perspective. It propagates the idea of loading components (*classes*) dynamically, and from arbitrary Internet locations. Modularity is based on the object-oriented paradigm, there is language support for safe communication with the environment and other applications through language-level objects. Java also has built-in support for concurrency, which is essential for decentralised applications. Above all, these mechanisms are platform-independent and hence allow the programmer to escape the lower realms of system programming.

As far as the programming language mainstream is concerned, Java heralded a shift of paradigm: an application is no longer seen as a monolithic entity built from statically anticipated components, but as a dynamic service process that acquires additional components as the need arises.

On the other hand, many of the open programming concepts found in Java are quite conservative and ad-hoc. In particular, its support for object exchange (*serialisation*) is comparably primitive and does not directly encompass higher-order use cases that would require transmission of code. The Remote Method Invocation protocol, *RMI* [WRW96], and other such frameworks later added to Java to address inter-process communication, work around this by transmitting class files separately if not available at destination site. But since classes are identified only by name this is a rather fragile approach.

The most significant weakness of Java, however, and our main concern in the context of this dissertation, is that its static type system is almost meaningless in the face of open programs, as we will discuss in Section 1.3.1.

1.2.2. Oz

While Java certainly has pioneered the open programming idea in the mainstream, more research-oriented languages have carried it further, seeking for more expressive and more principled incarnations of the respective mechanisms. In particular, the concurrent constraint programming language *Oz* [Smo95, VH04, Moz04] has become an important platform for investigating and implementing related concepts in a practical context [DKSS98, Kor06].

Open programming in Oz is centered around generic support for *pickling*, which allows almost arbitrary values (including procedures and their code) to be imported and exported by processes. Based on pickling, Oz features a flexible component system with lazy dynamic linking, and rich support for distributed programming. In particular, Oz pioneered the idea of representing components as pickles.

The main omission of Oz is the lack of a static typing discipline. Oz has been designed as a safe but untyped language to enable free experimentation with new ideas. As the language has stabilized, an obvious question is how to reconcile the evolved concepts with a type system. In brief, the practical part of our work is trying to give an answer to part of that question: in a nutshell, it takes the essence of the open programming facilities found in Oz and developes a typeful counterpart. Alice ML also improves on Oz by increasing simplicity and regularity of the underlying concepts.

1.3. Typed Open Programming

The characteristics of open programming prevent full static type checking – at least some checks have to be performed at runtime. For example, if a program loads an object from a file, the compiler has no way of knowing in advance what actual type this object will have, since it usually will have been constructed outside the respective program.

In this light it is not surprising that the most advanced support for open programming can be found in "dynamically typed" languages, like Oz or Lisp/Scheme. These languages simply make no assumptions about types, but instead perform dynamic checks every time an operation is about to be performed that requires a value to have a certain shape. Consequently, the aforementioned problem does not arise. It should be noted, however, that this approach often side-steps some of the issues of open programming. Some of the problems we will discuss exist in untyped languages as well. For example, unpickling is unsafe in the current implementation of Oz, despite its dynamic checks – the run-time system cannot guarantee that loaded code

is well-formed, for example, because that would again require some verification akin to type checking on the internal code format.

1.3.1. Java

When it comes to typed languages with support for open programming, the reference point surely is Java, which we already introduced in the previous section.

With respect to typing, Java is interesting because it uses a hybrid approach: it has a static type system, but also performs run-time checks like "dynamically typed" languages. The reason is that the type system actually is too weak to really encompass open programming: class types are essentially identified by their syntactic name only (more precisely, by name and class loader [LB98]), and no assumptions are checked about a class signature when it is loaded. Instead, checks are performed on individual method calls. That means that a class loaded at run-time under the name C needs to bear no resemblance to the class found under the same name at compile time. In an open program, invocation of a method of C may thus potentially result in a NoSuchMethodError or a related exception, i.e. a dynamic type error [LY96].

The effect can be demonstrated with a simple example, shown in Figure 1.2. Assume there are two classes, Database and App. The former implements an abstraction over database access, which allows global locking of the database file. The latter is an application using that class. There are two versions of the Database class: the first names the method to release a locked database release, while in the latter version the name has been changed to unlock. Now assume that the application has been compiled against version 1, but is run in the presence of version 2. At run-time, it will happily load the updated Database class, open the database file "/serve/my.db", and lock it. All these operations succeed, because the respective methods have not changed. But when the application tries to release the database, it will encounter a NoSuchMethodError exception, because the expected release method is not part of the class – leaving the database file in locked state with no way to recover!

The problem is that Java does not perform any structural check ensuring that App's assumptions about the Database class are still valid *before* allowing access to it. The checks are done only incrementally, but at that point it may already be too late. This is precisely the situation a type system should prevent.

Technically, Java's type system may hence be considered unsound in the presence of open programming (although it is not unsafe – besides other dynamic checks, safety of dynamic loading is ensured by a process called *byte code verification*). This lack of proper type soundness particularly has ramifications on inter-process communication through serialisation (via persistence or RMI): classes are identified by name, but there is no guarantee that different sites (or one site at different times) actual use the same classes. Unexpected deserialisation failures may be a result, or worse, values may successfully deserialise, but fail to meet semantic invariants of the local class implementation. In other words, Java cannot prevent accidental breach of abstraction safety across process boundaries.

A simple but overly restrictive solution to this problem might be to hash class files with a cryptographic checksum. This would rule out incompatible changes to a class, but also all *compatible* changes, i.e. simple interface extensions. In an open programming scenario such inflexibility is not desirable.

In Chapter 7 we will present a more pleasent approach to dynamic linking and get back to this example in Section 7.3.1.

```
// Database.java, version 1
class Database
{
    Database(String path) { ... }
    public void lock() { ... }
    public void release() { ... }
    ...
}
```

```
// Database.java, version 2
class Database
{
    Database(String path) { ... }
    public void lock() { ... }
    public void unlock() { ... }
    ...
}
```

```
// App.java
import org.my.database.*
class App
{
    public static void main(String[] args)
    {
        Database db = new Database("/serve/my.db");
        db.lock();
        ...   // use data base
        db.release();
    }
}
```

Figure 1.2.: Dynamic typing in Java

1.3.2. Dynamics

A more promising approach for integrating dynamic typing into a statically typed language has been known for a long time: *dynamics* add the bits of dynamic typing necessary to embrace operations like dynamic loading without compromising soundness of the type system. They have first been suggested by Mycroft in an unpublished draft [Myc83] and later worked out in detail by Abadi, Cardelli et al. [ACPP91].

The latter work proposes a single universal type called dynamic that is basically an infinite sum of all (monomorphic) types. Values of this type are constructed by injection:

dynamic $exp : \tau$

Projection then requires a case distinction over all types:

typecase exp **of**
x : int $\Rightarrow$ exp_{int}
x : bool $\Rightarrow$ exp_{bool}
x : $\alpha \times \beta \Rightarrow$ $exp_{\times}$
x : $\alpha \rightarrow \beta \Rightarrow$ $exp_{\rightarrow}$
else exp'

Dynamics can be used to define interfaces for type-safe input/output of language-level values, simply by restricting these I/O operations to values of type dynamic. For example,

write : string $\times$ dynamic $\rightarrow$ unit
read : string $\rightarrow$ dynamic

would be enough as a generic interface for persistence: because dynamic is a universal type, values of any type can, in principle, be communicated under its hood. After retrieving a dynamic value

with read, it has to be inspected with typecase to check it against the type expected by the application, or take appropriate failure measures otherwise.

The significant advantage of dynamics, as opposed to the hybrid approach taken by Java, is that they do not undermine the basic soundness properties of the type system. Dynamic typing – and the potential for dynamic type failure – is completely isolated in the explicit typecase construct. All other code is statically type-safe. The static type system does clearly indicate where knowledge about the type of values is limited, by assigning type dynamic. The boundaries of static typing are thereby evident in a program.

There has been a range of follow-up work improving on dynamics, particularly by allowing polymorphic content [LM93, ACPR95]. Later work has decoupled dynamic case switching on types by introducing *type analysis* as a stand-alone construct [HM95, CWM98, TSS00, Wei02], thereby allowing to reduce dynamics to values of existential type $\exists\alpha.\alpha$.

So, if everything is roses, why have dynamics not been widely adopted in typed programming languages? Our view is that there are two main problems with dynamics that have prevented their wide-spread adoption so far:

- *Pragmatically*, dynamics as proposed are too inconvenient for many applications. A general type case with the ability to handle polymorphism is complex and sometimes not straightforward to use. More seriously, it is too fine-grained for the purpose of open programming, because it works on the granularity of single values. When one wants to transfer a complex program component housing many entities it is highly undesirable to require encoding its content in, say, a single tuple value. Moreover, components may contain other entities than plain values, that cannot be directly stored in a value. Furthermore, type case is too inflexible to be used in evolving software systems, because the content type must be known very precisely in order to unpack a dynamic. Subtyping would be desirable, but raises coherence issues in conjunction with type case [ACPR95].

- *Technically*, the presence of dynamics (as well as general type analysis) destroys *parametricity* [Rey83, BFSS89, ACC93], a valuable property of polymorphically typed languages. Intuitively, parametricity means that an expression will always evaluate uniformly, no matter how any of its free types is instantiated, i.e. evaluation does not depend on types. That is a desirable abstraction property, for which Wadler coined the slogan *"theorems for free"* [Wad89]. For example, in plain polymorphic lambda calculus, a function of type $\forall\alpha.\alpha \to \alpha$ is known to be the identity function. This is no longer true with dynamics or more general forms of type analysis.

  The loss of parametricity particularly affects the semantics of abstract types: in the standard model of type abstraction, which is based on scoping of existential quantification [MP88], abstraction safety is lost because the representation of an abstract type can be rediscovered dynamically, effectively allowing the definition of casts between abstract types and their representations.

  Lack of parametricity also has significant impact on the implementation and efficiency of a language: because types become part of the operational semantics, they can no longer be erased, as it is done in almost all practical implementations of programming languages today. Instead, polymorphism requires a type passing implementation.

Consequently, no practical programming language implements dynamics in their full beauty, nor does any seriously use it to support open programming. Some languages, namely Clean [Pv00], Mercury [HCS$^+$01], and the GHC implementation of Haskell [MPO02] incorporate simple variants of dynamics, but except for Clean they make no use of it in their libraries. The GHC

implementation of dynamics even is unsafe, as it is not primitive but uses user-defined strings to identify types.

1.4. Contribution

In this dissertation we develop a concrete and realistic language design for typed open programming. We focus on how typing interacts with open programming, primarily considering modularity and dynamicity. We also cover other aspects of open programming, like concurrency and distribution, to a limited extent.

Our approach is based on the idea underlying dynamics, but does not suffer from the problems described above. In particular, it is easier for the programmer to understand and more convenient to use than dynamics. Furthermore, to reconcile it with type abstraction, we develop a formal semantics for type abstraction that is not compromised by the lack of parametricity.

In detail, our contributions are the following:

- We describe the design of a concrete, non-toy language, *Alice ML*, which is based on Standard ML [MTHM97], and incorporates a range of open programming features in a coherent way. It thus combines a particularly strong type system with flexible support for *typed* open programming. The presentation we give extends on previously published work [RLT+06, Ros06].

- In particular, we introduce *packages* as a variation of dynamics that differ in that they carry *modules* instead of plain values. This allows to exchange arbitrary bundles of objects in convenient ways. When unpacked, a package is matched against an expected interface in an intuitive manner already known from the module system. Moreover, it implies a natural notion of interface inclusion that makes these checks robust against evolutionary changes.

- While we cannot avoid the loss of *parametricity* in principle, our language design confines it to the module level – leaving parametricity intact for all uses of polymorphism occurring in the underlying core language. That maintains "theorems" where they are usually exploited, and allows for an efficient type erasing implementation.

- On top of packages and pickling we are able to define, as a layer of straightforward syntactic sugar, a powerful system of first-class *components*. It supports type-safe lazy dynamic linking, dynamic creation, and user-programmed linking policies for security.

- We address the problem of abstraction safety (which still persists if parametricity is given up for modules) by formulating a non-standard theory of abstract types in a calculus with *dynamic generation* of fresh type names instead of existential quantification. Unlike previously published work [Ros03a, Ros06], the calculus incorporates both type generation and *translucency*, expressed by *singleton kinds*, and hence captures the essentials of advanced module systems.

- The calculus employs *coercions* as a means of giving a reduction semantics for ADTs. By generalising to higher-order coercions, we are able to recover the semantics of generative *sealing* for creating abstractions *a posteriori*, as it is common in module systems. The calculus seems to be the first that combines higher-order coercions with dependent and singleton kinds, a combination that raises a number of technical difficulties.

- Our calculus also includes a simple but informative abstract semantics of *pickling*. This allows us to identify two independent forms of dynamic checking that have to be performed when unpickling.
- Our language design has been implemented in the *Alice Programming System* [Ali03], which is a full-scale programming environment available as open source software. The system is being used in research and teaching.

Altogether, these contributions show that typed open programming is indeed *possible*, thus substantiating our thesis. Because our approach even maintains abstraction safety across processes, we claim that it in fact enables *typeful* open programming (see Section 1.1.1). Moreover, we believe that the design of Alice ML is elegant and convenient enough to also make it suitable as a *practical* language for typed open programming, although a thorough evaluation of that stronger claim lies beyond the scope of the thesis.

Last but not least, our approach is basically *simple*. Its essence is packages and pickling, all other language features we discuss are mainly for convenience. The approach thus is potentially applicable to *any* programming language that provides (1) a structurally typed module system, and (2) a generic pickling mechanism – or any language to which these can be added. It does not matter what programming paradigm underlies the respective language - while we focus on a functional setting here, the aforementioned concepts and the notion of component we derive are not limited to it.

1.5. Structure

The rest of this dissertation will be split in two parts, developing our approach from two sides.

In Part 1 we address the practical side and present and motivate the design of the concrete programming language, Alice ML. We incrementally introduce the relevant language concepts, explaining their semantics informally, and demonstrating their use with concrete program examples. The notion of component is introduced as derived syntax on the language level.

- Chapter 2 gives a brief overview of the features of Alice ML and its relation to the underlying Standard ML (SML) programming language.
- Chapter 3 motivates and explains extensions to the SML *module system*.
- Chapter 4 introduces *packages* as the central means for dynamic typing.
- Chapter 5 describes *pickling*, the mechanism for importing and exporting higher-order language values.
- Chapter 6 introduces *futures*, enabling light-weight concurrency and lazy evaluation.
- Chapter 7 presents the dynamic *component* system, and describes its decomposition into a combination of the concepts from the preceding chapters.
- Chapter 8 explains the language's approach to *distributed programming*, which again is based on the concepts previously introduced.
- Chapter 9 gives a brief overview of the *implementation* and discuss possible future work with respect to language design and implementation.

In Part 2 we will cover the theoretical side by developing a formal calculus and type system modelling the essentials of this language. We distill the relevant concepts and put them into the context of a standard higher-order typed λ-calculus, for which we prove soundness properties as well as a moderate abstraction result. In this calculus, we can define sealing as derived syntax.

- Chapter 10 introduces the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-*calculus* and motivates each of its features by relating it to examples from Alice ML.
- Chapter 11 discusses the *type language* of the calculus, particularly the semantics of singleton kinds and the novel notion of *abstraction kinds*, and gives various properties, including decidability.
- Chapter 12 discusses the *term language* and its operational semantics, with focus on dynamic typing and type generation, and gives a soundness result and an abstraction property we call *opacity*.
- Chapter 13 extends the calculus with higher-order constructs to encompass *higher-order abstraction* and shows how these constructs allow the encoding of higher-order *sealing*, which is proved correct.
- Chapter 14 concludes and discusses possible directions for future work.

At the end of each chapter we discuss prior work related to the topic of the respective chapter.

Part I.

Introducing Alice ML

2. Overview

The following chapters provide an overview of the functional programming language *Alice ML*. Alice ML has been specifically designed to support typed open programming. It extends the conventional feature set of functional languages with a novel combination of concepts supporting concurrency, distribution, and particularly, type-safe import and export of program components – that is, typed open programming. We present the central concepts of Alice ML, motivate them with examples, and show how they play together as a coherent whole.

This presentation is not intended to be a language specification. We keep the description informal and concentrate on the essentials of the semantics. Theoretic considerations are left to the second part of this thesis, where we look at a more idealised language that lends itself better to theoretic study. A formal specification of a significant subset of Alice ML is given in a technical report [Ros05].

The material in this part of our thesis extends on previously published work describing aspects of the design of Alice ML [RLT+06, Ros06]. We will discuss it along with other related work at the end of each chapter.

To warm up, we start off in this chapter by giving a brief recap of Standard ML, on which Alice ML is based, and summarise the extensions that Alice ML provides. We also introduce the Alice Programming System [Ali03], that implements the Alice ML language. In the following chapters we then present the key concepts of Alice ML in more detail: higher-order modules (Chapter 3), dynamic typing with packages (Chapter 4), high-level import/export with pickling (Chapter 5), concurrency with futures (Chapter 6), the component system (Chapter 7), and distributed programming (Chapter 8). On the way, we introduce many of the open programming facilities provided by the Alice library, which make use of all the aforementioned concepts.

2.1. Standard ML Heritage

Alice ML is a functional programming language in the tradition of *ML*, a family of typed functional languages with pragmatic support for imperative programming. Alice ML has been designed as a mostly conservative extension of the ML incarnation known as *Standard ML* [MTHM97], or simply *SML*.

ML was originally developed in the late 1970s by Milner as a *Meta Language* for the LCF proof-checking system [GMW79]. Over the years, its development brought three major innovations to the field of programming languages: the introduction of *polymorphic typing* with *type inference* [Mil78, DM82] (the same idea already had been discovered earlier in the context of combinatory logic [Hin69]), a parametric module system based on dependent types [Mac86, Mac84], and rigorous formal specification of a complete language, the *Definition of Standard ML* [MTH90, MTHM97].

Today, there are several implementations of Standard ML, plus a major dialect called *Objective Caml* [Ler03], which particularly adds a rich object-oriented sublanguage. ML enjoys a prominent position in language research and teaching, thanks to its clean design and specification, and the expressive yet robust higher-order semantics. The main features of the ML family of languages as of today can be summarized as follows:

- **Functional core.** A higher-order functional language with a strict evaluation regime constitutes the core language.
- **Imperative features.** Besides a pure functional subset, imperative constructs like exception handling and mutable references are available.
- **Algebraic data types.** User-defined data types come with a concise notation for pattern matching.
- **Polymorphic type system.** A strong static type system provides parametric polymorphism and supports type inference.
- **Parametric module system.** The module system is a functional language on its own, with strong support for encapsulation and parameterisation.
- **Safe semantics.** No program can ever "go wrong", i.e. access computational resources in an invalid or unsafe way.

Due to its clean formal specification, SML is a particularly well suited vehicle for programming language research. Its comparatively expressive and well-studied type and module system is a good match for exploring typed open programming. Consequently, Alice ML has been designed as a conservative extension to the revised version of Standard ML, with some ideas borrowed from Objective Caml.

Giving an introduction to Standard ML, or to functional programming in general, is out of the scope of this work. We refer the interested reader to the available literature [Har06, Pau96, Ull97, HR99]. In the following, we assume a working knowledge of SML, or some other dialect of the ML family of languages. Where important, we will briefly summarize central concepts of SML alongside our presentation of Alice ML.

2.2. Extensions and Oz Heritage

Alice ML has been designed as a conservative extension of SML. Most SML programs can be readily interpreted as Alice ML programs. However, while Alice ML is backward compatible with SML, it also features significant extensions:

- **Futures.** A future is a place-holder for a yet undetermined value, usually computed by a concurrent thread. Different flavours of futures provide *laziness*, light-weight *concurrency*, and *promises*, a restricted form of logic variable.
- **Higher-order modules.** Structures, functors, and signatures can be defined locally and composed arbitrarily. In particular, signatures can contain signature members, and like types, these may be abstract or concrete.
- **Packages.** Modules may be passed as dynamically typed first-class values by injecting them into a special type known as `package`. A package value carries information about the contained module's signature. When accessing the package, the signature is verified by a dynamic type check. Packages are the basis for type-safe persistence and distribution. Thanks to higher-order modules, packages can contain arbitrary language entities.
- **Pickling.** A generic mechanism for import and export of language-level data structures, including code. A pickle is a self-contained, platform-independent external representation of an Alice ML value. The library interface to pickling is type-safe because it operates on packages.

- **Components.** Programs are decomposed into separate components that are connected via import relations. Linking of imported components is performed dynamically and on a by-need basis, and involves dynamic verification of signature assumptions. Components are expressed in terms of packages, pickling and lazy futures.

- **Distribution.** Alice processes on different sites can connect to each other and safely exchange almost arbitrary Alice data structures. Processes may create *proxies* to local functions that, when applied, transparently perform a remote procedure call to the original process. Futures play an important role to deal with asynchronicity and latency in distributed programming.

Each of these concepts is realised by augmenting the basic language with only a few simple and orthogonal, yet general and powerful constructs. These constructs extend the semantics of the original language in considerable ways, while maintaining most of its valuable properties.

Most of the features specifically targeting open programming – i.e. pickling, components and distribution – are inherited in one form or another from the concurrent constraint programming language Oz [Smo95, VH04, DKSS98, Kor06, Moz04]. We will discuss the relation where appropriate.

Packages and a strongly typed model of components are the primary contributions of Alice ML, and the main concern of this thesis. In the following chapters, we will introduce and motivate all of the above features, because they all play together to ease open programming. But we will particularly focus on packages and components and the issues with dynamic typing that they address.

2.3. The Alice Programming System

Alice ML has been implemented in a fully-featured programming system [Ali03]. Having at hand not only a toy implementation, but a fully-featured prototype that allows playing with the language under realistic conditions, proved immensely helpful during the design of Alice ML. The Alice System provides a compiler for the full language, an efficient platform-independent virtual machine with support for distributed programming, several interactive development tools, and extensive libraries.

Another central feature of the Alice Programming System is its rich support for constraint programming [Apt03, Sch02], based on the Gecode constraint programming library [Gec05, ST05]. Since constraint programming in Alice ML is purely a library issue and does not require special language support we will not discuss that aspect of Alice further in this thesis. We refer the reader to the Alice documentation [Ali03] for details.

The first pre-version of Alice (still based on the Mozart Programming System [Moz04]) was released in December 2002. The official 1.0 release of Alice followed in 2004, and there have been regular updates since. The Alice Programming System is freely available as open source software, and runs on all major platforms.

2.4. Summary

- Alice ML is a language designed for typed open programming.

- It is a mostly conservative extension of Standard ML.

- Notable extensions are future-based concurrency and laziness, higher-order modules, dynamically typed modules (packages), pickling, components, and distributed programming features.
- Alice ML has been fully implemented in the Alice Programming System, which is available as open source software.

3. Higher-Order Modules

Large-scale programming requires the ability to break down the complexity of programs, and to avoid duplicating work for different programs or different parts of a single program. Hence, *modules* allow the decomposition of programs into units that implement dedicated aspects of its logic and functionality. They particularly support the definition of abstractions, which reduces coupling and increases the potential for *re-use* in different contexts.

Modules thus play a central role in structuring ML programs. The ML module system still defines the state-of-the-art in language design for typed modular programming. The following is a brief summary of the main features of the SML module system:

- **Structures** are the basic form of module. They are containers that can carry arbitrary core language entities, like values and types, as well as nested structures. Structure members are named and can be accessed by dot notation.

- **Signatures** are the types of structures. They describe the members of a structure. Signatures are *translucent*, that is, types can be described either *concretely* (*transparent*, *manifest*) or *abstractly* (*opaque*). In the former case, the type equivalence is revealed, while in the latter it is not. Every structure has a *principal signature* that is fully concrete.[1]

- **Matching** is subtyping on signatures. The subtyping relation consists of two dimensions: *enrichment* allows a subsignature to extend a supersignature with additional members; *instantiation* allows a subsignature to concretise types that are abstract in the supersignature. The signature language contains syntax for *refining* a given signature along both dimensions.

- **Sealing** is an operation that ascribes a signature to a structure, using the syntax *strexp* :> *sigexp*. The identity of any type described abstractly in the signature is thereby hidden outside the sealed structure. Sealing is said to be *generative*, because it effectively generates new abstract types that are distinct from all others.

- **Functors** are (first-order) functions over structures. By applying a functor to a structure a new structure is computed whose content *and* signature can depend on the supplied argument. Functors are also said to be *generative*, because applying the same functor twice to the same argument produces distinct abstract types, if the functor returns a sealed structure.

- **Stratification** describes the fact that the module language is completely separated from the core language. Core language expressions cannot contain module expressions and declarations. Modules are solely a means for structuring large programs and describing program architectures.

When it comes to open programming, good language support for modularity is essential. The SML module system is quite advanced in comparison to most other languages, but still limited by its restriction to first-order parameterisation and its stratified setup. Since the design of

[1] Although not all principal signatures are expressible in the surface syntax.

the original SML module system there has been a long line of work on extending its expressive power (Section 3.4). Alice ML integrates some of that work by extending the module system of SML in three ways:

- **Higher-order functors**. Functors can be arbitrarily nested and parameterised over other functors.
- **Nested and abstract signatures**. Signatures can be wrapped in structures and be specified abstractly.
- **Local modules**. Modules can be defined within core let expressions.

These extensions allow new and more flexible forms of abstraction. Modular techniques are applicable at more fine-grained levels, which is particularly important for dealing with packages (Chapter 4). Simultaneously, the extensions generalise the module language such that it is rendered more regular and compositional than in plain SML. Most importantly, structures become a general container that can carry *all* sorts of language entities (values, types, modules, signatures, even fixity directives). Again, this is crucial to the expressive power of packages, and it is essential to the design of the Alice component system (Chapter 7).

Higher-order modules are neither new nor exclusive to Alice ML. Consequently, this chapter only gives some motivation for adding them and an overview of their specific design in Alice ML. Their semantics and theoretical underpinnings have been investigated extensively in literature [HMM90, MT94, HL94, Ler95, DCH03, Rus98].

3.1. Higher-Order Functors

The module language of Standard ML is first-order: functors can take structures as arguments and return structures as results, but they cannot accept or return other functors. Neither can functors be put into structures. The first-order restriction on functors can sometimes hamper modular design, even without the particular requirements of open programming.

Consider a conventional example: a compiler that consists of several phases (stages). Each phase translates one intermediate program representation into another one, using some contextual information. A suitable signature for describing a compiler phase might be the following:

```
signature PHASE =
sig
    type in_rep
    type out_rep
    type context
    val translate : context → in_rep → out_rep
end
```

A compiler is essentially a pipe of different phases. To construct it, we need a way to plug phases together. We can achieve this by repeatedly applying a functor for composing two consecutive phases:[2]

```
functor ComposePhases
    (structure Phase1 : PHASE
     structure Phase2 : PHASE where type in_rep = Phase1.out_rep) =
struct
    type in_rep = Phase1.in_rep
```

[2]The Alice ML compiler itself is constructed this way.

```
    type out_rep = Phase2.out_rep
    type context = Phase1.context × Phase2.context
    fun translate (c1, c2) = Phase2.translate c2 ∘ Phase1.translate c1
end
```

But what if each phase was additionally parameterised over some configuration structure? That is, each phase was not a structure, but a functor of the type

CONFIG → PHASE

for some given signature CONFIG describing the configuration structure. In that case, the composition functor had to take such functors as arguments, plus the actual configuration structure for passing it forth to its operands. In a first-order module system, we cannot express such a composition functor.

Building on a long line of previous work that is briefly discussed in Section 3.4, Alice ML extends SML's module system to a full higher-order language. Structure expressions are generalised to module expressions, that may consist of functor abstractions. A functor is a module expression of the form

fct *strid* : *sigexp* ⇒ *strexp*

The new keyword fct denotes a module-level lambda, analogous to fn denoting a core-level lambda in SML. A functor is applied using straightforward functional notation:

$strexp_1$ $strexp_2$

Figure 3.1 shows the basic syntax of Alice ML module expressions. Besides functor expressions, it contains unpacking, which belongs to Alice ML's package mechanism and will be discussed in Chapter 4, and expression forms related to futures, explained in Section 6.6. The syntax subsumes SML, but while SML separates the name spaces of structure identifiers and functor identifiers, Alice ML gives up this distinction in order to have a natural treatment of higher-order modules.[3]

Since structure expressions may consist of functors, a structure declaration may introduce a functor in Alice ML.[4] Functor declarations as known from SML have been degraded to a derived form, very much like the core provides function declarations with fun as sugar for plain val declarations. For convenience, they have been extended to support curried functors (Figure 3.2).

Along with modules themselves, the signature language has been extended to cover functor signatures, as is appearent from Figure 3.1. Functor signatures are dependent types [Mac86], so there needs to be a binder for the argument. We reuse the keyword fct for it:

fct *strid* : $sigexp_1 \rightarrow sigexp_2$

If the binder is not needed, i.e. $sigexp_2$ is not dependent on *strid*, a functor signature may be abbreviated as

$sigexp_1 \rightarrow sigexp_2$

[3] The merging of structure and functor name spaces introduces an incompatibility with SML. However, it rarely seems to matter in practice. The alternative route taken by Moscow ML [RRS00], which makes syntactic distinctions to keep the name spaces apart, was considered but regarded too subtle.

[4] Nevertheless, the keyword structure and the syntactic classes *strid*, *strexp*, etc. have been inherited unchanged from SML, although module would have been more appropriate – a new declaration phrase either had introduced unnecessary syntactic redundancy, or sacrificed compatibility with SML.

strexp	:=	*longstrid*	module identifier
		struct *dec* **end**	structure
		fct *strid* : *sigexp* ⇒ *strexp*	functor
		strexp strexp	functor application
		let *dec* **in** *strexp* **end**	local declaration
		strexp : *sigexp*	ascription
		strexp :> *sigexp*	sealing
		unpack *exp* : *sigexp*	unpacking
		lazy *strexp*	lazy evaluation
		spawn *strexp*	concurrent evaluation
sigexp	:=	*longsigid*	signature identifier
		sig *spec* **end**	structure
		fct *strid* : *sigexp* → *sigexp*	functor
		sigexp **where type** *tyvarseq longtycon* = *ty*	specialisation

Figure 3.1.: Module and signature expressions in Alice ML

Figure 3.2 defines the meaning of this and other derived forms. In particular, we allow abbreviating structures and signatures by just enclosing them in parentheses instead of a keyword pair, a generalisation of SML's derived forms for functor arguments that comes naturally with higher-order modules and currying and is convenient in conjunction with packages (Chapter 4).

With functor expressions and signatures, the module language represents a higher-order functional language. We can now formulate the desired higher-order version of the phase composition functor. For convenience, we make it a curried definition:

```
functor ComposePhases
    (type in_rep; type inter_rep; type out_rep)
    (MkPhase1 : CONFIG → PHASE where type in_rep = in_rep and out_rep = inter_rep)
    (MkPhase2 : CONFIG → PHASE where type in_rep = inter_rep and out_rep = out_rep)
    (Config : CONFIG) : PHASE =
let
    structure Phase1 = MkPhase1 Config
    structure Phase2 = MkPhase2 Config
in
    struct
        type in_rep = Phase1.in_rep
        type out_rep = Phase2.out_rep
        type context = Phase1.context × Phase2.context
        fun translate (c1, c2) = Phase2.translate c2 ∘ Phase1.translate c1
    end
end
```

A remaining nuisance is the need to add the auxiliary type parameters in_rep, out_rep and inter_rep to denote the resulting input/output representations as well as the intermediate representation shared between both phases. Without making inter_rep explicit, type sharing between the two phase parameters could not be expressed. Without making in_rep and out_rep explicit, type sharing between the parameters and the functor's result signature could not be expressed (the result signature is implicit in this example, but it is clear that without the constraints on the MkPhase functors, the resulting types Phase1.in_rep and Phase2.out_rep would be fresh and

Derived form	Equivalent form	
(*dec*)	**struct** *dec* **end**	
(*spec*)	**sig** *spec* **end**	
fct (*spec*) ⇒ *strexp*	**fct** *strid* : (*spec*) ⇒ *strexp*′	(*)
fct (*spec*) → *sigexp*	**fct** *strid* : (*spec*) → *sigexp*′	(*)
$sigexp_1$ → $sigexp_2$	**fct** *strid* : $sigexp_1$ → $sigexp_2$	(*)
functor *strid* (arg_1) ... (arg_n) ⟨:⟨>⟩ *sigexp*⟩ = *strexp*	**structure** *strid* = **fct** arg_1 ⇒ ... **fct** arg_n ⇒ *strexp* ⟨:⟨>⟩ *sigexp*⟩	
functor *strid* (arg_1) ... (arg_n) : *sigexp*	**structure** *strid* : **fct** arg_1 → ... **fct** arg_n → *sigexp*	

(*) The module identifier *strid* is fresh. Any identifier *id* bound in *spec* is replaced by *strid.id* in *strexp* (respectively, *sigexp*), yielding *strexp*′ (respectively, *sigexp*′).

Figure 3.2.: Syntactic sugar for Alice ML modules

fully abstract). Both of these limitations are a consequence of Alice ML not supporting *applicative* functors. Were the latter available, the composition functor could be formulated more succinctly:

```
functor ComposePhases
    (Config : CONFIG)
    (MkPhase1 : CONFIG → PHASE)
    (MkPhase2 : CONFIG → PHASE where type in_rep = MkPhase1(Config).out_rep)
    : PHASE =
let ... (* as before *)
```

Here, the language has been extended to allow type identifiers that contain functor applications, like MkPhase$_1$(Config).out_rep above. Obviously, such notation only makes sense if the functor is applicative, i.e. each application delivers the same abstract type identities – in contrast to SML's generative functors, where each application generates new type names. Although the example demonstrates the usefulness of applicative functors, the need for them does not appear to be very pressing in practice. We hence left the addition of applicative functors for future work.

3.2. Local Modules

Standard ML consists of two sublanguages: the core language that contains all the usual elements of a functional programming language, and the module language, which sits on top of it. Both sublanguages are strictly separated. In particular, core expressions cannot contain any module expression – they may only refer to structure names in scope.

Alice ML relaxes the strict stratification and allows modules to be declared in local let expressions. This change renders the language significantly more uniform.

The main purpose of local modules is to make working with *packages* (Section 4) feasible. Packages are values that carry modules. The need to be able to define modules locally follows immediately. We will see examples of this in Chapter 4.

Local modules can also be useful on their own. To see why, consider a function for ordering and removing duplicates from string lists, where the ordering is given as a parameter. Such a function can be easily implemented given an appropriate library functor for representing ordered sets and the ability to define modules locally:

```
fun sortWithoutDups compare =
let
    structure Set = MkRedBlackSet (type t = string; val compare = compare)
in
    Set.toList ∘ foldr Set.insert Set.empty
end
```

Note that due to the higher-order nature of the module language, even functors can be defined locally.

Another argument for local modules is catching exceptions. Module expressions may raise exceptions stemming from contained core expressions, failed module futures (Section 6.6), or dynamic type mismatches of packages (Chapter 4). Especially packages can cause module-level exceptions regularly, and they have to be handled programmatically. In SML, there is no way to handle such exceptions, they will always propagate to the toplevel and cause the program to terminate abnormally. With local modules on the other hand, it is often possible to wrap exception handlers around module declarations.[5] In the following example, suppose the Init functor may fail with an exception.

```
structure Helper = Init ()
fun f (x, y) = Helper.f (x, 2*y)
```

It is not possible to catch the potential exception – the program will just terminate, probably with printing a cryptic message about an uncaught exception. Such behaviour is only slightly better than the sort of uncontrolled crashes more low-level languages are infamous for. With local modules, the program can be rewritten to provide more user-friendly behaviour:

```
val f =
let
    structure Helper = Init ()
in
    fn (x, y) ⇒ Helper.f (x, 2*y)
end
handle e ⇒ (print "initialisation failed, please try again later\n";
            OS.Process.exit OS.Process.failure)
```

A refined design might even perform several retries for applying the functor.

3.3. Local and Abstract Signatures

Signatures are the types of modules. Like for core types, SML allows declaring named signatures for convenience. Unlike types, signatures may only be declared at the toplevel, and not, for example, as structure members.

3.3.1. Local Signatures

Alice ML removes any such restrictions on signature declarations. In the same way modules may be declared anywhere, it is legal to put a signature declaration into local scope. A particular consequence is that signatures may appear as structure members. And indeed, signatures may be projected from structures, using long signature identifiers.

Consider a factory module that provides an interface for creating certain structures:

[5] A more expressive alternative might be to extend the module language with exception handlers. However, we have not yet encountered practical examples that make the involved duplication in the language design worthwhile.

```
structure Factory =
struct
    signature SHAPE = sig type dim; ... end
    functor New (type dim) : SHAPE = struct type dim = dim; ... end
end
```

The signature and functor might be accessed as follows:

```
structure Shape : Factory.SHAPE = Factory.New (type dim = int)
```

But what is the signature of the structure Factory containing SHAPE? In order to enable expressing it, the Alice ML signature language supports nested signatures, by providing signature specifications. The signature of structure Factory might be specified as

```
signature FACTORY =
sig
    signature SHAPE = sig type dim; ... end
    functor New (type dim) : SHAPE where type dim = dim
end
```

Such a signature specification corresponds to a concrete type specification. That is, a structure matches the signature FACTORY if and only if it defines an equivalent signature member SHAPE.

3.3.2. Abstract Signatures

So far, local signatures contribute mere convenience – for example, the ability to put signatures into structures accounts for better name space management. Semantically, they could as well be lifted to the toplevel, or inlined. However, Alice ML goes one step further by introducing *abstract signatures specifications* that describe a nested signature abstractly:

```
signature sigid
```

An abstract signature specification can be matched by any signature definition, including functor signatures.

Nested, and especially abstract signatures, are less standard than the previous higher-order features we presented for modules. They increase the expressiveness of the module language significantly, particularly by enabling us to define *polymorphic functors*, i.e. functors that can operate on modules of arbitrary signature. The polymorphic Apply functor is an exemplification:

```
functor Apply (signature S; signature T) (F : S → T) (X : S) = F X
```

When the functor is applied, the desired signature instantiations have to be given explicitly, determining the actual functor signature:

```
structure Set = Apply (signature S = ORDERED; signature T = SET) MkRedBlackSet Int
```

The Alice ML library contains several polymorphic functors to provide certain generic module-level functionality. Most prominently, they are used in conjunction with packages (Chapter 4) and components (Chapter 7). We will see an example of this in Section 8.1.

The addition of abstract signatures is not a trivial extension. The module language becomes *impredicative*, as the following example demonstrates:

```
signature E = (signature S)
structure M = (signature S = E) : E
```

Essentially, we have introduced something close to Type : Type [ML71] into the system, which is well-known to cause problems like non-termination, undecidable type checking, or even logical inconsistency [MR86, Tv88]. In the presence of abstract signatures, type checking of an ML-like module system becomes undecidable [Lil97]. For example, the following snippet will send the type checking algorithm of the current Alice ML compiler into an infinite loop:

```
signature S =
sig
    signature A
    functor F (X : A) : sig end
end

signature T =
sig
    signature A
    functor F (X : S where signature A = A) : sig end
end

signature U = S where signature A = T

(*) Try to check U ≤ T
functor Loop (X : U) = X : T
```

Our experience is that in practice the undecidable type checking of Alice ML is not a problem. Examples like the above are contrived enough to not arise in practice. So far, we have not encountered a single instance of an undecidable example in the wild. It should also be noted that Objective Caml [Ler03] exhibits the same undecidability (for the same reasons), and its compiler exhibits the same non-terminating behaviour when confronted with the respective example. Nevertheless, experience seems to be similarly positive: no complaints from users about the compiler not terminating on such programs have ever been reported. However, this may be due to abstract signatures being extremely rare (probably non-existent) in Objective Caml programs, so it may not be safe to draw strong conclusions from this observation.

3.4. Related Work

Since the ML module system has been proposed by MacQueen [Mac84], there always has been the desire to lift its first-order restriction. However, at the time SML was defined, it was not clear how to cope with type abstraction and sharing in the presence of higher-order functors. Much effort has since gone into generalising functors to higher-order. First theoretical work was done by Harper, Mitchell & Moggi [HMM90]. Tofte & MacQueen formalised the stamp-based mechanism for higher-order functors that is implemented in SML of New Jersey [Tof94, MT94]. A simpler and more satisfactory explanation was the translucent sum calculus by Harper & Lillibridge [HL94, Lil97], which moved modules into a type-theoretic context. The mostly equivalent concept of manifest types simultaneously developed by Leroy [Ler94, Ler95] is the basis for the higher-order module system of Objective Caml.

Russo developed a theory for ML modules that is not based on dependent types [Rus99], and was able to develop a practical design for modules as first-class values [Rus00]. Based on Aspinall's more general idea of singleton kinds for capturing type sharing [Asp97], the unified, type-theoretic framework given by Dreyer, Crary & Harper [DCH03] subsumes most previous work on modules and provides a satisfactory answer to most theoretic questions.

Harper & Lillibridge's work unifies core and module language and hence subsumes abstract signatures. They showed undecidability of type checking in such a system [HL94]. Moscow

	MLton	SML/NJ	Moscow ML	O'Caml	Alice ML
higher-order functors	–	+	+	+	+
applicative functors	–	–	+	+	–
local modules	–	–	+	+	+
local signatures	–	–	+	+	+
abstract signatures	–	–	–	+	+
first-class modules	–	–	+	–	(+)

Figure 3.3.: Higher-order modules in different ML systems

ML [RRS00] also allows local signatures, but does not treat them as structure members, thus avoiding undecidability, for the price of being less expressive: it can neither express the FACTORY signature from Section 3.3, nor polymorphic functors.

The knowledgeable reader will realise that the higher-order module extensions of Alice ML turn its module system into a higher-order functional language that closely mirrors the module language of Objective Caml [Ler03, Ler94], except that functors are not applicative [Ler95]. Even abstract signatures are available in Objective Caml [Ler03], but are not formalised. As already noted, their presence renders type checking for Objective Caml undecidable as well.

Regarding scenarios for the use of higher-order functors, Ramsey describes the design of an extensible interpreter for the embedded language Lua, making intensive use of higher-order modules [Ram05].

Several existing ML implementations provide higher-order extensions to the ML module system. Figure 3.3 presents a quick overview for the major systems.

3.5. Summary

- The Standard ML module system is a first-order functional language layered on top of core ML.
- Its main features are strong type abstraction and large-scale parameterisation.
- Module types are called signatures and provide structural subtyping and a simple form of dependent typing.
- Based on previous work on higher-order modules, Alice ML generalises the module language and removes the strict core/module stratification.
- It also extends the module type system with nested and abstract signatures, which enables a form of module-level polymorphism.

4. Packages

ML style modules are an indispensable aid for large-scale programming, thanks to their ability to express complex modular abstractions. They provide namespacing, encapsulation, genericity, and architectural configuration in form of a small higher-order language with an expressive, strong type system. Higher-order modules even provide a semantic and linguistic foundation for non-trivial *architectural programming*, or *programming-in-the-large*.

However, all this expressive power is purely static: a program has to be a closed module expression. That is, the program must be determined and provided in its entirety at compile time, prior to running it (compile time may include static linking steps). The program's module expressions could actually be evaluated at compile time[1], and some implementations of Standard ML in fact do that [Els99]. Once built, configuration, functionality, and extent of a conventional ML program is fixed – in other words, programs are *closed*. This situation is at odds with the open programming scenario discussed in Section 1.2, where we identified the need to acquire program components dynamically, without them necessarily being available, or even anticipated, in advance. The ML module language provides nothing to address that need. Obviously, some additional amount of dynamism is required, including the ability to defer some type checking to runtime – in other words, we need some form of dynamic typing.

Alice ML employs the novel notion of *package* to resolve this tension. Packages complement the static module and type system of SML with the necessary amount of dynamic typing to gain the desired dynamic flexibility, while not compromising any of the advantages of static typing. More precisely, they add two important capabilities to the language:

- **Dynamic typing.** The static type system is complemented with a flexible, controlled form of dynamic typing.
- **Modules as first-class values.** Modules can be encapsulated as first-class values, enabling computations over modules.

Although this represents quite a dramatic change to the expressivity of the language, packages come with the following properties:

- **Type safety.** Soundness and abstraction properties of the static type system are not compromised.
- **Economic design.** Minimal conceptual overhead through maximal conceptual re-use.
- **Idiomatic use.** Well-understood idioms for module programming apply unchanged.

The package concept is a variation of the well-known idea of *dynamics* [ACPP91, LM93, ACPR95, Dug99]. The main difference is that they carry modules instead of plain values. As a consequence, they can reuse the module system's flexible subtyping relation for dynamic type checking, instead of necessitating a complicated `typecase` construct for matching types.

The presence of packages has severe implications on other aspects of the language. A closely related but more technical contribution of our work on Alice ML hence is a refined semantics

[1] Local modules as available in Alice ML (Section 3.2) generally cannot be evaluated statically, though.

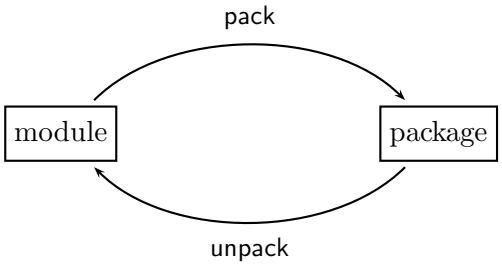


Figure 4.1.: Forming packages

for type abstraction through sealing. It is indispensable to make sealing coexist properly with dynamic typing: abstraction safety should not be compromised. The issue has been largely ignored in the various proposals for dynamics and related mechanisms in the literature. We will discuss the problem in Section 4.6. A formalisation of the refined semantics is discussed in Part II of this thesis. A formal semantics of packages and abstraction-safe sealing in the framework of the Definition of Standard ML [MTHM97] can be found in a technical report [Ros05].

We note at this point however that due to other features, Alice ML cannot fully guarantee abstraction safety either. More precisely, if and only if a value of abstract type is imported from outside the process, then the type system can ensure its type safety but not integrity with respect to the abstraction – the value may have been forged by extra-linguistic means. Such forging is impossible within the language, however. Section 5.4 will discuss this problem.

4.1. Basics

Packages are the exclusive means for integrating dynamic typing into the statically typed universe of Alice ML. A package is a first-class value of the primitive type package. Intuitively, it contains a module, along with a dynamic description of the module's signature, which we call the *package signature*. Package signatures exist only in the dynamic semantics, they are not tracked by the static type system. That property sets packages apart from other proposals for first-class modules [Rus00, DCH03], where the signature is always fixed statically.

There are only two basic operations on packages, depicted in Figure 4.2. A package is created by injecting a module, expressed by a structure expression *strexp* (which may denote a functor, Section 3.1) into the type package:

pack *strexp* : *sigexp*

This expression creates a package from the module expressed by *strexp*. The signature expression *sigexp* defines the package signature. Of course, the module expression must statically match this signature.

The inverse operation is projection, eliminating a package. The module expression

unpack *exp* : *sigexp*

takes a package computed by *exp* (which needs to have type package) and extracts the contained module – provided that the package signature matches the *target signature* denoted by *sigexp*. That is, unpacking performs a dynamic type check. If the dynamic check fails, the pre-defined exception Unpack is raised.[2] Statically, the whole expression has the signature *sigexp*.

For example, we can wrap the library structure Array into a package,

[2]In the current version of the Alice System that exception is named Mismatch.

exp	:=	...
		pack *strexp* : *sigexp* packing
strexp	:=	...
		unpack *exp* : *sigexp* unpacking

Figure 4.2.: Syntax of packages

---

val p = **pack** Array : ARRAY

and unpack it successfully using the same signature:

structure Array' = **unpack** p : ARRAY

Any attempt to unpack p with an incompatible signature will fail (but any supersignature will be admissible, see Section 4.3). On the other hand, all subsequent accesses to Array' or members of it are statically type-safe, no further checks are required.

4.2. Persistence

Before we discuss the semantics of packages and dynamic typing in more detail, let us first take a short detour to describe one of their main applications. Doing so allows us to continue the presentation along more interesting and intuitive examples.

Our main motivation for dynamic typing is type-safe import and export. For example, with packages, we can provide a type-safe interface to high-level *persistence*, i.e. I/O of language-level data structures to an external medium [ACPP89, OK93]. In Alice ML, this interface consists of two functions in the library structure Pickle:

val save : string × package → unit
val load : string → package

The save operation writes a package to a file of a given name. The inverse operation load retrieves a package from a file. The file will contain a so-called *pickle* (Chapter 5), a self-contained, platform-independent representation of the saved package. All pickle files contain a single value of the type package – reducing the problem of dynamically checking the type of imported values to the type checking performed by unpack. Since packages contain modules, and modules can embed arbitrary language entities (values, types, higher-order modules, even signatures), allowing only packages to be saved is not a restriction.

For example, instead of just wrapping the library module Array into a package as before, we can write it to disk, using the following idiomatic code:

save ("array.alc", **pack** Array : ARRAY)

As we will explicate in Chapter 5, the pickle contains the whole module, including its code! It can be loaded – by the same process or a different one – by composing the inverse operations, in reverse order:

structure Array' = **unpack** load "array.alc" : ARRAY

The obtained structure Array' can now be used as a substitute for Array – it is an identical copy (however, see Section 4.4 for the issue of type sharing). Again, all uses of Array' are statically type-safe. The only possible point of type failure is the unpack operation.

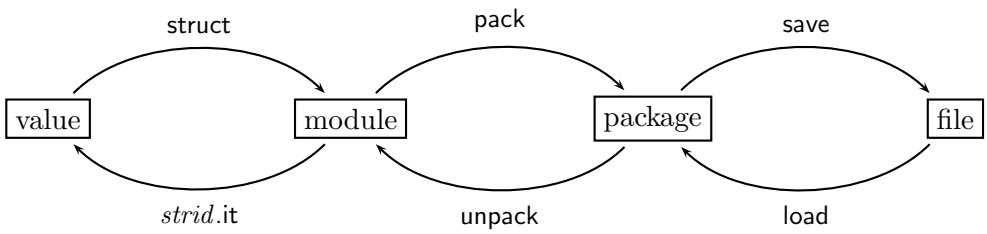


Figure 4.3.: Pickling values

The syntactic sugar for structures that was defined in Figure 3.2 is convenient for saving single values. As a convention borrowed from SML's top-level, we use the identifier it to name the single value in the auxiliary structure:

```
save ("five.alc", pack (val it = 5) : (val it : int))
```

Note how the inner pairs of parentheses abbreviate heavier occurrences of **struct** ... **end** and **sig** ... **end** keywords. To load it, we have to name the auxiliary structure, though:

```
val five = let structure Five = unpack load "five.alc" : (val it : int) in Five.it end
```

Figure 4.3 shows a diagram of the steps involved in making a value persistent and retrieving it, as realised by the above program snippets.

We delay presentation of the gory details of pickling and its semantics until Chapter 5. For now, the naive explanations given so far are sufficient preparation for more interesting examples of packages in the remainder of the current chapter. Moreover, as it will turn out in Section 7.2, save and load are not primitives, but are definable in terms of two more general functions that support dynamic import and export of entire components.

4.3. Dynamic Type Matching

Packages are a fully conservative extension to SML. They complement the static type system with dynamic typing. They do so *without* breaking the type system: all typing rules are still sound, because unpack completely *isolates* dynamic typing. The type of an unpack expression is statically determined by the explicit annotation. All subsequent uses of the resulting module are statically safe, because unpacking will only succeed if the package signature meets the static requirements. No further runtime type checks are necessary, nor can execution fail later due to inconsistent type assumptions. This is in contrast to languages with dynamic typing in the more conventional sense, where potentially every operation can fail due to type errors and hence requires checking.

Dynamic type checking for packages is performed on signatures. Signatures describe interfaces, and support a rich notion of subtyping, often called *matching* in ML nomenclature. The Definition of Standard ML [MTHM97] formalises it quite intuitively as a relation between environments that consists of two dimensions:

- **Enrichment.** The more specific signature may contain more fields than the less specific one.
- **Instantiation.** Abstract types in the less specific signature can be realised by concrete types in the more specific signature.

Subtyping hence allows a great amount of flexibility with respect to composing modules. In particular, it is robust against extension or specialisation of a module interface. That is essential for adequately describing program architectures in a modular manner.

Obviously, robustness against future extensions is even more desirable in – potentially ever-changing – dynamic applications. Packages provide it to a wide extent, by having the dynamic type check verify the package signature up to the subtype relation. When a client process retrieves a module from some external location, only minimum assumptions about its signature need to be made. The provider of the module is free to replace it with a richer version, as long as the new interface just extends or refines the previous one. All client code will continue to work. In fact, even incompatible changes of the interface will work with clients that did not make assumptions about that part of the module because they did not access it.

For example, consider a provider offering a package implementing efficient functional dictionaries over strings:

```
type α dict
val empty : α dict
val insert : α dict × string × α → α dict
val lookup : α dict × string → α option
val filter : (α → bool) × α dict → α dict
```

A client retrieves and unpacks the module as follows:

```
structure Rat = unpack load "Dict" :
    sig
        type α dict
        val empty : α dict
        val insert : α dict × string × α → α dict
        val lookup : α dict × string → α option
    end
```

Note that the signature does not mention the filter function, since the client does not use it. Consequently, the client will not break if the module one day is changed and extended to support the following signature:

```
type α dict
val empty : α dict
val insert : α dict × string × α → α dict
val lookup : α dict × string → α option
val filter : (string × α → bool) × α dict → α dict
val adjoin : α dict × α dict → α dict
```

Neither the added function adjoin nor the modified type of the filter function will affect the client, the signature still matches the one it assumed.

4.4. Dynamic Type Sharing

In Section 4.2 we claimed that the module Array' obtained by pickling and unpickling the original library module Array is an identical copy. Although that is true, there is a caveat: the ARRAY signature contains the abstract type array. The way we unpacked it, the type Array'.array will be statically incompatible with the original type Array.array. Since there generally is no way to determine statically what type identities are found in a package, all abstract types in the target signature must indeed be considered fully abstract – and hence different from any other – by the (static) type system. The copy can hence only substitute the original as far as type compatibility is not required.

However, type compatibility can be obtained easily – we just need to enforce it in the usual ML way, namely by putting *type sharing* constraints on the target signature:

```
structure Array' = unpack Pickle.load "array.alc" : ARRAY where type array = Array.array
```

With this formulation, the type Array'.array is statically known to be equal to Array.array. Of course, unpacking will only succeed if the package actually meets this requirement at runtime.

The constraint effectively expresses *dynamic type sharing*. By restricting the target signature we ensure static compatibility, for the price of precluding successful use of non-standard implementations of arrays. Much like for programming with functors, it depends on the application how much sharing is required. Section 4.7 will demonstrate more intricate uses of dynamic type sharing.

4.4.1. Package Signature Refinement

There is a subtlety involved in the previous example: dynamic type sharing works as demonstrated only because the package signature supplied with a pack expression is interpreted transparently. That is, writing

```
val p = pack Array : ARRAY
```

actually is equivalent to

```
val p = pack Array : ARRAY where type α array = α Array.array
```

This behaviour mirrors the semantics of SML's transparent ascription operator (:), but dynamically: the actual package signature is obtained by refining the ascribed signature with the concrete types found in the respective module. Technically, dynamic *selfification* [HL94], or *strengthening* [Ler94], is performed.

As we pointed out in [Ros06], it is worth noting that *without* the transparent interpretation, the fragment

```
val p = pack Array : ARRAY
structure Array' = unpack p
                 : ARRAY where type α array = α Array.array
```

would fail with an Unpack exception (as one would expect), but the contorted, yet seemingly equivalent example

```
val p = pack Array : ARRAY
structure Aux = unpack p : ARRAY
val p' = pack Aux : ARRAY where type α array = α Aux.array
structure Array' = unpack p'
                 : ARRAY where type α array = α Array.array
```

would still succeed. Obviously, such pathological behaviour is neither desirable nor useful, thus we chose the transparent interpretation.

In summary, the transparent interpretation maintains consistency between a package signature and the contained module. That allows a package to be unpacked with the most specific type of the contained module and saves the programmer from cluttering pack expressions with redundant where constraints to achieve type propagation.

Note that, due to the presence of abstract signatures (Section 3.3.2), the necessary refinement of the package signature cannot be determined statically in all cases. For example, consider a functor that corresponds to an η-expansion of the pack operator:

```
functor Pack (signature S) (X : S) = (val it = pack X : S)
```

Applying that functor to Array,

```
val p = let structure P = Pack (signature S = ARRAY) Array in P.it end
```

should have the same effect as the previous definition of p. Obviously, this requires the signature S to be dynamically refined inside the functor. Thus, package signatures are generally constructed at runtime.

4.5. Parametricity

The integration of dynamic typing has a severe impact on the semantics of the language: by utilising dynamic type sharing it is possible to dynamically test for type equivalences and have a program behave differently depending on the outcome of such a test. Consequently, evaluation is no longer *parametric* [Rey83, BFSS89, ACC93]. Intuitively, a polymorphically typed expression is parametric if its evaluation is independent of the concrete type instantiation. A language is said to be parametric if all polymorphism is parametric.

Parametricity is a valuable property for polymorphic languages, offering important advantages:

- *Type erasure.* Programs can be compiled and executed without maintaining costly type information at runtime.
- *Theorems* [Wad89]. Polymorphic types state strong invariants about terms, which allow deriving a variety of useful laws.
- *Abstraction* [Rey83, MP88]. It is possible to achieve encapsulation solely by abstracting or quantifying over types.

Looking closer, it is obvious that evaluation of *modules* cannot be parametric in the presence of packages – the behaviour of unpack must depend on dynamic type information. However, for Alice ML the semantics of dynamic types has been crafted such that the *core* language, where polymorphism is ubiquitous, is not affected. In particular, unlike functors, polymorphic functions are still fully parametric: the usual laws still hold, and they can be compiled using standard type erasure techniques.

This nicely fits the syntactic setup of ML: on the module level, passing types is always made explicit in the syntax. Core polymorphism, on the other hand, is completely implicit. Thus the syntax provides a clear model to the programmer: only types explicitly supplied in a program, by means of named type declarations, can potentially affect its operational behaviour, and induce a cost.

In order to maintain parametricity for core polymorphism we need strict separation between implicit and explicit types. More precisely, it is required that no operation consuming dynamic types – i.e. unpack and sealing (Section 4.1) – may ever depend on the instantiation of a polymorphic type variable. Fortunately, this comes for free, thanks to ML's syntactic treatment of type variables: both these operations require a signature to be written explicitly. Signature expressions can only refer to other type and signature declarations (possibly nested in structures), and there are only two places in type or signature declarations where type variables can occur:

1. in *type declarations* and *specifications*, where type variables on the right side always have to refer to variables bound on the left side,[3]

[3] Actually, that very restriction is missing from the 1997 revision of the SML Definition [MTHM97], but its absence has been confirmed as a mistake on part of the Definition, because it leads to unsoundness [Ros01].

2. in *value specifications*, where all type variables occurring in the type are interpreted as universally quantified (except for exception specifications, where they are explicitly disallowed).

Together, the closedness restriction on type declarations and the implicit local quantification in value specifications ensure that no dynamic type can depend on a polymorphic type variable, even with local modules (Section 3.2). That is a sufficient condition to maintain parametricity in the extended language Alice ML. For instance,

```
fun α mypack (x : α) = pack (val it = x) : (val it : α)      (*) illegal!
```

will not type check, because the local α in the signature is considered locally quantified (and hence different from the one bound at the surrounding declaration), and x does not have the universal type $\forall\alpha.\alpha$. Neither is it possible to type the package by using an auxiliary type declaration,

```
fun α mypack (x : α) =
    let
        type t = α      (*) illegal!
    in
        pack (val it = x) : (val it : t)
    end
```

because the local type declaration for t, containing a free occurrence of α, is not allowed by the SML Definition. There is no way to make the package signature refer directly or indirectly to the polymorphic type variable α, therefore the package signature cannot depend on it.

4.5.1. Working Around Parametricity

Maintaining parametricity in the core language is not without drawback – there is a tension between desirable properties and expressiveness. The fact that ordinary core-level evaluation cannot directly depend on types may appear to be a severe restriction in some dynamic scenarios. The expressive power of dynamic typing remains relatively limited, maybe too limited.

However, that objection can be diluted by the existence of work-arounds that allow emulating most of the missing expressiveness:

- Local modules and higher-order functors (Chapter 3) often enable lifting polymorphic function definitions to the module level, by turning them into functors. For example, the function mypack from above can be reformulated straightforwardly as a functor:

  ```
  functor MyPack (type t; val x : t) = (val p = pack (val it = x) : (val it : t))
  ```

  Thanks to local modules, a functor formulation is possible even for local functions. However, this technique potentially requires turning all polymorphic functions up the call chain into functors, which might quickly become unwieldy. Moreover, the module language is not Turing-complete, so not all desired functions may be expressible (still, it contains System F^ω).

- A more general work-around is to abuse packages as means for communicating types and modules as first-class values. By wrapping types into packages, they can be passed to core functions, which may unpack them locally in order to perform consecutive dynamic type operations. Of course, that approach could be deemed somewhat questionable, because it essentially means evading the static type system and relying on dynamic typing more than necessary.

- If this should prove to be insufficient, adding conventional first-class modules [Rus00] to the language would be a general solution.

So far, we have only encountered few interesting examples – in the context of what dynamic typing in Alice ML is intended for – which needed to employ any work-around, and the former two were adequate enough in those cases.

If the availability of non-parametric core functions should be deemed necessary, the advantages of parametricity could still be generally maintained. Instead of abolishing parametricity altogether, two kinds of type variables could be distinguished, namely conventional ones, which stay parametric and can be erased, and dynamic ones, that can be used to represent dynamic types. Parametric variables could be instantiated with types containing dynamic ones, but not vice versa. Something similar has been proposed by Dubois, Rouaix & Weis to support generic functions [DRW95]. It is relatively straightforward to extend ML type inference accordingly.

4.6. Abstract Types

An important purpose of a type and module system is erecting and statically verifying abstraction boundaries [Mor73b]. Type abstraction is the respective tool given to the programmer by the type system. It also is the central feature of the ML module system, where it is supported through *sealing* (Chapter 3). The ML type system guarantees *abstraction safety*: values of abstract type can only be constructed and deconstructed by the implementation of the abstraction itself. There is no means within the language that allows client code to break an abstraction.

As we saw in the previous section, Alice ML lacks full parametricity. In that situation, type abstraction cannot be guaranteed by means of static scoping of type variables, as in the standard models of abstract types that are based on existential quantification [MP88, Ros03a]. The addition of dynamic typing thus raises an important question: Should dynamic typing be required to respect abstraction? Or can we allow to explicitly overcome abstraction barriers by means of dynamic typing?

Alice ML takes a clear stance: type abstraction is a central feature of the ML type system, and no code should be able to sneak across an abstraction barrier. Abstraction safety shall be maintained at all times. For example, consider an abstract type for generating time stamps:

```
signature STAMP = (eqtype stamp; val stamp : unit → stamp)
structure Stamp :> STAMP =
struct
    type t = int
    val state = ref 0
    fun stamp () = (state := !state+1; !state)
end
```

The abstraction guarantees that every call to stamp actually delivers a fresh, distinct stamp. Hence it is crucial that the following code cannot be executed successfully:

```
val p = pack (val x = 13) : (val x : int)
structure Fake = unpack p : (val x : Stamp.t)
```

If the unpack operation succeeded, we would have forged a bogus stamp value – the stamp procedure would no longer be guaranteed to deliver distinct values. Hence we need unpack to fail in this example. Apparently, this means that Stamp.t must be a type different from int – dynamically!

Alice ML achieves the desired semantics by employing *dynamic generativity*. That is, every abstract type is represented by a dynamic type name, and this name is generated at runtime, when the respective declaration is evaluated. In particular, *sealing*, i.e. evaluating a structure expression of the form

strexp :> *sigexp*

generates a new name for every type specified abstractly in the signature *sigexp*. This happens each time the expression is evaluated, hence the functor

```
functor MkStamp () :> STAMP =
struct
    type t = int
    val state = ref 0
    fun stamp () = (state := !state+1; !state)
end
```

generates multiple distinct stamp types when applied multiple times – exactly as suggested by the static semantics of SML.

Dynamic type names are globally unique. In general, this is necessary to avoid forging of abstract values by other processes. Pickling (Section 4.2) allows arbitrary values to be made persistent or be exchanged between different processes, and this includes values of abstract type. In order to maintain abstraction safety, the type names generated in one process must hence be different from any type name generated in any other process, at any time.

4.6.1. Internal and External View of Abstraction

Though the dynamic semantics of sealing and type generativity are relatively straightforward, they can have subtle effects. In particular, sealing is performed *after* evaluating the sealed module itself. Consequently, the module's internals know nothing about the type names generated. As statically, a dynamic type abstracted via sealing is fully transparent inside the abstraction. Dynamic types crossing abstraction boundaries may hence not satisfy equivalences one might naively expect. For example, after evaluating[4]

```
structure M :> (type t; val f : t → string) =
struct
    type t = int
    fun f x = Int.toString x
    do Pickle.save ("m.alc", pack (val it = 37) : (val it : t))
end
```

the type M.t, being opaque, is different from the type occurring in the signature of the package written to the file, which is just int. Hence

```
structure It = unpack Pickle.load "m.alc" : (val it : M.t)
```

will fail. On the other hand,

```
structure It = unpack Pickle.load "m.alc" : (val it : int)
```

actually succeeds! Probably not what is desired. The problem is that there are two incompatible views of an abstract type, the *internal* and the *external* one. The type system ensures that the views are properly switched whenever a value of the abstract type crosses the abstraction boundary. If we, like in the example above, sidestep the static type system by passing such a value dynamically typed, then the type system cannot know about that. Abstractions need to be implemented such that this does not happen.

Fortunately, occasions where the abstraction has to construct a dynamic value of abstract type internally – like above – are rare. The desired effect can always be achieved by a two-staged construction:

[4] Alice ML allows to abbreviate the omnipresent "**val** _ = exp" with the declaration form "**do** exp".

```
structure M : (type t; val f : t → string) =
struct
    structure Abs :> (type t; val it : t; val f : t → string) =
    struct
        type t = int
        fun f x = Int.toString x
    end
    open Abs
    do Pickle.save ("m.alc", pack (val it = Abs.it) : (val it : Abs.t))
end
```

In general, the main implementation of an abstraction would go into a sealed, inner auxiliary structure like Abs. Only operations requiring access to the abstract type name are defined outside. Note that the outer signature ascription is transparent, in order not to generate a second level of abstraction that would defeat the whole purpose of the construction.

4.7. Typeful Dynamic Programming

No abstraction can be breached by means of dynamic typing. But is that behaviour useful? Can we really work with rock-solid abstract types in the context of open programming? For example, when a process pickles a value of abstract type, and the type was created by the process, how can the value ever be unpacked after that process has terminated? Due to the generative semantics of sealing, even a subsequent process running the same program will not be able to unpack it.

The answer is quite simple: one has to export and share the abstraction as well. Recall that all import/export is based on packages, and packages contain modules. Hence it is perfectly valid to pickle the module implementing an abstraction. In fact, dynamic type sharing can be employed for *typeful programming* [Car91] with dynamic types, when packages themselves contain the implementation of abstract types.

Consider a strategy game. It allows initiating a new campaign, and during a campaign arbitrary snapshots (saved games) can be stored that allow reverting to that point of the campaign later on. When the program is exited, the current campaign is pickled and can be continued next time the game is started. At any point, a stored snapshot can be loaded.

Different campaigns may use different configurations (e.g. map sizes). A snapshot only is valid in conjunction with the campaign that it belongs to. To prevent mixing up snapshots of incompatible campaigns, a snapshot can be modelled as an abstract type that is created along with a particular campaign. More precisely, a campaign is a structure with the signature

```
signature CAMPAIGN =
sig
    type world
    val getWorld : unit → world
    val setWorld : world → unit
    ...
end
```

where the abstract type world encapsulates the state of the campaign. A new campaign might be created by means of a functor:

```
functor MkCampaign (Config : CONFIG) :> CAMPAIGN = ...
structure Campaign = MkCampaign MyConfig
```

A fresh world type is generated along with an initial world state. A snapshot of a campaign can now be created by retrieving its current world and pickling it to a file:

```
Pickle.save ("snapshot", pack (val world = Campaign.getWorld ()) : (val world : Campaign.world))
```

As long as the application has not terminated, it is easy to reload previous snapshots of the running campaign:

```
structure W = unpack Pickle.load "snapshot" : (val world : Campaign.world)
do Campaign.setWorld W.world
```

When the application exits, it pickles the campaign itself:

```
Pickle.save ("campaign", pack Campaign : CAMPAIGN)
```

When the game is run the next time the user can choose to continue an existing campaign, at a given snapshot. This is implemented by unpickling the previous campaign instead of creating a new one, along with the selected snapshot:

```
structure Campaign = unpack load "campaign" : CAMPAIGN
structure W = unpack Pickle.load "snapshot" : (val world : Campaign.world)
do Campaign.setWorld W.world
```

Unpacking the snapshot will only succeed if it actually belongs to the given campaign. What we see in the example is an interesting instance of dynamic type sharing (Section 4.4): the signature used to unpack one module refers to a type dynamically obtained from another package. The generative semantics of sealing (Section 4.6) makes it possible to detect type sharing dynamically, while still not being able to discover the underlying implementation types.

In simple examples like the above, type sharing is straightforward to express, because the ascribed signature can easily be given inline. In order to deal with more complex examples, SML's module system already provides features for expressing more involved type sharing: type constraints on signatures, i.e. type specialisation using `where` as well as `sharing` specifications, both useful in the static type system, will come in handy for dynamic typing as well.

Assume that a campaign snapshot should not only contain the state of the game world, but also some meta information like when the snapshot was created, on what system, etc. In that case, it is more convenient to define a signature:

```
signature SNAPSHOT =
sig
    type world
    val world : world
    val date : date
    val system : string
    ...
end
```

The `world` type has to be abstract in the signature, because it will differ between uses. In order to concretise it when we ascribe the respective `unpack` operation, we simply put a respective constraint on the signature:

```
structure Campaign = unpack Pickle.load "campaign" : CAMPAIGN
structure World =
    unpack Pickle.load "snapshot" : SNAPSHOT where type world = Campaign.world
```

In general, the same language mechanisms are applicable for expressing dynamic sharing with packages as for expressing static sharing between functor parameters. No additional structure in the type language is necessary. The introduction of dynamic typing stays economic from a semantic point of view and, to a large extent, enables the programmer to reuse her knowledge about expressing type relations with modules.

4.8. Related Work

In previous work we have presented the basic design of packages in Alice ML [RLT+06], and we have formalised packages and pickling as part of a calculus of higher-order modules [Ros06] that we proved sound. The latter work gives a relatively direct type-theoretic account for the semantics of packages. We also have integrated a formalisation of packages directly into the formal language specification of Standard ML [Ros05].

The concept of dynamics has long been folklore. Dynamics were already proposed for the very purpose of open programming in an unpublished article by Mycroft [Myc83], and later made precise by Abadi, Cardelli, Pierce & Plotkin [ACPP91, ACPR95]. Their proposal involved a complex typecase construct for projection, in order to dispatch on the type of the dynamic. Our unpack operator is less expressive, although type dispatch can be simulated to a certain extend by a sequence of unpack operations with different signatures. On the other hand, unpack supports subtyping, hence making the use of packages more flexible and robust against changes. In our experience, this dimension actually is much more important for the applications dynamics are intended for.

Some typed languages offer slightly different variations of dynamic typing. Alanko [Ala04] gives an overview of the use of dynamic typing and reflection in typed languages.

The syntax for package injection and projection has been borrowed from Russo's work on first-class modules [Rus00, Rus98], and indeed there is a close relation. The fundamental difference between packages and first-class modules is that the latter are statically typed (i.e. the type of a module value describes its full signature), while packages are dynamically typed (the type package is abstract).

A package can be understood as a first-class module wrapped into a conventional dynamic. However, coupling both mechanisms enables unpack to exploit subtype polymorphism, which is not possible otherwise, due to the lack of subtyping in the ML core language. For example, assume two signatures $S \leq S'$. Given a package with signature S we can unpack it under signature S', without actually knowing S. With dynamics and first-class modules however, given a dynamic carrying a first-class module of signature S, i.e. a value of type $\langle S \rangle$, we first would have to unwrap the dynamic under type $\langle S \rangle$ – since there is no subtyping in the core language, we would have to know S exactly at this point. We could only go to S' *after* having projected the module. Clearly, this would severely weaken modularity and robustness.

4.9. Summary

- We introduce packages as a novel variant of dynamics; they carry a module along with its dynamic signature.
- Accessing a package requires a dynamic type-check against a static target signature.
- The type-check employs structural subtyping to make clients robust against interface changes.
- Based on packages, Alice ML provides type-safe persistence.
- Dynamic type sharing can be expressed with idioms known from ML modules, and enables typeful dynamic programming.
- Unlike with previous work on dynamics, dynamic typing is confined to the module language, the core language is still parametric.
- Abstraction safety is maintained within the language – type abstraction dynamically generates fresh, globally unique type names.

5. Pickling

The most important characteristic of an open program is the need to communicate with the outside world. More precisely, an open program has to exchange information with its environment, other programs, or other instances of the same program. In a high-level language, we want to be able to represent such information as language-level values wherever possible. That is, the language should enable us to import and export potentially arbitrary language-level *data structures*. In a higher-order language with first-class functions, that naturally includes functions, and hence code – opening up a whole range of applications for exchanging *functionality* and *behaviour*. Furthermore, in a language with strong emphasis on types and modules for structuring data and functionality, we want to be able to exchange entire program fragments in the form of modules.

The task of exporting a value from a process is often known as *serialisation* or *marshalling*. For reasons discussed in the next section, we prefer the somewhat less common term *pickling* [BJW87].

Unlike the features discussed in the previous chapters, pickling is not a language construct per se, but rather a generic mechanism underlying a number of other language constructs. Nevertheless, its central role and recurring semantic implications justify treatment in a separate chapter.

Pickling or related mechanisms have been employed in many previous languages to support persistence and inter-process communication, e.g. CLU [HL82], Modula-3 [BNOW95], or Java [RWWB96]. The Mozart System for Oz was the first to systematically and uniformly base its compilation and distribution model on pickles, in particular by supporting pickling of code [DKSS98]. Alice ML adopts the Oz approach and puts it into a typed context.

5.1. Pickles

In order to export a value from a process, it must be transformed into an external representation. To support open programming in a most general and safe way, we require several properties of such a representation:

- **Transparency.** The composition of pickling and unpickling a value should yield a copy that is observationally equivalent to the original value.
- **Universality.** The representation should support all possible types of values found in the language. In particular, it should readily support higher-order and user-defined types, as well as cyclic structures. The only exceptions are primitive types whose meaning is bound to the current process, so-called *resources* (e.g. file handles, thread names, etc.).
- **Closedness.** An external representation should always be self-contained. That is, it should contain the transitive closure of all values reachable by the one meant to be exported. For functions, it has to include their code.
- **Portability.** The representation should be independent from and portable between different computer architectures, operating systems, and implementations of the language.

- **Verifiability.** It should be possible to test the integrity of an external representation, i.e. the format must be self-describing enough to support checking that a given instance really represents a well-formed value of the language.
- **Security.** An external value should not be allowed to contain references to critical resources that might enable contained functions to silently perform unapproved, security-relevant actions when applied on an importing site.
- **Efficiency.** The space required for the representation of a value should be at most linear in the size it takes in memory. That particularly implies that *sharing* between values in a closure must be maintained.

We call an external representation meeting these requirements a *pickle*. This terminology emphasises the fact that it is really a self-contained representation, which is not the case for most existing serialisation or marshalling mechanisms, especially with respect to code.

Obviously, the requirement for universality demands pickling to be made available as a generic mechanism. A library or user-defined infrastructure, like often found in other languages, is not enough, because it could not deal with functions, abstract types, or modules, etc. In Alice ML, pickling, and its inverse, unpickling, thus need to be built-in services of the runtime system. Pickling takes a value and produces a copy that can be transferred to other processes. On a more technical level, pickling transforms the internal graph-like representation of an object in memory into a linear, platform-independent external representation [TKS06, Tac03]. From a pickle an equivalent copy of the original object can be reconstructed. A pickle includes the transitive closure of the respective object, i.e. the reachable subgraph. This graph can be cyclic.

For a rich language like ML, designing and specifying a pickle format that meets all of the above requirements is a complex endeavour. In particular, it implies giving a full specification of the external code format of the language. Furthermore, it involves specifying a low-level type system to support verification of code and data. This in turn has potential impact on the implementation of a runtime system for the language, because it must be able to produce this type information upon pickling.

Clearly, such questions are beyond the scope of this thesis. Here, we are only concerned with the design and semantics of the mere language, not its implementation. We hence stay in orbit of these issues and take the desired properties for granted, as far as they are transparent in the semantics.[1] We refer the interested reader to the works of Kornstaedt and Tack [TKS06, Kor06, Tac03] for an in-depth discussion of the design and implementation of a pickling mechanism.

Another aspect we generally remain naive about in this thesis is security. We consider security issues only as far as not making the language semantics violate obvious security concerns in irreparable manners, and by providing some hooks to implement more elaborate strategies.

5.2. Type checking and verification

In Alice ML, almost arbitrary language-level data structures can be pickled. How do we achieve type-safety if we allow values of arbitrary type to be retrieved from files at runtime? How can a load operation be typed, when it is impossible for the compiler to know what files will be accessed dynamically, and what types of values they contain?

Obviously, loading pickles generally requires dynamic checks to establish type-safety. We can distinguish two kinds of check:

[1] The current version of the Alice System does not yet meet all of the requirements. In particular, the pickle format does only support a rudimentary amount of verification, and no attempt has been made to actually make the system a secure platform.

- **Verification**. Checks *internal* consistence of a pickle, i.e. whether it represents a well-formed value (of a fixed type).
- **Dynamic Typing**. Checks *external* consistence, i.e. compares the type of the pickle with the one expected by the consuming code.

Both these checks are orthogonal: if the producer of a pickle is trusted, verification may be omitted; if the type of the value represented by the pickle is known statically, dynamic typing is not needed. In the latter case, the pickle need not include a description of its own type, because it can be determined from context. All combinations are possible – consider the following scenarios:

- Persistence: the content type of a file cannot be known statically, hence dynamic typing is required; however, if the environment is trusted, we may omit verification of the file.
- Typed communication channel: the channel type determines the type of values received over the channel, hence it need not be transmitted nor checked for each value (it is checked once when the connection is established). However, if the sender is untrusted, the individual values must still be verified to be well-formed with respect to that type.

Moreover, although both checks can be understood as a form of dynamic type checking, their nature is quite different: verification has to look at the structure of a pickle, while dynamic typing can be realised simply by comparing a type description that is part of the pickle (and might itself require verification). Both checks operate on different levels of abstraction. For example, verification must be able to see the representation of abstract types, while dynamic typing should not (Section 4.6). Verification might involve more (and lower-level) type information than visible in the type system of the language, because it has to encompass internal invariants of the low-level representation, especially of code.[2]

Thanks to packages, we can separate concerns: we uncouple dynamic typing from (un)pickling instead of building it into the pickling mechanism. We rely on packages as an independent mechanism, while unpickling itself only performs verification (if necessary).

Section 4.2 briefly introduced the primitives for persistence provided in Alice ML:

```
val save : string × package → unit
val load : string → package
```

All pickle files thus created contain values of one and the same type, package, thus supporting dynamic typing. The idiomatic usage of the primitives then is as follows:

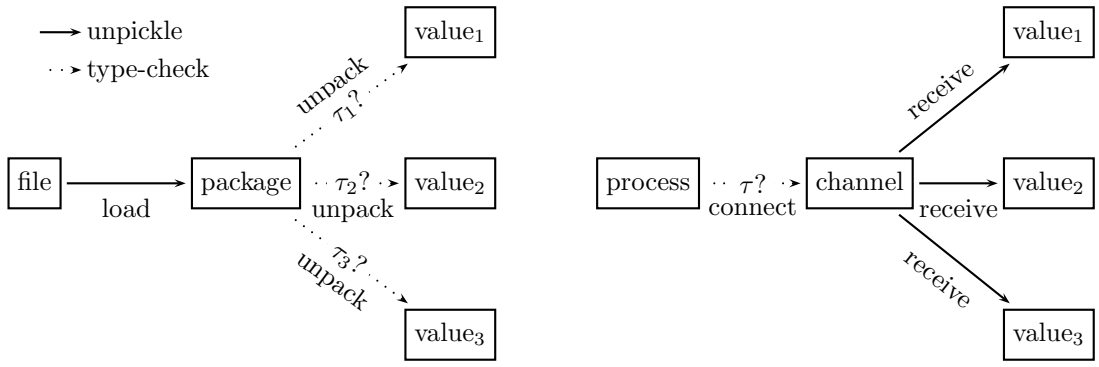
```
do save ("file.alc", pack Module : SIG)
structure Module' = unpack load "file.alc" : SIG'
```

The code clearly separates the two kinds of checks discussed above:

- The load operation has to perform verification, i.e. checks that the file contains a well-formed pickle, representing a package. Failure at this point is considered I/O failure.[3]
- The unpack operator has to check that the package signature matches the static assumptions, i.e. the target signature. Failure at this point is a dynamic type error.

[2] Alternatively, verification can also be substituted by *authentication*. In that approach, pickles are signed using cryptographic methods, and only pickles stemming from trusted authorities are accepted as well-formed. Obviously, this is less flexible, but might be sufficient for certain scenarios. We will not explore that possibility here.

[3] As mentioned, this check is not properly implemented in the current version of Alice.



(a) obtain once/check many

(b) check once/obtain many

Figure 5.1.: Scenarios for type-checking pickles

The design choice of uncoupling dynamic typing from pickling has several advantages:

1. Dynamic typing can be employed independent from pickling.

2. A pickle can easily be checked against multiple different types.

3. Either check can be omitted (independently) under circumstances where it is redundant.

The component system we are going to present in Chapter 7 will explore the first two points to realise link-time type checking. Components will be defined in terms of packages, but ad (1), they are first-class, so they are not necessarily obtained only from pickles; ad (2), a single component may be imported by several others, and hence has to be checked against multiple, potentially different signatures.

The third point is explored in the context of distributed programming, to enable efficient inter-process communication as described earlier. In Alice ML, *proxies* (Section 8.1) are an example of a typed communication mechanism, where we want to avoid transmitting repetitive type descriptions with every single value – a dynamic type check is only necessary for *establishing* a connection (when receiving a proxy), not for every *transmission*.

Two typical scenarios are shown in Figure 5.1. On one extreme there is the obtain-once/check-many scenario of persistence, where a pickle is loaded and then can be used under different types (and hence type-checked) multiple times. On the other end we have a check-once/obtain-many scenario of inter-process communication, where a type check is performed a priori, and then multiple pickles of this type are received. Thanks to the availability of packages as a separate mechanism, both scenarios can be supported efficiently. If packages and pickling were coupled into a single mechanism, both scenarios would require redundant work: in (a) we would have to unpickle the file multiple times, in (b) we would have to check the channel type multiple times.

Note that in this model pickles as such are not self-describing, in the sense that they have to be interpreted under a type known from context. However, the context is always unambiguous. Most operations will interpret a pickle as a package, in which case the pickle in fact contains a complete type description. But some primitives employing pickling can make other assumptions, as long as their use of pickling is encapsulated.

5.3. Resources and Security

Not all values that can be created by a process have a universal meaning *outside* this process. For example, a file handle will typically be local to a process. What should the interpretation of such a value be outside a process?

The ability to pickle and transmit higher-order values raises delicate security issues. If a process receives a function, how can it be sure that it is safe to apply it?

We collectively call values and operations with local meaning or security-relevant semantics *resources*. Alice ML takes the simplest possible path to deal with resources and the problems sketched above: resources are disallowed in pickles. We say that resources are *sited*, and any attempt to pickle a resource will be dynamically detected and yield the exception Sited.

Under this regime, pickles are always meaningful and secure[4] – code loaded from a pickle cannot perform any critical action without the receiving site explicitly giving it the capability to do so.[5] In particular, there deliberately is no implicit rebinding of resources [BHS+03] in Alice ML. If rebinding is desired, it must be programmed explicitly.

This restriction may seem to severely limit the utility of pickling. In a certain sense, pickles need to be 'pure' – they may not contain any objects whose behaviour is observable or undefined) outside a process. How can we still exchange behaviour that is 'effectful'? The answer is abstraction. When two sites want to communicate a function that makes use of resources then the source site has to abstract over the required resources, and the target site can then supply them (or choose not to do so). This abstraction can happen on the function level, or on the module level.

However, such functional (or functorial) abstraction can get cumbersome and inflexible in practice. In Alice ML, a more comfortable approach is to employ the component system (Section 7) to abstract over resources and other functionality *en-mass* by means of dynamically computed components (Section 7.2). On the target site, those components will then import all resources needed implicitly, without the target process needing to perform any explicit plugging. It can still control supplement of resources through the security mechanism of the component system if desired (Section 7.5).

5.3.1. State

There are three possible ways to reconcile state (i.e. mutable references, arrays, etc.) with pickling:

- **Sited State**. Stateful values are considered resources. Any attempt to pickle them results in failure.
- **Cloning**. Stateful values are copied. That is, each unpickling operation yields a fresh copy that is equivalent to the original, up to external references (i.e. *aliasing*).
- **Distributed State**. Stateful values are pickled as remote references. This implies that Alice ML processes implement distributed state.

Both cloning and distributed state compromise one of the requirements for pickles:

- Cloning violates the transparency property, because a copy is only equivalent up to aliasing.

[4] Strictly speaking, it is still possible for hostile code to perform denial-of-service attacks, e.g. by continuously allocating memory or spawning threads. This possibility has to be addresses by other means, which lie beyond the scope of the mechanism described here.

[5] A caveat regards *proxies*, which represent a capability to communicate with other processes but are not considered resources. See Section 8.4.3 for further discussion.

- Distributed state violates the closedness property, because the potential for dead references is high.

Note in particular that the aliasing problem implies that cloning can (silently) break stateful abstractions, i.e. it violates abstraction safety. To pick an arbitrary example, cloning a lock can cause critical but hard to debug concurrency errors.

Moreover, both choices come with significant complications and costs in semantics and implementation. For instance, for cloning in combination with concurrency and futures (Chapter 6) it is difficult to ensure that pickling always reflects an atomic *snapshot* of the program state, as one might desire. Achieving this not only requires a fixed point iteration – in general, a stable snapshot may even fail to exist, or may fail to respect critical sections of individual threads [Kor06, TKS06].

On the other hand, both cloning and distributed state can easily be programmed as (approximate) abstractions in Alice ML: cloning in terms of the transformation mechanism (see Section 5.5), and distributed state on top of the proxy mechanism (see Section 8.1). Hence, in this thesis, we take the simplest and cleanest choice and consider state as sited.[6]

5.4. Abstraction Safety

Type safety is guaranteed even when unpickling values from unreliable or untrusted sources. But what about abstraction safety? Is encapsulation maintained across abstraction boundaries?

The answer is: only partially. There is no possibility of breaching abstraction barriers *within* the language. The generative semantics of type abstraction ensures that an abstract types representation remains sealed for any language construct.

The situation is not as simple when we step *outside* the language, though. In general, an attacker cannot be prevented from forging a pickle containing a value of abstract type that is type-correct but does not adhere to the invariants of the abstraction. Verification can check the former, but has no way of validating the latter. Generally, not even the implementation of the abstraction itself can perform that check after the fact, e.g. when global invariants are involved.

In principle, abstraction safety in this scenario could be improved by forms of encryption: generative type abstraction can be seen as the generation of cryptographic *keys*, and sealing is analogous to encryption with the key. This relation has been investigated by Sumii & Pierce in their cryptographic λ-calculus [SP03, SP04]. However, such an approach would be operationally expensive, and basically require implementing the full machinery of higher-order coercions, that we develop in Chapter 13 of the theory part of this thesis, in the actual runtime system.

More seriously though, even encryption could not guarantee full extra-linguistic abstraction safety. The problem is that higher-order pickling is so expressive that it actually enables externalising the principal owning an abstraction, and thus the cryptographic key itself!

For example, given a structure defining an abstract type `t`, abstraction safety can be enforced by encryption as long as only *values* of type `t` are exported and imported. However, as soon as the *implementation* is exported – by pickling the respective structure –, an attacker may be able to intercept it, extract the key from its representation, and is then able to forge values of type `t`.

Consequently, we doubt that it is possible – in principle – to actually realise full abstraction safety in an open language. Alice ML achieves *intra-linguistic*, but not *extra-linguistic* abstraction safety.

[6] The current Alice System implements a cloning semantics for state, because it turned out to be useful in the low-level implementation of the system itself, where transformations were not available. This may change in future versions.

However, the programmer can protect her abstractions against pickling by making them intentionally sited. The Alice ML library simplifies this by providing a type sited with the following siganture:

```
eqtype α sited
val sited : α → α sited
val value : α sited → α
```

Any value of type τ sited is precluded from appearing in pickles. Internally, this is realised by simply pairing the wrapped value with some random resource. By wrapping the representation of an abstract type into sited an abstraction prevents any client from directly or indirectly using its values for import or export. That way abstraction safety is guaranteed even in the face of malice.

5.5. Transformations

Often the representation of data structures contains redundant information, usually to speed up certain operations on them. For example, a graph with N nodes might be represented as a sparse matrix of size N^2, or its representation might internally cache certain information, like pre-computed shortest path information. In cases like that it might be desirable to omit the redundant information from the pickle to reduce size, e.g. represent the graph by a simple but compact adjacency list.

To achieve this, the pickling mechanism can be extended with support for user-defined transformations. In the language, this can be enabled by providing a primitive functor as follows:

```
functor MkTransformed (type α internal; type α external
                       val externalize : α internal → α external
                       val internalize : α external → α internal) :
sig
    type α trans
    type to : α internal → α trans
    type from : α trans → α internal
end
```

The types are kept polymorphic to enable transformation of polymorphic types.

To create a data structure with customised external representation one has to apply the functor and then define the data structure in terms of the wrapper type delivered by the functor:

```
structure G = MkTransformed (type α internal = bool matrix
                             type α external = int × (int × int) list
                             fun externalize m =
                                 (Matrix.size m,
                                  Matrix.foldi (fn (i,j,b,l) ⇒ if b then (i,j)::l else l) [] m)
                             fun internalize (s,l) =
                                  Matrix.tabulate (s, fn p ⇒ List.exists (fn q ⇒ p = q) l))
type graph = unit G.trans
fun graph n = G.to (Matrix.tabulate (n, const false))
fun addEdge (g, n, m) = G.to (Matrix.update (G.from g, n, m, true))
```

Here, the type graph implements a directed graph. Its internal representation is a boolean matrix that contains true on all coordinates (i, j) where there is an edge from node i to node j (we assume that matrix is an immutable 2D array type, with a functional update operator). The external representation is an adjacency list. We need to use the constructor and destructor functions to and from to construct and access values of this transformable type.

When a value of the type graph is pickled, the pickler will implicitly apply the provided externalize function to transform it to the lighter external representation. Furthermore, the pickle will include the internalize function (which hence is required to be unsited) to enable the target site to convert back the representation at unpickling time.

For a polymorphic example, consider an implementation of cloneable references:

```
structure CRef =
struct
    structure R = MkTransformed (type α internal = α ref
                                 type α external = α
                                 val externalize = !
                                 val internalize = ref)
    type α ref = α R.trans
    fun new x = R.to (ref x)
    fun !r = !(R.from r)
    fun r:=x = (R.from r) := x
end
```

As this demonstrates, transformations can be a substitute for stateful pickling, because they are sufficient to program picklable references (albeit without atomic semantics).

There are two limitations to this approach:

1. The type α internal cannot be recursive in a way that requires the transformation wrapper to cross-cut the recursion – the functor argument would depend on its result.

2. The internalize function cannot be sited.

Should these limitations turn out to be too tight, the former restriction would be remedied by adding recursive modules [Rus01, DHC01, Dre04], which already exist in some dialects of ML [RRS00, Ler03]. The latter could be addressed by a registration mechanism, where the transformed type is explicitly introduced on the target site prior to unpickling. We leave working out the details of such a mechanism for future work.

5.6. Modules

Pickles can contain modules. These can occur either in packages, or in the closure of functions that reference them or define them locally. The pickle of a module contains all values and functions it provides, plus the runtime representation of all types it defines.

In order to keep pickles compact in the presence of modules, the semantics of closures has to be defined carefully. Consider the following example:

```
val listToString = String.concatWith "," ∘ List.map Int.toString
do save ("list-to-string.alc", pack (val it = listToString) : (val it : int list → string))
```

Does the closure of the function listToString include the entire modules String, List and Int, and transitively, all modules reachable in the dependency graph of their implementations? Clearly, a closure thus computed can get quite large in general, and would make pickling impractical. Besides the size of a pickle, it also potentially makes a lot of functional values sited, just because some module in the closure contains a sited value.

Hence, there is a slight twist in the semantics of Alice ML, which ensures that a pickle like the one above will only contain the individual values projected from structures, instead of the entire module. Technically, this is achieved by *hoisting* all structure projections to the largest possible scope, hence performing projections outside the closure [Kor06]. That is, the above function is equivalent to the following definition:

```
val Int_toString = Int.toString
val String_concatWith = String.concatWith
val List_map = List.map
val listToString = String_concatWith "," ∘ List_map Int_toString
```

In order to give a sense for the relevance of this transformation, consider the pickle created above. Thanks to the hoisting semantics, it will produce a file of 1181 bytes in the current version of the Alice System. This pickle contains the code of 8 functions appearing in the transitive closure. In a naive semantics, the pickle would contain 9 modules with a total of 210 functions instead! Pickling this set of modules takes 18966 bytes with the current library, a more than 16-fold increase. Keep in mind that this is a very simple example, only referencing the most primitive library modules – for more high-level modules the ratio is likely to get significantly worse.

It is worth noting that this optimisation crucially relies on the fact that ML is a module-centric language and *not* an object-oriented one. With modules, a value is free-standing, and does not carry any functionality of its own (unless it is of higher-order type). In contrast, in an object-oriented model every value carries all its methods – either directly, or indirectly through its class, which will always be part of an object's closure. No comparable transformation can be applied, because all methods remain reachable through the original object.

5.7. Related Work

The term *pickling* was originally coined in the context of databases and operating systems, by Birrell, Jones & Wobber [BJW87]. However, the first programming language that was equipped with a comparable mechanism already was CLU [HL82], which had a particular focus on type abstraction, and required the programmer to provide transformation functions to pickle values of abstract type. CLU later inspired a similar mechanism for the object-oriented languages Modula 3 [BNOW95] and Java [RWWB96]. Neither of these languages meets all of the requirements stated in Section 5.1. In particular, they remain limited with respect to universality, closedness, and portability. Only Java ensures the latter and performs verification on class files.

Oz [Smo95, DKSS98, VH04] and the Mozart Programming System directly inspired the pickling mechanism found in Alice ML. Like Alice ML, pickling in Oz meets all of the listed requirements except Verifiability. Since Oz is a dynamically checked language, dynamic typing is not employed.

Acute [SLW+05] is an ML-based language for distributed programming that is closest to Alice ML and also provides a generic pickling mechanism. Unlike in Alice ML, pickling is not separated from dynamic typing (Section 5.2). Also, Acute is not intended to "protect against fraud", so there is no concept of security: pickles are type-checked but not verified,[7] and unpickling performs uncontrolled implicit rebinding of resources.

Several other functional languages also provide pickling functionality, for instance SML of New Jersey [Luc00] and Objective Caml [Ler03]. In most of them pickling is not even a type-safe operation. The only notable exception is Clean, which features high-level I/O based on dynamics [Pil96].

Kornstaedt [Kor06] discusses the design space of many aspects of pickling in detail, and also provides a more in-depth comparison of the mechanisms available in the forementioned languages. The low-level semantics and implementation of pickling in the Alice System is described by Tack & Kornstaedt [TKS06, Tac03]. They also describe a generic minimisation mechanism

[7] As mentioned, Alice ML does not implement this yet either.

(implemented in the Alice System) that reduces the size of pickles by applying automaton minimization algorithms to the data graph.

We have given a high-level formal semantics of pickling in Alice ML in previous work [Ros06]. It closely mirrors the formal development in Chapter 12 of this thesis, but was in the framework of a module calculus modelling ML modules. Some of the aspects regarding the role of pickling in Alice ML we present here are also discussed in that and other earlier work [RLT+06].

Kennedy implements a limited form of pickling in form of a combinator library written in ML [Ken04]. This is attractive for more lightweight applications like simple data persistence, but as already mentioned, a library approach fails to meet most of the requirements imposed by an open programming scenario (Section 5.1): although surprsingly flexible, Kennedy's combinators are neither universal (especially, they cannot deal with function values), nor can they guarantee properties like transparency, closedness, portability, or security. Efficiency also is a major concern: maintaining sharing requires extra effort and can only be achieved for a statically bounded number of types, because it needs per-type environments.

5.8. Summary

- Pickling is a generic mechanism for exporting and importing language-level values; it is the basis for services like persistence and inter-process communication.
- Following Oz, pickles in Alice ML are self-contained, portable and higher-order (i.e. can contain code).
- Furthermore, Alice ML pickles are type-safe.
- Pickles are generally not abstraction-safe, because abstractions can be forged by extra-linguistic means. However, abstractions can be protected against pickling explicitly.
- Pickles cannot contain resources and are hence secure with respect to critical operations.
- We identify two orthogonal dynamic checks required when importing a pickle: internal verification and external type checking.
- We propose a type-based, language-level mechanism for customising pickling by employing user-defined transformations.
- Pickling is practical even in the presence of deep module dependencies.

6. Futures

Programs communicating with the outside world often have to deal with a multitude of inputs and events that can occur at arbitrary points in time. For example, with a graphical user interface a user can usually trigger actions in ways that are not necessarily sequential. The program might have to display and handle several interactive windows at the same time. Conventional sequential programming techniques cannot adequately handle such scenarios, particularly when the process has to stay active while waiting for input. Under such scenarios it is vital to employ *concurrency* to establish several simultaneous flows of control.

Concurrency – parallel or interleaved execution of multiple interacting sequential computations – has become an omni-present phenomenon in today's computing, with natural applications in many areas: interactive systems, servers, databases, operating systems, or distributed systems usually have to perform tasks in an inherently concurrent manner. Yet, few programming languages offer adequate language-level support for concurrent programming. For the purpose of open programming in Alice ML, it was deemed crucial.

Concurrency in Alice ML is based uniformly on the concept of *futures*, an expressive mechanism for implicit synchronisation between threads that was first proposed by Halstead [Hal85] to make automatic parallelisation of functional programs in his Multilisp language effective. A future is a transparent place-holder for a yet undetermined value that allows for implicit synchronisation based on data flow. The particular design of concurrency and futures in Alice ML is due to Smolka [Smo99]. It offers four different kinds of future:

- **Concurrent future.** Holds place for the result of an expression computed in its own thread. *All* threads evaluate an expression and possess a result, hence we speak of *functional threads*.

- **Lazy future.** Also stands for the result of a concurrently evaluated expression. However, the computation is delayed until another thread actually requires its result. With lazy futures, techniques from lazy functional programming can be employed directly.

- **Promised future.** Is created through an explicit handle called a *promise*. A promised future is eliminated by *fulfilling* the associated promise through an explicit operation. Promises are reminiscent of single-assignment variables or logic variables and allow the construction of data structures with 'holes'.

- **Failed future.** Replaces a future that could not be eliminated because the associated computation terminated with an exception. Whenever a failed future is accessed, the respective exception will be re-raised in the thread accessing it.

We will describe the different flavours of futures informally in the following sections. A formal semantics of futures has been given by Niehren, Schwinghammer & Smolka [NSS05].

Futures are a convenient generic mechanism for implicit communication and synchronisation between threads. As such they are comparatively simple, but expressive enough to enable formulation of a broad range of concurrency abstractions. Note that threads also have *shared state*, which allows them to communicate through conventional stateful data structures. Futures enable the implementation of the necessary locking mechanisms for controlling access to them.

Since the focus of this thesis lies elsewhere, we will only briefly discuss concurrent programming. We refer the interested reader to related literature on the subject for a more thorough discussion of the issues involved in designing and using concurrent programming languages. A good starting point presenting a different approach for integrating concurrency into SML is the book by Reppy [Rep99]. More specific examples demonstrating the use of futures can be found in [Smo99].

6.1. Concurrency

Future-based concurrency is very light-weight – any expression can be evaluated in its own thread. Thread creation is straightforward: a concurrent computation is forked off be prefixing an expression with the spawn keyword:

spawn *exp*

This phrase immediately evaluates to a fresh *concurrent future*, standing for the yet unknown result of *exp*. Simultanously, evaluation of *exp* is initiated in a new thread. As soon as the thread terminates, its result value globally replaces the future.

A thread is said to *touch* a future [FF95] when it performs an operation that requires the actual value the future stands for. A thread that touches a future is suspended automatically until the actual value is determined. That is, synchronisation on futures is implicit, there are no explicit operations to access a future. This is known as *data flow synchronisation*.

Thanks to futures, threads give results, and concurrency can be orthogonally introduced for arbitrary parts of an expression. For example, to evaluate all constituents of the application $e_1(e_2, e_3)$ concurrently, it is sufficient to annotate the application as follows:

(**spawn** e_1) (**spawn** e_2, **spawn** e_3)

Hence, threads with futures blend perfectly into the "everything is an expression" philosophy of functional programming languages. For that reason, we call them *functional threads*.

6.1.1. Synchronisation

A thread blocks when it touches a future. Only few operations touch futures:

- Procedure application touches the procedure value.
- Pattern matching touches the examined value (unless the pattern is a variable or wildcard).[1]
- Exception raising touches the exception value.
- Unpacking (Chapter 4) touches the package.
- Primitive operations (e.g. op+, op:=) touch some or all of their arguments.
- Structural primitive operations (e.g. op=, or pickling (Section 4.2)) may perform *deep* touches; that is, they traverse structured values (usually depth-first, left-to-right) and potentially touch all of the contained futures.[2]

[1] This implies that projection from a record or tuple, and access to a reference touch their respective argument, because they are all defined by pattern matching.

[2] The precise behaviour depends on the operation. For example, polymorphic equality will terminate traversal as soon as equality is discovered to be disentailed. Moreover, operations might abort with an exception – particularly when encountering a failed future (Section 6.3).

In particular, note that procedure application, by itself, does evaluate but *not* touch its argument. For instance, the expression

```
(fn x ⇒ 5) (spawn forever ())
```

where forever is a function that never terminates, does not block.

Futures provide for complex one-to-many communication and synchronisation. Consider the following example:

```
val offset = spawn (sleep (Time.fromSeconds 120); 20)
val table = Vector.tabulate (40, fn i ⇒ spawn fib (i + offset))
```

The first declaration starts a thread that takes two minutes to deliver the value 20. Given an appropriate definition of the Fibonacci function fib, the second expression will construct a vector of the 20th to 60th Fibonacci numbers. The computation of the individual table entries depends on offset, but since the entries are computed concurrently, construction of the table itself can proceed without delay. However, the respective threads computing the entries will all block until offset is determined. Consecutive code can access the vector without caring about the progress of the threads. If evaluation depends on a value that is not yet determined, it will automatically block as long as required.

Besides implicit synchronisation, Alice ML offers primitives for explicit synchronisation on futures:[3]

```
val await : α → α
val awaitEither : α × β → (α, β) alt
```

The function await is a variant of the identity function that touches its argument: if applied to a future, it blocks until the future has been replaced by a proper value. A straightforward abstraction using this function is a higher-order *barrier*, which applies a list of functions concurrently and then waits for all computations to terminate:

```
fun barrier fs = map await (map (fn f ⇒ spawn f ()) fs)
```

The function awaitEither implements *non-deterministic choice*: given two futures it blocks until at least one has been determined. It is sufficient as a primitive to encode complex synchronisation with multiple events. As a simple example, consider an abstraction for waiting with time-out:

```
fun awaitTimeout time x =
    case awaitEither (x, spawn sleep time) of
        | FST x ⇒ x
        | SND _ ⇒ raise Timeout
```

6.1.2. Asynchronicity

Functional threads allow turning a synchronous call to a procedure – say, f – into an *asynchronous* one by simply prefixing the application with spawn:

```
val result = spawn f(x, y, z)
```

The ease of making asynchronous calls even where a result is required is important in combination with distributed programming (Section 8), because it allows for *lag tolerance*: the caller can continue its computation while waiting for the result to be delivered. Data flow synchronisation ensures that it will wait if necessary, but at the latest possible time, thus maximising concurrency. In a distributed setting this is particularly useful as it helps hiding network latency for remote procedure calls (Section 8.1).

An interesting degenerate case arises where the thread's result is ignored, e.g.

[3] The type alt is defined in the Alice library as: **datatype** (α, β) alt = FST **of** α | SND **of** β

```
(spawn f (x, y, z) ; g (v, w))
```

Such usage emulates the behaviour of asynchronous calls like they are found in many other concurrent languages, where asynchronous calls can never return anything. It can be implemented as efficiently.

Sometimes it is desirable to define certain functions as asynchronous per se. Alice ML supports asynchronous function definitions with straightforward syntactic sugar for the function declaration syntax: if a function post is defined like

```
fun spawn post (x, y, z) = ...
```

then it is not necessary for the caller to spawn a thread at application point. The callee takes care of that, and the call returns immediately.

6.2. Laziness

ML employs an *eager* (or *call-by-value*) evaluation strategy: any arguments are evaluated before a function is applied. Eager evaluation has certain advantages. In particular, it makes algorithmic complexity (in space and time) fairly predictable, and coexists more peacefully with side-effects. However, certain algorithms and data structures are expressed more elegantly or more efficiently with a *lazy* (or *call-by-need*) evaluation strategy [Oka98]. It hence has become a common desire to marry eager and lazy evaluation.

The future mechanism provides an elegant way for supporting laziness: a *lazy future* is a form of future that is introduced analogously to a concurrent future, by prefixing an expression with the keyword lazy:

```
lazy exp
```

This phrase will not evaluate *exp*, but instead it returns a fresh lazy future, standing for the yet unknown result of *exp*. Evaluation is triggered when some thread first touches the future. At that moment the lazy future becomes a concurrent future, associated with a fresh thread performing the computation. Evaluation proceeds as for concurrent futures (Section 6.1).

In other words, lazy evaluation can be selected for individual expressions by prefixing them with the lazy keyword. For example, the expression

```
(fn x ⇒ 5) (lazy f ())
```

will *not* call f. A fully lazy evaluation regime can be emulated by prefixing *every* subexpression with the lazy keyword, but usually only few strategic annotations are necessary.

A simple example of the use of laziness is a generator for the (infinite) lazy stream of natural numbers:

```
fun enum n = lazy n :: enum (n+1)
```

Applying this function,

```
val nats = enum 0
```

just delivers a lazy future. Only when an element of the list is requested, the necessary prefix will be computed. For example, first evaluating List.nth (nats, 5) will trigger computation of the first five elements of nats. After that, nats is no longer a future, but has the following shape:

```
0 :: 1 :: 2 :: 3 :: 4 :: 5 :: _lazy
```

where _lazy stands for another lazy future representing the uncomputed tail of the list.[4] A consecutive evaluation of List.nth (nats, 3) does not require any recomputation. Note that only the spine of the list is lazy in this example, the elements are not constructed as futures.

In order to support the definition of lazy functions using equational clauses, Alice ML extends the definition of SML's sugared function declaration syntax with support for the lazy keyword (analogous to the use of spawn for defining asynchronous functions, Section 6.1). For example, consider a lazy variant of the standard map function, which can be applied to streams without blocking or unnecessarily triggering lazy suspensions:

```
fun lazy map f nil     = nil
      | map f (x::xs) = f x :: map f xs
```

which desugars into

```
val rec map = fn f ⇒ fn l ⇒
    lazy case (f, l) of
        | (f, nil) ⇒ nil
        | (f, x::xs) ⇒ f x :: map f xs
```

Note how the rewriting makes the pattern matching properly happening *inside* the lazy suspension.

Literature describes many uses of laziness, which are all applicable in Alice ML. However, the primary, but *implicit* source of laziness in Alice ML is the component system: every reference to an imported entity is in fact lazy, so that components are loaded only when needed. Components and the lazy linking involved will be described in Section 7.

It should be noted that Alice does in no way restrict the combination of laziness with effects. As with concurrency, it is good advice to keep side effects, and particularly the non-local use of mutable state, to a minimum in lazy computations.

6.3. Failure

ML is an impure language, in which evaluation can terminate with two possible outcomes: ordinary termination with a result value, or *exceptional* termination with an uncaught exception. What happens to a future when the associated thread terminates with an exception? Clearly, threads suspending on the future need to be notified of the exceptional condition, so that they do not dead-lock on a result that will never arrive.

This situation is dealt with by *failed futures*. Failed futures allow controlled propagation of exceptions between threads. Every failed future carries an exception. When a thread terminates exceptionally, the associated future becomes failed with the uncaught exception. Any attempt to touch the failed future re-raises this exception in the respective thread. For example, evaluating

```
val x = spawn raise Empty
```

does *not* raise an exception (in the current thread). However, a consecutive attempt to evaluate the expression x+1 will propagate the exception and re-raise Empty in the current thread.

A special error condition with respect to futures is the attempt to determine and replace a future with itself (it is perfectly valid to replace it with a *different* future). For example, a thread may return its own future, by exploiting recursion or side effects. Consider:

[4] This is precisely how the interactive Alice System would print nats at that point.

```
let
    val r = ref 5
in
    r := lazy !r;
    !r + 1
end
```

There is no way to eliminate the future in cases like this, an erroneous configuration that is called a *black hole* and cannot generally be detected statically. The error is flagged by failing the respective future with the special exception Cyclic. Consequently, the above code will raise this exception during evaluation of the addition.

Failed futures can be employed in different ways. One strategy is to ignore the possibility of exceptions due to failed futures in all places except some supervising control threads that have the responsibility for handling them and probably restarting certain computations. This is basically the philosophy of "letting it crash" that is favoured and successfully used in the concurrent language Erlang [Arm03]. Alternatively, default handlers can be installed *within* critical thread expressions, so that exceptions are ensured not to escape. No commitment is made by the language semantics, futures are the primitive that enables programming different strategies as higher-level abstractions.

6.4. Promises

Functional threads and lazy evaluation offer convenient means to introduce and eliminate futures. However, the direct coupling between a future and the computation delivering its value often is too inflexible, because it demands an initial commitment to the way the information is obtained. A thread might want to create a future without making such a commitment early on. It might even want to allow some other actor to deliver it. For such cases, *promises* are a more fine-grained mechanism that allows for creation and elimination of futures in separate operations.

Promises are available through a library structure named Promise, with the following signature:

```
type α promise
exception Promise
val promise : unit → α promise
val future : α promise → α
val fulfill : α promise × α → unit
val fail : α promise × exn → unit
```

A promise is an explicit handle for a future. It virtually states the assurance that a suitable value determining the future will be made available at some later point in time, fulfilling the promise. When a new promise is created with the procedure promise, a *promised future* is created along with it that can be obtained with the future function. A promised future largely behaves like a concurrent future, in particular by allowing data flow synchronisation. The difference is that it is not replaced automatically, but has to be eliminated by explicitly applying the fulfill function to its promise. A promise can also be 'broken' by means of the fail function, yielding a failed future carrying the corresponding exception.

A promise may only be fulfilled or failed once – after one of these operations was successfully performed any further attempt will raise the exception Promise. This yields a view of promises as single-assignment references, that differ from conventional references in that they are created uninitialised, but may only be assigned once. Dereferencing them prior to assignment delivers a future standing for the later (immutable) content. Figure 6.4 shows how the different operations on promises and references correspond.

type α promise	**type** α ref
val promise : unit $\to$ α promise	**val** ref : $\alpha \to \alpha$ ref
val future : α promise $\to \alpha$	**val** ! : α ref $\to \alpha$
val fulfill : α promise $\times \alpha \to$ unit	**val** := : α ref $\times \alpha \to$ unit

Figure 6.1.: Promises vs. references

```
fun append (l1, l2) =
let
    fun iter (p, nil)    = fulfill (p, l2)
      | iter (p, x::xs) = let val p' = promise () in fulfill (p, x::future p'); iter (p', xs) end
    val p = promise ()
in
    iter (p, l1); future p
end
```

Figure 6.2.: Tail-recursive append with promises

Promises allow the partial and top-down construction of data structures with holes, as exemplified by the tail-recursive formulation of the append function shown in Figure 6.2. They can also be used to create cyclic data structures:

```
val p = promise ()
val ones = 1 :: future p
do fulfill (p, ones)
```

The variable ones is now bound to an infinite list of 1s.[5] More generally, promises can be utilised to tie arbitrary recursive knots:[6]

```
val p = promise ()
val even = future p
fun odd n = n≠0 andalso even (n-1)
(*) no syntactic recursion between odd and even
fun even n = n=0 orelse odd (n-1)
do fulfill (p, even)
```

In conjunction with packages, promises can even be used to construct simple recursive modules (Section 6.6).

Despite these uses, the main purpose of promises is to support concurrent programming: for example, they can be used to implement streams and channels as lists with a promised tail, and they provide an important primitive for programming synchronisation, as we will see in the next section.

[5] An obvious consequence of the presence of futures is that datatypes definitions in Alice ML are no longer inductive, as in SML, but rather coinductive, as in Haskell or Objective Caml.

[6] Note that the same is possible with lazy futures and references:

```
val r = ref (fn _ ⇒ raise Domain)
val even = lazy !r
fun odd n = n≠0 andalso even (n-1)
fun even n = n=0 orelse odd (n-1)
dor := even
```

6.5. Locking

An important property of procedures or abstractions in concurrent programming is *thread safety*: a program fragment is thread-safe if no amount of concurrency can leave it in an inconsistent state. In Alice ML, all primitive operations that access or modify state are *atomic*. That is, when such an operation is performed, no other thread can interfere. For example, the effect of assigning a reference in one thread will not depend on other threads running concurrently.[7]

Of course, that alone is not sufficient for thread-safe programming: when multiple threads share mutable state, interference between the concurrent threads can quickly cause not physical, but logical state inconsistencies. It is thus imperative to synchronise access, usually by forms of locking on critical sections. Alice ML requires no primitive locking mechanisms to achieve that, such mechanisms can be fully bootstrapped from promises and references.

To that end, the only primitive we require is an atomic exchange operation for references. Atomic exchange is a variant of the fundamental test-and-set operation [Hal85]:

```
val exchange : α ref × α → α
```

The exchange operation alone is sufficient to express basic synchronisation mechanisms. However, without further primitives, their implementation would often require forms of polling. Along with futures and promises such polling can be circumvented.

As a simple example demonstrating this, Figure 6.3 presents a higher-order function implementing (non re-entrant) mutex locks for synchronising an arbitrary number of functions.[8] The following snippet illustrates its use to synchronise concurrent communication to standard output, by preventing execution of f and g to be interleaved (without making a commitment with respect to the order of execution of the calls):

```
val mutex = mkMutex ()
val f = mutex (fn x ⇒ (print "x = "; print x; print "\n"))
val g = mutex (fn y ⇒ (print y; print "\n"))
do spawn f "A"; spawn g "B"; spawn f "C"
```

A lock itself is represented by a reference. When it contains () the lock is free, whenever it contains a future the lock is taken. To take the lock, a fresh future is created and stored in the reference while simultaneously retrieving its previous content (via exchange). As soon as that content is determined to be () the pending function call can be executed; upon its return the lock is released by eliminating the new future. Care must be taken to also release the lock when the function terminates with an exception.[9]

As an aside, we would like to point out that this locking abstraction profits from the fact that SML prefers Cartesian formulations of n-ary functions – with curried functions, the mutex cannot be used as simple, but requires a careful η-expansion, which makes its use much more error-prone. The same observation will reoccur with other higher-order abstractions of similar shape, e.g. the proxy primitive (Section 8.1).

6.5.1. Promises Revisited

It should be noted that promises are derivable in a language with concurrent futures and an atomic exchange like we just introduced. Figure 6.4 shows a non-primitive implementation of

[7] While this may sound like a matter of course, it is not the case in Java, for instance, where assignment of scalars larger than 32 bit (e.g. double values) can in fact leave the variable in an inconsistent state and hence requires explicit locking [LY96].

[8] Similar functionality is part of the Alice library.

[9] Alice defines exp_1 finally exp_2 as syntactic sugar for executing a finaliser exp_2 after evaluation of exp_1 regardless of any exceptional termination, similar to the try... finally... expression or statement in other languages.

```
(*) mkMutex : unit → (α → β) → (α → β)
fun mkMutex () =
let
    val r = ref ()                                    (*) create lock
in
    fn f ⇒ fn x ⇒
    let
        val p = promise ()
    in
        await (exchange (r, future p));               (*) take lock
        f x
        finally fulfill (p, ())                       (*) release lock
    end
end
```

Figure 6.3.: Mutexes for synchronised functions

```
datatype α state = FREE | LOCKED | FULFILLED of α
type α promise = α state ref
exception Promise

fun promise () = ref FREE

fun poll p = case !p of FULFILLED x ⇒ x | _ ⇒ poll p
fun future p = spawn poll p

fun fulfill (p, x) =
    case Ref.exchange (p, LOCKED) of
      | FREE ⇒ p := FULFILLED x
      | LOCKED ⇒ raise Promise
      | FULFILLED y ⇒ (p := FULFILLED y; raise Promise)
```

Figure 6.4.: A non-primitive implementation of promises

promises. It relies on one polling thread for each created promise, whose task is to replace the corresponding future. The decomposition hence is not practical as an implementation strategy, but justifies the conceptual integration of promises as a library instead of language primitives.

Note that the implementation of the fulfill has to use locking (by employing Ref.exchange and the auxiliary state LOCKED) to avoid race conditions from threads that attempt to fulfill the same future concurrently.

6.6. Modules

Futures are not limited to the core language, entire modules can be represented by futures, too. In particular, module expressions can be evaluated lazily or concurrently. For instance, lazy evaluation can be imposed on a module computation by prefixing the respective expression with the lazy keyword (see the syntax of modules in Figure 3.1):

lazy *strexp*

Concurrent evaluation of modules using spawn is available analogously. Lazy modules are sometimes useful to perform functor applications lazily. However, the main importance of futures with respect to modules lies in the central role they play for the Alice ML component system (Chapter 7): lazy module futures are ubiquitous as a consequence of the lazy linking mechanism for components.

Like in the core language, a module future is *touched* by particular operations:

- Functor application touches the functor.
- Some primitive library functors touch some or all of their arguments.
- Applying certain primitive functions on packages (in particular pickling), touches the contained module.

It has to be stressed that projection from a structure (via a long identifier M.a) does *not* touch the respective structure. That is, structure selection is implicitly lazy, and a structure is only touched if one if its fields is touched itself. Hence, evaluating a structure lazily defers evaluation of its body until one of its fields is needed. Consider:

```
structure M = lazy struct fun id x = x; val _ = print "Now" end
val id = M.id
do print "Not"
val x = id 7
```

Merely evaluating M.id does not touch M, but applying it to an argument does – executing the above program fragment prints "NotNow". The lazy semantics for structure projection naturally extends to open declarations. It is motivated by the observation that structures are primarily used to define local scopes and organise name spaces. It is not desirable that mere grouping and qualified naming changes strictness – otherwise modular programming would become an obstacle for laziness (and particularly lazy linking of components – Chapter 7).

Also note that creating and accessing packages (Chapter 4) does not per se touch the respective module.

Module futures can be failed, e.g. if the module evaluation terminates with an exception. Exceptions from failed module futures can be caught because Alice ML allows modules to be declared in local scope (Section 3.2), which can be surrounded by a handler.

There are no promised module futures, but a module can be projected from a package (Chapter 4) that is a promised future, giving a similar effect. For example, we can create simple mutual recursion between modules:

```
val p = promise ()
structure E = lazy unpack future p : sig val even : int → bool end
structure O = struct fun odd n = n≠0 andalso E.even (n-1) end
structure E = struct fun even n = n=0 orelse O.odd (n-1) end
do fulfill (p, pack E : sig val even : int → bool end)
```

Note however, that any attempt to actually make type members mutually recursive will most likely result in a deadlock, because of the behaviour of type futures described below. Moreover, the module's signatures cannot be made mutually recursive. Hence, this approach is no substitute for a proper recursive module mechanism [Rus01, DHC01, Dre04].

6.7. Types

Types (which includes signatures) may reside in structures. The existence of module futures hence implies the existence of *type futures*. The notion of a type future is a novel concept that,

to the best of our knowledge, has not been considered before. Most of the times, however, the presence of type futures is not observable. The reason is that there are only very few operations that touch types:[10]

- Sealing touches the ascribed signature.
- Unpacking (Section 4) touches both, the package signature and the target signatures.
- Performing certain primitive operations on packages (in particular pickling), touches the contained signature.

Touching a type generally can trigger arbitrary computations, e.g. loading a component (Chapter 7). For instance:

```
structure T = lazy struct type t = int; val _ = print "Now" end
val p = pack (val n = 5) : (val n : int)
structure S = unpack p : (val n : T.t)
```

Evaluating the declaration for structure S will request the structure T, because its type member t is needed to decide the dynamic type check upon unpacking p.

Note that type declarations themselves never touch a type. In particular, type application is best thought of as being performed lazily. Constructing a package does not touch the package signature either. For example, evaluating

```
structure S = lazy struct val n = 5 end
structure T = lazy struct type α t = α end
val p = pack S : sig val n : int T.t end
```

will request neither S nor T. However, unpacking p later will request T (only).

6.8. Related Work

Futures were first introduced in Multilisp [Hal85], as a means to perform concurrent or delayed (lazy) computations. They were an attempt to make automatic parallelisation of functional programs effective. Flanagan & Felleisen later gave a formal semantics for futures [FF95]. The future mechanism found in Alice ML refines and extends that work. It has been proposed by Smolka [Smo99] and was formalised by Niehren, Schwinghammer & Smolka [NSS06]. Lazy type futures have been investigated formally in a recent thesis by Neis [Nei06].

Futures have already been available in more recent versions of Oz [Smo95, Moz04], but have never been formalised in the context of a relational language. However, futures are closely related to logic variables in relational languages like Oz or Prolog [Den85, SS94]. Concurrent logic programming introduced the idea of employing logic variables for synchronisation [Sha89, Smo95]. Unlike futures however, logic variables are bound via bi-directional unification. Consequently, they provide no distinction between read and write access and hence are relatively fragile with respect to concurrency abstractions. Promises like found in Alice ML have been introduced to avoid this problem [Smo99]. Basics of the concrete design of futures and promises in Alice ML are also described in previous work [RLT+06], on which this chapter extends.

The notion of promise has first been proposed by Liskov & Shrira [LS88] as a mechanism to hide network latency for remote procedure calls. For the same purpose, promises have been integrated into the untyped language E [Mil06], where they are directly coupled with the method

[10] All these operations potentially need to look at the whole structure of a type or signature, and hence perform deep touches. However, the specifics are left undefined and tend to depend on the internal representation of runtime types as well as the implementation of the respective algorithms, e.g. for checking subtyping.

invocation mechanism. Closely related are *I-structures*, as found in the Id language [ANP89]: instead of a single slot like a promise, they provide an array of slots. All these features differ from the notion of promise in Alice ML by not returning a future upon early read access, but blocking instead.

Lazy evaluation was first proposed by Friedman & Wise [FW76]. The SASL [Tur76] and Miranda [Tur85] languages by Turner pioneered its use as the fundamental evaluation strategy of a programming language. Today, the prime example of a language usually implemented based on lazy evaluation is the non-strict language Haskell [PH99].

Several concurrent extensions to ML and other functional languages have been proposed in the past. Reppy's Concurrent ML [Rep99] extends Standard ML with threads, channels, and first-class synchronisation events. Unlike with futures, synchronisation is explicit. Events can be composed, but no user defined events are possible. Channels are the primary means of communication. Concurrent ML is available as part of the SML of New Jersey distribution.

Facile [TLK96] is another extension of SML with concurrent and distributed programming features. Its concurrency model is based on CCS and the π-calculus [Mil89]. It provides channels and composable guards as its central communication and synchronisation construct.

More ambitious is the JoCaml language [CL99, FMS01], which extends Objective Caml with concurrency and distribution, based on the Join calculus [FGL+96], an extension of the π-calculus. The primary means of communication and synchronisation are *join patterns*, which atomically match a combination of values in a soup of messages. Join patterns enable formulation of concurrent systems in a comparably declarative style, which resembles finite state automata.

Erlang [Arm03] is an untyped functional language for concurrent and distributed programming. It uses an Actor-style approach [HBS73], where processes are represented by pure functions. They communicate over message channels that are implicitly associated with every process. Erlang has been developed in an industrial context and is probably the most successful and most widely used concurrent functional language so far.

Lately, *transactional memory* has been proposed as an alternative, lock-free paradigm for concurrent programming that is supposed to scale better than locking-based approaches [HMPH05], and has been implemented for Haskell [PH99]. Communication is through shared state, but instead of locking and synchronisation, critical state accesses are grouped into *transactions* which are executed quasi-atomically. If concurrent transactions cause a conflict, one of them is automatically rolled back and retried. The approach benefits greatly from Haskell's expressive type system, which allows to statically enforce purity of transactions.

6.9. Summary

- Alice ML adopts concurrency based on futures.
- Futures are transparent place-holders for yet undetermined values.
- Spawning a thread delivers a future of its result.
- Implicit data flow synchronisation automatically blocks and restarts threads that touch a future.
- Laziness is a simple extension where a thread does not start evaluation before its future is touched.
- Failure in a thread is captured by failed futures, which propagate exceptions in a synchronous manner when the future is touched.
- Promises decouple futures from threads and enable programming a wide range of concurrency abstractions.
- Novel in Alice ML is that modules and specifically types can be futures, too.

7. Components

Software of non-trivial complexity can neither be developed nor deployed as a monolithic block. To keep the development process manageable, and to allow flexible installation and configuration, software has to be split into functional building blocks that can be created separately, configured dynamically, and even be exchanged between processes. Such building blocks are called *components*.

We distinguish components from modules: while modules provide *logical* separation, name spacing, genericity, and encapsulation, components provide *physical* separation and dynamic composition. Modules are referred to by identifiers and static scoping rules, where components are identified by extra-linguistic means that are resolved dynamically. Both mechanisms complement each other. More precisely, components *contain* modules. It is the component system that enables closing over free references in a module implementation and hence turning it into a self-contained entity.

Alice ML incorporates a powerful notion of component that is a refinement and extension of the component system found in the Oz language[1] [DKSS98], which in turn was partially inspired by Modula-3 [BNOW95] and Java [GJS96]. It provides all of the following:

- **Separate compilation**. Components can be translated independently.
- **Lazy dynamic linking**. Loading can be performed automatically when needed.
- **Type safety**. Components carry strong type information and linking checks it.
- **Subtyping**. Type checking is tolerant against interface changes.
- **Static linking**. Components can be bundled into larger components off-line.
- **Dynamic creation**. Components can be computed and exported dynamically.
- **Sandboxing**. Custom *component managers* enable selective import policies.

The component system of Alice ML is based on a combination of different mechanisms presented in the previous chapters:

- *Higher-order modules* (Chapter 3), for encapsulating all possible language entities,
- *Packages* (Chapter 4), for dynamic type checking of imports,
- *Pickling* (Chapter 5), for representing components externally,
- *Futures* (Chapter 6), for performing linking lazily.

Packages already allow dynamic loading and exchanging of modules. However, these modules have to be fully evaluated and closed. If we wanted to delay evaluation, or to have it depend on other modules, then we would have to resort to functional (or functorial) abstraction. Components provide a much more comfortable and flexible means for achieving the same effect. Nevertheless, in Section 7.6 we will see that they can actually be expressed as a – relatively simple – functional abstraction. We will first present components as an independent feature before we show how precisely they relate to the mechanisms listed above.

[1] In Oz, components are called *functors* by slight abuse of terminology.

imp	:=	**import** *spec* **from** *scon*	import
			empty
		imp ⟨;⟩ *imp*	sequential
component	:=	*imp* ⟨;⟩ *dec*	

Figure 7.1.: Syntax of components

7.1. Compilation Units

Components are the unit of compilation as well as the unit of deployment in Alice ML. A program consists of a – potentially open – set of components that are created separately and loaded dynamically. Static linking (Section 7.8) allows both to be performed on a different level of granularity if desired, by bundling given components to form larger ones.

Every component provides a module – its *export* – and accesses an arbitrary number of modules retrieved from other components – its *imports*. Imported components are identified by URLs. Both, import and export interfaces, are strongly typed by ML signatures.

Each Alice ML source file defines, and is compiled into, a component:[2] the contained sequence of SML declarations is interpreted as a structure body, forming the export module. The respective export signature is inferred by the compiler. A component can access other components through a prologue of import declarations:

import *spec* **from** *string*

The SML signature specification *spec* in an import declaration describes the entities used from the imported structure, along with their type. All identifiers bound in the specification are in scope in the rest of the component. Because of Alice's higher-order module system (Chapter 3), these entities can include functors and even signatures. For instance, the following are valid imports:

```
import structure Pickle : PICKLE from "x-alice:/lib/system/Pickle"
import structure Server : sig val run : (α→β) → (α→β) end from "http://my.org/server"
import functor MkRedBlackMap (Key : ORDERED) : MAP where type key = Key.t
   from "x-alice:/lib/data/MkRedBlackMap"
```

The string in an import declaration contains the URL under which the component is to be acquired at runtime. Although the URL is hardwired into the code, its interpretation is completely up to the responsible *component manager* (Section 7.4), and hence configurable. Usually it is either a local file, an HTTP address, or a virtual URL denoting system library components (Alice ML uses the x-alice: scheme for this purpose).

To execute a program, a designated *root* component is *evaluated*, meaning that its defining declarations are evaluated in sequence, according to the dynamic semantics of the language. Loading of imported components is performed *lazily*, and every component is loaded at most once. Loading implies evaluation of the respective component. This process is referred to as *dynamic linking*. We defer discussion of the details of linking until Section 7.3.

Figure 7.1 summarises the basic syntax of components.

[2]Every single input into the interactive toplevel of the Alice System is also treated as a component.

7.1.1. Implicit import signatures

Compilation units are always syntactically closed. There are no free identifiers in a component, not even for most primitive operators like + (which are ordinary function names in SML) – they are all bound by some import. That enables separate compilation.

However, writing down the signatures for all imported modules would be tedious in practice. As syntactic sugar, Alice ML hence allows the type annotations in import specifications to be dropped. It suffices that the imported component are accessible (in compiled form) during compilation, so that the compiler can insert the respective types from their export signatures. For example, the previous import declarations could be abbreviated to

```
import structure Pickle from "x-alice:/lib/system/Pickle"
import structure Server from "http://my.org/server"
import functor MkRedBlackMap from "x-alice:/lib/data/MkRedBlackMap"
```

As an additional service, the compiler automatically thins implicit signatures by removing all entities that are not directly or indirectly referred in the remainder of the component. Doing so makes the compiled component maximally robust against eventual changes in parts of an interface that are not accessed.

The library of the Alice System is uniformly accessed as a set of components denoted by x-alice: URLs, like the Pickle and MkRedBlackMap modules above. For convenience and for compatibility with Standard ML, Alice ML allows to omit imports for modules from the Standard ML Basis library [GR04] by default. The respective import declarations are implicitly prepended to every compiled component source. Again, the compiler thins signatures as far as possible, and removes redundant imports introduced implicitly.

7.1.2. Example: A simple stand-alone application

Figure 7.2 shows a simple sample application consisting of two components, fib and main, which we will use as a running example in this chapter. The program expects an integer on the command line and outputs the corresponding Fibonacci number, with the help of the fib procedure supplied by the respective component. Note how the main component imports the fib function without explicitly giving its type. Also note that it uses several library structures without importing them explicitly. Even fib uses library functions, namely addition and subtraction.

Figure 7.3 shows how the same example will actually be rewritten by the compiler. It inserts the necessary imports for library entities and the type annotation for the explicit import. Note how the signatures of imported structures are recursively thinned down to the content actually required. The rewritten components are completely self-contained – they do not contain *any* free identifiers, not even for the most basic primitives.

Export signatures are not explicit in the code, but straightforward in this case: the export signature of the fib component only contains the function fib (as imported by main). The export signature of main is empty, thanks to the use of local.

7.2. Computed Components

Compilation is the most obvious, but not the only way to create components. Nor do components necessarily live in files. In fact, components are first-class entities in Alice ML, and can be constructed dynamically by an ML process. We call such components *computed components*, as opposed to compiled components (strictly speaking, a compiled component is just the special case of a component computed by the compiler, though).

```
(*) fib.aml
local
    fun fib'(i, j, 0|1) = j
      | fib'(i, j, n) = fib'(j, i+j, n−1)
in
    fun fib n = fib'(1, 1, n)
end
```

```
(*) main.aml
import val fib from "fib"
local
    val s = hd (CommandLine.arguments ())
    val t = Int.toString (fib (valOf (Int.fromString s)))
in
    do TextIO.print (t ^ "\n")
end
```

Figure 7.2.: A simple component example

```
(*) (fib.aml
import
    type int
    val + : int × int → int
    val − : int × int → int
from "x-alice:/lib/fundamental/Core"
local
    fun fib'(i, j, 0|1) = j
      | fib'(i, j, n) = fib'(j, i+j, n−1)
in
    fun fib n = fib'(1, 1, n)
end
```

```
(*) main.aml
import
    type int
    type string
    type unit = {}
    datatype α option = NONE | SOME of α
    datatype α list = nil | :: of α × α list
    val ^ : string × string → string
    val valOf : α option → α
    val hd : α list → α
    structure Int : sig val toString : int → string; val fromString : string → int option end
from "x-alice:/lib/fundamental/Core"
import
    structure CommandLine : sig val arguments : unit → string list end
from "x-alice:/lib/system/CommandLine"
import
    structure TextIO : sig val print : string → unit end
from "x-alice:/lib/system/CommandLine"
import val fib : int → int from "fib"
local
    val s = hd (CommandLine.arguments ())
    val t = Int.toString (fib (valOf (Int.fromString s)))
in
    do TextIO.print (t ^ "\n")
end
```

Figure 7.3.: The component example, rewritten by the compiler

```
structure Component :
sig
    type component

    exception Failure of url × exn
    exception Eval of exn
    exception NotFound
    exception Corrupt
    exception Sited

    val fromPackage : package → component
    val save : string × component → unit
    val load : string → component
    ...
end
```

Figure 7.4.: The Component structure

Within the language, a component is a first-class value of the abstract type component, which is defined in the library structure Component. Figure 7.4 shows an excerpt of its signature. Values of this type can either be constructed with the fromPackage function from that structure, or with a new syntactic form:[3]

comp *imp* **in** *spec* **with** *dec* **end**

Such a component expression has a syntactic structure not too different from a compilation unit. The main difference is that its export signature must be given explicitly, in form of a sequence of signature specifications *spec* between the keywords **in** and **with**. The environment obtained from the declarations *dec* must match this signature. Naturally, imports and declarations are not evaluated when the component is constructed, but when it is linked. Thus, a computed component can not only import other components, it can also perform sited operations and generate arbitrary side effects through functionality obtained from imported library components. This sets components apart from plain packages.

Note also that the imports scope over the signature – an export signature may depend on types defined in other components. In Section 7.6 the semantics of this feature will become clear.

The less visible but more important difference between compiled and computed components is that the latter need not be closed. Hence computed components can embody information that is obtained dynamically. That is useful for at least two purposes:

- *Pre-computation.* Through a staged building process, components can be created that readily provide data structures that are expensive to compute, or should be "statically generated".
- *Mobility.* Dynamic behaviour that depends on resources can be wrapped into a component and be passed to other processes.

[3] This syntax was not available in earlier versions of the Alice System; the following higher-order polymorphic functor from the Component structure had to be used as a substitute:

```
functor Create (signature S) (F : COMPONENT_MANAGER → S) : (val it : component)
```

Pre-computation already is enabled to a certain degree by packages and persistence (Section 4.2). However, packages are relatively limited: as pickles, they have to be closed, i.e. they cannot make use of any library functionality local to the site where they are loaded. In particular, they cannot easily provide behaviour that requires access to resources – e.g. input/output – because such resources and operations on them cannot be stored in pickles (Section 5.3). Components come to the rescue.

To demonstrate the utility of computed components, consider the example of a simple Hello World application. The following program computes a component that exports a function that *prints* a message containing the component's date of creation when invoked later:

```
val date = Date.toString (Date.fromTimeLocal (Time.now ()))
val component =
    comp
        import structure TextIO from "x-alice:/lib/system/TextIO"
    in
        val hello : unit → unit
    with
        fun hello () = TextIO.print ("Hello world! Created at " ^ date ^ "\n")
    end
do Component.save ("hello", component)
```

Note that date is computed outside the component, and hence included as a value. Component.save pickles a computed component to a file. Component files created this way behave exactly like components created by the compiler, they can be loaded or imported. For example, the simplest possible program utilising the component created above is the following:

```
import val hello : unit → unit from "hello"
do hello ()
```

Alternatively, we can use Component.load to load it as a first-class value, but it then requires an explicit component manager (Section 7.4) to evaluate it.

Packages can be directly converted to components with the Component.fromPackage function, which is sometimes convenient – we speak of *evaluated components* in this case, because they just export a constant module *value*. However, the key difference between a component and a simple package is that a component enables import of resources and other functionality local to the target site. As the example shows, computed components can thus have arbitrary, 'impure' behaviour, but are yet mobile, i.e. exportable through pickling. For instance, the example would not work if we omitted the import declaration for TextIO – then the local instance of TextIO.print would be in the closure of component, and pickling would fail with a Sited exception because print is a resourceful operation (Section 5.3). With the import, we effectively enforce *rebinding* of all scoped references to TextIO on the target site. However, the target site may choose to prevent the import if it wishes to restrict the capabilities of an untrusted component (Section 7.5).

Rebinding is particularly important for distributed programming. Exchanging dynamic behaviour between processes can be achieved by creating a *mobile* component serving two purposes:

- it closes over all entities obtained at creation site, thus containing the necessary dynamic information (like date above),
- it abstracts over all entities to be obtained at the target site, thus enabling pickling and (re)binding (like TextIO above).

The former is handled automatically by closure and the semantics of pickling. The latter can be controlled by the use of import declarations in the definition of the mobile component. The

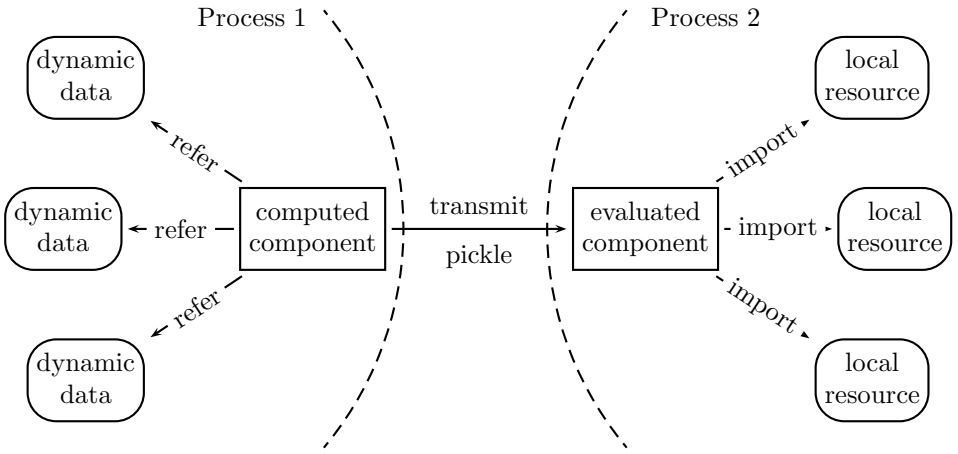


Figure 7.5.: Mobile components

rebinding itself is automatic, by the process of dynamic linking. Figure 7.5 illustrates this technique. We will show applications of this technique in the chapter on distributed programming (Chapter 8).

Note that, unlike procedural abstraction over resources, components are much more flexible. In particular, they reduce coupling: the client (the target site) does not need to know every detail about what resources are required by the component. The component can readily utilise every service that the client grants it access to, without the latter having to *explicitly* supply handles to every such service. This makes clients robust against changes in the *implementation* of components they dependent on (while subtyping already makes them robust against changes in *interfaces*). Effectively, this is achieved by employing an indirection (through the component manager responsible for linking, Section 7.4) to keep implementation details out of interfaces.

7.2.1. Pickling Components

The functions save and load from the Component structure allow storing components to disk and retrieving them. These are the basic primitives implementing persistence in Alice ML. In particular, load is the function ultimately used (by a component manager, Section 7.4) to load imported components from the file system.

The functions Pickle.save and Pickle.load that we had seen in Section 4.2 are in fact just simple wrappers defined as follows:

```
fun save (s, p) = Component.save (s, Component.fromPackage p)
fun load s = SingularComponentManager.eval (Component.load s)
```

In other words, there is only one uniform file format: all pickle files are actually (evaluated) components and can be used as such. Pickle.load can load even unevaluated components, and will evaluate them. In general however, components have imports, which requires cooperation from a component manager (Section 7.4). To keep unpickling secure (Section 5.3) and prevent accidental security breaches by untrusted components naively loaded as pickles and silently grabbing resources, Pickle.load employs a special degenerate component manager that rejects any further imports attempted by the loaded component – this basically is a simple form of sandboxing (Section 7.5). All evaluated components, especially those created with Pickle.save, will successfully load. For other components, a suitable explicit component manager is required to evaluate the component without failure.

7.3. Dynamic Linking

The components making up an application form a directed graph with respect to their import dependencies. A designated *root* is the main component of a program. Primitive library components providing system services and the most basic functionality make up the leaves of the graph.

To execute a program, its root component is *evaluated*. For compiled components, this means that the respective declarations are evaluated in sequence. As already explained, loading of imported components is performed lazily, and every component is loaded and evaluated only once. This is achieved by treating every cross-component reference as a lazy future (Section 6.2). Conceptually, the lazy thread suspended by the future is responsible for loading the component when it is touched.

The process of loading and evaluating a component requested as import (by another component) is referred to as *dynamic linking*. Linking a component involves several steps:

1. **Resolution**. The import URL is normalised relative to the URL of the current component.

2. **Acquisition**. If that URL is being requested for the first time, it is loaded.

3. **Evaluation**. If the component has been loaded afresh, its body is evaluated and its dynamic export signature computed.

4. **Type Checking**. The component's export signature is matched against the respective import signature.

Each of the steps can fail: for instance, the component might be inaccessible or malformed, evaluation may terminate with an exception, or type checking may discover a mismatch. Under each of these circumstances, all futures representing references to the component are failed (Section 6.3) with the standard exception Component.Failure (Figure 7.4):

```
exception Failure of url × exn
```

The URL denotes the requested component and the nested exception describes the precise cause of the failure. In particular, it can be an I/O exception, the exception Unpack (Section 4.1), or any of the following exceptions, which are also defined in the library structure Component (Figure 7.4):

```
exception NotFound          (*) component was not found
exception Corrupt           (*) component is malformed
exception Eval of exn       (*) component evaluation terminated with an exception
```

In the case of Eval, the uncaught exception is in turn nested.

Note that the import graph of a program may contain cycles. In the extreme, a component may even import itself, although this is hardly useful. Thanks to lazy linking, cycles pose no problem as long as at least one of the components on the cycle does not need its respective import immediately. Otherwise a deadlock may occur. In practice, cyclic imports are very rare. Programs typically rely on implicit import signatures inserted by the compiler (Section 7.1). By construction, such imports can never be cyclic, unless some of the components are changed and recompiled during the build process.[4]

[4] A trick that also works in Java.

```
// Database.aml, version 1
structure Database :>
sig
    type db
    val open : string → db
    val lock : db → unit
    val release : db → unit
    ...
end = struct ... end
```

```
// Database.aml, version 2
structure Database :>
sig
    type db
    val open : string → db
    val lock : db → unit
    val unlock : db → unit
    ...
end = struct ... end
```

```
// App.aml
import structure Database from "http://my.org/Database"
structure App =
struct
    val db = Database.open "/serve/my.db"
    do Database.lock db
    ...   (*) use data base
    do Database.release db
end
```

Figure 7.6.: Example: dynamic import in Alice ML

7.3.1. Example

We can now revisit the Java example we gave in the introduction in Section 1.3.1. Figure 7.6 shows how it can be recast in Alice ML.

Again assume that the interface of the Database component has been changed between compilation and runtime of App by renaming the release function to unlock. As before, executing App will trigger loading of the Database component when we open the database. But this time, loading implicitly performs a dynamic type check, matching the export signature found in the Database component (version 2) against the import signature demanded by App (for which the compiler will have inserted version 1). Because there is no function named release in the export signature, matching will fail, and linking will be aborted with a Failure exception. Consequently, the database will neither be opened nor put a lock on it.

Of course, this behaviour relies on the provider of the Dabase component to have chosen the right abstraction, which couples locking and unlocking into a single component. If both operations were provided by separate components, we would be left in the same situation as in Java. As always, types are no panacea, but employing typeful programming [Car91] by choosing the right abstraction can improve error protection.

7.4. Component Managers

To enable control over the process of linking imports, it is always performed with the help of a *component manager*. The main responsibility of a component manager is to locate and load requested components, and to maintain a table of components that have been loaded already. The default component manager is a module of the runtime library that is initialised on startup of an Alice ML process. It starts with an empty table and incrementally fills it as required by evaluation of the root component or any of its imports.

Concretely, a component manager appears as a structure with the signature shown in Figure 7.7. To a program, the responsible component manager is accessible not only implicitly for

```
signature COMPONENT_MANAGER =
sig
    exception Conflict

    val acquire : url → component
    val eval : component → package
    val enter : url × component → unit
    val lookup : url → package option
    val link : url → package
end
```

Figure 7.7.: The signature of a component manager

imports, but also explicitly as the library structure ComponentManager of that signature. Using it, a program can operate on first-class components and influence the manager in more direct ways. As apparent from its signature, a component manager provides several basic services on component values, most of which implement one of the linking steps enumerated in the previous section:[5]

- acquire retrieves a component, without actually evaluating or entering it,
- eval evaluates a component into a package containing its export,
- enter enters an export package into the table, raising Conflict if the URL is already taken,
- lookup retrieves a component from the table,
- link loads and enters a component from a specified URL.

Ultimately, eval is the only operation that actually consumes a value of type component – no other ways exist to access it. It will typically be used in distributed programming scenarios (Chapter 8), where components are exchanged between processes to achieve rebinding of resources via computed components (Section 7.2).

The function link combines the sequence of operations usually required to acquire and link a component given its URL. Its semantics is roughly equivalent to the following definition, except that it performs the necessary steps atomically:

```
fun link url = case lookup url of
                 | SOME p ⇒ p
                 | NONE ⇒ let val p = eval (acquire url) in enter (url, p); p end)
```

Note that there is no operation to remove a component from the table. While this is *not* a pre-requisite for achieving soundness – thanks to typing being based on generative type names – it is a desirable property of a component manager to grow monotonically, in order to maintain the property that every component is evaluated only once. If required, unloading of components can be simulated by creating a child component manager (Section 7.5), link a component with it, and then drop all references to it [Kor06].

[5]Some types currently differ slightly from Figure 7.7 in the actual Alice System, but the essence is the same.

7.5. Resources and Sandboxing

The relevance of component managers lies in their ability to control imports. That ability can be utilised to realise security measures. In an open setting it is important to handle untrusted components, and to restrict their capabilities. For example, not all components should be given unrestricted access to the local file system.

To deal with this, component managers can be employed to adopt the approach taken by Java: they can form a *sandbox* to execute untrusted components in. Sandboxing relies on three factors:

- **Sited Resources**. All critical operations and other objects that provide access to effectful services (e.g. input/output, system calls, various runtime services) are considered resources (Section 5.3). Resources cannot be pickled, hence there is no way to export them from a process. Consequently, no component imported into a process can contain any direct reference to resources.

- **Resource Acquisition through Component Managers**. The only way a component can gain access to resources is by importing them from so-called *primitive components* on the local site. Because all imports are controlled by a component manager, and primitive components are identified by a stable naming scheme, the responsible manager can thus decide what access to grant a given component.

- **Custom Managers**. It is possible to create custom managers that restrict the access to certain components and explicitly link untrusted components through them. A component manager is inherited, i.e. all imports directly or indirectly requested by a component will be linked using the same manager (or another custom manager that can only be defined in terms of that manager).

In other words, every component initially receives an (implicit) reference to exactly one resource: its component manager. It is then the responsibility of this manager to decide what further resources, and hence capabilities, are given to the component or its descendants.

The initial manager starting up an Alice process usually provides access to all resources. The resourceful *primitive components* made available by this manager are an integral part of the runtime system and do not exist independently from it.

On the language level, a custom manager simply is a user-defined implementation of the COMPONENT_MANAGER signature. For example, the Alice ML library provides a functor to create new managers with specialised behaviour. When a custom manager is created, then it can only use capabilities provided by its own 'parent' manager. In particular, the only way to provide access to primitive components is by forwarding the respective requests to the parent manager. Hence a custom manager can never grant more access than it has itself. Thus there is no way to bypass the restrictions of a manager – a custom manager represents a proper sandbox.

There are several possible ways in which a custom manager can restrict access:

- It can simply reject loading from specific system URLs altogether, and only forward requests it deems save.

- It can restrict the signature under which specific components are made available, by forwarding the request to its parent manager, but repackaging the result with a thinner signature (e.g. removing the operations for opening output files from TextIO).

- It can substitute critical components by security-sensitive wrappers, that dynamically check access per operation, using fine-grained policies (this corresponds to Java's Security Manager).

Either of these approaches is feasible and programmable. A thorough analysis of the design space is independent of language design issues as discussed here, and thus left for future work. However, the language semantics provide all necessary prerequisites.

Scharfstein describes one possible design and implementation of a sandboxing infrastructure for Alice ML [Sch06], which uses secure wrapper components performing dynamic checks with respect to user-configurable policies, but also enables the user of the sandbox to plug in her own behaviour. Noteworthily, sandboxing is realised fully within the language itself and does not require any direct support from the core system. A variation of this design is part of the current Alice ML library.

We should stress again that security is not the focus of this thesis, and that the above description is only meant to show that Alice ML is *potentially* secure. We do not claim to describe an end-to-end security concept. Nor does the current implementation of the language provide it.

7.6. Decomposition of the Component System

At first, components may look like a complex mechanism. In this section we will refute this presumption, by giving a simple reduction of components to functions and packages. The merit of this reduction is three-fold:

- It keeps the language conceptually simple.
- It defines component semantics without need for additional technical machinery.
- Soundness and related properties follow for free.

The close relation between components and the concepts presented in previous chapters – like modules, packages and futures – is obvious, so one might hope that there exists a simple reduction. And indeed, components can be understood as mere syntactic sugar.

7.6.1. Components

A component can be seen as a function that evaluates to a package (Chapter 4). The package encapsulates the export module and its signature.

More precisely, the abstract type component can be implemented as a higher-order function type:

type component = (url → package) → package

Its argument encapsulates the component manager, needed to acquire imports. Supplying the function evaluates the component.

Accordingly, a component expression

comp *imp* **in** *spec* **with** *dec* **end**

can be viewed as syntactic sugar for the function

fn import ⇒ **let** *imp′* **in pack** (*dec*) : (*spec*) **end**

where import is a reserved identifier and *imp′* is obtained from *imp* by rewriting every import declaration

import *spec* **from** s

Component syntax	Reduced syntax
imp ⟨;⟩ *dec*	**comp** *imp* **in** *spec′* **with** *dec* **end**
comp *imp* **in** *spec* **with** *dec* **end**	**fn** import ⇒ **let** *imp* **in** **pack struct** *dec* **end** : **sig** *spec′* **end** **end**
import *spec* **from** *s*	**open lazy unpack** import *s* : **sig** *spec* **end**

Notes: (1) *spec′* describes the principal signature of *dec*.
(2) import is a unique identifier.

Figure 7.8.: Reduction of components to packages

to[6]

open lazy unpack import s : (*spec*)

Likewise, a compilation unit can be rewritten, except that the package signature is derived and inserted by the compiler.

This simple transformation fully determines the semantics of components. In particular, it makes obvious where laziness applies, how dynamic type checking is performed for imports, and how acquisition of imported component is delegated to the component manager: every component receives the function import for acquiring its imports, as packages of their export.[7] Evaluation of a component produces a package that contains its export.

Figure 7.6 summarises the definition of components as syntactic sugar. Note that it relies only on packages, and the fact that structures are universal containers for all available language entities. The latter in fact is the main reason why we had to generalise ML modules to higher order (Chapter 3). If ML had no functors at all, the approach would still work! Likewise, if we did not have futures we could still express dynamic linking, except that we would have to perform it eagerly.

Consequently, the component system as presented is not necessarily limited to Alice ML or an ML-like language – it would be applicable to any language with a nested module system as long as it supports a comparable notion of signature. To represent components externally, a candidate language also would have to provide (or make feasible) a higher-order pickling mechanism. But apart from these two requirements, nothing in the basic approach is language-specific.

7.6.2. Examples

Reconsider the (rewritten) Fibonacci application from Section 7.1 (Figure 7.3). Figure 7.9 exhibits how the same example is decomposed according to the rules just described.

Note that the results are closed expressions. Moreover, the decomposition finally makes the inferred export signature explicit.

[6] The syntax "**open** *strexp*" is syntactic sugar for the declaration

localstructure*strid* = *strexp* **inopen***strid* **end**

where *strid* is a fresh identifier.

[7] For presentational purposes, we treat the type url as synonym for string here, although it is actually an abstract type demanding explicit conversion.

```
(*) fib
fn import ⇒
pack struct
    open lazy unpack import "x-alice:/lib/fundamental/Core" : sig
        ... (*) (import specs as in Figure 7.3
    end
    local
        fun fib'(i, j, 0|1) = j
          | fib'(i, j, n) = fib'(j, i+j, n−1)
    in
        fun fib n = fib'(1, 1, n)
    end
end : sig val fib : int → int end

(*) main
fn import ⇒
pack struct
    open lazy unpack import "x-alice:/lib/fundamental/Core" : sig ... end
    open lazy unpack import "x-alice:/lib/system/CommandLine" : sig ... end
    open lazy unpack import "x-alice:/lib/system/TextIO" : sig ... end
    open lazy unpack import "file:fib" : sig val fib : int → int end
    local
        val s = hd (CommandLine.arguments ())
        val t = Int.toString (fib (valOf (Int.fromString s)))
    in
        do TextIO.print (t ^ "\n")
    end
end : sig end
```

Figure 7.9.: The component example decomposed

7.6.3. Component Managers

The import function used in the component decomposition encapsulates a component manager. As explained in Section 7.4, its job is locating components and keeping a table of loaded components. When a component is requested for the first time, it is loaded, evaluated and entered into the table. It remains to be shown how component managers themselves can be implemented.

Figure 7.10 contains a simple model implementation of such a function. It locally defines a set of auxiliary functions that almost directly mirror the functionality seen in the COMPONENT_MANAGER signature (Figure 7.7). Apparently, most of these functions are straightforward. However, we show a quite limited version of acquire, which simply delegates to Component.load (Section 7.2) and thus can only handle file URLs. For other URL schemes (particularly http:) additional services may be accessed, which we will not describe here. We also ignore primitive components.

The main complication is URL resolution: all URLs in an import declaration have to be interpreted relative to the domain (*authority*) and path of the URL under which the importing component was acquired. This is necessary to make groups of components relocatable across directory structures and network domains. Consequently, the import function passed to a component must know about the component's associated URL, so that URL has to be passed as an additional parent argument – import' is the abstraction of import over this additional argument. Assuming existence of an auxiliary function *resolve* : url × url → url that syntactically resolves

```
exception Conflict
val table = ref [] : (url × package) list ref
val mutex = mkMutex ()

fun import' parent =
let
   fun acquire url =
       Component.load url handle exn ⇒ raise Failure (url, exn)

   fun lookup "x-alice:/lib/system/ComponentManager" =
       pack (
          structure ComponentManager =
          struct
              exception Conflict = Conflict
              val acquire = acquire
              val lookup = mutex lookup
              val enter = mutex enter
              val eval = eval' "."
              val link = import' "."
          end
       ) : (structure ComponentManager : COMPONENT_MANAGER)
     | lookup url =
       List.find (fn (x,_) ⇒ x = url) (!table)

   and enter (url, package) =
        if isSome (lookup url) then raise Conflict
        else table := (url, package) :: !table

   and eval' url component =
        component (import' url) handle exn ⇒ raise Failure (url, exn)

   fun link url =
       let  val url' = resolve (parent, url) in
            case lookup url' of
               | SOME package ⇒ package
               | NONE ⇒
                 let val package = lazy eval' url' (acquire url') in
                      enter (url', package); package
                 end
       end
in
   await ∘ mutex link
end
```

Figure 7.10.: A canonical component manager

a URL relative to another one, the internal link function can then perform the necessary resolution. When evaluating a component C, link passes along C's URL to the evaluation function eval', which constructs an import function from it that is suitable for loading C's own imports.

Explicit access to the component manager is enabled in this implementation by special-casing the internal lookup function on the system URL of the component manager. If applied to that URL, lookup just returns an appropriate package containing the required functionality. Note that these functions do not interpret URLs relative to a parent – no unambiguous notion of parent exists in their case, because the functions can be passed around first-class. Instead, URLs are pragmatically resolved relative to the local host and current working directory, which we indicate with "." here.

The component table is stateful. To achieve thread safety, the manager must perform locking. We reuse the mutex abstraction from Section 6.5 for this purpose, which synchronises functions via mutual exclusion. All functions that access the table and are passed outside the closure – namely lookup, enter and link – are synchronised by applying the mutex.

Note that the link function enters the component into the table immediately, but actually suspends loading and evaluation through laziness. This is necessary to achieve re-entrancy of the manager, which is required because evaluating the component might spawn new threads and request new components *before* returning. By returning from the synchronised link function before actually initiating evaluation, the mutex lock is released first and a deadlock is avoided. To compensate, we must ensure that the lazy future is touched after the lock is released. We do that by wrapping the synchronised link function with an explicit call to await to touch it.

7.6.4. Program Execution

Given the described reduction of components and component managers, execution of an Alice ML program can be thought of as evaluation of the simple application

```
import' "." root
```

where import' refers to the initial component manager, and *root* is the URL of the program's root component, resolved relative to the current host and working directory, which we again indicate by a dot URL.

In summary, all observable properties of program execution and dynamic linking follow from the decomposition we gave:

- *Lazy Linking*: Import is fully lazy. The first access to an imported component triggers its actual loading, evaluation, and type checking.

- *Type Propagation*: Export signatures are dynamic types. Type checking fully takes dynamic type equivalences into account. Through lazy types, a type check may trigger loading of other components.

- *Failure*: When a component could not be linked, all references will eventually become failed futures (with exception Failure), automatically propagating failure notices when necessary.

- *Concurrency*: Components can be linked concurrently. A component manager has to be thread-safe and properly re-entrant.

We elaborate the second point in the following section.

7.7. Type Propagation

The most subtle point implied by our interpretation of components is that export signatures are actually determined dynamically, due to the transparent interpretation of pack (Section 4.1). This enables complex type sharing: an export signature may mention a type that has been imported abstractly, still other components further down the dependency graph may match this type concretely. Consider three components:

```
(*) A
type t = int
val x = 5

(*) B
import type t; val x : t from "A"
type u = t
val y : u × bool = (x, true)

(*) C
import val y : int × bool from "B"
val z = #1 y
```

Here, B imports type t from A, but makes no assumptions about it – it just reexports it under the name u. The export signature of B thus contains no *static* information about the identity of u. Still, component C can successfully import it under the assumption u = int, because the actual export signature of B is determined *dynamically*, and reflects the respective type equivalence.

A related subtlety is the effect of dynamic type checking on lazy linking. Type checking an import may sometimes require loading a component that is imported only transitively, i.e. is not yet accessed directly, but whose dynamic export signature needs to be known to decide type checking further down the import chain. Consider the above example again. In order to evaluate z in component C, the pair y must be accessed from B, i.e. linking of B is triggered. To construct y, the value of x imported from A is not actually needed, it can be kept as a future, linking of A is not required for evaluation. However, in order to type-check the import of B in C, we must check u = int. Therefore, we have to inspect type u, which initially is a lazy reference to type t from A (Section 6.7). The type check can only proceed by touching this type future, thereby loading A after all.

Dynamic type information does not necessarily propagate from other components only, it can also be the result of an actual computation:

```
(* A *)
val p = if isFullMoon () then pack (type t = int; val x = 5)
                         else pack (type t = bool; val x = true)
structure M = unpack p : (type t; val x : t)
val y = M.x

(* B *)
import val y : int from "A"
val z = y + 1
```

In this example, linking A and B will only succeed during full moon.

In summary, the presented semantics guarantees that link-time type checking accounts for all available dynamic type information. Of course, practical programs are unlikely to be tempted to exploit this flexibility to the extreme. Examples like the last one are rarely useful as is, but the underlying semantics may be relevant for computed components.

```
fn import ⇒
let
    val fibPackage = lazy unpack fibComponent import : sig val fib : int → int end
    fun import' "fib" = await fibPackage
      | import'  url  = import url
in
    mainComponent import'
end
```

Figure 7.11.: The component example after static linking

7.8. Static Linking

When *developing* an application, it usually is good advice to split it into small enough components, so that they can be modified and compiled independently, probably by multiple developers. When the final application is *deployed*, however, it is preferable to have it consist of as few parts as possible, to ease installation, minimise potential for failure, and prohibit unlicensed reuse of private components. It is thus important to decrease the level of granularity of components when moving from individual programming tasks to the final product.

The Alice System supports this with a simple notion of *static linking*, or *bundling*: it provides a linker tool that takes a set of components, including one designated root component, and bundles them to form a single large component. In practice, the set can be computed automatically from the root component and its dependency graph, plus URL-based cut-off rules given by the user.[8]

From the set of components a new component is computed and pickled that incorporates all of them. The resulting component has the same export as the root component, and the collective imports of all components that were *not* included, but are imported by an included one. The linker checks that all pairs of import/export signatures of internalised import edges match, otherwise the whole operation is rejected with a type error message. Note also that it is not possible to link primitive components (Section 7.5), because they are resourceful and cannot be pickled. They are always cut off implicitly.

Interestingly, the semantics of static linking can be defined in terms of custom component managers. We give a brief sketch by means of a simple example.

Consider the Fibonacci application from Figure 7.2, in its decomposed version shown in Figure 7.9. We take the main component as root and exclude any library components. Slightly simplifying, main is statically linked with fib by computing and pickling a wrapper component whose desugared appearance is roughly equivalent to the expression shown in Figure 7.11. We assume that the original component values are bound to mainComponent and fibComponent in the context, and will hence be in the closure when the wrapper is pickled.

Basically, the new component creates a trivial custom component manager that treats requests for the URL "fib" of the bundled component specially. All other component requests are just forwarded to the parent manager. This specialised manager is then passed on to the root component main, ensuring that it will see the internalized version of fib when requesting it. Note that the relative evaluation order of the linked components is not changed: fib is still evaluated lazily.

It should be sufficiently clear how this approach scales to more complex examples. The main complication is that not only the root component has to receive the specialised manager, but

[8] If the linker were to be implemented in Alice ML itself, this would require limited reflective functionality on the representation of components, to allow the inspection of the import URLs and signatures of components.

all bundled components that import other bundled components. To deal with cyclic linking correctly, this requires the import' procedure to be defined in mutual recursion with the lazy package definitions – promises (Section 6.4) provide an easy way to realise this. We omit the details.

Another complication we refrain from discussing further is URL resolution: since URLs can be relative, the custom import function defined actually has to consider the URL of the importing component to correctly identify references to bundled components, similar to the way the parent URL is employed in the idealised component manager of Figure 7.10. This is tedious but not hard.

The linker of the actual Alice System also performs an additional optimization: since import/export signatures are checked at bundling time, it is statically known that the inserted **unpack** operations will not fail. Consequently, the linker can safely remove the corresponding dynamic type checks, saving space and time with respect to component representation and evaluation. However, this transformation cannot be expressed on the language level, due to the lack of static first-class modules in Alice ML.

7.9. Related Work

We have already described central aspects of the component system presented here in previously published work [Ros06, RLT+06], where we also gave a formal semantics of packages and pickling as part of a module calculus. That work omits many of the details given here, particularly regarding type propagation and compilation.

The languages most relevant with respect to our component design are Java [GJS96] and Oz [Smo95, DKSS98], in which many of the ideas originated, and Acute [SLW+05], which is a different ML-like language that incorporates a component system with comparable flexibility. We discuss them in detail below.

Kornsteadt [Kor06] describes and discusses many additional aspects in the design of a component system like the one found in Oz and Alice ML. In particular, he is concerned with component namespacing via URLs, URL resolution, the external representation of components via pickles, and static linking (bundling) in much more detail. Moreover, he describes an alternative decomposition of components that aligns with the model used in Oz, discussed below.

The problem of software configuration and dynamic linking has been approached from a more general direction by many authors. For example, in early work on the functional language Pebble, Burstall already describes the idea of regarding modules as functions and linking as application [Bur84]. Cardelli [Car97a] was the first to investigate linking formally, but he was only concerned with separate compilation and static linking. Glew and Morrisett [GM99] consider type-safe static linking on the level of assembly language. Flatt & Felleisen [FF98] proposed a calculus of first-class *units*, which are very similar to our components, though defined as monolithic primitives. They support dynamic and static linking (the latter through a special compound construct), but no lazy linking or sandboxing. Unlike in our system, a client that links (*invokes*) a component must supply all its imports, which decreases modularity. Ancona & Zucca [AZ02, AFZ04] have developed a series of formal calculi that abstractly model components and type-safe and lazy linking, but they maintain components as a complex primitive concept. Duggan [Dug02] also gives a formal semantics for a language with dynamic linking, but is mainly concerned with the issue of recursive linking.

Dean [Dea97] investigates issues of type safety in a language with dynamic linking in the style of Java. None of these issues arise in Alice ML, because the semantics of components is defined purely by decomposition into (safe) language features like modules and packages.

There is relatively little work that investigates concrete language design in the context of ML or similar higher-order typed languages. There have been different proposals for defining separate compilation for ML [BA99, SMCH06], but they do not provide dynamic linking or any of the other advanced features provided by our approach.

Modula-3 and Oberon

Some languages in the Modula tradition, noticeably implementations of Oberon [RW92] and Modula-3 [BNOW95], already were centered around a module system with dynamic linking. The Oberon System even went a step further: it constitutes a whole operating system that consists of a library of dynamically loaded components [WG92]. However, the module and type systems of these languages are comparably simple, and neither performs dynamic type checks at link time. Hence they are inherently unsafe.

Java

Java [GJS96] has been the first major language with a serious focus on open programming, and as such quite successful and influential. Our approach to dynamic linking and sandboxing through component managers has clearly been inspired by Java's concept of *class loader* [LB98]. Java is object-oriented, so instead of modules, Java components (*class files*) carry classes. Full verification of the well-formedness of class files is carried out upon loading. However, Java performs no structural type checking when a class is loaded, subsequent method calls may cause a NoSuchMethodError exception any time, undermining the type system to the point that Java has to be considered a dynamically checked language as soon as it is actually used for open programming.

In contrast, the dynamic type check performed by Alice ML ensures that all invariants of the static type system are maintained – execution cannot fail later due to a type error. More precisely, the use of laziness may delay type errors in Alice ML, but the package semantics prevents evaluating a call into a component successfully before *all* assumptions on a given import edge have been checked. In Java, any number of inter-class calls may execute successfully (and potentially initiate side effects) before a subsequent call uncovers an actual inconsistency in the interface assumptions. See our introductory example in Section 1.3.1.

Runtime checks are performed at individual method calls. Java uses a simple form of nominal typing for that: every class is identified by a pair of its syntactic name and the responsible class loader. Clearly, this approach is more vulnerable to soundness problems, and indeed, early versions of the Java Virtual Machine were unsound in this regard [DFW96]. The issue is *namespace consistency* between runtime classes introduced by multiple class loaders, and the classes seen by the compiler [LB98]. Ensuring type safety and integrity of the runtime relies on a one-to-one mapping between classes as types and classes as components, and thus requires certain properties from loaders. These cannot be enforced for user-defined class loaders. Consequently, the Java runtime not only has to ensure that no class loader is ever invoked twice for the same class, it also has to double-check the name of classes delivered by a loader.

None of these problems exist in Alice ML, where type identity is established by generativity, and type safety thus is completely independent from the identity of components. Consequently, component managers are safe by construction, since they can be expressed solely in terms of proper language constructs, which are known to be type-safe. Even nonsensical managers, e.g. ones that deliver a different component each time the same URL is requested, cannot compromise soundness.

Java has a built-in mechanism for serialization that even supports versioning. However, it is not interchangeable with class files and not generic. In particular, code cannot be serialised, only

class names can be used to represent code in serialised objects, which is a comparatively inflexible and fragile abstraction. Consequently, there is no equivalent to computed components. Instead, if required, classes have to be exchanged manually as class files, separate from objects. In the case of distributed programming, the Java RMI (*remote method invocation*) framework [WRW96] encapsulates transmission of class files in the library. Classes can be loaded as first-class entities, but as such they can only be accessed indirectly per Java's reflection mechanism, which precludes any form of static type checking.

Java was originally intended for writing applets running in Web browsers. For that purpose, Java provides sandboxing as an integral part of its runtime infrastructure. Custom class loaders allow to control access to certain classes. Every class loader is accompanied by a *security manager* that checks access on a per-operation basis. In more recent versions, the sandbox model has been refined with a system of cryptographically verified trust and more fine-grained, user-defined permissions and policies. The sandboxing infrastructure needs to be built into the core system in Java, in contrast to our approach, where it can be realised fully within the language.

Java Archives (JARs) are bundled collections of Java class files. Unlike static linking in Alice ML, creation of Java archives does not perform type checks, and intra-archive linking can still fail at runtime.

Scala

Scala [Ode05] is a hybrid object-oriented/functional language on top of the Java infrastructure. It has a very powerful static type and class system [OCRZ03] that subsumes Java as well as much of ML modules, and additionally provides features like multiple inheritance, mixins, and views.

Still, open programming in Scala suffers from the same shortcomings as in Java, due to its approach of reusing most of Java's runtime and library. It hence inherits most of Java's problems regarding open programming. In particular, the expressiveness of the static type system does not carry over to dynamic typing, because Scala types are *erased* to Java types during compilation.

".NET"

The Microsoft ".NET" Common Language Runtime [Mic03] is a framework that is very similar to the Java Virtual Machine and library in most aspects related to open programming, but is meant to support multiple languages. Unlike Java it allows components (*assemblies*) to be created dynamically via its reflection mechanism, but like Java class files, assemblies can only contain static data of very limited form.

Oz

The overall design of the component system for Alice ML was directly inspired by Oz [Smo95, VH04], which also has a first-class component system with lazy linking, computed components, managers, and a definition by decomposition [DKSS98, Kor06]. Dynamic linking in Oz in turn was partially inspired by Java's class loading mechanism.

Oz components are called *functors*. Like our components, they depend on a number of imports identified by URLs, and provide a record of values as export. Linking is performed by a *module manager*, which directly corresponds to our component manager. Functors are represented as pickles externally, although these are not interchangeable with plain pickles like in our design (all functor files are pickles, but not vice versa). As an extension to the system presented here, Oz provides syntactic sugar for creating computed components directly during compilation.

With Oz being an untyped language, no dynamic type-checking is performed for imports. The decomposition of Oz components thus differs from the one in our approach (Section 7.6): instead of being abstracted over an indirect linking function, Oz functors are directly represented as functions over the tuple of import modules. This approach is inapplicable in a typed setting, because there would be no fixed representation type for components: import signatures are dependently typed, which precludes passing imports as homogeneous variable-length vectors – thus component functions would actually have to vary in arity. Linking in the Oz model requires reflective capabilities that cannot be typed either. Moreover, the model cannot capture the actual type-checking semantics, especially the fact that export signatures are dynamically computed by evaluating the component. Consequently, the interplay between dynamic type-checking and lazy types could not be reflected properly either.

Another difference between the two approaches is that Oz makes the *linking policy* [Kor06] (e.g. use of laziness) the responsibility of the manager, while our approach could allow every component to select it on a per-import basis. This has the arguable advantage that it renders the semantics of imports more transparent and the manager itself becomes somewhat simpler. Another advantage we see in our approach is that imports are clearly modelled as implementation details of a component, which mirrors the language-level situation more accurately.[9]

Acute

Acute [SLW+05] is an experimental, ML-like language for typed open programming, probably closest in spirit to Alice ML. It addresses many issues of open programming, including type-safe pickling (marshalling), abstraction safety, dynamic linking, and even versioning [Sew01, BHS+03].

Although Acute has no explicit notion of component, modules can be lazily imported from files or URLs very much like in Alice ML. The module system itself is simple: only structures are supported, functors are not available.

Instead of providing first-class or computed components, pickling in Acute incorporates a different mechanism for controlling rebinding: individual modules can be declared as "imported", which means that they are potentially rebound on the target site when the pickle is loaded. The precise extent of rebinding is controlled by decorating a program with symbolic marks. A pickling operation can then initiate rebinding up to a particular mark. This approach crucially requires programs to be viewed as linear sequences of module definitions. To control rebinding properly, a programmer might have to construct a program such that a suitable global ordering on all its modules and marks is imposed. Hence the approach appears to be inherently unmodular; it is not clear how the ordering could be maintained in the presence independently developed libraries, which might even have conflicting requirements on imports shared in a diamond-graph manner.

In contrast, Alice ML's computed components allow rebinding to be controlled locally and much more fine-grained. On the other hand, they do not enable rebinding in pre-computed parts of the component – i.e. abstraction after the fact. That is possible with Acute's mark mechanism, although even marks cover only a limited class of cases. For example, there seems to be no way to control rebinding of modules that do not appear in the static program text, but were implicitly received by a preceding unmarshalling operation.

Acute takes a more monolithic approach to language design: its mechanisms are all built-in as comparatively complex language constructs, with no obvious reduction into simpler concepts

[9] Only static linking requires knowledge about the import list, at least if computation of the dependency graph should be automatic.

like in our approach. For example, there is no equivalent to component managers, the complete linking process is primitive in the language semantics.

A severe consequence is that linking strategies cannot be customised by the programmer. Due to that – and Acute's uncontrolled implicit rebinding – it is not clear how sandboxing or other security mechanism could be programmed. Static linking can be simulated to a limited extend using Acute's inclusion mechanism for compiled code.

Acute provides a number of interesting features not available in Alice ML. Most notably, module definitions and imports can be decorated with versioning constraints, which are checked during linking. Moreover, Acute gives means to redefine the implementation of existing modules, by explicitly breaking type abstractions in a type-safe manner (i.e. without changing the actual representation of abstract types). The latter violates abstraction safety, but Sewell et al. argue that it is necessary to support software evolution in practice [SLW+05].

7.10. Summary

- Alice ML recasts the component system from Oz in a typed context.
- Components complement ML's static module system with type-safe, lazy dynamic linking.
- Alice ML is unique in that components are strongly typed by dynamic higher-order module signatures.
- Through structural signature subtyping, link-time type checking is robust against interface changes.
- Every source file is interpreted as a component definition with an implicit signature.
- Components can also be computed dynamically, and thus allow processes to exchange behaviour that encapsulates dynamic information *and* depends on local resources.
- Customisable component managers give control over imports and enable security policies for resources.
- Static bundling allows formation of large components from smaller ones.
- The component system can be explained solely in terms of modules, packages and futures. The model differs from previous work in that export signatures are computed from concrete dynamic type information.
- The essentials of the component system are not language-specific and can in principle be adapted to any language with a structural module system, because they only require packages and a pickling mechanism.

8. Distribution

Networks are ubiquitous in today's computing world. Fast local networks connect individual machines and enable them to share resources, either in hardware or software. Slower global networks – particularly the Internet – connect computers and local networks all over the world and enable them to communicate and exchange various kinds of data.

Many applications today are distributed over multiple communicating and cooperating machines:

- Classical *client/server* applications, where multiple client programs benefit from services provided by a (single) server program. The most dominant example of this architecture today is the World Wide Web.

- *Peer-to-peer* applications, where many programs connect to each other and form a dynamic network for storing, locating and exchanging information. File sharing software is a popular example.

- *Cluster and grid computing*, where a large number of machines shares computational resources to solve costly tasks in parallel. For example, this concept is found in render farms creating computer-generated imagery (CGI), or search engines indexing large sets of data [DG04].

- *Mobile agents*, where autonomous programs migrate between different sites to gather information, manage a network, deploy software, or perform other tasks. Ironically, the most wide-spread incarnation of this principle today are computer viruses.

A lot can be said about distributed programming, its applications, techniques, and the problems involved. However, within the scope of this thesis, we are mainly concerned with the problem of establishing type safety across process boundaries. We deliberately stay ignorant about other delicate aspects of distributed programming – like network latency, failure recovery, etc. Excellent work exists in literature that covers these issues, one good starting point is the thesis of Armstrong [Arm03]. The rest of this chapter restricts itself to describing the basic language primitives that exist in Alice ML to support distributed programming, along with a few simple examples that illuminate questions of typing.

A distributed system consists of a number of interacting *processes*. Every process executes a *program*. The processes run on potentially different *sites* in a network. We will use the terms *process* and *site* almost interchangeably, to emphasise that processes are usually located on physically different locations in a network.

The goal of having language support for distribution is to allow processes to be expressed as high-level Alice ML programs. The design we present offers the following desirable properties:

- **Network transparency**. A process can obtain references to values in another process (*remote values*), which are handled in (almost) the same way as local values. Hence the same abstraction mechanisms and idioms can be applied for local and remote operations and communication.

- **Network awareness**. Although *use* of remote values is transparent, *creation* is not. A program has explicit control over the introduction of – potentially expensive or insecure – inter-process references to local entities.
- **Synchronicity and Asynchronicity**. All communication is synchronous by default. However, by employing futures synchronous communication can easily be turned asynchronous.
- **Type safety**. Use of remote values is statically type-safe. Only when establishing a connection to another process dynamic type checks need to be employed. All properties of the static type system are maintained.
- **Resource security**. A connection to another process cannot leak security-relevant local resources to another site. Code imported on a site can only gain access to resources if explicitly granted.

The principal characteristic of distributed programming is that processes cooperate via *inter-process communication* over logical *connections*. We roughly distinguish two phases of activity for connections:

1. *Establishing* a connection.
2. *Communicating* over that connection.

A particular goal of language support for distribution is that we want to be able to express inter-process communication without having to drop below the language level – which still is the predominant approach in practice, where inter-process communication is often performed in terms of HTTP protocols, or even directly on the socket level.

It turns out that distributed programming can be based on only a few high-level primitives that suffice to hide all the embarrassing details of low-level connections:

1. **Proxies**, mobile wrappers for stationary functions that transparently perform remote procedure calls when applied.
2. **Tickets**, dynamically generated URLs that enable a process to retrieve a module from another process.
3. **Remote execution**, which allows a process to spawn new processes on remote machines.

Proxies are the abstraction for a connection in Alice ML, while the other two mechanisms provide convenient ways to establish such connections, addressing two different scenarios of distributed programming.

8.1. Proxies

A variety of different primitives has been proposed in literature to realise communication between processes. In a functional language like ML, function invocation is the fundamental way for communicating *within* a process. It is natural to use the same mechanism for communication *between* processes. Alice ML hence adopts *remote procedure calls* (RPCs) [BN84] – or rather *remote function application* – as the most obvious and minimal choice for generalising ML to a distributed language. That is, a thread in an Alice ML process can call a function that actually resides in another process. Alternative communication mechanisms can be programmed as abstractions on top of RPCs.

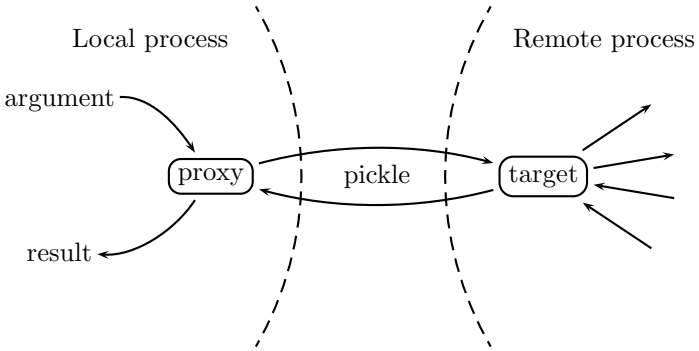


Figure 8.1.: Inter-process communication with proxies

Being expressed by function calls – in a higher-order language – inter-process communication naturally inherits a number of useful properties:

- Connections are first-class and mobile.
- Communication is statically typed.
- Communication is two-way and synchronous.
- Multiple communications between two processes can be performed concurrently.

To perform RPCs, Alice ML employs the notion of a *proxy function*. A proxy is a mobile wrapper for a stationary function: it can be pickled and transferred to other processes independent of the wrapped function. When applied, the call to the proxy is automatically forwarded to the site where the wrapped function resides – we speak of the latter as *target site* and *target function*. Both argument and result of the call are transferred between proxy and target by means of pickling (Chapter 5). Figure 8.1 illustrates the communication underlying a remote procedure call over a proxy.

Only one primitive function is needed to make proxies available in the language:

```
val proxy : (α → β) → (α → β)
```

Applied to a target function, it creates a respective proxy. For instance, the declaration

```
val f = proxy (fn s ⇒ print (s ^ "\n"))
```

defines a simple proxy function that can be called to print a message on the target site. (Note that the proxy primitive takes advantage of SML's preference for Cartesian functions, cf. the respective discussion about the mutex abstraction in Section 6.5.)

Proxies are *transparent*, in the sense that invocation behaves almost identical to a direct invocation of the target function where both are accessible. That is, an RPC is invoked using ordinary application syntax (as the type of the proxy function suggests):

```
f "Hello world!"
```

The only immediate difference between a proxy call and a direct call is the use of pickling, implying that a deep touch (Section 6.1.1) is performed on its argument and result, and that no resources can be passed to and from a proxy (to maximise network transparency, pickling is used even if the target resides in the same process as the caller). Naturally, pickling and network communication also introduce additional potential for failure, as we will discuss below.

A proxy is *mobile*: it can be pickled and transferred to other processes, where it can still be transparently applied without the local process being required to have explicit knowledge

of the originating site – this knowledge is encapsulated in the proxy. The important feature here is that, when pickling a proxy, the target function is *not* pickled along with it – contrast this to pickling of ordinary functions that have other functions in their closure (Section 5.1). In particular, the target function can be sited (Section 5.3) without compromising mobility of the proxy. This is the only deviation that Alice ML makes from the closedness principle for pickles (Section 5.1) – it is crucial however to allow capturing connections to processes in pickles (because obviously, processes cannot be included in a pickle).

A connection to another process now simply consists of a proxy that has been created in that process. Receiving a proxy can be understood as establishing a connection. Communication is engaged by calling a proxy. For instance, if we pickle the above proxy function to a file,

```
Pickle.save ("/home/rossberg/hello", pack (val it = f) : (val it : string → unit))
```

and load the pickle in a second process (that has access to the same file system) then we can send a greeting to the first process:

```
structure H = unpack Pickle.load "/home/rossberg/hello" : (val it : string → unit)
do H.it "Hello world!"
```

Note that we could not have pickled the target function itself, because print is a sited operation!

Proxies are *typed* connections. A proxy invocation is statically type-safe because proxies can only be received through pickles, where type consistency is always ensured. In other words, once the connection is established – by successfully receiving the proxy – no further type checks are required. However, the runtime system may need to perform verification (Section 5.2) of arguments or results received for proxy calls, if it does not trust the other process (Section 8.4).

All invocations of a proxy are synchronous: the call to H.it in the above example does not return before the message has been printed in the remote process. However, asynchronous calls can easily be achieved by employing futures (Section 6.1.2):

```
spawn H.it "Hello world!"
```

Now we have turned the application into an asynchronous remote call – it immediately evaluates to a future. In other words, futures give us the ability to use just one uniform RPC semantics while still providing synchronous *and* asynchronous calls, orthogonally. More interestingly, unlike asynchronous calls in most other languages, futures in Alice ML enable even asynchronous calls to return results. This feature allows to conveniently formulate bi-directional query/reply communication while still hiding network latency as much as possible.

The function proxy is provided in the structure Remote of the Alice library, whose signature is shown in Figure 8.2. Besides other functions explained in the following sections, it also contains a higher-order polymorphic functor Proxy, which conveniently allows wrapping all functions in a given structure (or functor) into proxies in one go. For example, we can give another Alice ML process access to local I/O by wrapping the whole TextIO structure and pickling the result:

```
Pickle.save ("myio", pack Proxy (signature S=TEXT_IO; structure X=TextIO) : TEXT_IO)
```

The Proxy functor uses the dynamic signature information to operate in a type-directed manner. Consequently, it is able to handle even curried functions correctly, by recursively wrapping each partial application in its own proxy. For example,

```
Proxy (signature S = sig val f : string → string → string end
       structure X = struct fun f s1 = (print s1; fn s2 ⇒ (print s2; s1^s2)) end)
```

returns a structure containing a wrapper for f as follows:

```
proxy (fn s1 ⇒ proxy (f s1))
```

```
structure Remote :
sig
    exception Proxy of exn
    exception SitedArgument
    exception SitedResult
    exception Protocol of string

    val proxy : (α → β) → (α → β)
    val offer :  package → url
    val take :  url → package
    val run :   string × component → package
    functor Proxy (signature S; structure X : S) : S
end
```

Figure 8.2.: The Remote structure

This function behaves equivalently to the following direct formulation:

```
proxy (fn s1 ⇒ (print s1; proxy (fn s2 ⇒ (print s2; s1^s2))))
```

As this example demonstrates, repeated wrapping generally is the only correct behaviour for curried functions, because every partial application might produce a sited closure. Curried *functors* are treated in a similar manner.

8.1.1. Example: Remote References

As a simple demonstration of an abstraction that can be expressed in terms of proxies, consider the problem of distributed state. If references are sited – which we assumed in Section 5.3.1 – then they cannot be shared among processes. There is no direct way to communicate a local reference to or access it from another processes. However, proxies make it possible to implement *remote references* as a simple abstract type with the following signature:

```
signature REMOTE_REF =
sig
    type α rref
    val rref : α → α rref
    val ! : α rref → α
    val := : α rref × α → unit
end
```

Since this signature is practically identical to that of ordinary references, remote references can be used as a substitute where desired. Unlike ordinary references however, they can be passed to other processes in a first-class manner (without copying them). The implementation is simple:

```
structure RRef :> REMOTE_REF =
struct
    type α rref = {put : α → unit, get : unit → α}

    fun rref x =
        let
            val r = ref x
        in
            {put = proxy (fn x ⇒ r := x),
```

```
                get = proxy (fn () ⇒ !r)}
        end
    fun !{put,get} = get ()
    fun {put,get}:=x = put x
end
```

A remote reference is represented by a pair of two proxies for read and write access. The target functions of these proxies share an ordinary reference in their closure. When a remote reference is pickled and passed to another process only the proxies are included, the state itself is kept in the original process. Similar abstraction can be built for all kinds of sited entities.

8.1.2. Proxy Failure

A proxy call can fail for a number of reasons:

- The argument value is sited and cannot be transferred.
- The result value (or exception) is sited and cannot be transferred.
- Verification fails on either end.
- The target process is dead or otherwise unreachable.
- Lower-level communication errors or network failures occur.

In all these cases, the call raises the exception Proxy(e), to distinguish the failure from exceptions raised by the target function. In the first three cases, the nested exception e will be SitedArgument, SitedResult, Protocol to indicate the cause, respectively. The latter cases may be indicated with e being another exception from the SML library, for example IO.Io upon errors in the underlying I/O layer.

Consider the following simple examples:

```
val g = proxy (fn () ⇒ print)
val h = proxy (fn f ⇒ f)
do g () "Hello"                 (*) raises Proxy(SitedResult)
do h print "Hello"              (*) raises Proxy(SitedArgument)
do h (proxy print) "Hello"
```

The first two calls will fail, because print is sited. The third call is OK, however, because a proxy is never sited. It will print Hello.

It is not considered a failure if the target function itself terminates with an exception. Instead, the exception will simply be tunnelled back to the caller. For instance,

```
val k = proxy (fn () ⇒ raise Domain)
do k ()
```

will simply raise Domain. If the exception is sited however (because it carries a sited argument), the caller will see the exception Proxy(SitedResult) instead.

8.2. Tickets

We saw that in order to establish a connection, a proxy has to be transferred between two processes. The only way this can be achieved is by means of pickling (Chapter 5). In the example from the previous section we relied on persistence (Section 4.2) and shared access to

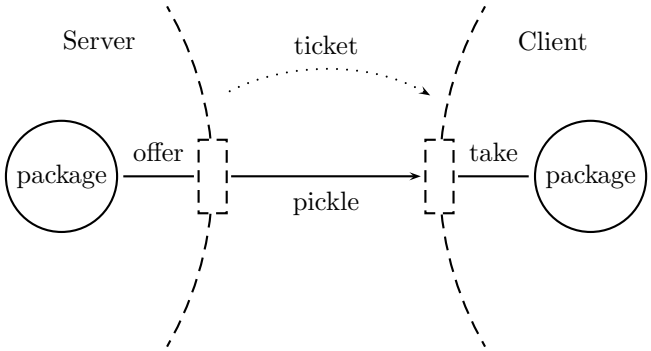


Figure 8.3.: Package exchange via ticket

a single file system to exchange the pickle. Obviously, this is a crude and inflexible mechanism that is inapplicable across network domains.

To address this, Alice ML provides a simple transfer mechanism for initiating connections, which has been adopted from Mozart [Moz04]: a process – let us call it the *server* – can make available a package (Chapter 4) for download in the network, using the library primitive

```
val offer : package → url
```

Offering a package employs pickling in a way similar to persistence. Given a package, the offer function returns a globally unique, dynamically generated URL, known as a *ticket*. The ticket identifies the server and the individual package in the network. Of course, a process can offer arbitrarily many packages, generating multiple different tickets.

A ticket can be communicated to the outside world by conventional means, such as web pages, email, phone, or pigeons. Another process – the *client* – can use a ticket to retrieve the corresponding package from the server, using the complementary primitive:

```
val take : url → package
```

After receiving the package, it can then be opened using unpack, which dynamically checks that the package signature matches the client's expectations. As a result, the downloaded module is available to the client program. Figure 8.3 illustrates the procedure graphically. Note that an offered package can be taken multiple times.

In order to establish a permanent connection, the package must contain proxies. Once the connection is thus established, these proxies serve as permanent communication channels. Noticeably, this is the only point where a dynamic type check is necessary. From now on, static type checking suffices to ensure that all communication is well-typed.

8.2.1. Bi- and multi-directional Connections

Obviously, the proxy mechanism directly supports a client/server architecture. However, so far, only the client can call the server side, over the proxies it received. To get a symmetrical connection, it suffices to send client-side proxies back to the server as arguments to higher-order proxies that the server provided for this purpose.

More complex communication patterns can be established by passing proxies back and forth to other connected processes, for instance, to enable different clients to communicate directly with each other. In a similar vein, more general peer-to-peer applications could be implemented.

Note again that, once the initial connection has been made, all calls through proxies are statically typed. In particular, extending the connection to bi-directional or multi-directional communication does not require any further dynamic type checks.

8.2.2. Example: Chat Room

As a simple example for a client/server architecture with bi-directional communication, consider a chat program. The following is a minimalist, yet complete, implementation of such an application. It consists of a chat server, that prints a ticket when started. Using this ticket, clients can connect to the server.

Both sides need to agree on a signature for the exchanged package. Basically, it describes the server interface:

```
signature SERVER =
sig
   val register :   {send : string → unit} → unit
   val broadcast : {name : string, message : string} → unit
end
```

Clients can register with the server, after which they will receive all messages sent by other clients, and they can broadcast messages themselves.

Here is the full code for the server component:

```
val clients = ref nil
fun register client = clients := client :: !clients
fun broadcast {name, message} =
    List.app (fn {send} ⇒ spawn send (name ^ ": " ^ message)) (!clients)

structure Server = (val register = proxy (mkMutex () register)
                    val broadcast = proxy broadcast)
val ticket = offer (pack Server : SERVER)
do TextIO.print (ticket ^ "\n")
```

The server simply keeps a list of registered clients (represented by their `send` functions), and broadcasting iterates over this list and forwards the message to each. In order to avoid having to wait for each client in turn to receive the message, sending happens asynchronously, using `spawn` (Section 6.1.2). Moreover, since the client list is stateful, we have to avoid race conditions when several clients try to register at the same time. The exported `register` function is hence synchronised on a mutex lock (Section 6.5).

The code for a client is even simpler:

```
val [ticket, name] = CommandLine.arguments ()

structure Server = unpack take ticket : SERVER
do Server.register {send = proxy TextIO.print}

fun loop () = case TextIO.inputLine TextIO.stdIn of
                | NONE ⇒ OS.Process.exit OS.Process.success
                | SOME message ⇒ (Server.broadcast {name, message}; loop())
do loop ()
```

It expects a valid server ticket and a user name on the command line, registers with the server, and simply forwards everything typed by the user to the server (if registered, the user will see her own messages as an echo). Note that the call to `register` is a proxy call, passing another proxy as argument, thereby establishing the bi-directional connection.

Obviously, this implementation is very spartan: there is no notification of other clients connecting or disconnecting, nor is there any error handling. However, the basic principles are there, and enriching the implementation accordingly is largely straightforward.

With respect to multi-directional connections, the client/server interface could be extended such that the server can hand send functions from one client to another. The latter client could then send private messages to the former itself, directly, without further assistance from the server.

8.3. Remote Execution

In the client/server setting, client processes choose independently to connect to a known server process. A different scenario of distributed programming arises in applications like cluster computing. There, we usually find a *master* process (sometimes known as the *manager*) that shifts computational tasks to a number of *slaves* (or *workers*). In this scenario, it is the central master who initiates connections, by actually spawning new processes on remote machines.

To support this scenario, the Remote module of the Alice ML library features a further function:

```
val run : string × component → package
```

This function performs most of the required procedure: it connects to a remote machine by using a low-level service (such as ssh), the host name is given by the string argument. On that machine it starts a fresh Alice ML process as slave. The slave immediately connects to the master to receive the component argument, and evaluates the component. It sends back the resulting module as a package.

The transmitted component will typically be computed (Section 7.2) in the master process. By capturing proxies defined *outside* of the component, and by creating proxies *inside* it and exporting them, a two-way communication is immediately established.

8.3.1. Example: Distributed Search

Let us illustrate remote execution – and other features of Alice ML – by showing the implementation of a distributed application, namely a distributed solver for constraint programming [Apt03, Sch02]. This example is taken from [RLT+06].

In the context of constraint programming, a solver is a program that explores a *search tree* in order to find the solutions of a given constraint problem. Nodes of the tree represent choice points, leaves represent either solutions or failures (where previous choices are inconsistent). From a logical point of view, searching amounts to traversing a tree and asking the status of each leaf node.

In a distributed setting, a number of workers perform the search, such that each worker explores a different subtree. The interesting information, that is the solutions, are transmitted back to a manager. The manager also organises the search. In the following, we focus on the distribution aspect. Details about the search itself can be found in [TL04], which contains a formalisation of the underlying abstractions, performing efficient backtracking and vital optimisations such as *branch and bound*.

The interface between the workers and the manager can be represented as shown in Figure 8.4 [Sch02]:

- find is sent by an idle worker to request a new job, that is, the path of a subtree that remains to be explored.

- collect is send by a worker when it finds a solution. The message will contain the respective solution.

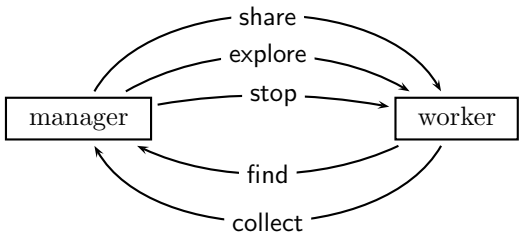


Figure 8.4.: Distributed search interface

- share is used by the manager to ask a worker whether it can give away a subtree that remains to be explored. The worker is required to answer either negatively, or positively by providing the path associated with the corresponding unexplored subtree.
- explore commands a worker to explore the subtree at a given path.
- stop is used to interrupt a worker when the search is finished.

The implementation of the distributed search engine consists of two components: the manager and the worker. The manager creates workers by using the Remote.run primitive. Each worker has the same interface:

```
signature WORKER =
sig
    val share   : unit → path option
    val explore : path → unit
    val stop    : unit → unit
end
```

Workers are components dynamically computed by the manager (Section 7.2). They capture proxies to the two functions find and collect of the manager interface in their closure. The manager hence defines these functions as follows:

```
val find = proxy (fn () ⇒ (* request new subtree from a worker and return its path *))
val collect = proxy (fn sol ⇒ (* store sol; request new subtree and explore *))
```

Since a worker is a computed component, the definition of a worker also takes place in the manager. Basically, we create a computed component that exports the three operations share, explore, and stop from the WORKER signature. Naturally, they are also defined as proxies. But since the component is evaluated on the remote site, the proxies are created there and thus represent an inverse connection:

```
val worker =
    comp
        import structure Gecode from "x-alice:/lib/Gecode"
    with
        include WORKER
    in
        val share   = proxy (fn () ⇒ (* find some unexplored subtree *))
        val explore = proxy (fn path ⇒ (* explore the given subtree *))
        val stop    = proxy (fn () ⇒ OS.Process.exit OS.Process.success)
    end
```

The library used for constraint solving, named *Gecode* [Gec05, ST05], is a native library that is sited (Section 5.3). Thus, each worker needs to acquire this library by importing it locally on the remote site.

In the implementation of explore, two special cases are interesting. If the exploration is finished, the worker asks for some more work by calling find. If a solution sol is found, it is transmitted to the manager by performing an asynchronous call:

```
spawn collect sol
```

In both cases, a remote procedure call is performed, since the corresponding functions find and collect are proxies.

In order to create multiple workers, the manager applies the run function repeatedly:

```
val hosts = (* list of host names *)
val workers = List.map (fn host ⇒ Remote.run (host, worker)) hosts
```

The workers are remembered simply as a list of packages, encapsulating the respective connections.

Now, the search starts by sending the root path of the search tree to the first worker of the list, then asking it for some work to give to other workers:

```
structure W1 = unpack List.hd workers : WORKER
do W1.explore root
do List.map (fn w ⇒ let structure W = unpack w : WORKER in W.explore (W1.share ()) end)
            (List.tl workers)
```

From now on, the manager handles concurrent requests from workers. For example, the collect message stores the given solution in a list, which must be protected using a locking mechanism (Section 6.5).

Noticeably, the list of collected solutions can be returned immediately when the search engine starts, in the form of a future. The list will then be built concurrently while solutions are sent to the manager.

As the last code snippet demonstrates, we have to perform a dynamic type check each time a worker is selected from the list. This is an unfortunate consequence of the lack of statically typed first-class modules in Alice ML – we have to abuse packages to store the modules as first-class values in a list (Section 4.5). While it would not increase expressiveness, type safety in this example obviously could profit from the addition of first-class modules to the language.

8.4. Safety

To round up our presentation of distribution in Alice ML, let us have a look on how safety and security are maintained for programs that participate in distributed programming.

8.4.1. Type Safety and Verification

We already discussed type safety in the previous sections:

- *Proxies* are typed communication channels. Hence communication itself cannot violate type safety *within* the language. Communication with untrusted partners may require verification of received data on the implementation level, though (Section 5.2).
- Modules retrieved via the *ticket* mechanism are wrapped in packages, thus enforcing an explicit dynamic type check before being accessed. The packages are received as pickles, and unpickling can perform verification if necessary.

- *Remote execution* will verify and dynamically type-check the mobile component on the remote site, and the result package transmitted back will likewise be checked on the original site.

There are no other ways of communication (except for low-level string-based communication on sockets), so we conjecture that this setup leaves no hole that could allow communication to compromise type safety. Short of a formalisation of inter-process communication, we cannot provide a formal proof for this, of course.

One important limitation is verification of pickles containing proxies. Since proxies are not closed values, verification cannot fully check their consistency. Doing so would require cooperation from the originating process – which in turn cannot necessarily be trusted. Hence verification can generally check only consistency of the local end of a proxy connection. However, whenever there is an inconsistency with the other end of that connection, it will be captured by verification of the communicated values. Unlike the dynamic type check, this check can fail late, on a connection that has already been used successfully. Unfortunately, this situation is principally unavoidable in an open setting.

The sceptical reader may ask whether the separation of dynamic type checking and verification (Section 5.2) is worthwhile in the face of this limitation. We believe that it makes a significant difference, because both kinds of check address different classes of errors:

- *Verification* is meant to protect against malice, violations should be extremely rare in practice – ideally, non-existent.
- *Dynamic type checks* protect against accident and occasional incompatibilities – they are expected to arise much more frequently, and often have to be handled programmatically.

Considering these observations, it seems desirable to detect dynamic type mismatches as early as possible – that is, earlier than malice *can* be detected. The presented approach achieves this.

If communication through proxies was not conducted in a typed manner (but instead through a port-like generalisation of the ticket mechanism, for instance), then each transmitted piece of data would have to be dynamically type-checked. According to the above argument that makes for weaker static invariants.

There also is a cost issue: for proxies in particular, verification often can be significantly cheaper than a dynamic type check. For instance, verifying that something is a valid integer has almost no cost. On the other hand, dynamic type checking would not only cost more in terms of an additional check per communicated value, it also increases the cost of pickling, transmission and verification themselves, because the dynamic type information has to be transmitted and verified as well – and it tends to be larger than the actual data in those cases.

8.4.2. Resources

Having ensured type safety (which implies memory safety), another major safety concern is unauthorised access to critical resources (Section 5.3). A malicious process could try to transfer resources to or from another process it communicates with.

Fortunately, this is prevented by two simple design choices that apply to all data received by a process:

- *Pickling.* All communication is performed in terms of pickles, hence no resources can be transmitted (Section 5.3).
- *No hidden linking.* Proxies do communicate values, not components. Hence no implicit linking is performed that could provide access to local resources behind the local program's back. Likewise, transmission using the ticket mechanism is restricted to plain packages.

This should explain why the ticket mechanism transfers plain packages instead of first-class components (Section 7.2), although the latter are in fact more general: we want take to be realisable as a secure operation (Section 5.3). Like with the load operation for persistence (Section 7.2), the restriction to packages prevents any implicit rebinding from happening (Section 7.4), and thus calling functions received over the network is always secure with respect to local resources, unless they are explicitly supplied.

Only remote execution in fact *does* transmit a component to a remote site and evaluates it there unasked, implicitly linking possible imports. However, network and operating system permissions should be sufficient to guarantee that no untrusted site is able to spawn a process on a given site in the first place.[1] Assuming proper security measures on this level, the remote site can implicitly trust the originating site and proceed linking the component. Several refinements are conceivable: for example, a site could be configured for restricted remote execution only, by running all processes invoked from remote sites in pre-defined sandboxes (Section 7.5). However, such considerations lie outside the language itself, hence we will not explore them here.

Of course, components are by no means precluded from communication. On the contrary: since they are first class, they can easily be transmitted. However, transmitted components will be explicit, and require the receiver to explicitly evaluate them under a component manager of his choice. Hence security cannot be breached by accident. The receiver can employ a sandbox component manager (Section 7.5) if he does not trust the producer of the component.

8.4.3. Other Security Concerns

It should be noted that the security model we sketched is only concerned with resources. Other considerations are beyond the scope of this work.

In particular, the proxy mechanism as described in this chapter does not meet strong security requirements: the semantics of pickling does not prevent proxies from unknowingly being imported and called by a process. It is thus possible to foist *"phone home"* functionality on a process. Such unwarranted communication may be deemed unacceptable under many circumstances. More seriously, proxies are the single form of remote reference in Alice ML, and as such face the risk of being forged. An attacker may use a forged proxy to perform a call into a process.

Currently, Alice ML does not address such concerns. At least, leakage of local information is only possible if a process explicitly makes it available for external communication in some way: an imported proxy can only communicate back information explicitly passed to it in a local call, and a remote call to a local function can only be forged if the local process ever created a proxy for it.

We leave it as future work to develop a tighter and more comprehensive security regime that enables controlling the use of proxy communication and other possible issues. With regard to proxies, one possible approach might be to make proxies themselves resources and rely on an extended transformation mechanism (Section 5.5) that allows target sites to customise internalisation of proxies.

8.5. Related Work

This chapter extends on previous work on the design of Alice ML [RLT+06]. The proxy mechanism is also described in more detail by Kornstaedt [Kor06], who focusses on the lower-level implementation underlying it. In particular, he shows that proxy creation can be reduced to pickling and two further primitives for creating and addressing call targets, the rest can be realised

[1] In practice, many security holes can occur in this setup, but they are outside the reach of the language.

within the source language. We conjecture that even those two primitives could be implemented by means of the language-level transformation mechanism for pickling abstract values that we described in Section 5.5, on top of a conventional socket library. Similar ideas have already been explored by Ohori, who developes a typed translation of high-level inter-process communication operations into low-level primitives in an ML-like language [OK93].

Oz and Mozart inspired the ticket mechanism of Alice ML for dynamic connectivity that avoids the need for a centralised server for establishing connections [HRBS98]. Unlike Alice ML however, Mozart provides true distributed state: reference cells as well as futures and logic variables can be accessed remotely, and are in fact mobile, making for a significantly more expressive semantics than most of the previously mentioned languages. Consequently, no explicit facility for remote communication is needed, channels or proxies can in fact be programmed as abstractions. The price is a significantly more complex semantics and language implementation [HRB+99]. Although Alice ML inherits most of Mozart's open programming facilities, it was a conscious design decision to forgo its rich distribution features and avoid the ensuing semantic complexity. Distribution in Alice ML can uniformly be based on its standard pickling mechanism, which is not the case for Mozart, where multiple modes of pickling with differing semantics regarding state and synchronisation are required.

Many programming languages feature some form of support for distributed programming, but mostly in the form of comparatively low-level libraries. There are much fewer languages that have been designed with distribution in mind. Even among those there is a wide spectrum of support, ranging from simple mechanisms for higher-level inter-process communication, over distributed state, to fully-fledged thread mobility.

The first language with language-level support for inter-process communication probably was CLU [LAB+79, HL82]. It had a built-in mechanism for remote procedure calls that allowed values of *transmissible* types to be passed to other processes, using a simple form of pickling. However, no runtime checks were performed, communication hence was unsafe.

Modula-3 Network Objects [BNOW95] use a similar approach, but based on objects. Since pickling is not higher-order, only objects of transmissible types, for which *stub modules* have been statically created at the client site, can be called remotely. Connections are established with the help of a separate *agent server*, where objects can be registered for remote access. The ticket mechanism we presented allows for relatively easy implementation of such a server within the language, but also enables less centralised approaches.

Java's remote method invocation (RMI) [WRW96] inherits most of the ideas from Modula-3, but can generate and transfer stubs automatically at runtime. Because Java is dynamically checked, RMI also is safe, albeit not type-safe in the conventional sense: because local instances of classes might differ, the underlying class loading may yield classes with incompatible signatures (Section 7.9).

Early examples of languages specifically designed for distributed programming include the untyped imperative language Linda [ACG86], which allowed processes to synchronise and communicate through a shared memory pool called the *tuple space*, where vectors of values could be stored and retrieved by a form of pattern matching. Unlike later approaches, including ours, inter-process communication is thereby based on a paradigm completely different from local means of data transfer, which makes it much less transparent.

Obliq [Car95] is an experimental untyped object-oriented language that allows objects to migrate over the network and performs remote method invocations transparently. Mutable objects stay stationary, but mobile proxies are created implicitly. An explicit copy operation allows stateful objects to be cloned on remote sites. Obliq probably was the first language featuring mobile code, but the implementation used a source language representation for transfer.

Erlang [Arm03] is an untyped distributed language for embedded telecommunications systems.

Processes can be spawned on different nodes in a network and communicate through channels. Although Erlang is a higher-order language, functions cannot directly be communicated. Since Erlang is used in industrial-strength, massively distributed systems, it has a particular focus on dealing with robustness and failure recovery, where it provides some unique mechanisms such as process linking, which allows failure to be detected for whole groups of processes. Erlang primarily targets embedded systems, consequently it is not concerned with security or inhomogeneous networks.

In the world of typed functional programming, Facile [TLK96] extends Standard ML with facilities for concurrency and distributed programming inspired by the π-calculus [Mil89]. It allows new processes to be initiated on remote sites, similar to our Remote.run primitive, but not from something as rich as components. Instead of remote procedure calls, remote communication takes place through typed channels. There is no direct equivalent to Alice ML's ticket mechanism. Instead, to achieve dynamic connectivity, Facile also requires taking an indirection through a cetnral *structure server*, which allows making persistent ML structures. A structure is retrieved from the server by requesting a module with a suitable signature, which naturally implies a form of dynamic signature check. If several structures match a given signature, the last one stored is returned.

JoCaml [CL99] takes a similar stake as Facile, but extending Objective Caml [Ler03] and with concurrency being based on the richer Join Calculus [FGL+96]. Inspired by Obliq, it allows processes to migrate over the network. Similar to most of the previous approaches, a central *name server* storing channel names is needed to establish connections, which is less flexible than Alice ML's ticket concept. Communication is type-safe but not secure, because resources are implicitly rebound during transfer.

Distributed Caml [WAM99] is another distributed extension of Objective Caml. It does not provide for dynamic connectivity, all processes have to be spawned from a single site, and execute the same program. More distributed dialects of ML have been proposed, including ParaML [BNSS94] and Distributed ML [CK92].

Acute [SLW+05] is another experimental ML-like language with support for open programming. It provides only simplistic support for inter-process communication. Every process has one implicit communication channel. Like in Alice ML, communication is based on pickling, but all communication is dynamically typed per transmission instead of initiating typed communication by a single type check. Acute has a primitive for *thunkifying* running threads into first-class values that can be pickled, which provides a limited form of thread mobility. However, due to the lack of distributed state, this form of mobility is rather fragile, because it can silently capture and duplicate stateful entities, especially locks, undermining the safety of concurrent abstractions. Implicit rebinding of resources increases expressiveness compared to Alice ML, but makes communication inherently insecure.

8.6. Summary

- Remote procedure calls are the main means of inter-process communication in Alice ML.
- All data transfer is reduced to pickling.
- Proxies transparently perform remote procedure calls; they are type-safe first-class functions that represent live connections for communication.
- Connections can be provided by offering a module containing proxies on the network.
- Connections are established by retrieving such a module.

- Connections can be opened by offering a module that is then referred to by a URL called a ticket.
- Alternatively, a process can itself spawn new processes remotely.
- Only establishing connections requires flexible type checking; communication itself only needs verification, which may be cheaper.
- Distribution is secure with respect to resources.

9. Implementation and Outlook

Alice ML, as presented in the preceding chapters, has been implemented as part of the Alice Programming System [Ali03] to allow evaluation of and experimentation with its concepts under realistic conditions. The Alice System is freely available as open source software. Pre-built packages are available for major platforms.

9.1. Architecture of the Alice System

The Alice implementation consists of five major parts:

- **SEAM**. The *Simple Extensible Abstract Machine* [BK02, BK03, Sea04] is a portable infrastructure for building virtual machines. It implements generic services like memory management, thread management, pickling [TKS06], etc.

- **Alice VM**. A virtual machine is needed to execute Alice ML programs. The VM is constructed on top of the SEAM infrastructure. It defines the Alice *abstract code* format and implements several interpreters for it that execute this code inside the VM. Noticeably, two of these interpreters employ just-in-time compilation of the abstract code into more efficient formats to speed up execution (either native machine code, or efficient byte code [Mül06]).

- **Compiler**. The compiler is bootstrapped within Alice ML itself and supports the full language as described in this thesis. The compiler can be accessed as a batch tool, but also online from within Alice ML programs, as part of the Alice library, to compile source code dynamically.

- **Library**. The library consists of the obligatory parts of the Standard ML Basis library [GR04], as well as modules that support specific features of Alice ML, e.g. component managers and distribution [Kor06]. The runtime representation of types is also part of the library. Moreover, the library contains rich bindings for programming graphical user interfaces, database access, and other services.

- **Tools**. The Alice System comes with several software development tools, including compiler, static linker (Section 7.8), an interactive top-level with graphical user interface, and graphical tools.

SEAM and the Alice VM have been implemented in C++, while the rest of the system is almost entirely bootstrapped in Alice ML.

9.2. Other Language Extensions in Alice ML

Besides the fundamental features presented in the preceeding chapters, Alice ML incorporates a number of other minor extensions to Standard ML:

- **Structural Datatypes**. Unlike SML, Alice ML does not treat datatype declarations generatively. That is, given two datatypes with identical definitions, they will be compatible. This eases distributed use of datatypes, because communicating processes are not forced to share a common definition a priori.

- **Extensible Types**. Alice ML generalises the extensible type concept underlying the exception type in SML: in Alice ML, the user can define her own extensible types, plus constructors thereof [Mac93]. Unlike exceptions, such types can be parameterised and hence have polymorphic constructors. Constructor declarations for extensible types are generative. They hence provide an elegant and high-level means to dynamically generate globally unique names in Alice ML, including the ability to have name-dependent types (via constructor arguments).

- **Syntactic Sugar**. Alice ML also provides a variety of minor syntactic enhancements over SML, including an extended pattern matching language, additional record features, finalisation, assertions, and wildcards in type annotations.

Detailed descriptions of these features can be found in the Alice manual [Ali03].

9.3. Limitations

The Alice System implements most of the Alice ML language as described in this thesis. However, the current version still has limitations in a number of areas:

- **Pickle Verification**. The most severe omission of the current Alice implementation is the lack of verification for pickles, as described in Section 5.2. Verification for higher-order pickles is an interesting and non-trivial problem. It requires a typed code format as well as the ability to check the type of heap data structures at runtime (because a pickle basically is an extract from the heap). Addressing these issues has been consciously left for future work.

- **Security**. The implementation has not been tailored to security. For example, the Alice System is equipped with a foreign function interface (FFI) that allows linking of so-called *native components* implemented in C++. Native components are employed for bindings to external libraries as well as most primitive libraries (e.g. TextIO), which ultimately implement resourceful operations. Currently, the runtime system does not restrict the import of native components, such that they represent a potential hole in the security structure. Other holes may exist in the architecture, parituclarly with respect to proxies.

- **Distributed Garbage Collection**. Since distribution in Alice ML mainly relies on pickling (i.e. copying) and does not support distributed state, there is no need for full-scale distributed garbage collection. However, proxies (Section 8.1) represent a form of inter-process reference in Alice ML. Currently, a function for which a proxy has been constructed can never be collected, thus potentially creating a space leak. A form of distributed garbage collection would be necessary to address this shortcoming.

- **Native Threading**. SEAM implements threads purely in software, using its own scheduling mechanism. It does not yet enable employment of system threads. Consequently, an Alice ML program cannot yet take advantage of multi-processor machines and multi-core processors.

We hope to be able to address some of these limitations in the future.

9.4. Future Work

9.4.1. Possible Extensions

Alice ML already is a comparably rich language. However, practice occasionally raises the desire for extending it with further features that would integrate naturally:

- **Dynamic Import URLs**. The language as presented requires all URLs given with import declarations to be constant strings. It is natural to ask – especially in the presence of computed component (Section 7.2) – whether it should be possible to compute these URLs dynamically.
- **First-class modules**. The parametricity restriction on the core language (Section 4.5) and the distributed search example from Section 8.3.1 showed that, besides packages, it is desirable to also have statically typed first-class modules in the language.
- **Applicative Functors and Components**. The higher-order functor example in Section 3.1 exhibited that higher-order modules are of limited use without applicative functor signatures. More seriously, fully generative type abstraction can be a hurdle for distributed programming and persistence, because it always forces pickling the abstractions themselves (Section 4.6). Compile-time generative types would be more flexible here. However, care has to be taken to prevent unsoundness from mixing runtime and compile-time generative types in invalid ways (in particular, a compile-time type may not refer to imported types that are not known to be fully compile-time themselves).
- **Reflection**. The dynamic typing facility provided by packages could be driven further towards richer reflection primitives, which, for example, would allow selecting a structure member whose label is computed at runtime. Such operations are supported by many dynamically checked languages, including Java. Even constructing structures at runtime is a possibility. For example, component frameworks like JavaBeans [Sun97] heavily rely on reflection. It is unclear however, to what extent such features would be desirable in a language like Alice ML.

Most of these features are reasonably well-understood in literature and we believe that they could be integrated into Alice ML without fundamental problems.

9.4.2. Language Specification

So far, we have not produced a *formal specification* of the full Alice ML language, as described in the previous chapters. The remaining parts of this thesis develop a formal semantics for an idealised subset of the language. However, formalising the full language probably is a daunting task. In particular, it is not possible to simply extend the SML language definition [MTHM97] with a specification of the new language features we added. There are a number of difficulties:

- *Concurrency.* The SML definition defines the operational semantics in a big-step style. The concurrent nature of Alice ML would require replacing this by a small-step semantics.
- *Futures.* Making precise the synchronisation behaviour of non-trivial operations – e.g. nested pattern matching, deep strict operations like polymorphic equality, and particularly the details of touching type futures during dynamic type checks – are tedious to describe precisely. In the case of dynamic types a complete specification might in fact unwantedly over-constrain implementations and their choice of algorithm for testing subtyping and type equivalence (Section 6.6).

- *Closures.* An operation like pickling traverses the whole transitive closure of a value. In the presence of futures or resources it is observable what is included in this closure. The extent of closures – particularly of functions or modules – must hence be fully defined. In particular, such a definition may enforce – or prohibit – certain program transformations and optimisations by the compiler (consider hoisting of structure projections as described in Section 6.6).

- *Principal signatures.* Since the compiler is required to derive the export signature of compilation units (Section 7.1), the language definition must fix the details of this process. In other words, it must require derivation of principal types and signatures for compilation units.

- *Undecidability.* Type checking of Alice ML's higher-order modules is undecidable (Section 3.3.2). In order to guarantee portability, it may be necessary to fix details of the type checking algorithm in the language definition, to ensure that non-termination (or, equivalently, abortion of compilation) is encountered for the same set of programs with every implementation. Dynamic type checking is affected by this problem too, so fixed behaviour is in fact a prerequisite for a fully specified operational semantics. Alternatively, it may be worth reconsidering the decision to include abstract signatures – given first-class modules, some or most of its current uses in the Alice ML library may be dispensible.

- *Library.* Much of the functionality of Alice ML is available not as syntax, but in the form of library primitives (e.g. the operations on first-class components and component managers). A realistic language specification hence would have to include significant portions of the library. For the existing library, details about sitedness and strictness with respect to futures would have to be added for all operations.

The above aspects only cover the *internal* semantics of the language. If the specification is meant to be truly *open*, i.e. make different implementations interoperable dynamically, then it must also cover the *external* semantics, where even harder problems arise (this certainly is not an exhaustive list):

- *Pickling.* An open definition must specify a typed pickle format. This would in fact include a complete specification of the external code representation, plus a verification algorithm.

- *Distribution.* Likewise, the specification would have to fix a protocol for inter-process communication and distributed collection of proxies.

- *Generativity.* To avoid name clashes on global scope, the language specification must determine the format and generation algorithm for type and constructor names.

Despite these problems, we believe that it would be highly interesting – and doable! – to embark on such an endeavour and work out a language specification for Alice ML in future work. At least an internal semantics covering most of the above issues seems a realistic goal for scientific work. A comprehensive external semantics appears to be beyond a scientific project, and would only be practical as an industrial standardisation effort. Still, certain aspects might prove to be interesting from a scientific point of view, and investigating an idealised sublanguage certainly would be manageable goal.

9.4.3. Implementation

- **Pickle verification.** Our language design, and the theory we develop in the second part, assume that pickles are verified upon unpickling. As mentioned above, this has

not been implemented yet, and is not a trivial problem. It has to be investigated how much dynamic type annotations are required to enable this (e.g. for closures, generative constructors, abstract data types).

- **Security.** Although the design of Alice ML does consider security to some extend, no end-to-end security regime has been put in place. Some aspects of the design, particularly proxies, may be too lax regarding serious security requirements. Numerous approaches to security are described in literature, and it remains open which are best-suited for the language and implementation, and how they can be integrated.

- **Efficient type representation.** The implementation of packages requires runtime operations on complex type structures (ML module signatures). The current implementation of Alice ML uses a comparatively naive representation. It might be interesting to investigate techniques like incremental minimal construction to improve space and time behaviour of runtime types. This will be complicated by the requirement to make runtime types thread-safe. Some preliminary work has been done by Paltzer in his Bachelor's Thesis [Pal05].

- **Case studies.** The Alice ML design has been evaluated on a handful of relatively simple applications, e.g. a distributed search engine, a distributed multi-user game, and a small chat program. More complex case studies would be necessary to get confident that the language and its implementation really scale.

Part II.

Theory

10. A Calculus for Components

In the first part of this work we presented Alice ML, a programming language designed for typed open programming. We motivated its design and discussed examples of its use in practice, backing up our thesis (Chapter 1) from the practical perspective. However, we described the Alice ML semantics only informally.

In order to approach and substantiate our thesis from the theoretical side, the second part of this work will develop a formal semantics. Clearly, a semantics for the full language is far out of scope for a work like this, so we will restrict ourselves to core elements of the Alice ML semantics that we isolate and model in a formal calculus, $\lambda^{\omega}_{\mathrm{SA}\Psi}$, for which we prove relevant properties. Since our main interest is in typing issues, this calculus will largely focus on the semantics of types.

The type system of Alice ML is quite intricate, statically *and* dynamically. While the static aspects are well-known from ML and investigated extensively in literature on polymorphic type systems [Pie02] and modules [HMM90, MT94, HL94, Ler95, Rus98, DCH03], Alice ML also exhibits a number of dynamic aspects not present in conventional languages:

- **Dynamic type matching**. Packages require arbitrarily complex types (signatures) to be compared at runtime.
- **Dynamic type checking**. Pickling requires arbitrary pieces of code to be type-checked at runtime.
- **Dynamic type sharing**. Type matching has to check equivalences between statically undetermined types at runtime.
- **Dynamic type generativity**. Maintaining abstraction safety in the presence of dynamic type sharing requires a notion of runtime type generation.
- **Dynamic sealing**. Types may not just be generated individually, but in fact by sealing of entire (higher-order) modules.
- **Lazy types**. Module-level laziness, particularly with linking, induces the notion of lazy type variables that are triggered by runtime operations on types.

Our calculus will be an idealised functional language that provides all these features, except for the last (see below). We believe that $\lambda^{\omega}_{\mathrm{SA}\Psi}$ provides a uniform and simplest possible account for the mechanisms that form the essence of Alice ML's dynamic type system.

On the other hand, although it is intended to model the essentials of Alice ML, we will not provide a formal translation from the language into $\lambda^{\omega}_{\mathrm{SA}\Psi}$– we restrict ourselves to giving an intuitive understanding of the relation between the two, based on examples. As clarified already, a formalisation of the full Alice ML language – either directly or by translation – is beyond the scope of this work.

In the remainder of this chapter we will introduce the main concepts of $\lambda^{\omega}_{\mathrm{SA}\Psi}$. We will do so incrementally, starting with a standard polymorphic λ-calculus and enriching it step by step with constructs modelling individual aspects of the language. These are the following:

- **Higher-order polymorphic λ-calculus** to represent the core language.
- **Existential types** and **product kinds** to express modules.
- **Singleton kinds** and **subtyping** to express translucent signatures.
- **Dynamic type analysis** to model dynamic type matching.
- **Type generativity** with **coercions** for recovering abstraction safety.
- **Higher-order coercions** and **generativity** to explain sealing.
- **Pickling operations** to capture the semantics of pickles.

We will motivate each feature in turn, relating it to simple examples from Alice ML. More detailed explanations of the calculus and a precise semantics will be given in later chapters, where we deal with its individual features and the meta-theory in depth.

The main technical contribution of our work is a semantics for sealing as higher-order dynamic type generativity, in the presence of singleton kinds. As it turns out, the combination of these two features is surprisingly complicated, mainly due to the presence of dependent kinds that is enforced by singletons (Section 10.3). The dependency of kinds on types requires not only the introduction of (higher-order) coercions on the term level for mitigating between generated types and their representations, but also demands for similar coercions on the type level. The work presented in this thesis thus significantly extends on our earlier work on generativity [Ros03a], which only considered plain polymorphic λ-calculus, and it gives an alternative view on packages and pickling, which we first formally described in the context of a module calculus [Ros06], but without considering type abstraction.

The complete calculus we devise is not simple. However, it is relatively canonical, and we believe that it cannot be simplified further without depriving it of essential expressiveness. The interesting aspects of the type system of Alice ML are in the interaction and interference of different features, particularly type abstraction, dynamic typing, and type sharing. They cannot be modelled without modelling all these features. However, we split the development in two parts: first we develop a basic calculus with a straightforward but inflexible notion of type abstraction, which is sufficient to study most of the interesting properties. Then we extend it with more realistic higher-order abstraction and show that it maintains these properties.

One interesting aspect of Alice ML's dynamic typing semantics not covered by the above list is laziness: the combination of dynamic type analysis induces a notion of *lazy types*, because type analysis may need to trigger a lazy suspension binding type variables. Neis has given an account of lazy types in a higher-order polymorphic λ-calculus [Nei06], but unfortunately, the type equivalence we consider in our system is significantly more complex due to singleton kinds (Section 10.3) and relies on the environment, so that his approach cannot readily be transplanted. We thus leave the integration of lazy types into our calculus as future work.

10.1. Core Language: Higher-order Polymorphic λ-calculus

ML is a mostly functional programming language, and in this thesis, we only consider its functional subset. The canonical choice for a formal model of such a language is the λ-calculus [Bar92, Pie02]. Specifically, because ML is polymorphically typed, it naturally maps to the polymorphic λ-calculus [Gir71, Rey74, Bar92, Pie02], also known as System F [Gir71]. Since the λ-calculus and its use for modelling basic programming language features are completely standard [Pie02], we are only going to recap it briefly, by means of a few representative examples. Specifically, we refer to Harper & Mitchell's work on the type structure of ML [HM93] for a

$$
\begin{array}{lll}
\text{kinds} & \kappa & ::= \Omega \mid \kappa \to \kappa \mid \kappa \times \kappa \\
\text{types} & \tau & ::= \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \tau\tau \mid \langle\tau,\tau\rangle \mid \tau{\cdot}1 \mid \tau{\cdot}2 \\
\text{terms} & e & ::= x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e,e\rangle \mid \mathsf{let}\langle x,x\rangle = e \;\mathsf{in}\; e \\
 & & \quad \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle\tau,e\rangle_\tau \mid \mathsf{let}\langle\alpha,x\rangle = e \;\mathsf{in}\; e \\
\text{environments} & \Gamma & ::= \cdot \mid \Gamma,\alpha{:}\kappa \mid \Gamma,x{:}\tau
\end{array}
$$

Figure 10.1.: Syntax of λ^ω

thorough discussion on the relation between ML and typed λ-calculus. In particular, they study in detail the relationship between implicitly Hindley/Milner style [Hin69, DM82] and explicitly typed systems modelling ML.

Figure 10.1 shows the syntax of a standard variant of the (impredicative) higher-order polymorphic λ-calculus (System F_ω). Like Girard's original formulation [Gir71] it includes existential types. It has been further enriched with straightforward binary Cartesian products on term and type level. On the term level, we use a pattern matching let construct instead of projections to deconstruct pairs, for notational symmetry with existential types. From now on, we will refer to this basic calculus as λ^ω.

Many of our examples will assume, without further notice, that the bare calculus has been extended with built-in integers, Booleans or other standard types, plus corresponding constants.

This calculus captures the essentials of ML. For instance, consider the following ML definitions:

fun area (x : int, y : int) = x × y
fun α twice (x : α) = (x,x)
fun (α,β,γ) compose (f : α → β) (g : γ → α) (x : γ) = f (g x)
val h = compose area twice

These declarations consist of little more than lambda abstraction and application and can be translated into λ-calculus almost literally, mapping polymorphic type variables in the ML code to type lambdas. We only have to insert type applications:

$area = \lambda p : int{\times}int.\ \mathsf{let}\langle x,y\rangle = p \;\mathsf{in}\; x \times y$
$twice = \lambda\alpha : \Omega.\ \lambda x : \alpha.\ \langle x,x\rangle$
$compose = \lambda\alpha : \Omega.\ \lambda\beta : \Omega.\ \lambda\gamma : \Omega.\ \lambda f : \alpha \to \beta.\ \lambda g : \gamma \to \alpha.\ \lambda x : \gamma.\ f\,(g\,x)$
$h = compose\ (int{\times}int)\ int\ int\ area\ (twice\ int)$

Higher-order type expressions in λ^ω generalise parameterised type definitions. For instance, the ML declarations

type α pair = α × α
type (α,β) func = α pair → β

can be transliterated into λ^ω-types as follows:

$pair = \lambda\alpha : \Omega.\ \alpha \times \alpha$
$func = \lambda\alpha : \Omega.\ \lambda\beta : \Omega.\ pair\ \alpha \to \beta$

Parameterised type definitions map to functions on the type level. For that reason, we have employed F_ω, although ML only represents a limited subset of the full higher-order system. Incidentally, all kind annotations in our examples are ground (i.e. kind Ω). That is no accident: the ML core language cannot express higher-order types, only first-order type functions are possible. However, the full higher-order power of the calculus unfolds as soon as we consider modules.

One noteworthy omission of the calculus when compared to the ML core language is the absence of a fixed point operator. Unlike ML, bare F_ω cannot express general recursion, so that it in fact is terminating [Gir71, Bar92]. We do not add a primitive for recursion because the dynamic typing construct we will introduce in Section 10.4 actually allows expressing a fixed point operator for any inhabited type. It hence is unnecessary to make recursion primitive.

10.2. Modules: Existential Types and Higher-order Quantification

Modules play an important role in ML. Moreover, Alice ML's dynamic typing facilities and component system are centred around modules.

The ML module system has originally been proposed as a language based on dependent types [Mac86, Mac84]. That approach induces quite a heavyweight meta-theory, which had to be tamed in a long line of work [HMM90, MT94, HL94, DCH03]. It has repeatedly been suggested that dependent types are overkill for representing modules, and that most of its features can be sufficiently approximated in a simpler language based on a second-order calculus [Rus99]. We hence follow more recent work [HMM90, Rus99, Dre05] and express modular structure in terms of λ^ω with only moderate extensions.

The plain λ^ω-calculus already is expressive enough to cover most of ML's module language. In particular, the existential types of λ^ω have a close correspondence to signatures: the signature

```
signature COMPLEX =
sig
    type complex
    val i : complex
    val mk : real → real → complex
    val re : complex → real
    val im : complex → real
    val mul : complex → complex → complex
end
```

can be interpreted as a λ^ω-type as follows (reading $\times$ as a right-associative operator):

$$\begin{aligned} &\mathit{COMPLEX} = \\ &\quad \exists \mathit{complex} : \Omega.\, \mathit{complex} \times (\mathit{real} \to \mathit{real} \to \mathit{complex}) \times (\mathit{complex} \to \mathit{real}) \\ &\qquad\qquad \times (\mathit{complex} \to \mathit{real}) \times (\mathit{complex} \to \mathit{complex} \to \mathit{complex}) \end{aligned}$$

Signatures with more than one type can be expressed either by nested quantifiers, or in a more regular manner, by quantification over product kinds [HS00]. For example,

```
signature GENERIC_COMPLEX =
sig
    type base
    type complex
    val i : complex
    val mk : base → base → complex
    val re : complex → base
    val im : complex → base
    val mul : complex → complex → complex
end
```

can be mapped to the type

$$\begin{aligned} &\mathit{GENERICCOMPLEX} = \\ &\quad \exists \langle \mathit{base}, \mathit{complex} \rangle : (\Omega \times \Omega).\, (\mathit{complex} \times (\mathit{base} \to \mathit{base} \to \mathit{complex}) \\ &\qquad\qquad \times (\mathit{complex} \to \mathit{base}) \times (\mathit{complex} \to \mathit{base}) \\ &\qquad\qquad \times (\mathit{complex} \to \mathit{complex} \to \mathit{complex})) \end{aligned}$$

Literature on ML modules often refers to the type component in such an existential type as the *static* (or compile-time) part of a structure, and to the term component as its *dynamic* (or run-time) part [HMM90, Sto05, Dre05]. As a meta-syntactic abbreviation, we will use the notation $\mathrm{Stat}(\tau)$ and $\mathrm{Dyn}(\tau)$ to refer to both these parts of a given existential type τ, i.e. if $\tau = \exists\alpha{:}\kappa_1.\tau_2$, then $\mathrm{Stat}(\tau) = \kappa_1$ and $\mathrm{Dyn}(\tau) = \tau_2$. This notion can be generalised to other types, but we will not need it for our examples.

We note in passing that parameterised type specifications, as in

sig type α set ... **end**

would materialise as quantifiers with higher kind in λ^ω, i.e. $\exists set : \Omega{\to}\Omega.\tau$ for this particular example.

In correspondence with the type level, a *value* of existential type (called an *existential pair*, or simply an *existential* throughout this thesis) mirrors the concept of a structure. It is not hard to see that a sealed structure definition like

```
structure C :> COMPLEX =
struct
    type complex = real × real   (*)  polar; invariant: for all (a,th), 0 ≤ th < 2×pi
    val i = (1.0, pi/2.0)
    fun mk x y = (sqrt (x×x + y×y), atan2 y x + pi)
    fun re (a,th) = a × cos th
    fun im (a,th) = a × sin th
    fun mul (a1,th1) (a2,th2) = (a1×a2, rem (th1 + th2, 2×pi))
end
```

can be reflected by the following λ^ω-term (we use pattern matching λ-notation to abbreviate local let expressions in obvious ways, and n-ary tuple syntax $\langle t_1, \ldots, t_n\rangle$ as a syntactic abbreviation for the nested tuple $\langle t_1, \langle t_2, \ldots \langle t_{n-1}, t_n\rangle \ldots\rangle\rangle$):

$$
\begin{array}{l}
C = \langle real \times real, \\
\quad \langle\langle 1, pi/2\rangle, \\
\quad\ \lambda x : real.\ \lambda y : real.\ \langle sqrt(x \times x + y \times y), arctan2\ y\ x + pi\rangle, \\
\quad\ \lambda\langle a, th\rangle : real \times real.\ a \times cos\ th, \\
\quad\ \lambda\langle a, th\rangle : real \times real.\ a \times sin\ th, \\
\quad\ \lambda\langle a_1, th_1\rangle : real \times real.\ \lambda\langle a_2, th_2\rangle : real \times real. \\
\qquad\quad \langle a_1 \times a_2, rem(th_1 + th_2, 2 \times pi)\rangle\rangle \\
\quad \rangle_{COMPLEX}
\end{array}
$$

In particular, existential quantification hides the identity of the representation type, mirroring the type-abstracting effect of sealing in the ML version, thus protecting the representational invariant of the implementation. We will elaborate on this observation – and its shortcomings – in Section 10.5.

Existentials have to be *opened* with a respective elimination form to access their components. As Cardelli & Leroy showed [CL90], the dot notation $M.x$ used for modules can be translated systematically into open expressions: roughly, every module has to be opened at the outermost scope – that is, immediately at its binding point. All dot accesses can then be substituted by suitable (sequences of) projections from the two variables bound by the open expression. Putting everything together, the ML code

```
structure C : COMPLEX = struct ... end
val it = (fn c : C.complex ⇒ C.re (C.mul c c)) (C.mk 4.0 3.0)
```

can be represented by a λ^ω-expression of the following form (we use nested pattern matching notation to hide additional let expressions):

$$\mathsf{let}\langle complex, \langle i, mk, re, im, mul\rangle\rangle = \langle real \times real, \langle \ldots\rangle\rangle_{COMPLEX}$$
$$\mathsf{in}\ (\lambda c{:}complex.re\ (mul\, c\, c))\ (mk\ 4\ 3)$$

Where structures translate to existential types and pairs, functors translate to universal types and functions. Consider a functor

```
functor F (C : COMPLEX) =
struct
    type t = C.complex
    val z = C.mk 0.0 0.0
end
```

A naive attempt of a translation might be into a term of type

$$COMPLEX \to \exists t : \Omega.\ t$$

Unfortunately, this is not faithful, because it cannot properly propagate the type information from the argument to the result type of the functor, due to the lack of dependent types – the returned type t would be fully abstract. In fact, if we had not re-exported C.complex as t, the expression could not be typed at all, because the type variable α bound in $COMPLEX$ would have to escape its scope in order to express the type of z.

A proper translation thus requires splitting the $COMPLEX$ signature into its static and dynamic parts, such that the types in the static part can scope not only over the dynamic part, but over the whole codomain of the function, by means of universal quantification:

$$\lambda complex : \mathrm{Stat}(COMPLEX).\ \lambda\langle \_, mk, \_, \_, \_\rangle : \mathrm{Dyn}(COMPLEX).\ \langle complex, mk\, 0\, 0\rangle$$

This expression is typed as

$$\forall complex : \mathrm{Stat}(COMPLEX).\ \mathrm{Dyn}(COMPLEX) \to \exists \alpha : \Omega.\ complex$$

At call site the translation analogously requires passing types and terms separately.

In a similar vein, the λ^ω formulation is not compositional when it comes to nesting of structures. Consider the following signature:

```
signature S =
sig
    structure C : COMPLEX
    type t
    val f : C.complex → t
end
```

To translate this signature and capture the dependency of the type of f on the nested structure C, we again need to split the COMPLEX signature and then lift the static part to the outer quantifier:

$$\exists\langle complex, t\rangle : (\mathrm{Stat}(COMPLEX) \times \Omega).\ (\mathrm{Dyn}(COMPLEX) \times (complex \to t))$$

Examples like this demonstrate why modules are usually interpreted using dependent types: existential types do not allow projection but require opening instead – which limits scope and is only possible on the term level. Fortunately, the non-dependent approach we sketched can be applied systematically, yielding a whole-program translation scheme known as *phase splitting* [HMM90, Sto05, Dre05], where every module is split into its static and its dynamic part. The former can be represented as a type of potentially higher kind, the latter as a term of potentially higher-order polymorphic type. We will not repeat the details of this translation

here but be content with having given an intuitive understanding of the underlying idea. The interested reader is referred to the aforementioned works for deeper enlightenment.

As it turns out, we will not be able to forgo dependent typing entirely: with the introduction of singleton kinds in Section 10.3 we will have to move to dependent *kinds*. However, these are less critical than dependent *types*. In particular, their addition does not affect the phase distinction between type checking and evaluation, since they are purely a static matter.

10.3. Type Sharing and Translucency: Singleton Kinds and Subtyping

While λ^ω can already encode most of ML modules in a relatively direct manner, one important feature cannot be expressed as easily. The ML signature language allows specifications of types in two ways: *abstractly* or *concretely*. As we saw in the previous section, an existential type corresponds to a signature with an abstract type specification. But there is no counterpart in λ^ω to a signature with a concrete (or *manifest*) type specification, like in

```
signature REAL_COMPLEX =
sig
    type base = real
    type complex
    val i : complex
    val mk : base → base → complex
    val re : complex → base
    val im : complex → base
    val mul : complex → complex → complex
end
```

which in ML, given the signature GENERIC_COMPLEX from the previous section, can be abbreviated by a short-hand:

```
signature REAL_COMPLEX = GENERIC_COMPLEX where type base = real
```

To address this shortcoming, we follow the route of recent literature on ML modules [Sto00, DCH03, Dre05], and employ the concept of *singleton kinds* [Asp95, SH06] as uniform means to express transparent type information, and thus so-called *translucent* signatures with mixed concrete/abstract type information.[1]

A singleton is a new form of kind, written $\mathrm{S}(\tau)$, which classifies all types that are provably equivalent to τ. Such a kind is inhabited only by (the equivalence class) of a single type, hence the name. Any type τ can be assigned its own most specific singleton kind $\mathrm{S}(\tau)$.

Figure 10.2 summarises the syntactic changes to the kind language of λ^ω necessary to encompass singletons. We will refer to the resulting calculus as $\lambda^\omega_{\mathrm{S}}$. Besides the addition of singleton kinds themselves, the plain arrow and pair kinds of λ^ω have to be generalised to dependent products and sums [ML71, Bar92], to accommodate the appearance of types within kinds. For example, the kind $\Sigma\alpha{:}\Omega.\mathrm{S}(\alpha)$ can be assigned to all pairs of two equivalent types and is thus more specific than $\Sigma\alpha{:}\Omega.\Omega$.

Our kind language thereby is an almost unmodified instance of the type system with singletons that was investigated by Stone & Harper [SH06]. This allows us to directly re-use most of their techniques and results. In particular, we will follow their approach of conveniently defining singletons at higher kind, written $\mathrm{S}(\tau : \kappa)$, as mere syntactic sugar over the basic language specified in Figure 10.2.

Given singleton kinds, the REAL_COMPLEX signature can be expressed as

[1] Singletons were first introduced as types instead of kinds, but obviously they can be used one level up as well.

$$\text{kinds} \quad \kappa \quad ::= \quad \Omega \mid \mathrm{S}(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa$$

$$\begin{array}{lcl} \kappa_1 \to \kappa_2 & := & \Pi\_{:}\kappa_1.\kappa_2 \\ \kappa_1 \times \kappa_2 & := & \Sigma\_{:}\kappa_1.\kappa_2 \end{array}$$

Figure 10.2.: Syntax extensions for $\lambda^{\omega}_{\mathrm{S}}$ (λ^{ω} with singleton kinds)

$$\begin{array}{l} \mathit{REALCOMPLEX} = \\ \quad \exists\langle \mathit{base}, \mathit{complex}\rangle : (\mathrm{S}(\mathit{real}) \times \Omega).\, (\mathit{complex} \times (\mathit{base} \to \mathit{base} \to \mathit{complex}) \\ \qquad\qquad \times (\mathit{complex} \to \mathit{base}) \times (\mathit{complex} \to \mathit{base}) \\ \qquad\qquad \times (\mathit{complex} \to \mathit{complex} \to \mathit{complex})) \end{array}$$

That is, an abstract type specification amounts to quantification over base kind Ω (or some higher kind in the case of parameterised types), while a concrete specification is likewise represented by quantification, but with a singleton kind. Since $\tau : \mathrm{S}(\tau)$ is always derivable, a type will successfully match a compatible singleton kind annotation. The whole type actually is equivalent to

$$\begin{array}{l} \exists\langle \mathit{base}, \mathit{complex}\rangle : (\mathrm{S}(\mathit{real}) \times \Omega).\, (\mathit{complex} \times (\mathit{real} \to \mathit{real} \to \mathit{complex}) \\ \qquad\qquad \times (\mathit{complex} \to \mathit{real}) \times (\mathit{complex} \to \mathit{real}) \\ \qquad\qquad \times (\mathit{complex} \to \mathit{complex} \to \mathit{complex})) \end{array}$$

In ML, a signature with a concrete type specification matches a signature of similar form, but with an abstract specification in place for the respective type. For example,

sig type t = int × int **end** matches **sig type** t **end**

To capture this form of subsumption, $\lambda^{\omega}_{\mathrm{S}}$ incorporates a notion of *subkinding*, induced by a basic rule deriving $\mathrm{S}(\tau) \leq \Omega$ for any well-formed singleton $\mathrm{S}(\tau)$. Thus, a concrete type (of singleton kind) can be passed wherever a type is specified abstractly (with kind Ω).

The subkinding relation is extended to *subtyping* by inducing inclusion of quantified types. That is, we have $\exists\alpha{:}\kappa.\tau \leq \exists\alpha{:}\kappa'.\tau'$ whenever $\kappa \leq \kappa'$ (and $\tau \leq \tau'$). Universal quantifiers act similarly, but are contravariant. As we will see later (Section 10.4), subtyping is necessary to express dynamic signature matching, as it occurs in Alice ML when unpacking a package (Section 4.1).

10.4. Dynamic Typing: Type Analysis

The $\lambda^{\omega}_{\mathrm{S}}$-calculus encompasses the essential expressiveness of the ML language and its static type and module system with reasonable accuracy. As such, it is relatively standard, and described in the literature in minor variations [Sto05, Dre05]. However, our main interest lies in extending the language with *dynamic* typing, which is what we turn to now.

Alice ML offers *packages* as its central means for dynamic typing (Chapter 4). Packages are basically a variant of *dynamics* [Myc83, ACPP91, LM93, ACPR95], but carrying modules. Since we can already express modules in our calculus, it is sufficient to integrate a simple form of dynamics.

Looking closer, dynamics are actually two features in one: (1) they provide a way to store type information in a value, and (2) they allow dispatching on such dynamic type information. Existential types already provide the first, so it is a worthwhile simplification to provide dynamic

type switching as a separate feature, and express dynamics in terms of these two, more primitive mechanisms.

Like dynamics, dynamic *type analysis* has been investigated extensively in literature [HM95, DRW95, Gle99, CW99, TSS00, Wei02, CWM02, VWW05]. Most of these works introduce a typecase construct of varying complexity for dispatching on the structure of a type, sometimes along with a similar construct typerec on the type level, to encompass higher-order types.

We restrict ourselves to a much simpler construct, which just provides a way to compare two types, and branch according to the result:

$$\mathsf{case}\, e_1{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}\ e_3$$

Evaluating this expression compares the dynamic instantiations of the type τ_1 of value e_1 with τ_2. If they match, the e_2 branch is taken, with x bound to the value of e_1, otherwise e_3 is evaluated. However, the comparison is not for type *equivalence*, but for *subtyping*! That is, if $\tau_1 \leq \tau_2$ then e_2 is chosen. In other words, our type case is reminiscent of a *checked downcast*. More formally, it is a variant of Girard's J operator [Gir71], or rather Harper & Mitchell's *TypeCond* operator [HM99], enriched with subtyping (Section 12.3.3).

Given a primitive for dynamic type analysis, an equivalent to the Alice ML package type can be defined in our system as $\exists\alpha{:}\Omega.\alpha$, which we will abbreviate to package in the following. The intuition is obvious: a package is a value of some type named α, paired with the actual type. The equivalent to injecting some value into type package hence is nothing more than plain existential construction.

Consider the following Alice ML example:

```
val p = pack (val it = 5) : (val it : int)
```

The straightforward translation is

$$p = \langle \mathit{int}, 5 \rangle_{\mathsf{package}}$$

This scales to more complex modules in the obvious way. For instance, here is a module including a type:

```
val q = pack (type t = int; fun f (n : int) = n) : (type t = int; val f : t → t)
```

It could be expressed as follows:

$$q = \langle \exists t : \mathrm{S}(\mathit{int}).\ t \to t, \langle \mathit{int}, \lambda n : \mathit{int}.\ n \rangle_{\exists t:\mathrm{S}(\mathit{int})} \rangle_{\mathsf{package}}$$

Note the twofold use of existential types in this example: the inner existential expresses the structure, the outer the package.

Since constructing a package obviously is straightforward, unpacking is the more interesting direction: given a value of package type and a *target signature* (cf. Section 4.1) it should yield the contained module, if and only if the dynamic signature is a subtype of the target signature. Consider the following piece of Alice ML code:

```
unpack p : (val it : int)
```

It is not difficult to express this operation in terms of existential opening and type case:

$$\mathsf{let}\langle \alpha, x' \rangle = p\ \mathsf{in}\ \mathsf{case}\, x' : \alpha\ \mathsf{of}\ x : \mathit{int}.\ x\ \mathsf{else}\ \bot_{\mathit{int}}$$

Because we do not have exceptions in the calculus, we use $\bot$ to indicate the exceptional case (we will see in Section 12.3.3 that we can find a diverging computation for all interesting types).

The second of the above packages, q, can be unpacked successfully in at least two different ways:

structure M_1 = **unpack** q : (**type** t = int; **val** f : t → t)
structure M_2 = **unpack** q : (**type** t; **val** f : t → t)

M_1 unpacks it under its original, most precise signature. M_2 however uses a supersignature where t is left abstract. Both can be translated directly:

$$M_1 = \mathsf{let}\langle\alpha, M'\rangle = q \text{ in } \mathsf{case}\, M' : \alpha \text{ of } M : (\exists t : \mathrm{S}(int).\ t \to t).\ M \text{ else } \bot$$
$$M_2 = \mathsf{let}\langle\alpha, M'\rangle = q \text{ in } \mathsf{case}\, M' : \alpha \text{ of } M : (\exists t : \Omega.\ t \to t).\ M \text{ else } \bot$$

The latter example demonstrates why we needed to built in subtyping (Section 10.3) into our calculus: the case expression in M_2 does only evaluate successfully because $\exists t : \mathrm{S}(int).\ t \to t \leq \exists t : \Omega.\ t \to t$. Unlike static uses of subsumption [Sto05, Dre05], we cannot express the conversion between these types by η-expansion, because the package type is not statically known.

10.5. Loss of Parametricity and Abstraction Safety

Although the type case construct may look rather innocent at first, it has serious implications on the properties of the calculus. As said above, it represents a variant of Girard's J combinator [Gir71, HM99], and Girard showed that such an operator inherently destroys the *parametricity* property of the polymorphic λ-calculus.

Intuitively, parametricity means that a well-formed term always reduces in the same way, regardless of potential substitution of free type variables [Rey83, BFSS89, ACC93]. Obviously, a type case enables inspection of the dynamic instantiation of a type variable, and it can evaluate to different branches, accordingly.

As already explained in Section 4.5, a number of useful properties are consequences of parametricity, and they get lost along the way. For example, unlike λ^ω, the extended calculus is not terminating (Section 12.3.3). More seriously, though, *type abstraction* can no longer be expressed by conventional means.

Type abstraction is one of the most important tools of modular programming [Mor73a, Mor73b, MP88, HP05]. In the ML module system type abstraction is achieved through *sealing* (Chapter 3): the representation of a type defined by a structure is hidden to clients by ascribing a signature containing an abstract type specification. For example, recall our structure defining complex numbers:

```
signature COMPLEX =
sig
    type complex
    val i : complex
    val mk : real → real → complex
    val re : complex → real
    val im : complex → real
    val mul : complex → complex → complex
end

structure C :> COMPLEX =
struct
    type complex = real × real  (* polar; invariant: for all (a,th),  ≤ th < 2×pi *)
    val i = (1.0, pi/2.0)
    fun mk x y = (sqrt (x×x + y×y), atan2 y x + pi)
    fun re (a,th) = a × cos th
    fun im (a,th) = a × sin th
    fun mul (a1,th1) (a2,th2) = (a1×a2, rem (th1 + th2, 2×pi))
end
```

This declaration defines an abstract type C.complex, whose representation is hidden from clients – they can only use the operations from the COMPLEX signature to construct, consume, and inspect values of this type.

If the language provides no way to breach an abstraction barrier erected through the use of abstract types it is called *abstraction safe*. Standard ML and most similar languages have this property.

In their classic paper, Mitchell & Plotkin observed that type abstraction is closely related to existential quantification [MP88]. In Section 10.2, we already saw how the complex number ADT can be expressed in λ^ω as an existential:

$$\begin{aligned} C = \langle & \mathit{real} \times \mathit{real}, \\ & \langle \langle 1, \mathit{pi}/2 \rangle, \\ & \ \lambda x : \mathit{real}.\ \lambda y : \mathit{real}.\ \langle \mathit{sqrt}(x \times x + y \times y), \mathit{arctan2}\ y\, x + \mathit{pi} \rangle, \\ & \ \lambda \langle a, \mathit{th} \rangle : \mathit{real} \times \mathit{real}.\ a \times \mathit{cos}\ \mathit{th}, \\ & \ \lambda \langle a, \mathit{th} \rangle : \mathit{real} \times \mathit{real}.\ a \times \mathit{sin}\ \mathit{th}, \\ & \ \lambda \langle a_1, \mathit{th}_1 \rangle : \mathit{real} \times \mathit{real}.\ \lambda \langle a_2, \mathit{th}_2 \rangle : \mathit{real} \times \mathit{real}. \\ & \qquad \langle a_1 \times a_2, \mathit{rem}(\mathit{th}_1 + \mathit{th}_2, 2 \times \mathit{pi}) \rangle \rangle \\ & \rangle_{\mathit{COMPLEX}} \end{aligned}$$

Abstraction safety in this approach crucially relies on parametricity. To be accessed, the existential has to be opened, as in

$$\mathsf{let} \langle \gamma, \mathit{ops} \rangle = C\ \mathsf{in}\ e$$

But only because the body e – which represents the client code of the structure – is parametric in the bound type variable γ representing the abstract type *complex*, it is guaranteed not to be able to break the abstraction: it simply cannot perform any computations that depend on the representation of γ, i.e. the concrete type that will be substituted during reduction.

Unfortunately, the loss of parametricity caused by the addition of dynamic type analysis invalidates this solely static model of type abstraction. Specifically, we can construct an expression that breaks the abstraction of the complex number ADT by dynamically inspecting the abstract type and consecutively constructing a value of the abstract type without using the *mk* operator from its signature:

$$\mathsf{let} \langle \gamma, \mathit{ops} \rangle = C\ \mathsf{in} \ldots \mathsf{case}\ \langle 0, -3 \times \mathit{pi} \rangle : \mathit{real} \times \mathit{real}\ \mathsf{of}\ x : \gamma.\ x\ \mathsf{else}\ i \ldots$$

The type case in this expression is perfectly well-typed. Dynamically, it checks whether $\mathit{real} \times \mathit{real}$ and the abstract type γ are equivalent, and in case they are, interprets the pair $\langle 0, -3 \times \mathit{pi} \rangle$ as a complex value. And the check will indeed succeed, because the reduction rule for the let expression opening the existential will substitute γ by $\mathit{real} \times \mathit{real}$, implying $\gamma \equiv \mathit{real} \times \mathit{real}$. By evaluating this expression, we have forged a value of the abstract type for complex numbers. Note that indeed, $\langle 0, -3 \times \mathit{pi} \rangle$ does not adhere to the intended invariant of the implementation, which would require $-3\pi \in [0; 2\pi[$.

There seem to be two principal ways to restore abstraction safety in the presence of dynamic type analysis:

1. Disallow dynamic analysis of certain types. This approach has been suggested by Harper & Morrisett [HM95], who propose statically distinguishing between analysable and non-analysable types, supposedly via the kind system.

2. Prevent substitution of (existentially) quantified types. Abadi, Cardelli, Pierce & Rémy [ACPR95] propose to simply replacing the type variable bound by open with a "fresh" type constant during evaluation.

The first solution is overly restrictive for our purposes, because it would preclude values of certain types – specifically, all abstract types – to be put in packages. Clearly, such a limitation would render packages insufficient as a basis for a component system. We need a *full reflexivity* type analysis mechanism, i.e. all types have to be analysable – or rather, comparable – dynamically.[2]

The idea put forth by Abadi et al. is more promising. It amounts to changing the reduction rule for opening existientials to:

$$\mathsf{let}\langle\alpha, x\rangle = \langle\tau, v\rangle \ \mathsf{in}\ e \quad \rightarrow \quad e[t/\alpha][v/x]$$

where t is a fresh type constant. Obviously, the representation type could no longer be analysed transparently.

Unfortunately, this modification has two fundamental problems:

- The notion of "fresh type constant" is not formally defined. It needs to be made precise what dynamic freshness means.
- Worse, the rule destroys type preservation.

To see how type preservation is lost, consider the following example:

$$\mathsf{let}\langle \mathit{complex}, \langle i, \mathit{mk}, \mathit{re}, \mathit{im}, \mathit{mul}\rangle\rangle = C \ \mathsf{in}\ (\lambda c : \mathit{complex}.\ \mathit{re}\, c)\, i$$

It is easy to verify that this term is well-typed in λ^{ω} . Nevertheless, after applying the above reduction rule (and the implicit deconstructions of the nested operator tuple) the term becomes:

$$(\lambda c : t.\, (\lambda\langle a, \mathit{th}\rangle : \mathit{real} \times \mathit{real}.\ a \times \cos\ \mathit{th})\ c)\ \langle 1, \mathit{pi}/2\rangle$$

which is no longer typable! In two places there is a clash between the abstract type t – which now stands for *complex* – and its representation type $\mathit{real} \times \mathit{real}$: the tuple $\langle 1, \mathit{pi}/2\rangle$ representing the complex number i simply has type $\mathit{real} \times \mathit{real}$ where the outer λ-expression expects t, and vice versa, c has type t where the implementation of the *re* function expects a pair.

The approach we are going to present in the following takes up the idea of generating fresh types to maintain abstraction safety, but avoiding the above problems:

- Instead of "fresh type constants" it introduces an explicit construct for dynamically *generating* type names, independent from existential types.
- Type inconsistencies between abstract types and their representations are addressed by introducing *coercions* for mitigating between the types.

The basic approach has been described previously in a paper of ours [Ros03a]. However, in the paper we did not consider the interaction with singleton kinds and the resulting dependency of kinds on types, which turns out to be a severe complication. Moreover, the current work features a more modular approach to recording the isomorphism between generated types and their representations, by using the kind system. It also simplifies higher-order coercions by treating them as syntactic sugar.

[2] We borrow the term *full reflexivity* from Trifonov, Saha & Shao [TSS00], who introduced it in a slightly different context to express the absence of any restriction on the *syntactic form* of types that are available for analysis (no such restriction is necessary for the simple form of type case used in this thesis).

10.6. Abstraction Safety: Dynamic Generativity

Formalisms for describing dynamic generation of fresh *value* names are well developed. For example, the name restriction form $\nu n.P$ is a central feature of the π-calculus [SW01] and can be viewed as an expression that generates a new name n with local scope. Pitts' λ_ν-calculus [PS93] transfers that idea to the λ-calculus, although with a different formulation.

However, we are not concerned with value names but with *type* names. Consequently, although we adopt a similar construct, it has a different form:

$$\mathsf{new}\,\alpha \approx \tau\ \mathsf{in}\ e$$

This expression generates a new type denoted by the type variable α, bound within e (and thus subject to standard α-conversion rules). To track generated type names, we will employ a *type heap* that is maintained by the reduction relation and maps type variables to kinds.

Because uninhabited types are rarely interesting, every new type has to be associated with a *representation type* τ. The new type differs from τ, i.e. the static type system as well as dynamic type analysis will distinguish between both types. However, they are known to be *isomorphic* within the scope of the new construct. The isomorphism can be exploited by means of *coercions* that mitigate between the two types.

We take the most conservative approach and require these coercions to be performed as explicit operations. That is, we introduce two symmetric expression forms:

$$\{e\}^+_\tau \qquad \text{and} \qquad \{e\}^-_\tau$$

Given an abstract type τ with representation τ', the upward coercion $\{e\}^+_\tau$ converts e of type τ' into τ, while the downward coercion $\{e\}^-_\tau$ does the inverse. Reduction then allows nested inverse coercions to *cancel* each other:

$$\{\{v\}^+_\tau\}^-_\tau \quad \to \quad v$$

Note that the representation type τ' is known from context and hence does not need to be annotated. We will discuss the design space surrounding coercions in Section 12.5. As a simple example, the expression

$$\mathsf{new}\,\alpha \approx \mathit{int}\ \mathsf{in}\ (\lambda x : \alpha.\,\{x\}^-_\alpha)\,\{666\}^+_\alpha$$

is well-formed having type *int* and results in the integer 666.

A coercion requires the annotated type τ to be an abstract type, as generated by the new operator. Moreover, it must be associated with a suitable representation type. How do we keep track of this information?

While our previous work [Ros03a] used special environment entries, we take a more uniform approach in this thesis, which is inspired by singleton kinds: analogous to the latter, we introduce a new form of *abstraction kinds*, written $\mathrm{A}(\tau)$, which are assigned to abstract types with representation type τ. Like singleton kinds, abstraction kinds record the fact that a type is isomorphic to another type. But unlike singletons, where this isomorphism can be exploited *implicitly* by the type system (through type equivalence and subsumption), abstraction kinds can only be utilised *explicitly*, by means of coercions.

Note that the type language presented here is higher-order, hence a type of abstraction kind is not necessarily a type variable bound by new – that is the reason why coercions are actually annotated with a *type*, not a simple type variable.

Figure 10.3 summarises all changes to the syntax of $\lambda^\omega_{\mathrm{S}}$ for extending it with dynamic type analysis, generativity, and corresponding coercions and abstraction kinds. We will refer to the calculus incorporating all these extensions as $\lambda^\omega_{\mathrm{SA}}$.

$$
\begin{array}{lrcl}
\text{base kinds} & \hat{\kappa} & ::= & \Omega \mid \mathrm{A}(\tau) \\
\text{kinds} & \kappa & ::= & \hat{\kappa} \mid \mathrm{S}_{\hat{\kappa}}(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa \\
\text{terms} & e & ::= & \ldots \mid \mathsf{case}\, e{:}\tau\, \mathsf{of}\, x{:}\tau.e\, \mathsf{else}\, e \mid \mathsf{new}\, \alpha \approx \tau\, \mathsf{in}\, e \mid \{e\}^+_\tau \mid \{e\}^-_\tau
\end{array}
$$

Figure 10.3.: Syntax extension of $\lambda^{\omega}_{\mathrm{SA}}$ (λ^{ω} with dynamic type analysis and generativity)

In this calculus, we can express the ADT of complex numbers in such a way that it is dynamically abstraction safe by generating a fresh type, and strategically inserting adequate coercions:

$$
\begin{array}{lcl}
C' & = & \mathsf{new}\, c \approx \mathit{real} \times \mathit{real}\, \mathsf{in} \\
 & & \langle c, \\
 & & \ \langle \{\langle 1, \mathit{pi}/2\rangle\}^+_c, \\
 & & \ \ \lambda x : \mathit{real}.\ \lambda y : \mathit{real}.\ \{\langle \mathit{sqrt}(x \times x + y \times y), \mathit{arctan2}\ y\ x + \mathit{pi}\rangle\}^+_c, \\
 & & \ \ \lambda z : c.\ \mathsf{let}\langle a, \mathit{th}\rangle = \{z\}^-_c\ \mathsf{in}\ a \times \mathit{cos}\ \mathit{th}, \\
 & & \ \ \lambda z : c.\ \mathsf{let}\langle a, \mathit{th}\rangle = \{z\}^-_c\ \mathsf{in}\ a \times \mathit{sin}\ \mathit{th}, \\
 & & \ \ \lambda z_1 : c.\ \lambda z_2 : c.\ \mathsf{let}\langle a_1, \mathit{th}_1\rangle = \{z_1\}^-_c\ \mathsf{in} \\
 & & \qquad\qquad \mathsf{let}\langle a_2, \mathit{th}_2\rangle = \{z_2\}^-_c\ \mathsf{in} \\
 & & \qquad\qquad \{\langle a_1 \times a_2, \mathit{rem}(\mathit{th}_1 + \mathit{th}_2, 2 \times \mathit{pi})\rangle\}^+_c\rangle \\
 & & \rangle_{\mathit{COMPLEX}}
\end{array}
$$

The previous attempt to break the abstraction using type analysis will no longer succeed:

$$
\begin{array}{l}
\mathsf{let}\langle \mathit{complex}, \langle i, \mathit{mk}, \mathit{re}, \mathit{im}, \mathit{mul}\rangle\rangle = C' \\
\mathsf{in}\ \ldots\, \mathsf{case}\ \langle 0, -3 \times \mathit{pi}\rangle : \mathit{real} \times \mathit{real}\ \mathsf{of}\ x : \mathit{complex}.\ x\ \mathsf{else}\ i \ldots
\end{array}
$$

The use of type case in this expression is still well-typed. However, reduction now yields

$$
\mathsf{new}\, c \approx \mathit{real} \times \mathit{real}\ \mathsf{in}\ \ldots\, \mathsf{case}\ \langle 0, -3 \times \mathit{pi}\rangle : \mathit{real} \times \mathit{real}\ \mathsf{of}\ x : c.\ x\ \mathsf{else}\ i \ldots
$$

That is, *complex* gets substituted by the fresh type name c. Since $\mathit{real} \times \mathit{real} \leq c$ does not hold in our system, the type analysis will fail, evaluating to i instead of a forged value.

In Section 12.9 we will formally prove that abstraction safety holds for all types introduced by new.

10.7. Sealing: Higher-order Coercions and Generativity

So far, to build an abstraction, its implementation has to use coercions internally, in order to meet the intended signature type. We speak of *a priori* abstraction: an implementation must be tailored to a particular signature. On the other hand, abstraction based on existential types or sealing happens *a posteriori*: arbitrary parts of a given implementation's type are just hidden away without affecting the implementation itself. Can we recover that flexibility?

The answer is yes: we will show that we are able to define a sealing notation $e :> \tau$ as pure syntactic sugar in our calculus. Given an abstract "signature" type τ and a suitable implementation e, it systematically constructs an expression e' that coerces the whole implementation e into the desired type, in an abstraction-safe manner. Here, τ typically will be existentially quantified. The expansion is defined such that e' will generate fresh types for all quantifiers that are abstract, i.e. do not have singleton kind.

We will proceed in two steps. First, we generalise the calculus to support *higher-order generativity*. The new expression presented so far can only generate types of kind Ω. We can define a higher-order extension, written

$$
\mathsf{new}\, \alpha{:}\kappa \approx \tau\ \mathsf{in}\ e
$$

that allows arbitrary kind.

Along with higher-order generativity come *higher-order abstraction kinds*, $\mathrm{A}(\tau : \kappa)$, which are assigned to type variables in higher-order generators. They will be defined as a derived form in a manner very analogous to higher-order singletons (Section 10.3).

After having tackled higher-order generativity, the more difficult step towards the definition of a sealing operator is the introduction of *higher-order coercions*. To that end, we introduce the following notation:

$$\{e : \alpha{:}\kappa.\tau\}^{+}_{\tau_+ \approx \tau_-}$$

This expression is meant to coerce $e : \tau[\tau_-/\alpha]$ to the abstract signature type $\tau[\tau_+/\alpha]$, provided $\tau_+ : \mathrm{A}(\tau_- : \kappa)$. The type variable α is used as a placeholder that marks all the positions in τ where a concrete change of types has to occur. The former, basic coercions arise as the special case where the *residual type* τ is simply α. For example, given $\gamma : \mathrm{A}(\mathit{int} : \Omega)$, the notation

$$f \quad = \quad \{(\lambda\langle x, y\rangle : \mathit{int} \times \mathit{int}.x + y) : \alpha{:}\Omega.\, (\alpha \times \mathit{int} \to \alpha)\}^{+}_{\gamma \approx \mathit{int}}$$

ought to describe an expression of type $\gamma \times \mathit{int} \to \gamma$. Note that the first component of the function's argument type is made abstract, but not the second.

For simple types, the reduction of these higher-order coercions is relatively straightforward. The most interesting case are function types:

$$\{v : \alpha{:}\kappa.\tau_1 \to \tau_2\}^{+}_{\tau_+ \approx \tau_-} \;\to\; \lambda x_1{:}\tau_1[\tau_+/\alpha].\{(v\,\{x_1 : \alpha{:}\kappa.\tau_1\}^{-}_{\tau_+ \approx \tau_-}) : \alpha{:}\kappa.\tau_2\}^{+}_{\tau_+ \approx \tau_-}$$

The definition introduces an η-expansion inserting the necessary coercions on the result and, inversely, the argument of potential applications. For the term f above, reduction will yield the moral equivalent of the following term, using pattern matching notation:

$$\lambda\langle x', y'\rangle : \gamma \times \mathit{int}.\{f\,\langle\{x'\}^{-}_{\gamma}, y'\rangle\}^{+}_{\gamma}$$

The function takes a pair of arguments (one abstract, one concrete), coerces the former back to its representation, applies the original function, and coerces back the result to the abstract type.

As it turns out, defining higher-order coercions is far from trivial in the case of quantified types, i.e. where the residual type τ is either $\forall\alpha{:}\kappa.\tau'$ or $\exists\alpha{:}\kappa.\tau'$. In earlier work [Ros03a], we had given a straightforward definition for universal types. However, the type system considered was roughly equivalent to that of λ^{ω}, and did not have singleton kinds. Due to the presence of dependent kinds that is implied by singletons (Section 10.3), the same definition is no longer valid in a system like $\lambda^{\omega}_{\mathrm{S}}$, as we will discuss in Section 13.2.3. A correct definition is surprisingly intricate, and requires resorting to a similar notion of coercions on the type level, where we introduce the analogous notation

$$\{\tau : \alpha{:}\kappa'.\kappa\}_{\tau_+/\tau_-}$$

to coerce a type τ of kind $\kappa[\tau_-/\alpha]$ to kind $\kappa[\tau_+/\alpha]$.

With higher-order generativity and coercions in place we have the necessary ingredients to define sealing as follows:

$$e :> \exists\alpha{:}\kappa.\tau \quad := \quad \mathsf{let}\langle\alpha, x\rangle = e \;\mathsf{in}\; \mathsf{new}\,\alpha'{:}\kappa \approx \alpha \;\mathsf{in}\; \langle\alpha', \{x : \alpha{:}\kappa.\tau\}^{+}_{\alpha' \approx \alpha}\rangle_{\exists\alpha:\kappa.\tau}$$

Effectively, this definition takes the types from the "structure" e and replaces them by freshly generated names. Moreover, to keep the structure well-formed under the original signature type, its term part has to be coerced from the "old" representation types (bound to α) to the new abstract types (denoted by α'). The types thus are fully opaque, statically as well as dynamically, as is required to achieve abstraction safety.

For example, given the encoding of structure implementation C from Section 10.2, the expression

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \cdots \mid \Psi \\ \text{terms} & e & ::= & \cdots \mid \mathsf{pickle}\, e \mid \psi(e) \mid \mathsf{unpickle}\, x \Leftarrow e \;\mathsf{in}\; e \;\mathsf{else}\; e \end{array}$$

Figure 10.4.: Syntax of $\lambda^{\omega}_{\mathrm{SA}\Psi}$ (extension of $\lambda^{\omega}_{\mathrm{SA}}$ with pickling)

C :> COMPLEX

will translate to an expression that is basically an η-expansion of C' as given in Section 10.6.

For all syntactic definitions presented we will derive and prove correct suitable well-formedness and equivalence rules that show the consistency of the definitions. These proofs, which are mostly by induction on the structure of types and kinds, represent the most elaborate technical contribution of this thesis. While, for the most part, they are not difficult in principle, the complexity of the syntactic definitions (in particular for higher-order coercions over quantified types and kinds) makes them surprisingly subtle and long-winded.

After having defined sealing in terms of generativity and coercions, these constructs can be regarded as internal. The programmer does not have to be concerned with them directly. We have thus recreated the situation in ML. Moreover, the encoding of sealing we have given mirrors the actual compilation strategy for sealing used in the Alice ML compiler.

10.8. Pickling

Our main motivation for going through all the hassle of adding dynamic typing to a statically typed language is the desire to enable high-level dynamic import and export of language-level values. Dynamic typing is one key primitive for type-safe exchange of values with the outside world. However, in order to export a value, it has to be transformed into a self-contained external representation, a *pickle*. Furthermore, when such a pickle is loaded, its integrity has to be checked, because the static type system cannot make any guarantees about entities such as files that are beyond the control of the language's run-time system.

To achieve full safety, two kinds of checks have to be performed:

1. *Dynamic type matching* to ensure *external* consistency of unpickling, i.e. whether the pickle and the code loading it agree about its type.

2. *Dynamic type checking* to check *internal* consistency of the pickle, i.e. whether it actually is a well-formed value of the type it pretends to have.

The first check is what is provided by packages – or type case expressions in our calculus, respectively. But we have not yet coped with the second check.

To address this, we add a final extension to our calculus, which provides a high-level account for the semantics of pickling. Terms are enriched with three new constructs:

- $\mathsf{pickle}\, e$ creates a pickle of the value computed by e.

- $\mathsf{unpickle}\, x \Leftarrow e \;\mathsf{in}\; e_1 \;\mathsf{else}\; e_2$ takes a pickle e and extracts its value, binding it to x in e_1; if the pickle is malformed, e_2 is evaluated instead.

- $\psi(v)$ represents a pickle itself, i.e. a value that is a serialised representation of the value v. Pickles are assigned the new type Ψ.

Figure 10.4 summarises the extensions. We refer to the resulting calculus as $\lambda^{\omega}_{\mathrm{SA}\Psi}$.

The key characteristic of the pickling formalism is that there is no requirement for the term v in a pickle $\psi(v)$ to actually be well-formed! This models the fact that in practice, pickles can be created outside the runtime, by extra-linguistic means, and the language has no way to enforce that they are well-formed. Indeed, a pickle may be deliberately forged by an attacker (in a real language, a pickle might not even be syntactically valid, but we abstract from that possibility in our calculus, since it does not change the principle problem). This liberty is visible in the typing rule for pickles, which will have the following form:

$$\frac{\Gamma \vdash \square}{\Gamma \vdash \psi(v) : \Psi}\,(\mathrm{FV}(v) \subseteq \mathrm{Dom}(\Gamma))$$

This rule does not have any premise regarding v, except that it has to be closed with respect to the environment (this side condition is needed for technical ease only, namely to maintain the calculus' usual Type Variable Containment lemma, which says that a well-formed term contains no unbound variables).

If pickles may be malformed, how can we establish soundness? This requires *verification* when a pickle is loaded. In our model, verification simply amounts to a well-formedness side condition in the reduction rules for unpickle:

$$\begin{array}{lcll} \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 & \rightarrow & e_1[v/x] & \text{if } \Gamma \vdash v : \mathsf{package} \\ \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 & \rightarrow & e_2 & \text{otherwise} \end{array}$$

Upon deconstruction, it is checked that a pickle actually contains a well-formed representation of a value of package type package (Section 10.4). As we will explain in Section 12.4.2, the environment Γ is needed to capture previously generated types, as these may occur within a pickle. Except for those, Γ is empty, reflecting the requirement that a pickle be a closed value.

Despite the robustness achieved by run-time verification, we want to preclude that malformed pickles can be created from within the language. A programmer has to invoke $\mathsf{pickle}\, e$ to create pickles – the form $\psi(v)$ should be considered inaccessible in the surface language. A pickle expression first evaluates its operand e and then creates a pickle from the result. Unlike the values in a pickle, the type system requires the operand e to be a well-formed term.

Note that we do not actually model processes: neither does our calculus have concurrency (which is orthogonal to the issues discussed here), nor does it encompass local state. The only bit of state existent in our calculus is the heap of generated types, which simply acts as being global to all computation. This is a simplifying assumption that does not reflect the practical language, but abstracts from distracting detail that does not seem to add much technical insight to the problem at hand.

Also for simplicity, our calculus does not model resources either. An account for resources in the spirit of Alice ML would be relatively straightforward to add: it amounts to extending the pickle operator to conditional form and adding a side condition to its primary reduction rule that ensures that no resource names occur in v. Resources themselves could simply be represented as names drawn from a specific set.

10.9. Summary

- The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus models the essentials of the Alice ML type system.
- This includes dynamic type matching, type generativity, type sharing, and pickling.
- It is based on the higher-order polymorphic λ-calculus (System F_{ω}).

- Singletons express type sharing.
- Type analysis expresses dynamic typing.
- Type generativity is an explicit construct.
- Coercions are used to switch between abstract types and their representation.
- Pickling delays well-formedness checks for terms until reduction.

11. The Type Language

The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus incorporates an expressive static type system as well as non-trivial means of dynamic typing. In this chapter we discuss the language of types and its properties.

We will first give an overview of the type system. Most of it is standard, except for singleton and abstraction kinds. We will hence concentrate on those, first giving a recap of singletons and their implications and then discussing design issues surrounding abstraction kinds, which are closely related to singletons in most ways.

We state a number of relevant properties of the system. In the last section, we give algorithmic formulations of all judgements of the system, which are correct and decidable. We will need this result when we turn to the semantics of type case in Chapter 12. We only state the main theorems and propositions in this chapter. Auxiliary propositions and lemmata and all proofs can be found in Appendix C.

Most of what we discuss in this chapter is covered in literature, such that we can keep the discussion brief and concentrate on the most interesting issues. As far as the type language is concerned, abstraction kinds are the primary innovation over previous work.

11.1. Basic System

The abstract syntax of $\lambda^{\omega}_{\mathrm{SA}\Psi}$ is shown in Figure 11.1. As usual, we identify phrases (terms, types, and kinds) up to renaming of bound variables. Except where noted otherwise, we also use the distinct variable convention, whereby all bound variables in a phrase are assumed to be distinct from each other, and from any occurring free variable.

We write $\mathrm{FV}(t)$ for the set of free variables (including term and type variables) in the phrase t, defined as usual. We use the notation $t[t'/z]$ for the capture-avoiding substitution of the phrase t' for the variable z in the phrase t. Moreover, we use γ to range over finite substitutions and write $\gamma(t)$ for the capture-free application of such a substitution to a phrase t.

The static semantics of the calculus consists of eight judgements:

(environment validity)	$\Gamma \vdash \Box$
(kind validity)	$\Gamma \vdash \kappa : \Box$
(kind equivalence)	$\Gamma \vdash \kappa \equiv \kappa' : \Box$
(kind inclusion)	$\Gamma \vdash \kappa \leq \kappa' : \Box$
(type validity)	$\Gamma \vdash \tau : \kappa$
(type equivalence)	$\Gamma \vdash \tau \equiv \tau' : \kappa$
(type inclusion)	$\Gamma \vdash \tau \leq \tau' : \kappa$
(term validity)	$\Gamma \vdash e : \tau$

Figures 11.2 to 11.6 show the inference rules defining these judgements, except for term validity, which we delay until Chapter 12.

$$
\begin{array}{llll}
\text{base kinds} & \hat{\kappa} & ::= & \Omega \mid \mathrm{A}(\tau) \\
\text{kinds} & \kappa & ::= & \hat{\kappa} \mid \mathrm{S}_{\hat{\kappa}}(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa \\
\text{types} & \tau & ::= & \alpha \mid \Psi \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \\
 & & & \mid \lambda\alpha{:}\kappa.\tau \mid \tau\tau \mid \langle \tau, \tau \rangle \mid \tau{\cdot}1 \mid \tau{\cdot}2 \\
\text{terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e, e \rangle \mid \mathsf{let}\langle x, x \rangle = e \;\mathsf{in}\; e \\
 & & & \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle \tau, e \rangle \mid \mathsf{let}\langle \alpha, x \rangle = e \;\mathsf{in}_\tau\; e \\
 & & & \mid \mathsf{new}\,\alpha \approx \tau \;\mathsf{in}_\tau\; e \mid \{e\}^+_\tau \mid \{e\}^-_\tau \mid \mathsf{case}\,e{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}_\tau\; e \\
 & & & \mid \mathsf{pickle}\,e \mid \psi(e) \mid \mathsf{unpickle}\,x \Leftarrow e \;\mathsf{in}\; e \;\mathsf{else}_\tau\; e \\
\text{environments} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\kappa
\end{array}
$$

Figure 11.1.: Syntax of $\lambda^{\omega}_{\mathrm{SA}\Psi}$

Environment Validity $\boxed{\Gamma \vdash \Box}$

$$(\textsc{Nempty})\ \frac{}{\cdot \vdash \Box}$$

$$(\textsc{Ntype})\ \frac{\Gamma \vdash \kappa : \Box}{\Gamma, \alpha{:}\kappa \vdash \Box}\ (\alpha \notin \mathrm{Dom}(\Gamma)) \qquad (\textsc{Nterm})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma, x{:}\tau \vdash \Box}\ (x \notin \mathrm{Dom}(\Gamma))$$

Figure 11.2.: $\lambda^{\omega}_{\mathrm{SA}\Psi}$ environment validity

11.1.1. Environments

A typing *environment* Γ is a finite ordered sequence of variables associated with classifiers.[1] It collects assumptions about the kinds of free type variables, and the types of free term variables. We usually omit the initial empty environment when writing out such a sequence, and we write Γ_1, Γ_2 for appending two environments, with the obvious meaning.

The *environment validity* judgement $\Gamma \vdash \Box$ (Figure 11.2) defines well-formedness of a typing environment Γ. It holds when every type and kind appearing in it is well-formed with respect to the preceding subenvironment, and all variables bound in it are distinct. Because of the latter property, well-formed environments can be interpreted as partial functions, and we use functional notation $\Gamma(z)$ to denote the classifier associated with the (term or type) variable z, respectively. $\mathrm{Dom}(\Gamma)$ denotes the set of variables bound by Γ.

The type system is set up such that derivability of a judgement implies validity of the environment involved:

Proposition 1 (Environment Validity)**.** *Every derivation $\Gamma \vdash \mathcal{J}$, where $\mathcal{J}$ stands for any of the judgements of the system, contains a subderivation $\Gamma \vdash \Box$.*

11.1.2. Kinds

The kind language consists of ground kind Ω, higher-order kinds formed from standard dependent products and sums, and singleton and abstraction kinds. We distinguish ground and abstraction kinds as *base* kinds, because singletons may only be formed over these base kinds.

[1] We use the term *environment* instead of the more common *context* to avoid confusion with evaluation contexts.

Kind Validity $\boxed{\Gamma \vdash \kappa : \square}$

$$(\textsc{Komega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega : \square} \qquad (\textsc{Kabs})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) : \square} \qquad (\textsc{Ksing})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) : \square}$$

$$(\textsc{Kpi})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \square} \qquad (\textsc{Ksigma})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \square}$

$$(\textsc{KQomega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega \equiv \Omega : \square}$$

$$(\textsc{KQabs})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \square} \qquad (\textsc{KQsing})\ \frac{\Gamma \vdash \tau \equiv \tau' : \hat{\kappa} \quad \Gamma \vdash \hat{\kappa} \equiv \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \equiv \mathrm{S}_{\hat{\kappa}'}(\tau') : \square}$$

$$(\textsc{KQpi})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \quad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \equiv \Pi\alpha{:}\kappa_1'.\kappa_2' : \square}$$

$$(\textsc{KQsigma})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \quad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \equiv \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}$$

Kind Inclusion $\boxed{\Gamma \vdash \kappa \leq \kappa' : \square}$

$$(\textsc{KSomega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega \leq \Omega : \square}$$

$$(\textsc{KSabs})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \mathrm{A}(\tau') : \square} \qquad (\textsc{KSsing})\ \frac{\Gamma \vdash \tau \equiv \tau' : \hat{\kappa} \quad \Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \mathrm{S}_{\hat{\kappa}'}(\tau') : \square}$$

$$(\textsc{KSabs-left})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \Omega : \square} \qquad (\textsc{KSsing-left})\ \frac{\Gamma \vdash \tau : \hat{\kappa} \quad \Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \hat{\kappa}' : \square}$$

$$(\textsc{KSpi})\ \frac{\Gamma \vdash \kappa_1' \leq \kappa_1 : \square \quad \Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \square \quad \Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2' : \square}$$

$$(\textsc{KSsigma})\ \frac{\Gamma \vdash \kappa_1 \leq \kappa_1' : \square \quad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \leq \kappa_2' : \square \quad \Gamma \vdash \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \leq \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}$$

Figure 11.3.: $\lambda^{\omega}_{\mathrm{SA}\Psi}$ kind validity, equivalence and inclusion

However, higher-order singletons over arbitrary kind are definable (Section 11.2.2). Likewise, higher-order abstraction kinds can be defined (Section 13.1.2).

Since singleton and abstraction kinds contain types, well-formedness and equivalence of kinds is not purely syntactic. The *kind validity* judgement $\Gamma \vdash \kappa : \Box$ (Figure 11.3) specifies when a kind κ is well-formed under a given environment Γ. It is mostly standard [SH06], with the exception of abstraction kinds, which follow exactly the same rules as singletons, however.

The *kind equivalence* judgement $\Gamma \vdash \kappa \equiv \kappa' : \Box$ (Figure 11.3) defines an equivalence relation on kinds. It is defined in a straightforward inductive manner, modulo equivalence of constituent types (rules KSING and KABS).

The *kind inclusion* judgement $\Gamma \vdash \kappa \leq \kappa' : \Box$ (Figure 11.3) on the other hand defines a preorder on kinds, regulating when a type of the more specific subkind κ may be used in contexts requiring kind κ'. Subkinding is induced by singletons, hence we will discuss it in Section 11.2.

Kind equivalence and subkinding imply well-formedness of the involved kinds (therefore the suggestive pseudo classifier "$: \Box$"):

Proposition 2 (Validity of Kind Judgements)**.**

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*.*
2. *If* $\Gamma \vdash \kappa \leq \kappa' : \Box$*, then* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*.*

For some rules (namely KSPI and KSSIGMA) maintaining this property requires explicit premises.

Other basic structural properties are the following:

Proposition 3 (Reflexivity of Kind Judgements)**.**

1. *If* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa : \Box$*.*
2. *If* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \leq \kappa : \Box$*.*

Proposition 4 (Transitivity of Kind Judgements)**.**

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \equiv \kappa'' : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa'' : \Box$*.*
2. *If* $\Gamma \vdash \kappa \leq \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \leq \kappa'' : \Box$*, then* $\Gamma \vdash \kappa \leq \kappa'' : \Box$*.*

Proposition 5 (Symmetry of Kind Equivalence)**.**
If $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \vdash \kappa' \equiv \kappa : \Box$*.*

Proposition 6 (Antisymmetry of Kind Inclusion)**.**
If and only if $\Gamma \vdash \kappa \leq \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \leq \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*

Note that transitivity and symmetry have to be proved, as they are not enforced by explicit rules, as one might expect.

11.1.3. Types

The type language is a mostly standard version of System F_ω. Besides the usual F_ω types, $\lambda^{\omega}_{SA\Psi}$ provides type products $\langle \tau_1, \tau_2 \rangle$ and respective projections, $\tau{\cdot}1$ and $\tau{\cdot}2$. Moreover, the ground type constant Ψ is used to classify pickles. Note that the calculus is impredicative, which is crucial to be able to express dynamics (Section 10.4).

To ease representation, we sometimes use auxiliary syntactic classes of *paths* and *constructors*, which both are subclasses of types. They are defined in Figure 11.7.

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\textsc{Tvar})\ \frac{\Gamma \vdash \Box}{\Gamma \vdash \alpha : \Gamma(\alpha)} \qquad (\textsc{Tpsi})\ \frac{\Gamma \vdash \Box}{\Gamma \vdash \Psi : \Omega}$$

$$(\textsc{Tarrow})\ \frac{\Gamma \vdash \tau_1 : \Omega \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad (\textsc{Ttimes})\ \frac{\Gamma \vdash \tau_1 : \Omega \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 : \Omega}$$

$$(\textsc{Tuniv})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa_1.\tau_2 : \Omega} \qquad (\textsc{Texist})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa_1.\tau_2 : \Omega}$$

$$(\textsc{Tlambda})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2}{\Gamma \vdash \lambda \alpha{:}\kappa_1.\tau_2 : \Pi \alpha{:}\kappa_1.\kappa_2} \qquad (\textsc{Tapp})\ \frac{\Gamma \vdash \tau_1 : \Pi \alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\, \tau_2 : \kappa_2[\tau_2/\alpha]}$$

$$(\textsc{Tpair})\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \Box}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Tfst})\ \frac{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}1 : \kappa_1} \qquad (\textsc{Tsnd})\ \frac{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]}$$

$$(\textsc{Text-sing})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau)}$$

$$(\textsc{Text-pi})\ \frac{\Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2' \quad \Gamma, \alpha{:}\kappa_1 \vdash \tau\, \alpha : \kappa_2 \quad \Gamma \vdash \Pi \alpha{:}\kappa_1.\kappa_2' : \Box}{\Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Text-sigma})\ \frac{\Gamma \vdash \tau{\cdot}1 : \kappa_1 \quad \Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \Box}{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Tsub})\ \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash \kappa \leq \kappa' : \Box}{\Gamma \vdash \tau : \kappa'}$$

Figure 11.4.: $\lambda^{\omega}_{\mathrm{SA}\Psi}$ type validity

Well-formedness of types is specified by the *type validity* judgement $\Gamma \vdash \tau : \kappa$ (Figure 11.4), which classifies type τ with kind κ under environment Γ. Most of the rules are standard, with the provision that type functions and type products are dependently kinded by Π and Σ-kinds (rules Tlambda and Tpair). Also, due to the presence of subkinding, there is a subsumption rule (rule Tsub). Less familiar are the extensional rules Text-sing, Text-pi and Text-sigma, which allow deriving more precise singleton kinds and will be discussed in Section 11.2.

Type equivalence is defined by a judgement $\Gamma \vdash \tau \equiv \tau' : \kappa$ (Figure 11.4). Due to singletons, equivalence depends on the environment as well as the kind at which the types are considered (Section 11.2). Most are straightforward syntax-directed rules, plus the standard symmetry and transitivity rule (reflexivity is admissible, see below), and a subsumption rule (rule TQsub). Again there are extensional rules (TQext-sing, TQext-pi and TQext-sigma), which are the core of the singleton mechanism, in that they allow deriving additional type equivalences from singleton kinds. They will be explained in Section 11.2.

Note that the type equivalence rules do not include any $\beta\eta$-equivalences on higher-order types

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\text{TQVAR})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \alpha \equiv \alpha : \Gamma(\alpha)} \qquad (\text{TQPSI})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Psi \equiv \Psi : \Omega}$$

$$(\text{TQARROW})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \Omega}$$

$$(\text{TQTIMES})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \Omega}$$

$$(\text{TQUNIV})\ \frac{\Gamma \vdash \kappa \equiv \kappa' : \square \quad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \equiv \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TQEXIST})\ \frac{\Gamma \vdash \kappa \equiv \kappa' : \square \quad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \equiv \exists \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TQLAMBDA})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \quad \Gamma, \alpha{:}\kappa_1 \vdash \tau \equiv \tau' : \kappa_2}{\Gamma \vdash \lambda \alpha{:}\kappa_1.\tau \equiv \lambda \alpha{:}\kappa_1'.\tau' : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQAPP})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Pi \alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Gamma \vdash \tau_1\, \tau_2 \equiv \tau_1'\, \tau_2' : \kappa_2[\tau_2/\alpha]}$$

$$(\text{TQPAIR})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1 \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2[\tau_1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau_1', \tau_2' \rangle : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQFST})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1} \qquad (\text{TQSND})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]}$$

$$(\text{TQEXT-SING})\ \frac{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau'') \quad \Gamma \vdash \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}$$

$$(\text{TQEXT-PI})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau\, \alpha \equiv \tau'\, \alpha : \kappa_2 \quad \Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2' \quad \Gamma \vdash \tau' : \Pi \alpha{:}\kappa_1.\kappa_2''}{\Gamma \vdash \tau \equiv \tau' : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQEXT-SIGMA})\ \frac{\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1 \quad \Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQSYMM})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau' \equiv \tau : \kappa} \qquad (\text{TQTRANS})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \quad \Gamma \vdash \tau' \equiv \tau'' : \kappa}{\Gamma \vdash \tau \equiv \tau'' : \kappa}$$

$$(\text{TQSUB})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \quad \Gamma \vdash \kappa \leq \kappa' : \square}{\Gamma \vdash \tau \equiv \tau' : \kappa'}$$

Figure 11.5.: $\lambda^{\omega}_{\mathrm{SA}\Psi}$ type equivalence

Type Inclusion $\boxed{\Gamma \vdash \tau \leq \tau' : \kappa}$

$$(\text{TSEQUIV})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau \leq \tau' : \kappa}$$

$$(\text{TSARROW})\ \frac{\Gamma \vdash \tau_1' \leq \tau_1 : \Omega \quad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' : \Omega}$$

$$(\text{TSTIMES})\ \frac{\Gamma \vdash \tau_1 \leq \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' : \Omega}$$

$$(\text{TSUNIV})\ \frac{\Gamma \vdash \kappa' \leq \kappa : \Box \quad \Gamma, \alpha{:}\kappa' \vdash \tau \leq \tau' : \Omega \quad \Gamma \vdash \forall \alpha{:}\kappa.\tau : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \leq \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TSEXIST})\ \frac{\Gamma \vdash \kappa \leq \kappa' : \Box \quad \Gamma, \alpha{:}\kappa \vdash \tau \leq \tau' : \Omega \quad \Gamma \vdash \exists \alpha{:}\kappa'.\tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \leq \exists \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TSTRANS})\ \frac{\Gamma \vdash \tau \leq \tau' : \kappa \quad \Gamma \vdash \tau' \leq \tau'' : \kappa}{\Gamma \vdash \tau \leq \tau'' : \kappa}$$

Figure 11.6.: $\lambda^{\omega}_{\text{SA}\Psi}$ type inclusion

paths	π	$::=$	$\alpha \mid \pi\,\tau \mid \pi{\cdot}1 \mid \pi{\cdot}2 \mid \Psi \mid \tau \to \tau \mid \tau \times \tau \mid \forall \alpha{:}\kappa.\tau \mid \exists \alpha{:}\kappa.\tau$
constructors	χ	$::=$	$\alpha \mid \Psi \mid \lambda \alpha{:}\kappa.\tau \mid \chi\tau \mid \langle \tau, \tau \rangle \mid \chi{\cdot}1 \mid \chi{\cdot}2$

Figure 11.7.: Paths and constructors

– they are admissible in the presence of singleton kinds [SH06] (Section 11.2.3).

The *type inclusion* judgement $\Gamma \vdash \tau \leq \tau' : \kappa$ (Figure 11.6) defines the subtyping relation. Subtyping is induced solely by subkinding on the kind annotations of quantified type variables (rules TSUNIV and TSEXIST). The remaining rules are largely standard again. At higher kinds, subtyping coincides with equivalence (via rule TSEQUIV). We leave out higher-order subtyping, because (1) it is not required to express ML modules: in terms of modules it would correspond to subtyping on parameterised signatures – but those do not even exist in ML, and (2) to the best of our knowledge, there is no known algorithm for higher-order subtyping in the presence of singleton kinds.

Like with kinds, type equivalence and subtyping judgements imply well-formedness of the involved types. Similarly, the kind derived is always valid under the given environment:

Proposition 7 (Validity of Type Judgements)**.**

1. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \kappa : \Box$*.*
2. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\Gamma \vdash \kappa : \Box$*.*
3. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$*, then* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\Gamma \vdash \kappa : \Box$*.*

Again, some rules have explicit premises to maintain these properties (rules TQPAIR, TQEXT-PI, TQEXT-SIGMA, TSUNIV, TSEXIST).

We have similar structural properties as for kinds:

Proposition 8 (Reflexivity of Type Judgements)**.**

1. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau : \kappa$*.*
2. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \leq \tau : \kappa$*.*

Proposition 9 (Antisymmetry of Type Inclusion)**.**
If and only if $\Gamma \vdash \tau \leq \tau' : \kappa$ *and* $\Gamma \vdash \tau' \leq \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*

Proving the latter property has to be done by inversion on the subtyping algorithm presented in Section 11.4.4 and depends on its completeness.

Transitivity and symmetry hold trivially, because they are stated by explicit rules (rules TQtrans, TStrans, and TQsymm).

11.1.4. Terms

The term level consists of the usual F_ω terms, plus the extensions for type analysis, type generation, coercions, and pickling that we introduced in the previous chapter. The term language will be described in Chapter 12, along with the last judgement of the type system, *term validity* $\Gamma \vdash e : \tau$, which specifies well-formedness of terms e (Section 12.1).

11.2. Singletons

Singleton kinds are the most intricate feature of the $\lambda^{\omega}_{\mathrm{SA}\Psi}$ type system. Since they are also less standard than the rest of the basic system, and because our notion of abstraction kinds was inspired by them, we will give a brief presentation of their semantics.

Singleton kinds have substantial ramifications on the type system and its meta theory, e.g. by inducing dependent kinds and subkinding, and making type equivalence dependent on both the environment and the classifying kind. Finding an algorithmic formulation and proving it decidable becomes non-trivial. Fortunately, Stone & Harper have already done all the hard work of sorting out these issues [SH06]. To harvest their hard work, we have set up our system as close as possible to theirs. Our language is almost identical to theirs, with mostly straightforward extensions relative to their term language that can mostly be treated as sugar for constants of base types. We also adopt the well-formedness and equivalence rules of their system with only cosmetic modifications. This allows us to re-use their results almost directly.

In the following, we illustrate the main ideas of their system, recast in our calculus. For discussion of the finer points we encourage the reader to refer to their article, and Stone's thesis [Sto00].

11.2.1. Ground Singletons

Singletons come with three central rules:

$$(\textsc{Text-sing})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau)} \qquad (\textsc{KSsing-left})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \hat{\kappa} : \Box}$$

$$(\textsc{TQext-sing})\ \frac{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau'') \qquad \Gamma \vdash \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}$$

The first rule is the introduction rule for singletons: it allows the kind of any base type to be strengthened to singleton kind. Application of this rule is a step also known as *selfification* [HL94]. Intuitively – and referring back to the ML context – it is motivated by the obvious desire to make programs like the following type check:

```
structure M :> (sig type t end) = (struct type t = int end)
functor F (X : sig type t = M.t end) = ...
structure N = F M
```

A possible $\lambda^{\omega}_{\mathrm{SA}\Psi}$ translation is:

$$\begin{array}{l}\mathsf{let}\langle t_M, \_\rangle = \langle \mathit{int}, \diamond\rangle_{\exists t:\Omega.1}\ \mathsf{in}\\ \mathsf{let}\, F = \lambda X : \mathrm{S}_\Omega(t_M).\cdots\ \mathsf{in}\\ F\, t_M\end{array}$$

Here, the type t_M has the *natural* kind Ω, but should of course match the kind $\mathrm{S}_\Omega(t_M)$ required by the functor parameter signature – otherwise the functor application would be ill-typed. Thanks to selfification, the kind of t_M can be strengthened accordingly.

Conversely, the second of the above rules, KSsing-left, enables forgetting a singleton kind via subsumption. It corresponds to the basic subtyping rule of the ML signature language: a signature with an abstract type specification "type t" can always be matched by a signature "type t = τ" with a concrete specification. For example, the following program should be valid:

```
functor F (X : sig type t end) = ...
structure M :> (sig type t = int end) = (struct type t = int end)
structure N = F M
```

A $\lambda^{\omega}_{\mathrm{SA}\Psi}$ interpretation is:

$$\begin{array}{l}\mathsf{let}\, F = \lambda t_X{:}\Omega.\cdots\ \mathsf{in}\\ \mathsf{let}\langle t_M, \_\rangle = \langle \mathit{int}, \diamond\rangle_{\exists t:\mathrm{S}_\Omega(\mathit{int}).1}\ \mathsf{in}\\ F\, t_M\end{array}$$

The subtyping rule allows the kind $\mathrm{S}_\Omega(\mathit{int})$ of t_M to be weakened to Ω, such that the functor application is well-typed.

The most important rule is the third, TQext-sing: given two arbitrary types with the same singleton kind, they can be derived to be equivalent.[2] Thanks to this rule, terms corresponding to code like the following become well-typed:

```
structure M :> (sig type t = int; val f : t → int end) = ...
fun g (n : int) = M.f n
```

Consider a $\lambda^{\omega}_{\mathrm{SA}\Psi}$ encoding:

$$\begin{array}{l}\mathsf{let}\langle t_M, f_M\rangle = \langle\cdots\rangle_{\exists t:\mathrm{S}_\Omega(\mathit{int}).\, t\to\mathit{int}}\ \mathsf{in}\\ \lambda n{:}\mathit{int}.f_M\, n\end{array}$$

In this example, the function f_M requires an argument of type t_M, which is different from the type *int* assigned to n. However, since both types have kind $\mathrm{S}_\Omega(\mathit{int})$ (the latter by selfification), they are equivalent according to the above rule and n can also be assigned the former type by the usual subsumption rule for equivalent types.

Note how the rule TQext-sing makes type equivalence dependent on the kind: without the singleton kind information we would generally be unable to derive the equivalence. Stone & Harper give the following example of two types that actually exhibit different behaviour with respect to equivalence, dependent on the observing kind:

[2] Stone & Harper use an asymmetric equivalence rule of the form $\dfrac{\Gamma \vdash \tau : \mathrm{S}(\tau')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}(\tau')}$ instead. The symmetric formulation is slightly more convenient for our purposes.

$$
\begin{array}{rcl}
\mathrm{S}(\tau : \Omega) & := & \mathrm{S}_\Omega(\tau) \\
\mathrm{S}(\tau : \mathrm{A}(\tau')) & := & \mathrm{S}_{\mathrm{A}(\tau')}(\tau) \\
\mathrm{S}(\tau : \mathrm{S}_{\hat{\kappa}}(\tau')) & := & \mathrm{S}_{\hat{\kappa}}(\tau') \\
\mathrm{S}(\tau : \Pi\alpha{:}\kappa_1.\kappa_2) & := & \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2) \\
\mathrm{S}(\tau : \Sigma\alpha{:}\kappa_1.\kappa_2) & := & \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha])
\end{array}
$$

Figure 11.8.: Higher-order singletons

$\tau_1 = \lambda\alpha{:}\Omega.\alpha$
$\tau_2 = \lambda\alpha{:}\Omega.\mathit{int}$

Both these types can be assigned kind $\Pi\alpha{:}\Omega.\Omega$, under which they clearly differ. However, by subsumption both also can be given the contravariant superkind $\Pi\alpha{:}\mathrm{S}_\Omega(\mathit{int}).\Omega$, under which they are equivalent, because the only valid argument is type *int*, for which they both deliver the same result!

11.2.2. Higher-Order Singletons

Singleton kinds are only defined on types of base kind. What about higher-order types? For example, how can the following ML signature be expressed?

sig type α pair $= \alpha \times \alpha$ **end**

The specification for type pair would correspond to a singleton $\mathrm{S}(\lambda\alpha{:}\Omega.\alpha{\times}\alpha)$. The kind of the contained type function is $\Omega \to \Omega$, however (i.e. $\Pi\alpha{:}\Omega.\Omega$). Thus we need singletons at higher-order kind to express it.

Fortunately, it is not necessary to introduce higher-order singletons as primitive – they can be defined inductively as syntactic sugar. Figure 11.8 gives a respective definition following Stone & Harper [SH06]. The only exception are abstraction kinds, which require primitive support (Section 11.3.1).

Higher-order singletons are defined such that generalised versions of the singleton rules presented previously can be shown as propositions. These rules are given in Figure 11.9. Here and elsewhere, we will use the convention to name admissible rules by appending an asterisk *.

Theorem 10 (Admissibility of Higher-Order Singleton Rules)**.**
The rules KSING*, TEXT-SING, KQSING*, KSSING*, KSSING-LEFT* *and* TQEXT-SING* *are admissible.*

Establishing the admissibility proofs crucially relies on extensional validity and equivalence rules for higher-order kinds that we have not discussed yet. Namely, the kinding rules TEXT-PI and TEXT-SIGMA (Figure 11.4) allow singleton kind selfification to be pushed through η-expansions. For example, given $\Gamma = \alpha{:}(\Sigma\alpha_1{:}\Omega.\Omega)$, rule TEXT-SIGMA enables deriving the most precise kind $\Gamma \vdash \alpha : \mathrm{S}_\Omega(\alpha{\cdot}1) \times \mathrm{S}_\Omega(\alpha{\cdot}2)$. Analogously, with the extensional rule TEXT-PI, we can derive $\beta{:}(\Pi\alpha_1{:}\Omega.\Omega) \vdash \beta : \Pi\alpha_1{:}\Omega.\mathrm{S}_\Omega(t\,\alpha_1)$, which indicates the kind of a type function that always deliver the same result as β – hence this is exactly the definition of a higher-order singleton at Π-kind, i.e. we have derived $\Gamma \vdash \beta : \mathrm{S}(\beta : \Omega \to \Omega)$.

Similar extensionality rules are necessary for the type equivalence relation (Figure 11.5): rule TQEXT-PI says that two type functions can be considered equivalent (under a given kind) when they deliver the same result for every argument (of the given argument kind). Likewise, rule TQEXT-SIGMA says that pairs are equivalent whenever both their projections are.

Kind Validity $\boxed{\Gamma \vdash \kappa : \Box}$

$$(\text{KSING}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) : \Box}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \Box}$

$$(\text{KQSING}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \equiv \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \equiv \mathrm{S}(\tau' : \kappa') : \Box}$$

Kind Inclusion $\boxed{\Gamma \vdash \kappa \leq \kappa' : \Box}$

$$(\text{KSSING}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \leq \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \mathrm{S}(\tau' : \kappa') : \Box} \qquad (\text{KSSING-LEFT}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \kappa : \Box}$$

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\text{TEXT-SING}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau : \mathrm{S}(\tau : \kappa)}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\text{TQEXT-SING}^*)\ \frac{\Gamma \vdash \tau : \mathrm{S}(\tau'' : \kappa) \qquad \Gamma \vdash \tau' : \mathrm{S}(\tau'' : \kappa)}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}(\tau'' : \kappa)}$$

Figure 11.9.: Admissible rules for higher-order singletons

11.2.3. $\beta\eta$-Equivalences

An interesting side-effect of extensionality is that all $\beta\eta$-rules for higher-order types (Figure 11.10) become admissible as well:

Theorem 11. *The equivalence rules* TQAPP-BETA*, TQLAMBDA-ETA*, TQFST-BETA*, TQSND-BETA* *and* TQPAIR-ETA* *are admissible.*

For example, for typing $\tau = (\lambda\alpha{:}\Omega.\tau_1)\,\tau_2$, we can use extensionality (rule TEXT-PI) and selfification (rule TEXT-SING) to derive $\Pi\alpha{:}\Omega.\mathrm{S}_\Omega(\tau_1)$ as the kind of the λ-expression. The application rule (TAPP) then yields $\mathrm{S}_\Omega(\tau_1)[\tau_2/\alpha]$ as the kind of τ. By rule TQEXT-SING we can finally derive that $\tau \equiv \tau_1[\tau_2/\alpha]$.

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\text{TQ}_{\text{APP-BETA}}{}^*)\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2 \qquad \Gamma \vdash \tau_1 : \kappa_1}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 \equiv \tau_2[\tau_1/\alpha] : \kappa_2[\tau_1/\alpha]}$$

$$(\text{TQ}_{\text{LAMBDA-ETA}}{}^*)\ \frac{\Gamma \vdash \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2\,\alpha \equiv \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQ}_{\text{FST-BETA}}{}^*)\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle\tau_1, \tau_2\rangle{\cdot}1 \equiv \tau_1 : \kappa_1} \qquad (\text{TQ}_{\text{SND-BETA}}{}^*)\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle\tau_1, \tau_2\rangle{\cdot}2 \equiv \tau_2 : \kappa_2}$$

$$(\text{TQ}_{\text{PAIR-ETA}}{}^*)\ \frac{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \langle\tau{\cdot}1, \tau{\cdot}2\rangle \equiv \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

Figure 11.10.: Admissible rules for $\beta\eta$-equivalences

11.3. Abstraction Kinds

Using new, we can generate new types that are "isomorphic" to given representation types. But what precisely do we mean by "isomorphic" to their representation? The simplest answer would be: they are equivalent. In other words, given $\mathsf{new}\,\alpha \approx \tau\ \mathsf{in}\ e$, we could assign α the singleton kind $\mathrm{S}_\Omega(\tau)$. However, in that case, new would be nothing more than a type-let, and α indistinguishable from τ, statically and dynamically. But we want to distinguish them – at least dynamically.

We do so by introducing a new form of kind, which we call *abstraction kind*, written $\mathrm{A}(\tau)$ and employ the typing rule

$$(\text{E}_{\text{NEW}})\ \frac{\Gamma, \alpha{:}\mathrm{A}(\tau) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha \approx \tau\ \mathsf{in}\ e : \tau_2}$$

Similar to a singleton kind $\mathrm{S}_{\hat{\kappa}}(\tau)$, an abstraction kind $\mathrm{A}(\tau)$ expresses that all types that inhabit it are isomorphic to τ. But while singleton types allow the isomorphism to be utilised *implicitly* by the type system, its usage has to be made *explicit* in the case of abstraction kinds. That is, given $\tau : \mathrm{A}(\tau')$, the type τ is *not* equivalent to τ', but values of either type can always be *coerced* into the other (Section 12.5).

An abstraction kind $\mathrm{A}(\tau)$ is a subkind of Ω, thereby enabling type names generated by new to be used not only for coercions, but at the same time denote the generated abstract type. For instance, given

$$\mathsf{new}\,\alpha \approx \mathit{int}\ \mathsf{in}\ \ldots\,\lambda x{:}\alpha.x\,\ldots$$

α has kind $\mathrm{A}(\mathit{int})$. By subsumption however we can also assign $\alpha : \Omega$, which is required to make its use as argument type for the λ-expression well-formed.

11.3.1. Singletons over Abstraction Kinds

Since we extend the Stone/Harper system with abstraction kinds, we have to make sure that the higher-order singleton rules hold for our definition of singletons at abstraction kind. Because abstraction types are opaque there is no extensionality principle that can be employed. Neither

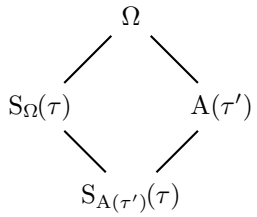


Figure 11.11.: Lattice of base kinds

is it sufficient to rely on the subkinding $\mathrm{A}(\tau) \leq \Omega$ and define

$$\mathrm{S}(\tau : \mathrm{A}(\tau')) \quad := \quad \mathrm{S}_\Omega(\tau)$$

With this definition, it would be impossible to prove rule KSsing-left* correct, because it would not hold for the case $\kappa = \mathrm{A}(\tau')$. Consequently, we have to make singletons over abstraction kind primitive, using labelled singleton notation $\mathrm{S}_{\hat{\kappa}}(\tau)$, where $\hat{\kappa}$ ranges over base kinds Ω and $\mathrm{A}(\tau)$.

Moreover, in order for KSsing* to hold, and to maintain principal kinds, we need to lift subkinding $\mathrm{A}(\tau) \leq \Omega$ to singletons, i.e. establish $\mathrm{S}_{\mathrm{A}(\tau)}(\tau') \leq \mathrm{S}_\Omega(\tau')$ for all suitable τ, τ' (rule KSsing). Figure 11.11 gives an overview of the subkinding relations between the different base kinds.

11.4. Algorithmic Formulations

The specification of the type system presented in the previous sections is declarative. It does not give a ready description of algorithms for checking well-formedness, equivalence and inclusion of types or kinds. However, we need a syntax-directed algorithmic formulation for several purposes:

- to establish a shape invariance property on type equivalence and subtyping, which is needed for parts of the soundness proof in Chapter 12;
- to prove *decidability* of the subtyping judgement, without which the operational semantics of type case would not be well-defined;
- to give a constructive formulation for actually implementing the type system.

11.4.1. Type and Kind Equivalence

The main complication in giving algorithmic formulations of our type system is the type equivalence judgement, on which all other judgements depend. We would like to employ the standard normalise-and-compare approach, but a normalisation strategy for types is not directly obvious, due to singleton kinds, which may introduce additional equivalences based on kind information, and hence yield additional reduction possibilities dependent on kinds. Fortunately, Stone & Harper [SH06] have already given a suitable normalization algorithm, and it is not difficult to extend it to our system.

Figure 11.13 shows the recast algorithm. To distinguish from the declarative formulations, we use the notation $\Gamma \triangleright \mathcal{J}$ instead of $\Gamma \vdash \mathcal{J}$ for all algorithmic judgements.

The algorithmic type equivalence judgement, *type comparison* $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$, takes, as an additional input, the required kind κ. It uses the usual normalize-and-compare strategy to check equivalence. However, the *type normalization* algorithm is more intricate than usual. It is a kind-driven algorithm: for ground types, it first computes the weak-head normal form of the

paths	$\hat{\pi}$	::=	$\alpha \mid \pi\,\tau \mid \pi{\cdot}1 \mid \pi{\cdot}2 \mid \Psi \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau$
path contexts	P	::=	$\_ \mid P\,\tau \mid P{\cdot}1 \mid P{\cdot}2$

Figure 11.12.: Paths

type, also called a *path* (Figure 11.12). It then proceeds normalising remaining constituent types by *path normalization*. For higher-order types, i.e. a function or pair, the judgement looks at the η-expansions. To η-expand a lambda into a normal form it also has to normalize its kind annotation, because it may contain types.

Weak-head normalization is performed by the judgement $\Gamma \triangleright \tau \Rrightarrow \pi$. It implements a straightforward algorithm performing small step β-reductions. Since the types were first η-expanded, and are hence in long-η-normal form, no η-reduction is needed. The only interesting bit is that the reduction also performs singleton reduction, whenever a path's *natural kind* is singleton, as discovered by another auxiliary judgement. Thus normalization is dependent on the environment.

Algorithmic kind equivalence is checked by the *kind comparison* judgement, $\Gamma \triangleright \kappa \equiv \kappa'$ (Figure 11.14). It is based on *kind normalization*, which implements a straightforward syntax-directed algorithm for normalizing all constituent types.

While soundness of the algorithm is easy to show, given a handful of suitable lemmata, completeness is more involved. Stone & Harper show completeness using ternary Kripke-style logical relations [MM91] over set-valued domains that collect all equivalent phrases. We do not have to repeat the complete proofs, but can harvest the fact that our kind language is very similar to their type language. It is only extended with abstraction types (which act like singletons with respect to kind equivalence and like base kinds otherwise) and with a simple labelled variant of singletons (which does not produce much complication). Our type normalization algorithm differs from the paper in a minor detail, namely that for ground types, path normalization may return an arbitrary kind. The reason is that there is no unique base kind in our system, but Ω has non-singleton subkind $\mathrm{A}(\tau)$ that may be recorded in the environment. This change does not affect the proofs, however.

Our type language is significantly richer than Stone & Harper's term language, but can be easily encoded by treating the different ground types as (families of) higher-order constants in standard ways.

Theorem 12 (Soundness of Algorithmic Kind and Type Comparison)**.**

1. *If* $\Gamma \triangleright \tau \Rrightarrow \pi$ *and* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \pi : \kappa$*.*
2. *If* $\Gamma \triangleright \pi \Rrightarrow \pi' \rightrightarrows \kappa'$ *and* $\Gamma \vdash \pi : \kappa$*, then* $\Gamma \vdash \pi \equiv \pi' : \kappa$*.*
3. *If* $\Gamma \triangleright \tau \Rrightarrow \tau' \leftleftarrows \kappa$ *and* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*
4. *If* $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*
5. *If* $\Gamma \triangleright \kappa \Rrightarrow \kappa'$ *and* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*
6. *If* $\Gamma \triangleright \kappa \equiv \kappa'$ *and* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*

Theorem 13 (Completeness of Algorithmic Kind and Type Comparison)**.**

1. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \triangleright \tau \Rrightarrow \tau'' \leftleftarrows \kappa$ *and* $\Gamma \triangleright \tau' \Rrightarrow \tau'' \leftleftarrows \kappa$ *for some* τ''*.*
2. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$*.*

Type Comparison

$\Gamma \rhd \tau \equiv \tau' \leftleftarrows \kappa$ if $\Gamma \rhd \tau \Rrightarrow \tau'' \leftleftarrows \kappa$ and $\Gamma \rhd \tau' \Rrightarrow \tau'' \leftleftarrows \kappa$

Type Normalization

$\Gamma \rhd \tau \Rrightarrow \pi' \leftleftarrows \hat{\kappa}$ if $\Gamma \rhd \tau \Rightarrow \pi$ and $\Gamma \rhd \pi \Rrightarrow \pi' \rightrightarrows \hat{\kappa}'$

$\Gamma \rhd \tau \Rrightarrow \pi' \leftleftarrows \mathrm{S}_{\hat{\kappa}}(\tau')$ if $\Gamma \rhd \tau \Rightarrow \pi$ and $\Gamma \rhd \pi \Rrightarrow \pi' \rightrightarrows \hat{\kappa}'$

$\Gamma \rhd \tau \Rrightarrow \lambda\alpha{:}\kappa_1'.\tau' \leftleftarrows \Pi\alpha{:}\kappa_1.\kappa_2$ if $\Gamma \rhd \kappa_1 \Rrightarrow \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \rhd \tau\,\alpha \Rrightarrow \tau' \leftleftarrows \kappa_2$

$\Gamma \rhd \tau \Rrightarrow \langle \tau_1, \tau_2 \rangle \leftleftarrows \Sigma\alpha{:}\kappa_1.\kappa_2$ if $\Gamma \rhd \tau{\cdot}1 \Rrightarrow \tau_1 \leftleftarrows \kappa_1$ and $\Gamma \rhd \tau{\cdot}2 \Rrightarrow \tau_2 \leftleftarrows \kappa_2[\tau{\cdot}1/\alpha]$

Path Normalization

$\Gamma \rhd \alpha \Rrightarrow \alpha \rightrightarrows \Gamma(\alpha)$

$\Gamma \rhd \Psi \Rrightarrow \Psi \rightrightarrows \Omega$

$\Gamma \rhd \tau_1 \to \tau_2 \Rrightarrow \pi_1 \to \pi_2 \rightrightarrows \Omega$ if $\Gamma \rhd \tau_1 \Rrightarrow \pi_1 \leftleftarrows \Omega$ and $\Gamma \rhd \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$

$\Gamma \rhd \tau_1 \times \tau_2 \Rrightarrow \pi_1 \times \pi_2 \rightrightarrows \Omega$ if $\Gamma \rhd \tau_1 \Rrightarrow \pi_1 \leftleftarrows \Omega$ and $\Gamma \rhd \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$

$\Gamma \rhd \forall\alpha{:}\kappa_1.\tau_2 \Rrightarrow \forall\alpha{:}\kappa_1'.\pi_2 \rightrightarrows \Omega$ if $\Gamma \rhd \kappa_1 \Rrightarrow \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \rhd \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$

$\Gamma \rhd \exists\alpha{:}\kappa_1.\tau_2 \Rrightarrow \exists\alpha{:}\kappa_1'.\pi_2 \rightrightarrows \Omega$ if $\Gamma \rhd \kappa_1 \Rrightarrow \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \rhd \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$

$\Gamma \rhd \pi\,\tau \Rrightarrow \pi'\,\tau' \rightrightarrows \kappa_2[\tau/\alpha]$ if $\Gamma \rhd \pi \Rrightarrow \pi' \rightrightarrows \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \rhd \tau \Rrightarrow \tau' \leftleftarrows \kappa_1$

$\Gamma \rhd \pi{\cdot}1 \Rrightarrow \pi'{\cdot}1 \rightrightarrows \kappa_1$ if $\Gamma \rhd \pi \to \pi' \rightrightarrows \Sigma\alpha{:}\kappa_1.\kappa_2$

$\Gamma \rhd \pi{\cdot}2 \Rrightarrow \pi'{\cdot}2 \rightrightarrows \kappa_2[\pi{\cdot}1/\alpha]$ if $\Gamma \rhd \pi \to \pi' \rightrightarrows \Sigma\alpha{:}\kappa_1.\kappa_2$

Head Normalization

$\Gamma \rhd \tau \Rrightarrow \pi$ if $\Gamma \rhd \tau \Rrightarrow_1 \tau'$ and $\Gamma \rhd \tau' \Rrightarrow \pi$

$\Gamma \rhd \tau \Rrightarrow \tau$

Head Reduction

$\Gamma \rhd P[(\lambda\alpha{:}\kappa.\tau_1)\,\tau_2] \Rrightarrow_1 P[\tau_1[\tau_2/\alpha]]$

$\Gamma \rhd P[\langle \tau_1, \tau_2 \rangle{\cdot}1] \Rrightarrow_1 P[\tau_1]$

$\Gamma \rhd P[\langle \tau_1, \tau_2 \rangle{\cdot}2] \Rrightarrow_1 P[\tau_2]$

$\Gamma \rhd P[\pi] \Rrightarrow_1 P[\tau]$ if $\Gamma \rhd \pi : \mathrm{S}_{\hat{\kappa}}(\tau)$

Natural Kinds

$\Gamma \rhd \alpha : \Gamma(\alpha)$

$\Gamma \rhd \Psi : \Omega$

$\Gamma \rhd \tau_1 \to \tau_2 : \Omega$

$\Gamma \rhd \tau_1 \times \tau_2 : \Omega$

$\Gamma \rhd \forall\alpha{:}\kappa_1.\tau_2 : \Omega$

$\Gamma \rhd \exists\alpha{:}\kappa_1.\tau_2 : \Omega$

$\Gamma \rhd \pi\,\tau : \kappa_2[\tau/\alpha]$ if $\Gamma \rhd \pi : \Pi\alpha{:}\kappa_1.\kappa_2$

$\Gamma \rhd \pi{\cdot}1 : \kappa_1$ if $\Gamma \rhd \pi : \Sigma\alpha{:}\kappa_1.\kappa_2$

$\Gamma \rhd \pi{\cdot}2 : \kappa_2[\pi{\cdot}1/\alpha]$ if $\Gamma \rhd \pi : \Sigma\alpha{:}\kappa_1.\kappa_2$

Figure 11.13.: Algorithmic type comparison

Kind Comparison

$\Gamma \triangleright \kappa \equiv \kappa'$	if $\Gamma \triangleright \kappa \Rrightarrow \kappa''$ and $\Gamma \triangleright \kappa' \Rrightarrow \kappa''$

Kind Normalization

$\Gamma \triangleright \Omega \Rrightarrow \Omega$	
$\Gamma \triangleright \mathrm{A}(\tau) \Rrightarrow \mathrm{A}(\tau')$	if $\Gamma \triangleright \tau \Rrightarrow \tau' \leftleftarrows \Omega$
$\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) \Rrightarrow \mathrm{S}_{\hat{\kappa}}(\tau')$	if $\Gamma \triangleright \tau \Rrightarrow \tau' \leftleftarrows \hat{\kappa}$
$\Gamma \triangleright \Pi\alpha{:}\kappa_1.\kappa_2 \Rrightarrow \Pi\alpha{:}\kappa_1'.\kappa_2'$	if $\Gamma \triangleright \kappa_1 \Rrightarrow \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 \Rrightarrow \kappa_2'$
$\Gamma \triangleright \Sigma\alpha{:}\kappa_1.\kappa_2 \Rrightarrow \Sigma\alpha{:}\kappa_1'.\kappa_2'$	if $\Gamma \triangleright \kappa_1 \Rrightarrow \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 \Rrightarrow \kappa_2'$

Figure 11.14.: Algorithmic kind comparison

Kind Matching

$\Gamma \triangleright \Omega \leq \Omega$	
$\Gamma \triangleright \mathrm{A}(\tau) \leq \Omega$	
$\Gamma \triangleright \mathrm{A}(\tau) \leq \mathrm{A}(\tau')$	if $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \Omega$
$\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) \leq \hat{\kappa}'$	if $\Gamma \triangleright \hat{\kappa} \leq \hat{\kappa}'$
$\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) \leq \mathrm{S}_{\hat{\kappa}'}(\tau')$	if $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \hat{\kappa}$ and $\Gamma \triangleright \hat{\kappa} \leq \hat{\kappa}'$
$\Gamma \triangleright \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2'$	if $\Gamma \triangleright \kappa_1' \leq \kappa_1$ and $\Gamma, \alpha{:}\kappa_1' \triangleright \kappa_2 \leq \kappa_2'$
$\Gamma \triangleright \Sigma\alpha{:}\kappa_1.\kappa_2 \leq \Sigma\alpha{:}\kappa_1'.\kappa_2'$	if $\Gamma \triangleright \kappa_1 \leq \kappa_1'$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 \leq \kappa_2'$

Figure 11.15.: Algorithmic kind matching

3. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$, *then* $\Gamma \triangleright \kappa \Rrightarrow \kappa''$ *and* $\Gamma \triangleright \kappa' \Rrightarrow \kappa''$ *for some* κ''.

4. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$, *then* $\Gamma \triangleright \kappa \equiv \kappa'$.

From soundness and completeness of the algorithmic equivalence judgements it follows immediately that the equivalence relations on kinds and types are decidable:

Corollary 14 (Decidability of Equivalence)**.**

1. *Given* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$, *it is decidable whether* $\Gamma \vdash \kappa \equiv \kappa : \Box$ *holds.*

2. *Given* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$, *it is decidable whether* $\Gamma \vdash \tau \equiv \tau' : \kappa$ *holds.*

11.4.2. Subkinding

Given an algorithm for type equivalence, it is easy to define an algorithm to check subkinding. It is shown in Figure 11.15: *kind checking*, $\Gamma \triangleright \kappa : \Box$, verifies that kind κ is well-formed under environment Γ, using straightforward structural recursion. The cases directly mirror the declarative rules. To check singletons and abstraction kinds, the algorithmic type equivalence check is employed on the constituent types.

Theorem 15 (Soundness of Algorithmic Kind Matching)**.**
If $\Gamma \triangleright \kappa \leq \kappa'$ *and* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$, *then* $\Gamma \vdash \kappa \leq \kappa' : \Box$.

Kind Checking

$\Gamma \triangleright \Omega : \square$	
$\Gamma \triangleright \mathrm{A}(\tau) : \square$	if $\Gamma \triangleright \tau \leftleftarrows \Omega$
$\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) : \square$	if $\Gamma \triangleright \hat{\kappa} : \square$ and $\Gamma \triangleright \tau \leftleftarrows \hat{\kappa}$
$\Gamma \triangleright \Pi\alpha{:}\kappa_1.\kappa_2 : \square$	if $\Gamma \triangleright \kappa_1 : \square$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 : \square$
$\Gamma \triangleright \Sigma\alpha{:}\kappa_1.\kappa_2 : \square$	if $\Gamma \triangleright \kappa_1 : \square$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 : \square$

Kind Synthesis

$\Gamma \triangleright \alpha \rightrightarrows \mathrm{S}(\alpha : \Gamma(\alpha))$	if $\alpha \in \mathrm{Dom}(\Gamma)$
$\Gamma \triangleright \Psi \rightrightarrows \mathrm{S}_\Omega(\Psi)$	
$\Gamma \triangleright \tau_1 \to \tau_2 \rightrightarrows \mathrm{S}_\Omega(\tau_1 \to \tau_2)$	if $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$
$\Gamma \triangleright \tau_1 \times \tau_2 \rightrightarrows \mathrm{S}_\Omega(\tau_1 \times \tau_2)$	if $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$
$\Gamma \triangleright \forall\alpha{:}\kappa_1.\tau_2 \rightrightarrows \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2)$	if $\Gamma \triangleright \kappa_1 : \square$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \leftleftarrows \Omega$
$\Gamma \triangleright \exists\alpha{:}\kappa_1.\tau_2 \rightrightarrows \mathrm{S}_\Omega(\exists\alpha{:}\kappa_1.\tau_2)$	if $\Gamma \triangleright \kappa_1 : \square$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \leftleftarrows \Omega$
$\Gamma \triangleright \lambda\alpha{:}\kappa_1.\tau_2 \rightrightarrows \Pi\alpha{:}\kappa_1.\kappa_2$	if $\Gamma \triangleright \kappa_1 : \square$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \rightrightarrows \kappa_2$
$\Gamma \triangleright \tau_1\ \tau_2 \rightrightarrows \kappa_2[\tau_2/\alpha]$	if $\Gamma \triangleright \tau_1 \rightrightarrows \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \triangleright \tau_2 \leftleftarrows \kappa_1$
$\Gamma \triangleright \langle\tau_1, \tau_2\rangle \rightrightarrows \kappa_1 \times \kappa_2$	if $\Gamma \triangleright \tau_1 \rightrightarrows \kappa_1$ and $\Gamma \triangleright \tau_2 \rightrightarrows \kappa_2$
$\Gamma \triangleright \tau{\cdot}1 \rightrightarrows \kappa_1$	if $\Gamma \triangleright \tau \rightrightarrows \Sigma\alpha{:}\kappa_1.\kappa_2$
$\Gamma \triangleright \tau{\cdot}2 \rightrightarrows \kappa_2[\tau{\cdot}1/\alpha]$	if $\Gamma \triangleright \tau \rightrightarrows \Sigma\alpha{:}\kappa_1.\kappa_2$

Kind Analysis

$\Gamma \triangleright \tau \leftleftarrows \kappa$	if $\Gamma \triangleright \tau \rightrightarrows \kappa'$ and $\Gamma \triangleright \kappa' \leq \kappa$

Figure 11.16.: Algorithmic kind synthesis

Theorem 16 (Completeness of Algorithmic Kind Matching)**.**
If $\Gamma \vdash \kappa \leq \kappa' : \square$*, then* $\Gamma \triangleright \kappa \leq \kappa'$*.*

Again, decidability follows immediately:

Corollary 17 (Decidability of Kind Inclusion)**.**
Given $\Gamma \vdash \kappa : \square$ *and* $\Gamma \vdash \kappa' : \square$*, it is decidable whether* $\Gamma \vdash \kappa \leq \kappa' : \square$ *holds.*

11.4.3. Kind Synthesis

We now give an algorithm for checking well-formedness of types and synthesising suitable kinds. Figure 11.16 shows the respective algorithm, including two auxiliary ones. *Kind checking*, $\Gamma \triangleright \kappa : \square$, verifies that kind κ is well-formed under environment Γ, using straightforward structural recursion. *Kind synthesis*, $\Gamma \triangleright \tau \rightrightarrows \kappa$ computes the principal kind κ for a given τ. It also uses straightforward structural recursion, returning singleton kinds for ground types. The auxiliary *type analysis* judgement $\Gamma \triangleright \tau \leftleftarrows \kappa$ checks whether a type has a given input kind κ by first inferring the principal kind and then testing whether they are in subkinding relation.

Theorem 18 (Soundness of Algorithmic Kind Synthesis)**.**

1. *If* $\Gamma \triangleright \kappa : \square$ *and* $\Gamma \vdash \square$*, then* $\Gamma \vdash \kappa : \square$*.*
2. *If* $\Gamma \triangleright \tau \rightrightarrows \kappa$ *and* $\Gamma \vdash \square$*, then* $\Gamma \vdash \tau : \kappa$*.*

Type Matching

$\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$	if $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$and $\Gamma \triangleright \pi \sqsubseteq \pi'$
$\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$	if $\kappa \neq \Omega$ and $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$

Path Matching

$\Gamma \triangleright \chi \sqsubseteq \chi'$	if $\Gamma \triangleright \chi \equiv \chi' \leftleftarrows \Omega$
$\Gamma \triangleright \tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'$	if $\Gamma \triangleright \tau_1' \leq \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leq \tau_2' \leftleftarrows \Omega$
$\Gamma \triangleright \tau_1 \times \tau_2 \sqsubseteq \tau_1' \times \tau_2'$	if $\Gamma \triangleright \tau_1' \leq \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leq \tau_2' \leftleftarrows \Omega$
$\Gamma \triangleright \forall \alpha{:}\kappa.\tau \sqsubseteq \forall \alpha{:}\kappa'.\tau'$	if $\Gamma \triangleright \kappa' \leq \kappa$ and $\Gamma, \alpha{:}\kappa' \triangleright \tau \leq \tau' \leftleftarrows \Omega$
$\Gamma \triangleright \exists \alpha{:}\kappa.\tau \sqsubseteq \exists \alpha{:}\kappa'.\tau'$	if $\Gamma \triangleright \kappa \leq \kappa'$ and $\Gamma, \alpha{:}\kappa \triangleright \tau \leq \tau' \leftleftarrows \Omega$

Figure 11.17.: Algorithmic type matching

3. *If* $\Gamma \triangleright \tau \leftleftarrows \kappa$ *and* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma \vdash \tau : \kappa$.

Theorem 19 (Completeness of Algorithmic Kind Synthesis)**.**

1. *If* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma \triangleright \kappa : \Box$.
2. *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \triangleright \tau \rightrightarrows \kappa'$ *and* $\Gamma \vdash \kappa' \leq \mathrm{S}(\tau : \kappa) : \Box$.
3. *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \triangleright \tau \leftleftarrows \kappa$.

Once more, this gives us decidability:

Corollary 20 (Decidability of Type Validity)**.**

1. *Given* τ *and* $\Gamma \vdash \Box$, *it is decidable whether there is a* κ *such that* $\Gamma \vdash \tau : \kappa$ *holds.*
2. *Given* τ *and* $\Gamma \vdash \Box$ *and* $\Gamma \vdash \kappa : \Box$, *it is decidable whether* $\Gamma \vdash \tau : \kappa$ *holds.*

From completeness of kind analysis it also follows immediately that kind synthesis in fact computes principal kinds, and hence that our system enjoys principal (least) kinding:

Corollary 21 (Principality of Kinding)**.** *If* $\Gamma \vdash \tau : \kappa$, *then there is a* κ' *such that for all* κ'' *with* $\Gamma \vdash \tau : \kappa''$ *it holds that* $\Gamma \vdash \kappa' \leq \kappa'' : \Box$.

11.4.4. Subtyping

Finally, we can give an algorithm for deciding subtyping, which we adapt from Stone [Sto00]. Figure 11.17 shows the respective *type matching* algorithm.

Subtyping coincides with equivalence at higher kinds, thus type matching just resorts to checking equivalence for higher kinds. For kind Ω, the types are put into weak-head normal form and matched by the syntax-directed *path matching* judgement.

Soundness of the algorithm is straightforward:

Theorem 22 (Soundness of Algorithmic Type Matching)**.**

1. *If* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$, *then* $\Gamma \vdash \tau \leq \tau' : \kappa$.
2. *If* $\Gamma \triangleright \pi \sqsubseteq \pi'$ *and* $\Gamma \vdash \pi : \Omega$ *and* $\Gamma \vdash \pi' : \Omega$, *then* $\Gamma \vdash \pi \leq \pi' : \Omega$.

Transitive Type Inclusion $\boxed{\Gamma \vdash \tau \leq^* \tau' : \Omega}$

$$(\text{TSEQUIV}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \tau \leq^* \tau' : \Omega}$$

$$(\text{TSARROW}^*)\ \frac{\begin{array}{cc}\Gamma \vdash \tau \equiv \tau_1 \to \tau_2 : \Omega & \Gamma \vdash \tau' \equiv \tau_1' \to \tau_2' : \Omega \\ \Gamma \vdash \tau_1' \leq^* \tau_1 : \Omega & \Gamma \vdash \tau_2 \leq^* \tau_2' : \Omega\end{array}}{\Gamma \vdash \tau \leq^* \tau' : \Omega}$$

$$(\text{TSTIMES}^*)\ \frac{\begin{array}{cc}\Gamma \vdash \tau \equiv \tau_1 \times \tau_2 : \Omega & \Gamma \vdash \tau' \equiv \tau_1' \times \tau_2' : \Omega \\ \Gamma \vdash \tau_1 \leq^* \tau_1' : \Omega & \Gamma \vdash \tau_2 \leq^* \tau_2' : \Omega\end{array}}{\Gamma \vdash \tau \leq^* \tau' : \Omega}$$

$$(\text{TSUNIV}^*)\ \frac{\begin{array}{cc}\Gamma \vdash \tau \equiv \forall \alpha{:}\kappa_1.\tau_2 : \Omega & \Gamma \vdash \tau' \equiv \forall \alpha{:}\kappa_1'.\tau_2' : \Omega \\ \Gamma \vdash \kappa_1' \leq \kappa_1 : \Box & \Gamma, \alpha{:}\kappa_1' \vdash \tau_2 \leq^* \tau_2' : \Omega\end{array}}{\Gamma \vdash \tau \leq^* \tau' : \Omega}$$

$$(\text{TSEXIST}^*)\ \frac{\begin{array}{cc}\Gamma \vdash \tau \equiv \exists \alpha{:}\kappa_1.\tau_2 : \Omega & \Gamma \vdash \tau' \equiv \exists \alpha{:}\kappa_1'.\tau_2' : \Omega \\ \Gamma \vdash \kappa_1 \leq \kappa_1' : \Box & \Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \leq^* \tau_2' : \Omega\end{array}}{\Gamma \vdash \tau \leq^* \tau' : \Omega}$$

Figure 11.18.: Transitive type inclusion

Completeness is more involved: due to quantified types, proving transitivity of the algorithm would depend on a weakening property for the matching algorithm, where kinds in the environment can be replaced by subkinds. Unfortunately, normalisation is not stable under weakening to a singleton kind. In his thesis [Sto00], Stone proves it directly on the algorithmic formulation, proceeding in two steps: first he proves completeness only for the sublanguage of constructors, then for the full language. However, these proofs rely on the predicativity of his system, which ours does not enjoy (Section 11.1).

We hence have to use an indirect proof. We first formulate an alternative subtyping judgement, which has transitivity built in, and prove it sound and complete with respect to the original judgement. We then show that the algorithmic formulation is complete with respect to the transitive form, and consequently, also for the original judgement.

Figure 11.18 gives a transitive formulation of the subtyping judgement. It is easy to show sound:

Proposition 23 (Soundness of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \vdash \tau \leq \tau' : \Omega$*.*

It is not hard to show that the judgement is in fact transitive:

Proposition 24 (Transitivity of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq^* \tau' : \Omega$ *and* $\Gamma \vdash \tau' \leq^* \tau'' : \Omega$*, then* $\Gamma \vdash \tau \leq^* \tau'' : \Omega$*.*

From transitivity we can then conclude completeness:

Proposition 25 (Completeness of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq \tau' : \Omega$*, then* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*.*

Finally, we show completeness of the type matching algorithm by relating it to the transitive judgement:

Theorem 26 (Completeness of Algorithmic Type Matching)**.**

1. *If* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$*.*
2. *If* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \triangleright \tau \Rrightarrow \pi$ *and* $\Gamma \triangleright \tau' \Rrightarrow \pi'$ *and* $\Gamma \triangleright \pi \sqsubseteq \pi'$*.*
3. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$*, then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$*.*

From completeness of kind and type matching follows decidability:

Proposition 27 (Decidability of Type Inclusion)**.**
Given $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, it is decidable whether* $\Gamma \vdash \tau \leq \tau' : \kappa$ *holds.*

11.5. Related Work

11.5.1. Typed Lambda Calculi

There exists countless works on λ-calculi, type systems, and their use as models for programming languages. We refer the reader to two standard references, namely Barendregt's handbook chapter on Lambda Calculi with Types [Bar92], which presents the basic theory from untyped calculus to dependent types and the lambda cube, and Pierce' book on Types and Programming Languages [Pie02], which gives an introduction to, and extensive overview of, the use of lambda calculus as an idealised programming language, particularly exploring different dimensions of type systems.

11.5.2. Singletons

Singleton types were first proposed as a means for expressing modular specifications by Aspinall [Asp95, Asp97]. He introduces higher-order singletons $\mathrm{S}(M : A)$ as primitive,[3] which produces the minor technical complication that there is an infinite sequence of subtypes $\mathrm{S}(M : A) \geq \mathrm{S}(M : \mathrm{S}(M : A)) \geq \ldots$ that has to be tamed by making all its elements equivalent with an additional rule. Interestingly, his system does not have a separate equivalence judgement, but encodes it as validity of the form $\Gamma \vdash M : \mathrm{S}(M' : A)$. The system does not have η-equivalence.

As already mentioned, our system is closely based on the work of Stone & Harper [SH06]. They introduce a simple lambda calculus with singleton types. The kind language of our system is almost identical to their type language, except for the addition of abstraction kinds (and singletons thereof). We adopted our type and kind equivalence algorithm directly from their article, with only minor modifications to encompass the extensions. Our type language is richer than their term language. However, with respect to equivalence and subkinding, the built-in type constructors of our type language can simply be treated as additional higher-order constants that do not affect the meta theory and algorithms. A variant of their system with a slightly richer type language practically identical to ours (except for the lack of existential types) appears in Stone's chapter on Type Definitions [Sto05].

One substantial extension that our system adds to the picture is subtyping, which would correspond to a notion of "subterming", or value subsumption, that does not exist in Stone & Harper's system. However, in his thesis [Sto00] Stone studies a more complicated system

[3] He uses the syntax $\{M\}_A$ for this purpose.

featuring singleton kinds *and* singleton types, and subkinding as well as subtyping. He even discusses an extension with intensional type analysis. In a first approximation, our system can be seen as a subsystem of that one, except for the addition of abstraction kinds and type generativity. However, his system is predicative, while ours allows fully impredicative type formation. That is crucial for our purposes, because we could not express dynamics (and hence packages) with a predicative restriction – a package is represented by the polymorphic type $\exists\alpha{:}\Omega.\alpha$, and must be able to embed other polymorphic values. Thanks to predicativity, Stone can prove completeness of the subtyping algorithm directly on the algorithmic formulation, while we were forced to take the detour through an auxiliary transitive subtyping judgement to show transitivity of the algorithm.

Singletons have also been employed by Dreyer, Crary & Harper in their work on type systems for higher-order modules [DCH03]. The language they describe consists of two layered languages: terms and modules. Type equivalence depends on module equivalence, due to projection of types from modules. Singletons appear in the module language, which induces dependent module types. Earlier work on higher-order module systems used *translucent sums* [HS00, Lil97] or, equivalently, *manifest types* [Ler94, Ler95] instead of singletons. They can basically be interpreted as a restricted form of singletons that can only be introduced in syntactically explicit positions, and thus are somewhat more tractable.

11.5.3. Type Names, Environment and Abstraction Kinds

Many systems deal with type names for which an isomorphism must be recorded in the environment. For example, Stone describes a simple system that deals with type definitions by recording them as special entries in the environment [Sto05], but the types are just synonyms for existing types.

In previous work [Ros03a], we also used special environment entries of the form $\alpha \approx \tau$ to record generated type names and their representations. Vytiniotis, Washburn & Weirich use a separate type isomorphism environment for the same purpose [VWW05]. In both works, type names can only be used for coercing between the type and its representation.

Dreyer also expresses type abstraction with explicit type generation to cope with recursive modules [Dre07]. For that purpose, he separates generation from definition of abstract type names, and hence requires multiple different environment entries that realise a simple effect system to ensure linearity of the bindings. See Section 12.10.3 for a more detailed comparison.

Where type names merely act as synonyms, singleton kinds represent a more uniform alternative to specialised environment entries. We are not aware of any previous work that has applied the same generalisation to non-synonym type names, thus arriving at a notion equivalent to our abstraction kinds.

11.6. Summary

- The type language of $\lambda^{\omega}_{\mathrm{SA}\Psi}$ is impredicative and higher-order with the usual built-in constructors, and including type pairs.
- The kind level has singleton kinds and abstraction kinds and is dependently kinded.
- Singletons induce subkinding and subtyping.
- Higher-order singletons are definable as a derived concept.
- Abstraction kinds classify explicit type isomorphisms.

- Standard $\beta\eta$-rules for type functions and pairs are admissible thanks to singletons.
- The judgements of the system are Environment, Kind and Type Validity, Type and Kind Equivalence, Type and Kind Inclusion.
- We give algorithms for deciding all judgements of the type system.

12. The Term Language

In this chapter we present the term level of our calculus, its typing rules, and the operational semantics. Again, most of the system is fairly standard, and we concentrate the discussion on our additions: type analysis, type generation, coercions, and pickling.

We then state standard soundness properties for the operational semantics, which are essentially straightforward given the type system meta theory developed in the last chapter. However, we first have to know that the type system is decidable, since unpickling has to perform type checking for verification.[1] Hence, we develop an algorithmic formulation of the typing rules.

Finally, we give a moderate abstraction result that establishes that type generation and coercions are indeed sufficient to ensure abstraction safety. Similar to the last chapter, auxiliary propositions and all proofs can be found in Appendix D.

12.1. Typing

Figure 12.1 shows the syntax of our term language. It consists of the standard constructions from System F_ω – functions, pairs, polymorphic abstractions, existential packages – enriched with type analysis, type generation, coercions, and pickles with respective operations. For symmetry with existential types, pairs are eliminated by a pattern matching let construct instead of projection. We discuss the other extensions individually in the following sections.

Well-formedness and typing of terms is specified by the rules of the *term validity* judgement, $\Gamma \vdash e : \tau$ (Figure 12.2). Besides the non-standard extensions, whose semantics we will describe in the following sections, the rules bear little surprises. The only exception is rule ECLOSE for forming existentials: unlike the standard approach to existential types, we do not require a type annotation on existential packages. The typing rule keeps the type τ_2 fully transparent. This corresponds closely to the situation in ML, where a structure expression always is assigned a fully transparent signature. The type can then be made "more abstract" by using subsumption. Thanks to singleton kinds, we can still assign a principal type to existentials, simply by deriving singleton kind for the witness τ.

In several places it will become convenient to have a unit type available. Also, a conventional let-expression will come in handy. We assume the following encodings:

$$\begin{array}{rcl} 1 & := & \forall\alpha{:}\Omega.\,\alpha \to \alpha \\ \diamond & := & \lambda\alpha{:}\Omega.\,\lambda x{:}\alpha.\,x \\ \mathsf{let}\,x = e_1\;\mathsf{in}\;e_2 & := & \mathsf{let}\langle x, \_\rangle = \langle e_1, \diamond\rangle\;\mathsf{in}\;e_2 \end{array}$$

Similar to the other judgements of the system (Chapter 11), the structure of the typing rules implies that the classifying type τ derived by the judgement is valid under the environment Γ:

Proposition 28 (Validity of Term Validity Rules)**.**
If $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \tau : \Omega$*.*

[1] As noted in Chapter 3, the type system of Alice ML is in fact undecidable. For the calculus, we prefer a "cleaner" approach, especially since abstract signatures, which make Alice ML undecidable, are rather of cursory utility.

$$\begin{array}{rrcl} \text{terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e, e\rangle \mid \mathsf{let}\langle x, x\rangle = e \;\mathsf{in}\; e \\ & & & \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle \tau, e\rangle \mid \mathsf{let}\langle \alpha, x\rangle = e \;\mathsf{in}_\tau\; e \\ & & & \mid \mathsf{new}\,\alpha \approx \tau \;\mathsf{in}_\tau\; e \mid \{e\}^+_\tau \mid \{e\}^-_\tau \mid \mathsf{case}\,e{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}_\tau\; e \\ & & & \mid \mathsf{pickle}\,e \mid \psi(e) \mid \mathsf{unpickle}\,x \Leftarrow e \;\mathsf{in}\; e \;\mathsf{else}_\tau\; e \end{array}$$

Figure 12.1.: Syntax of $\lambda^{\omega}_{\mathrm{SA}\Psi}$

Term Validity $\boxed{\Gamma \vdash e : \tau}$

$$(\textsc{Evar})\ \frac{\Gamma \vdash \square}{\Gamma \vdash x : \Gamma(x)}$$

$$(\textsc{Elambda})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad (\textsc{Eapp})\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2}$$

$$(\textsc{Epair})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

$$(\textsc{Eproj})\ \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\langle x_1, x_2\rangle = e_1 \;\mathsf{in}\; e_2 : \tau}$$

$$(\textsc{Egen})\ \frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\tau} \qquad (\textsc{Einst})\ \frac{\Gamma \vdash e : \forall\alpha{:}\kappa.\tau \quad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\,\tau_2 : \tau[\tau_2/\alpha]}$$

$$(\textsc{Eclose})\ \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash \langle \tau, e\rangle : \exists\alpha{:}\kappa.\tau_2}$$

$$(\textsc{Eopen})\ \frac{\Gamma \vdash e_1 : \exists\alpha{:}\kappa.\tau_2 \quad \Gamma, \alpha{:}\kappa, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathsf{let}\langle \alpha, x\rangle = e_1 \;\mathsf{in}_\tau\; e_2 : \tau}$$

$$(\textsc{Enew})\ \frac{\Gamma, \alpha{:}\mathrm{A}(\tau_1) \vdash e : \tau_2 \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha \approx \tau_1 \;\mathsf{in}_{\tau_2}\; e : \tau_2}$$

$$(\textsc{Eup})\ \frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \{e\}^+_{\tau_1} : \tau_1} \qquad (\textsc{Edn})\ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \{e\}^-_{\tau_1} : \tau_2}$$

$$(\textsc{Ecase})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{case}\,e_1{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_2 \;\mathsf{else}_\tau\; e_3 : \tau}$$

$$(\textsc{Epickle})\ \frac{\Gamma \vdash e : \exists\alpha{:}\Omega.\alpha}{\Gamma \vdash \mathsf{pickle}\,e : \Psi} \qquad (\textsc{Epsi})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \psi(v) : \Psi}\ (\mathrm{FV}(v) \subseteq \mathrm{Dom}(\Gamma))$$

$$(\textsc{Eunpickle})\ \frac{\Gamma \vdash e_1 : \Psi \quad \Gamma, x{:}(\exists\alpha{:}\Omega.\alpha) \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{unpickle}\,x \Leftarrow e_1 \;\mathsf{in}\; e_2 \;\mathsf{else}_\tau\; e_3 : \tau}$$

$$(\textsc{Esub})\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau' : \Omega}{\Gamma \vdash e : \tau'}$$

Figure 12.2.: $\lambda^{\omega}_{\mathrm{SA}\Psi}$ term validity

12.1.1. Principality

A technical problem with the rich type language in our calculus is that it does not automatically lead to *principal* (or *least*) types. There are two problems.

The Join Problem. The rules for branching constructs, ECASE and EUNPICKLE, demand that the type of both branches are the same. That would require finding the least upper bound (*lub*) of the respective principal types. However, a lub does not generally exist in the higher-order system we face. To see why, first note that contravariance makes lubs interdependent with greatest lower bounds (*glb*s). Moreover, finding the lub or glb of a quantified type involves finding the lub or glb of kinds. Now consider the following two kinds:

$$\Sigma\alpha : \Omega \to \Omega.\, \mathrm{S}(\alpha\, 1) \qquad \text{and} \qquad \Sigma\alpha : \Omega \to \Omega.\, \mathrm{S}(1)$$

These kinds have at least two common subkinds, namely $\Sigma\alpha : \Omega{\to}\mathrm{S}(1).\, \mathrm{S}(1)$ (which is equivalent to $\Sigma\alpha : \Omega{\to}\mathrm{S}(1).\, \mathrm{S}(\alpha\, 1)$) and $\Sigma\alpha : (\Pi\beta{:}\Omega.\mathrm{S}(\beta)).\, \mathrm{S}(1)$ (equivalent to $\Sigma\alpha : (\Pi\beta{:}\Omega.\mathrm{S}(\beta)).\, \mathrm{S}(\alpha\, 1)$), neither of which is more general than the other.

The Avoidance Problem. In rules EOPEN and ENEW the bound type variable α may not appear free in the resulting type τ, in order to avoid it escaping its scope. This requires finding the least supertype of the body's principal type that does not mention α. Unfortunately, in general such a type does not uniquely exist [DCH03, GP98]. For example, consider a type containing the kind

$$(\Omega \to \mathrm{S}(\alpha)) \times \mathrm{S}(\alpha)$$

The obvious choice for a superkind avoiding α would be $(\Omega \to \Omega) \times \Omega$, but that completely forgets the type sharing between the second component and the result of the first. There is a more precise choice that still records it, namely $\Sigma\beta : (\Omega \to \Omega).\, \mathrm{S}(\beta\, \tau)$, for some arbitrary τ. However, since τ is arbitrary, there are actually infinitely many choices, and all of them are incomparable with β being abstract.

In order to side-step these issues, we equip the critical expression forms with type annotations that restore principality. These annotations are necessary solely for the sake of decidable type checking (Section 12.7), they are not otherwise required for soundness. We will omit the annotations most of the time, to avoid clutter.

12.2. Reduction

Figure 12.3 defines an operational semantics for $\lambda^{\omega}_{\mathrm{SA}\Psi}$. Following the established Wright & Felleisen approach [WF94], we employ a small step formulation using evaluation contexts E. An evaluation context is a expression with exactly one occurrence of a hole marker $\_$. We write $E[e]$ for the result of replacing the hole in E with the expression e, possibly capturing free variables of E, provided the result is well-formed.

Reduction is defined on *configurations* C, which are expressions paired with a *heap*. Heaps capture the type names generated with new (Section 12.4).

The one-step reduction relation is denoted by $\to$. It is the least binary relation on contexts satisfying the rules in Figure 12.3. We write $C \to C'$ if C reduces to C' in one step according to this relation, and $C \to^* C'$ if it does so in zero or more steps.

Values and Contexts

$$
\begin{array}{lrcl}
\text{values} & v & ::= & \lambda x{:}\tau.e \mid \langle v, v\rangle \mid \lambda\alpha{:}\kappa.e \mid \langle \tau, v\rangle \mid \{v\}^+_\tau \mid \psi(v) \\
\text{contexts} & E & ::= & \_ \mid E\,e \mid v\,E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathsf{let}\langle x, x\rangle = E \;\mathsf{in}\; e \\
 & & & \mid E\,\tau \quad \mid \langle \tau, E\rangle \quad \mid \mathsf{let}\langle \alpha, x\rangle = E \;\mathsf{in}\; e \\
 & & & \mid \{E\}^+_\tau \mid \{E\}^-_\tau \mid \mathsf{case}\,E{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}\; e \\
 & & & \mid \mathsf{pickle}\,E \mid \mathsf{unpickle}\,x \Leftarrow E \;\mathsf{in}\; e \;\mathsf{else}\; e \\
\text{heaps} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tau) \\
\text{configurations} & C & ::= & \Delta; e
\end{array}
$$

Reduction rules

$$
\begin{array}{lll}
(\textsc{Rapp}) & \Delta; E[(\lambda x{:}\tau.e)\,v] & \to \Delta; E[e[v/x]] \\
(\textsc{Rproj}) & \Delta; E[\mathsf{let}\langle x_1, x_2\rangle{=}\langle v_1, v_2\rangle \;\mathsf{in}\; e] & \to \Delta; E[e[v_1/x_1][v_2/x_2]] \\
(\textsc{Rinst}) & \Delta; E[(\lambda\alpha{:}\kappa.e)\,\tau] & \to \Delta; E[e[\tau/\alpha]] \\
(\textsc{Ropen}) & \Delta; E[\mathsf{let}\langle \alpha, x\rangle{=}\langle \tau, v\rangle \;\mathsf{in}_{\tau'}\; e] & \to \Delta; E[e[\tau/\alpha][v/x]] \\
(\textsc{Rnew}) & \Delta; E[\mathsf{new}\,\alpha{\approx}\tau \;\mathsf{in}_{\tau'}\; e] & \to \Delta, \alpha{:}\mathrm{A}(\tau); E[e] \\
(\textsc{Rcancel}) & \Delta; E[\{\{v\}^+_{\tau_1}\}^-_{\tau_2}] & \to \Delta; E[v] \\
(\textsc{Rcase1}) & \Delta; E[\mathsf{case}\,v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_1[v/x]] \quad \text{if } \Delta \vdash \tau_1 \le \tau_2 : \Omega \\
(\textsc{Rcase2}) & \Delta; E[\mathsf{case}\,v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_2] \quad \text{if } \Delta \nvdash \tau_1 \le \tau_2 : \Omega \\
(\textsc{Rpickle}) & \Delta; E[\mathsf{pickle}\,v] & \to \Delta; E[\psi(v)] \\
(\textsc{Runpickle1}) & \Delta; E[\mathsf{unpickle}\,x{\Leftarrow}\psi(v) \;\mathsf{in}\; e_1 \;\mathsf{else}\; e_2] & \to \Delta; E[e_1[v/x]] \quad \text{if } \Delta \vdash v : \exists\alpha{:}\Omega.\alpha \\
(\textsc{Runpickle2}) & \Delta; E[\mathsf{unpickle}\,x{\Leftarrow}\psi(v) \;\mathsf{in}\; e_1 \;\mathsf{else}\; e_2] & \to \Delta; E[e_2] \quad \text{if } \Delta \nvdash v : \exists\alpha{:}\Omega.\alpha
\end{array}
$$

Figure 12.3.: Reduction

Values are defined as a subset of expressions as usual and consist of λ-abstractions, and tuples or existentials with nested values. Furthermore, they include values coerced to abstract type, and pickles.

The reduction relation given defines a standard call-by-value, left-to-right evaluation order. It consists of the usual β-rules for the standard constructs, plus appropriate rules for dealing with the special features of $\lambda^{\omega}_{\mathrm{SA}\Psi}$. These will be explained in detail in the following sections.

12.3. Type Analysis

Of our extensions to bare λ-calculus terms, type analysis probably is the most standard, covered extensively in literature [HM95, DRW95, Gle99, CW99, TSS00, Wei02, CWM02, VWW05]. However, most type analysis constructs described in literature are of considerable complexity, where we strive for something as simple as possible, just about rich enough to support a package-like mechanism.

12.3.1. Semantics

Our type case simply takes two types and checks whether they are in subtyping relation. Hence, there are two reduction rules, one for each possible branch (Figure 12.3). In those rules, the subtyping check is expressed by a suitable side condition containing a subtyping judgement. We use the notation $\Gamma \nvdash \mathcal{J}$ to express that the judgement $\Gamma \vdash \mathcal{J}$ does *not* hold.

The environment Δ under which the check is performed reflects the current type heap, as we will discuss in Section 12.4.2. Apart from generated type names the considered environment is always empty, because reduction is never performed under binders, and the involved types are hence necessarily closed up to the type heap.

The typing rule ECASE for case is straightforward (Figure 12.2). Obviously, the types to be compared need to be of ground kind Ω. But note that this and the need to give a witness term e_1 for τ_1 is not an actual restriction: types at arbitrary kind can still be compared freely, courtesy of the following abbreviation:

$$\begin{aligned}&\mathsf{case}\,\tau_1 \le \tau_2 : \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ :=\\ &\quad \mathsf{case}\, f : (\forall\alpha:\kappa\to\Omega.\,\alpha\,\tau_2\to 1)\ \mathsf{of}\ x : (\forall\alpha:\kappa\to\Omega.\,\alpha\,\tau_1\to 1).\ e_1\ \mathsf{else}\ e_2\end{aligned}$$

where $f = \lambda\alpha:\kappa\to\Omega.\,\lambda x:\alpha\,\tau_2.\,\diamond$. The compared types τ_1 and τ_2 appear in inverted order in the expansion, because they are in contravariant position. Using this sugar it is possible to express simple generic functions, although we will not exploit that here.[2]

12.3.2. Packages

Thanks to type case, we can define packages as follows:

$$\begin{aligned}\mathsf{package} &:= \exists\alpha{:}\Omega.\alpha\\ \mathsf{pack}\,e:\tau &:= (\lambda x{:}\,\mathsf{package}\,.x)\,\langle\tau, e\rangle\\ \mathsf{unpack}\,x \Leftarrow e_1 : \tau\ \mathsf{in}\ e_2\ \mathsf{else}_\tau\ e_3 &:= \mathsf{let}\langle\alpha, x'\rangle = e_1\ \mathsf{in}\ \mathsf{case}\,x'{:}\alpha\ \mathsf{of}\ x{:}\tau.e_2\ \mathsf{else}_\tau\ e_3\end{aligned}$$

The identity function used in the expansion of pack acts as a type constraint, forcing the existential to actually have type package. In order to be able to deal with failure programatically, unpack has to be modelled as a branching construct, similar to unpickle.

It is not difficult to see that we can derive the following evaluation rules:

$$\begin{aligned}\Delta; E[\mathsf{unpack}\,x{\Leftarrow}(\mathsf{pack}\,v:\tau):\tau'\ \mathsf{in}\ e_2\ \mathsf{else}_\tau\ e_3] &\to^* \Delta; E[e_2[v/x]] \quad \text{if } \Gamma_H \vdash \tau \le \tau' : \Omega\\ \Delta; E[\mathsf{unpack}\,x{\Leftarrow}(\mathsf{pack}\,v:\tau):\tau'\ \mathsf{in}\ e_2\ \mathsf{else}_\tau\ e_3] &\to^* \Delta; E[e_3] \quad \text{if } \Gamma_H \not\vdash \tau \le \tau' : \Omega\end{aligned}$$

The package notation enables us to translate the dynamic type sharing example from Section 4.7, in a relatively direct way, from Alice ML into $\lambda^{\omega}_{\mathrm{SA}\Psi}$ (omitting the else branches from unpickle and unpack expressions):

```
unpickle p1 ⇐ campaignfile in
  unpack campaign ⇐ p1 : (∃world : Ω. (1 → world) × (world → 1) × ··· ) in
    let⟨world, getWorld, setWorld, ...⟩ = campaign in
    unpickle p2 ⇐ snapshotfile in
      unpack snapshot ⇐ p1 : (∃world : S(world). world × date × ··· ) in
        let⟨world, w, d, ...⟩ = snapshot in
        ...
```

The sharing is expressed by a singleton kind in the target signature for *snapshot* that refers to the type variable *world* taken from the *campaign* structure.

One aspect we omit here for simplicity is the automatic signature refinement discussed in Section 4.4.1. According to the discussion, the expression

[2] For full genericity it would be necessary to be able to inspect type terms inductively. But we are not concerned with generic programming here.

$$\mathsf{pack}\langle int, 1\rangle : \exists\alpha{:}\Omega.\alpha$$

should yield the existential

$$\langle \exists\alpha{:}\mathrm{S}_\Omega(int).1, \langle int, \diamond\rangle\rangle$$

whereas the definition given above only produces

$$\langle \exists\alpha{:}\Omega.1, \langle int, \diamond\rangle\rangle$$

Implementing such refinement generically would require to extract and separate the kind information $\mathrm{Stat}(\tau)$ embedded in an arbitrary type τ – which basically means defining a proper module language by an appropriate inductive phase splitting translation. We give a sketch of such a definition in Appendix B. Short of that, a properly refined package type can always be given directly.

12.3.3. Recursion

Our type case operator is a variant of Girard's J operator [Gir71, HM99] (Section 12.10.1). Girard showed that a profound implication of the J operator (and the corresponding loss of parametericity, Section 10.5) is that – unlike plain F_ω– a calculus providing it is no longer terminating. In particular, it is easy to construct a fixed point operator, say on type $int \to int$ [ACPP89]. In our system the following is one possible definition (for better readability, we use the package notation defined in the previous section):

$$\begin{aligned}\mathsf{fix}_{int\to int} = \lambda f : (int{\to}int) \to (int{\to}int).\\ \quad \mathsf{let}\ f' = \lambda p : \mathsf{package}\,.\ \lambda x : int.\\ \qquad \mathsf{unpack}\ f' \Leftarrow p : \mathsf{package} \to (int{\to}int)\ \mathsf{in}\ f\,(f'\,p)\,x\ \mathsf{else}\ 0\\ \quad \mathsf{in}\ f'\,(\mathsf{pack}\ f' : \mathsf{package} \to (int{\to}int))\end{aligned}$$

It is easy to check that this term is well-typed, and that its application to a suitable function meets the standard fixed point equation. The basic trick of the encoding is that the recursive type usually needed to construct fix is avoided by hiding the recursive occurrence inside the package type – thanks to type case, we can still recover it dynamically.

Analogously, we can define a fixed point operator $\mathsf{fix}_{\tau_1\to\tau_2}$ for any types τ_1, τ_2 where τ_2 is inhabited. We will write $\bot_\tau$ for a diverging computation of type τ, which can easily be constructed from $\mathsf{fix}_{1\to\tau}$ as follows:

$$\bot_\tau = \mathsf{fix}_{1\to\tau}(\lambda f : 1 \to \tau.\ \lambda x : 1.\ f\,x)\,\diamond$$

With this encoding, recursion and divergence are restricted to inhabited types, for which a witness value is available (as a default value for the else branch of the unpack inside fix, although it will never be reached). It is easy to see that a general fixed point operator cannot be expressed, because the type case does not allow the construction of terms that would otherwise be uninhabited – the constituent term e_3 needs to be a witness of the whole expression's result type [HM99]. For our purposes, this limited form of recursion is good enough. It would be straightforward to add a primitive fixed point operator to the calculus if needed.

12.4. Type Generation

The next feature we turn to is type generation. It is supported in our calculus through a single syntactic construct, namely new expressions.

12.4.1. Type Heap

Evaluating new is supposed to generate a fresh type. We employ an additional *heap* in which generated types are allocated. The heap is simply a list of type variables classified by abstraction kinds. A configuration $\Delta; e$ then represents a snapshot of a computation. We consider the heap Δ in a configuration as a (multiple) binder, and hence configurations are equivalent modulo α-renaming.

Typing can be extended to configurations by the following straightforward rule, which takes advantage of the fact that heaps are a syntactic subclass of environments:

$$(\textsc{Cvalid})\ \frac{\Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \Delta; e : \tau}$$

Thanks to Environment Validity (Section 11.1.1), the premise enforces that Δ is well-formed.

As an example of a computation involving the heap, consider the following reduction sequence:

$$\begin{array}{ll} & \cdot; (\lambda f{:}\mathit{int}{\to}\mathit{int}.f\ (\mathsf{new}\,\alpha \approx \mathit{int}\ \mathsf{in}\ f\ 4))\ (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta \approx \mathit{bool}\ \mathsf{in}\ x) \\ \to (\textsc{Rapp}) & \cdot; (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta \approx \mathit{bool}\ \mathsf{in}\ x)\ (\mathsf{new}\,\alpha \approx \mathit{int}\ \mathsf{in}\ (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta' \approx \mathit{bool}\ \mathsf{in}\ x)\ 4) \\ \to (\textsc{Rnew}) & \alpha{:}\mathrm{A}(\mathit{int}); (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta \approx \mathit{bool}\ \mathsf{in}\ x)\ ((\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta' \approx \mathit{bool}\ \mathsf{in}\ x)\ 4) \\ \to (\textsc{Rapp}) & \alpha{:}\mathrm{A}(\mathit{int}); (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta \approx \mathit{bool}\ \mathsf{in}\ x)\ (\mathsf{new}\,\beta' \approx \mathit{bool}\ \mathsf{in}\ 4) \\ \to (\textsc{Rnew}) & \alpha{:}\mathrm{A}(\mathit{int}), \beta'{:}\mathrm{A}(\mathit{bool}); (\lambda x{:}\mathit{int}.\,\mathsf{new}\,\beta \approx \mathit{bool}\ \mathsf{in}\ x)\ 4 \\ \to (\textsc{Rapp}) & \alpha{:}\mathrm{A}(\mathit{int}), \beta'{:}\mathrm{A}(\mathit{bool}), \beta{:}\mathrm{A}(\mathit{bool}); 4 \end{array}$$

It terminates with a configuration of the form $\Delta; v$, where $v = 4$ is the resulting value, and $\Delta = \alpha{:}\mathrm{A}(\mathit{int}), \beta'{:}\mathrm{A}(\mathit{bool}), \beta{:}\mathrm{A}(\mathit{bool})$ represents the created type heap containing the three new types named α, β and β'. Note that, thanks to the variable convention and appropriate α-conversion, the generated types are all different, although β and β' both stem from the same expression and have the same representation *bool*.

It should be noted that type generation is fully dynamic, i.e. the number of type names generated is neither determined statically nor bounded. The following non-terminating expression will actually generate an infinite number of types:

$$(\mathsf{fix}_{1\to 1}\,\lambda f{:}1{\to}1.\lambda x{:}1.\,\mathsf{new}\,\alpha \approx \tau\ \mathsf{in}\ f\ x)\ \diamond$$

12.4.2. Analysing Generated Types

The whole point about introducing dynamically generated types is to make them available in the subtyping check performed by type case. Thus we must allow the types in a case expression to refer to names from the heap. Consequently, the reduction rules for type case must embrace the heap. More precisely, the side conditions expressing the subtyping check must be using a suitable *heap environment* reflecting the bindings in the heap. Fortunately, since heaps have been defined such that they form a syntactic subclass of environments, we can use a heap Δ as an environment directly in the rules.

Let us return to our previous example, which resulted in

$$\Delta; v \quad = \quad \alpha{:}\mathrm{A}(\mathit{int}), \beta'{:}\mathrm{A}(\mathit{bool}), \beta{:}\mathrm{A}(\mathit{bool}); 4$$

Obviously, a dynamic type test comparing the generated types, for instance,

$$\Delta; \mathsf{case}\,\beta \leq \beta' : \Omega\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$

(using the abbreviation defined in Section 12.3.1) will evaluate to 0 as desired, because $\Delta \vdash \beta \leq \beta' : \square$ cannot be derived – the two types are incompatible. Note how this behaviour corresponds to the static semantics of generative functors in ML, where

```
functor F () = struct type t = bool end :> sig type t end
structure M1 = F ()
structure M2 = F ()
```

yields incompatible types M1.t and M2.t.

In Section 12.9 we will state formally that generated types maintain abstraction safety.

12.5. Coercions

To construct values of abstract type, values of the corresponding representation type have to be coerced into the new type. Our calculus demands such coercions to be explicit. An extensive discussion of this design choice is only possible when considering higher-order abstraction, and hence appears in Section 13.6.1.

There are two symmetric expression forms, with dual typing rules:

$$(\textsc{Eup})\ \frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau : \mathrm{A}(\tau')}{\Gamma \vdash \{e\}^+_\tau : \tau} \qquad (\textsc{Edn})\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau : \mathrm{A}(\tau')}{\Gamma \vdash \{e\}^-_\tau : \tau'}$$

Upward and downward coercions can be seen as the introduction and elimination forms of abstract types. An upward coercion $\{v\}^+_\tau$, where v is a value, is considered a value. Such a value can be eliminated by the respective downward coercion, thanks to the Rcancel reduction rule.

Due to the side conditions in the rules, coercions are only available within the lexical scope of the corresponding type generator, thus the transition across abstraction boundaries can only be triggered within the lexical scope of the generator, which is the crucial property ensuring abstraction. Moreover, the type system guarantees that only coercions belonging to the same type can cancel each other out.

Note that the type τ used for coercion is *not* necessarily a type variable, although ultimately only new can create types of abstraction kind, which have to be referred to as variables. However, due to the higher-order nature of our type system, τ may still be an arbitrary type expression, as in

$$\lambda\alpha : \Omega \to \mathrm{A}(\mathit{int}).\, \lambda x : \alpha\ \mathit{bool}.\, \{13\}^+_{\alpha\ \mathit{bool}}$$

12.6. Pickling

The last feature of our calculus is pickling. Pickles are values of type Ψ, as the corresponding typing rules reveal (Figure 12.2). The content of a pickle, however, always has to be an expression of type $\exists\alpha{:}\Omega.\alpha$ – that is, a package.

The rule Epsi does *not* require the embedded value v to be well-formed, hence modelling potentially malformed pickles. To simplify manners however, the rule syntactically restricts pickle values to be closed with respect to the environment Γ. This is to maintain the usual Variable Containment property, which states that all free variables of a well-formed term are bound by the environment (Appendix C.1).

Reduction rules for pickling implement the expected semantics (Figure 12.3): pickle just reduces to a pickle value (rule Rpickle), and unpickle branches dependent on whether the pickle contains a well-formed package (rules Runpickle1 and Runpickle2). Similar to the subtyping check performed by type case (Section 12.3), this is expressed by a side condition consisting of a suitable typing judgement for the pickled value. And like for type case, the side condition has to embrace the type heap, because pickles may refer to abstract types (in the value as well as the type of the embedded package).

To clarify the interplay between dynamic type checking and dynamic type matching when accessing a pickled package, consider the following four examples (for brevity we omit branches and variable binding of the unpack and unpickle operations):

1. $\mathsf{unpack}\,(\mathsf{unpickle}\ \mathsf{pickle}(\mathsf{pack}\ \lambda x{:}1.\diamond : 1{\to}1)) : 1{\to}1$
2. $\mathsf{unpack}\,(\mathsf{unpickle}\ \mathsf{pickle}(\mathsf{pack}\ \lambda x{:}1.\diamond : 1{\to}1)) : 1{\to}1{\to}1$
3. $\mathsf{unpack}\,(\mathsf{unpickle}\ \psi(\mathsf{pack}\ \lambda x{:}1.x\,\diamond : 1{\to}1)) : 1{\to}1$
4. $\mathsf{unpack}\,(\mathsf{unpickle}\ \mathsf{pickle}(\mathsf{pack}\ \lambda x{:}1.x\,\diamond : 1{\to}1)) : 1{\to}1$

The first three expressions are all (statically) well-typed, but only the first will evaluate successfully. The second will fail due to a dynamic type error in unpack, the third due to a verification error during unpickling. The last example is rejected by the (static) type system, because the pickled value is not denoted by a well-formed expression. Contrast this to the third example, which represents a (statically valid) malformed pickle.

12.7. Algorithmic Type Checking

One of the most unusual aspects of pickling is that the operational semantics of $\lambda^{\omega}_{\mathrm{SA}\Psi}$ may involve type-checking arbitrary expressions. We saw that this is simply expressed as a side condition in the reduction rules RUNPICKLE1 and RUNPICKLE2 (Figure 12.3). To make the rules well-defined however (and thus for soundness), we have to show that this side condition is in fact decidable. That is, we have to give an algorithm for type checking $\lambda^{\omega}_{\mathrm{SA}\Psi}$-terms.

Figure 12.4 shows the type checking algorithm. It consists of three judgements: the syntax-directed *type synthesis* judgement $\Gamma \triangleright e \rightrightarrows \tau$ computes a (principal) type τ for a given term e. The auxiliary *type analysis* judgement $\Gamma \triangleright e \leftleftarrows \tau$ checks a term against a required type τ, up to subtyping. Finally, another auxiliary judgement derives a principal type in weak-head normal form. It is used where a premise requires a type of a particular shape. The main structural difference to the declarative formulation is the usual removal of the subsumption rule, which has been built-in into the rules where it is actually needed.

Correctness of the algorithm is relatively straightforward to show.

Theorem 29 (Soundness of Algorithmic Type Synthesis)**.**
Let $\Gamma \vdash \square$.

1. *If* $\Gamma \triangleright e \rightrightarrows \tau$, *then* $\Gamma \vdash e : \tau$.
2. *If* $\Gamma \triangleright e \rightrightarrows\!\!\!\!\rightarrow \pi$, *then* $\Gamma \vdash e : \pi$.
3. *If* $\Gamma \triangleright e \leftleftarrows \tau$ *with* $\Gamma \vdash \tau : \Omega$, *then* $\Gamma \vdash e : \tau$.

Theorem 30 (Completeness of Algorithmic Type Synthesis)**.**

1. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \rightrightarrows \tau'$ *with* $\Gamma \vdash \tau' \leq \tau : \Omega$.
2. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \rightrightarrows\!\!\!\!\rightarrow \pi$ *with* $\Gamma \vdash \pi \leq \tau : \Omega$.
3. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leq \tau' : \Omega$, *then* $\Gamma \triangleright e \leftleftarrows \tau'$.

Because the algorithm is syntax-directed, it terminates, and hence implies decidability:

Corollary 31 (Decidability of Term Validity)**.**

1. *Given* e *and* $\Gamma \vdash \square$, *it is decidable whether there is a* τ *such that* $\Gamma \vdash e : \tau$ *holds.*
2. *Given* e *and* $\Gamma \vdash \square$ *and* $\Gamma \vdash \tau : \Omega$, *it is decidable whether* $\Gamma \vdash e : \tau$ *holds.*

Type Synthesis

$\Gamma \triangleright x \rightrightarrows \tau$	if $\Gamma(x) = \tau$
$\Gamma \triangleright \lambda x{:}\tau_1.e_2 \rightrightarrows \tau_1 \to \tau_2$	if $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma, x{:}\tau_1 \triangleright e_2 \rightrightarrows \tau_2$
$\Gamma \triangleright e_1\, e_2 \rightrightarrows \tau_1$	if $\Gamma \triangleright e_1 \Rightarrow\!\!\!\!\!\rightarrow \tau_2 \to \tau_1$ and $\Gamma \triangleright e_2 \leftleftarrows \tau_2$
$\Gamma \triangleright \langle e_1, e_2\rangle \rightrightarrows \tau_1 \times \tau_2$	if $\Gamma \triangleright e_1 \rightrightarrows \tau_1$ and $\Gamma \triangleright e_2 \rightrightarrows \tau_2$
$\Gamma \triangleright \mathsf{let}\langle x_1, x_2\rangle = e_1 \text{ in } e_2 \rightrightarrows \tau$	if $\Gamma \triangleright e_1 \Rightarrow\!\!\!\!\!\rightarrow \tau_1 \times \tau_2$ and $\Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \triangleright e_2 \rightrightarrows \tau$
$\Gamma \triangleright \lambda\alpha{:}\kappa_1.e_2 \rightrightarrows \forall\alpha{:}\kappa_1.\tau_2$	if $\Gamma \triangleright \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \triangleright e \rightrightarrows \tau_2$
$\Gamma \triangleright e_1\, \tau_2 \rightrightarrows \tau_1$	if $\Gamma \triangleright e_1 \Rightarrow\!\!\!\!\!\rightarrow \forall\alpha{:}\kappa_2.\tau_1$ and $\Gamma \triangleright \tau_2 \leftleftarrows \kappa_2$
$\Gamma \triangleright \langle \tau_1, e_2\rangle \rightrightarrows \exists\alpha{:}\kappa_1.\tau_2$	if $\Gamma \triangleright \tau_1 \rightrightarrows \kappa_1$ and $\Gamma \triangleright e_2 \rightrightarrows \tau_2$ and $\alpha \notin \mathrm{Dom}(\Gamma)$
$\Gamma \triangleright \mathsf{let}\langle \alpha_1, x_2\rangle = e_1 \text{ in}_\tau\ e_2 \rightrightarrows \tau$	if $\Gamma \triangleright e_1 \Rightarrow\!\!\!\!\!\rightarrow \exists\alpha{:}\kappa_1.\tau_2$ and $\Gamma \triangleright \tau \leftleftarrows \Omega$ and $\Gamma, \alpha_1{:}\kappa_1, x_2{:}\tau_2 \triangleright e_2 \leftleftarrows \tau$
$\Gamma \triangleright \mathsf{new}\,\alpha \approx \tau_1 \text{ in}_\tau\ e_2 \rightrightarrows \tau$	if $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau \leftleftarrows \Omega$ and $\Gamma, \alpha{:}\mathrm{A}(\tau_1) \triangleright e_2 \leftleftarrows \tau$
$\Gamma \triangleright \{e\}^+_{\tau_1} \rightrightarrows \tau_1$	if $\Gamma \triangleright \tau_1 \rightrightarrows \mathrm{S}_{\mathrm{A}(\tau_2)}(\tau_3)$ and $\Gamma \triangleright e \leftleftarrows \tau_2$
$\Gamma \triangleright \{e\}^-_{\tau_1} \rightrightarrows \tau_2$	if $\Gamma \triangleright \tau_1 \rightrightarrows \mathrm{S}_{\mathrm{A}(\tau_2)}(\tau_3)$ and $\Gamma \triangleright e \leftleftarrows \tau_1$
$\Gamma \triangleright \mathsf{case}\, e_1{:}\tau_1 \text{ of } x_2{:}\tau_2.e_2 \text{ else}_\tau\ e_3 \rightrightarrows \tau$	if $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau \leftleftarrows \Omega$ and $\Gamma \triangleright e_1 \leftleftarrows \tau_1$ and $\Gamma, x{:}\tau_2 \triangleright e_2 \leftleftarrows \tau$ and $\Gamma \triangleright e_3 \leftleftarrows \tau$
$\Gamma \triangleright \mathsf{pickle}\, e \rightrightarrows \Psi$	if $\Gamma \triangleright e \leftleftarrows \exists\alpha{:}\Omega.\alpha$
$\Gamma \triangleright \psi(v) \rightrightarrows \Psi$	
$\Gamma \triangleright \mathsf{unpickle}\, x \Leftarrow e_1 \text{ in } e_2 \text{ else}_\tau\ e_3 \rightrightarrows \tau$	if $\Gamma \triangleright e_1 \leftleftarrows \Psi$ and $\Gamma \triangleright \tau \leftleftarrows \Omega$ and $\Gamma, x{:}\exists\alpha{:}\Omega.\alpha \triangleright e_2 \leftleftarrows \tau$ and $\Gamma \triangleright e_3 \leftleftarrows \tau$

Normal Type Synthesis

$\Gamma \triangleright e \Rightarrow\!\!\!\!\!\rightarrow \pi$	if $\Gamma \triangleright e \rightrightarrows \tau$ and $\Gamma \triangleright \tau \Rrightarrow \pi$

Type Analysis

$\Gamma \triangleright e \leftleftarrows \tau$	if $\Gamma \triangleright e \rightrightarrows \tau'$ and $\Gamma \triangleright \tau' \leq \tau \leftleftarrows \Omega$

Figure 12.4.: Algorithmic type synthesis

12.8. Soundness

We are now prepared to state soundness of the type system with respect to the operational semantics. We use the standard approach of proving *preservation* (reduction preserves typing) and *progress* (well-typed terms can always be reduced) [WF94]. With the formal development of the previous chapter, the proofs are almost straightforward.

For showing preservation of cancelled coercions, it is important to know that two abstract types that are known to be equal at kind Ω are actually the same abstract type, with the same representation:

Proposition 32 (Representation Equivalence)**.**
Let $\Gamma \vdash \tau_1 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_2')$.

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2 : \Omega$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_1' \equiv \tau_2' : \Omega$.
2. *If* $\Gamma \vdash \tau_1 \leq \tau_2 : \Omega$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_1' \equiv \tau_2' : \Omega$.

Preservation then is mostly standard:

Theorem 33 (Preservation)**.**

1. *If* $\Delta \vdash e : \tau$ *and* $\Delta; e \to \Delta'; e'$ *with* $E = \_$*, then* $\Delta' \vdash e' : \tau$*.*
2. *If* $\cdot \vdash C : \tau$ *and* $C \to C'$*, then* $\cdot \vdash C' : \tau$*.*

The progress proof requires the usual "canonical forms" lemma, but here respecting the heap:

Proposition 34 (Canonical Values)**.**
Let $\Gamma \vdash v : \tau$*.*

1. *If* $\Gamma \vdash \tau \leq \tau' : \Omega$ *and* $\Gamma \vdash \tau' : \mathrm{A}(\tau_1)$*, then* $v = \{v_1\}^+_{\tau_2}$*.*
2. *If* $\Gamma \vdash \tau \leq \Psi : \Omega$*, then* $v = \psi(v_1)$*.*
3. *If* $\Gamma \vdash \tau \leq \tau_1 \to \tau_2 : \Omega$*, then* $v = \lambda x{:}\tau_1'.e_2$*.*
4. *If* $\Gamma \vdash \tau \leq \tau_1 \times \tau_2 : \Omega$*, then* $v = \langle v_1, v_2 \rangle$*.*
5. *If* $\Gamma \vdash \tau \leq \forall \alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \lambda \alpha{:}\kappa_1'.e_2$*.*
6. *If* $\Gamma \vdash \tau \leq \exists \alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \langle \tau_1, v_2 \rangle$*.*

The formulation of the progress theorem also has to embrace the type heap:

Theorem 35 (Progress)**.**
If $\Delta \vdash e : \tau$*, then either* $e = v$*, or* $(\Delta; e) = (\Delta; E[e_1]) \to (\Delta'; E[e_1']) = (\Delta'; e')$*.*

The proof is mostly standard.

12.9. Opacity

By having proved soundness, we have shown that our calculus actually models a *type safe* language. But we have not yet shown that it also is *abstraction safe* (Section 1.1.1). We do so by formalising suitable notions of *secrecy* and *authentication* and proving them for values of abstract type.

More precisely, we show a simple parametricity property that we call *opacity*. We consider a term e that has free occurrences of a type variable $\alpha : \Omega$ and a term variable $x : \alpha$. The term e can be viewed as a *client* of some abstraction. If we now substitute the free variables by a concrete *implementation*, then reduction of e will proceed disregardless of what implementation we chose – as long as the type τ we substitute for α has abstraction kind (note that α is not assigned abstraction kind, otherwise e would logically be part of the *implementation* of the abstraction, and of course *not* opaque). Consequently, the client cannot have been able to look at the representation type of α or access the value behind x.

However, this only gives secrecy. For authentication, we need to show that e can even use a function from the abstraction that *expects* a value of type α, without changing the outcome of the evaluation either. We model this by allowing e to refer to a second variable $f : \alpha \to 1$. A function of this type can have two possible outcomes: termination (with $\diamond$) or divergence. By assuming that the substituted function terminates on the value substituted for x, but not making any assumptions for other possible values, we can prove authentication by observing that reduction still proceeds uniformly – because if f was applied to a value different from x reduction would be unpredictable. Hence e can not have forged a value of type α.

Opacity, as stated below, combines these characterisations of secrecy and authentication into a single formulation. It is a rather technical formulation, but the proof is essentially a straightforward induction on derivations. Interestingly, the proof is completely independent from the use of coercions. This indicates that (explicit) coercions are inessential to the system.

The core of the authentication half of the proof actually is a straightforward lemma showing that we cannot construct a value of variable type α if $\alpha : \Omega$ (note that the kind assumption is crucial, as for a type of kind $\mathrm{S}_{\hat{\kappa}}(\tau)$ or $\mathrm{A}(\tau)$ we often *could* write down a value).

Proposition 36 (Abstractness)**.** *If* $\Gamma = \Gamma_1, \alpha{:}\Omega, x{:}\alpha, \Gamma_2$ *and* $\Gamma \vdash e : \tau$ *with* $\Gamma \vdash \tau \leq \alpha$*, then* e *is not a value.*

For opacity itself, we only consider two different substitutions (i.e. implementations), since the result obviously generalises to an arbitrary number of different substitutions. Because of type case and unpickling, we must first show that type inclusion and term validity stay independent of the substitution as well.

Theorem 37 (Opacity)**.**
Let $\Gamma = \alpha{:}\Omega, x{:}\alpha, f{:}\alpha{\to}1$*, and* $\gamma_i = [\alpha_i/\alpha] \cup [v_i/x] \cup [v_i'/f] \cup [\alpha'/\alpha' \mid \alpha' \in \mathrm{Dom}(\Delta')]$ *with* $\alpha \notin \mathrm{Dom}(\Delta, \Delta')$ *and* $\alpha_i \notin \mathrm{Dom}(\Delta)$ *and* $\Delta, \gamma_i(\Delta') \vdash \gamma_i : \Gamma, \Delta'$ *and* $\Delta; v_i'\, v_i \to^* \Delta; \diamond$ *for* $i \in \{1, 2\}$*.*

1. *Let* $\Gamma, \Delta' \vdash \tau : \kappa$ *and* $\Gamma, \Delta' \vdash \tau' : \kappa$*.*
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(\tau) \leq \gamma_1(\tau') : \gamma_1(\kappa)$*, then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) \leq \gamma_2(\tau') : \gamma_2(\kappa)$*.*

2. *Let* $\Gamma, \Delta' \vdash \square$*.*
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(e) : \gamma_1(\tau)$*, then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(e) : \gamma_2(\tau)$*.*

3. *Let* $\Gamma, \Delta' \vdash e : \tau$*.*
   If and only if $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to^* \Delta, \gamma_1(\Delta'), \Delta_1; v'$*, then* $\Delta_1 = \gamma_1(\Delta'')$ *and* $v' = \gamma_1(v)$ *with* $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to^* \Delta, \gamma_2(\Delta'), \gamma_2(\Delta''); \gamma_2(v)$*.*

The proof of (1) and (2) is by indirection through the Algorithmic Type Matching and Type Synthesis judgements (Appendix D.4).

Note that the opacity theorem assumes that the client – the expression e – has no access to the type heap containing the abstract type substituted. This is an optimistic assumption. As we explained in Section 5.4, in practice an open scenario may, under certain circumstances, enable a malicious client to access parts of the type heap "out of scope", and thus circumvent opacity. The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus cannot directly express such a situation, though. We leave the development of an adequate refinement that may address this particular problem to future work (Section 14.2).

12.10. Related Work

12.10.1. Type Analysis and Dynamics

The type case construct in our calculus is a (weaker) variant of Girard's J operator [Gir71, HM99]. Girard defines it as a constant of type $\forall\alpha{:}\Omega.\forall\beta{:}\Omega.\alpha \to \beta$ with the following reduction rule:

$$\begin{array}{lcll} J\,\tau_1\,\tau_2\,v & \to & v & \text{if } \tau_1 \equiv \tau_2 \\ J\,\tau_1\,\tau_2\,v & \to & 0\,\tau_2 & \text{if } \tau_1 \not\equiv \tau_2 \end{array}$$

Here, 0 is an auxiliary universal constant of type $\forall\alpha{:}\Omega.\alpha$, used as a default value instead of a failure branch.

Even closer to our construct is a variant of J discussed by Mitchell & Harper [HM99]. Their operator $\mathit{TypeCond} : \forall\alpha{:}\Omega.\forall\beta{:}\Omega.\alpha \to \beta \to \beta$ has the following reduction rules:

$$\begin{array}{lcll} \mathit{TypeCond}\,\tau_1\,\tau_2\,v_1\,v_2 & \to & v_1 & \text{if } \tau_1 \equiv \tau_2 \\ \mathit{TypeCond}\,\tau_1\,\tau_2\,v_1\,v_2 & \to & v_2 & \text{if } \tau_1 \not\equiv \tau_2 \end{array}$$

This operator does not require a 0 constant, and modulo syntax is almost identical to our type case, except that we extend it with subtyping (and have higher-order types). Mitchell & Harper demonstrate that even the weaker *TypeCond* is sufficient to destroy normalization. They point out that a type *Dynamic* implicitly introduces a form of impredicativity (since it can be expressed as $\exists\alpha.\alpha$), and that impredicativity is essential for expressing diverging computations.

Richer forms of type analysis have been extensively discussed in programming language literature, either as stand-alone constructs, or in conjunction with dynamics.

Dynamics were first formalised by Abadi, Cardelli Pierce & Plotkin [ACPP91]. In their system, dynamics were eliminated directly by a type case construct that allowed pattern matching on the dynamic type. In the original system, only monomorphic types were possible, but they could contain pattern variables that captured constituent types. Later work by the same and other authors [ACPR95, LM93] extended this idea to polymorphic types, which raises a number of issues about scoping of type and pattern variables. Abadi et al. also discuss the interaction with subtyping, pointing out that subtyping interferes badly with pattern variables, because there no longer always is a unique type to bind them to. Our type case supports subtyping, but no pattern variables, so no such problems arise.

Other works investigate type-based dispatch as a stand-alone feature. In particular, Harper & Morrisett introduce *intensional type analysis* as a more general construct for analysing types or type variables [HM95]. Their main motivation is a typed intermediate language that can cope with heterogeneous representations in a compiler. Their system has a typerec construct that is not simply a pattern matching construct, but a primitive recursion operator on higher-order types. Moreover, they also introduce an analogous Typerec on the type level and show that many interesting applications (in particular representation optimisations) could not be typed without it. The underlying type system is predicative, ensuring termination. Later work extended this approach to polymorphic types [CW99, TSS00, Wei02], and discusses how to reify types on the term level in order to allow type erasure despite type analysis [CW99, CWM02].

In previous work [Ros06], we modelled packages directly as part of an expressive module calculus and proved it sound. The calculus correctly models the package signature refinement discussed in Sections 4.4.1 and 12.3.2. However, it does not support type abstraction. Besides packages, it also features pickling, very much like shown here. We are not aware of any other work that formalises pickling on this level of abstraction.

We also have previously given a semantics of packages directly within the framework of the formal language specification of Standard ML [Ros05]. However, that framework is too complex to make proving of interesting formal properties tractable.

12.10.2. Term Name Generation

Dynamic generation of term names is a common need in programming languages and related systems. Most prominently, the π-calculus [Mil99, SW01] provides the name restriction expression $\nu n.P$ as one of its central feature, which can be considered as a name generator. However, unlike in our system, generated names are not recorded on a heap, but maintained by floating outwards through a set of reduction rules for *scope extrusion*. A globally accumulated sequence of ν-binders could be interpreted as a heap.

Pitts' λ_ν-calculus [PS93] transfers explicit name generation to the typed λ-calculus. It extends plain λ-calculus by introducing a special type ν and the expression form $\nu n.e$ for generating

values of that type. The only operation available on such names is comparison, $\mathsf{if}\, e_1 = e_2\, \mathsf{then}\, e_3\, \mathsf{else}\, e_4$, where e_1 and e_2 must evaluate to names. Like our system, the operational semantics uses an explicit type heap.

The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus subsumes λ_ν, because using existential types, it can express generation of term names as a derived form: term names n can be represented as existential packages $\langle \alpha, \diamond \rangle$ carrying a type name α to identify the name. Term name generation then maps to type name generation and name comparison to a use of type case (making use of the abbreviation from Section 12.3.1):

$$\begin{array}{rcl} \nu & := & \exists \alpha{:}\Omega.1 \\ \nu n.e & := & \mathsf{new}\, \alpha_n{\approx}1\, \mathsf{in}\, \mathsf{let}\, n{=}\langle \alpha_n, \diamond \rangle\, \mathsf{in}\, e \\ \mathsf{if}\, e_1{=}e_2\, \mathsf{then}\, e_3\, \mathsf{else}\, e_4 & := & \mathsf{let}\langle \alpha_1, \_ \rangle{=}e_1\, \mathsf{in}\, \mathsf{let}\langle \alpha_2, \_ \rangle{=}e_2\, \mathsf{in}\, \mathsf{case}\, \alpha_1{\leq}\alpha_2{:}\Omega\, \mathsf{then}\, e_3\, \mathsf{else}\, e_4 \end{array}$$

It is easy to verify that λ_ν reduction can be simulated under this encoding, and that its typing rules are admissible.

12.10.3. Abstraction Safety and Type Generation

Literature on type abstraction goes back at least as far as Morris' paper from 1973, where he suggests that types should not primarily be seen as extensional classifications of values, but rather as abstract, intensional descriptions of their properties, enforcing authentication and secrecy [Mor73b]. Reynolds later put this view in a nutshell by observing that *"type structure is a syntactic discipline for enforcing levels of abstraction"* [Rey83].

The first formal treatment of the notion of abstract type in the framework of type theory is due to Mitchell & Plotkin, who give existential quantification types as a natural explanation for type abstraction [MP88]. In their model, abstract types are actually first-class and have to be opened explicitly for use. However, the type can only be used locally. MacQueen criticised this closed scope restriction as too inflexible for modular purposes, and proposed dependent types as an alternative [Mac86], which then let to the development of the ML module system

The existential model for type abstraction breaks in the presence of dynamic type analysis. Weirich in fact demonstrated that in a non-parametric setting arbitrary values of existential type can be cast back and forth to and from their actual representation type [Wei00]. Although this problem has long been folklore, it has not received much attention in literature.

As already discussed in Section 10.5, Abadi et al. address this matter in their paper on polymorphic dynamics [ACPR95] by proposing a modified reduction rule for opening existentials, where the type variable is substituted by a fresh type instead of the witness. However, as we demonstrated, this proposal destroys type preservation, so that soundness cannot be established by standard means.

Harper & Morrisett also mention the issue in their discussion of intensional type analysis [HM95]. They briefly suggest to distinguish analysable from non-analysable types, supposedly by instrumenting the kind system. As we already pointed out, any such approach is not applicable for our purposes, because expressing packages requires a *fully reflexive* solution, where all types can be analysed.

In earlier work [Ros03a] we already addressed the abstraction problem by introducing an explicit type generation construct for generating fresh abstract type names. Unlike the system in this thesis, the type system considered in the paper did not feature singleton kinds, and hence could only express abstract types but not translucency. Both are essential ingredients of ML-style module systems. Also, we only proved secrecy, but not authentication in that work.

Although being simple in spirit, to our knowledge there was no previous work that isolated the dynamic aspect of type generativity for abstraction and formalises it in a calculus. One

notable exception is Sewell, who uses sealing for modelling certain aspects of type abstraction in the presence of dynamic typing and gives a dynamic semantics [Sew01]. However, in his system generated abstract types are recorded with singleton kinds in a global environment, so that opacity is not maintained dynamically. Followup work [LPSW03] uses type generativity to maintain abstraction safety, similar to our approach. However, type name generation is *static* in that system, i.e. performed at compile-time, not at run-time. The names are generated as cryptographic hashes over the implementation of the module defining the type, such that they are even stable across compiles. This property allows for easier and more flexible sharing between processes, but is not safe for stateful abstractions. Their system does not have polymorphism. Consequently, even though it can express type sharing through a variant of singleton kinds, *dynamic* type sharing cannot be expressed. No opacity result is given for their system. It would not hold unconditionally, due to another feature of the system, which allows to intentionally breach an abstraction barrier and access the representation from outside its implementation. The authors argue that such a feature is needed to support software evolution, though it destroys abstraction safety.

Glew presented a calculus for generating new tagged types at runtime and dispatching on them [Gle99]. His system is more complex than ours in order to allow for hierarchical types, but it is not fully reflexive since untagged types cannot be analysed.

Vytiniotis, Washburn & Weirich combine type analysis with type generation, very much like our system [VWW05]. They have a richer type case with first-class, composable case arms. On the other hand, they do not have singleton kinds. They give a proof of soundness, but do not state any Opacity result. We will discuss their system in more detail in Section 13.6, once we have introduced higher-order coercions.

Dreyer employs explicit generativity to model type abstraction of recursive modules [Dre07]. For that purpose, he sets up a system somewhat similar to ours, but separates generation ($\mathsf{new}\ \alpha \uparrow \kappa\ \mathsf{in}\ e$) and definition ($\mathsf{set}\ \alpha :\approx \tau\ \mathsf{in}\ e$) of type names. The type heap in his system hence distinguishes multiple states for the type names it records, and the type system contains a simple effect system to ensure linearity of the definitions. He interprets all abstract types as iso-recursive, such that they are not equivalent to their (unfolded) representation and the respective fold and unfold operations take the role of basic coercions. Because the isomorphism is only applicable in the scope of the respective set construct, we conjecture that his system would enjoy a similar Opacity property as ours in the presence of dynamic type analysis.

The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus also reveals close similarities to the cryptographic lambda calculus by Sumii and Pierce [SP03, SP04]: new correspond to key generation and sealing/unsealing to encryption/decryption operations in that calculus. However, their calculus is untyped, and decryption may fail dynamically. As pointed out in Section 5.4, this work may point out a direction for improving extra-linguistic abstraction safety in Alice ML.

While module theories usually account for *generativity* as well, they do so solely on the static level of typing rules. In fact, all of the influential theories for ML modules [Ler95, Lil97, DCH03] are not full calculi, but merely type systems, that side-step the issue of reduction. The presence of ad-hoc typing rules encompassing type abstraction precludes a type-preserving reduction semantics.

12.10.4. Opacity and Proof Techniques

A topic we have only scratched in this thesis is the proof-theoretic side of (type) abstraction. There is a rich body of literature on proof techniques for studying properties such as *representation independence* or *observational equivalence*, which formulate when two implementations of an abstraction are equivalent, e.g. in the sense of behaving equivalently in all possible con-

texts. This is a notoriously hard problem that classically has been addressed with semantic methods such as logical relations [Mit91, PA93, Pit00, SP03]. However, for polymorphic types, these techniques crucially rely on parametricity [Rey83, BFSS89, ACC93]. It is not clear if and how they can be adapted to a non-parametric setting like the one fundamental to the issues we address with the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus.

Our opacity result can be interpreted as a form of representation independence or observational equivalence over a fixed and very limited signature (the entities bound in Γ).

More recently, there has been a trend towards more syntactic methods that tend to be less powerful but also less heavyweight [Pit98, Pit05]. In non-deterministic settings – especially when concurrency is involved – co-inductive methods like *bisimulations* [Mil99, SW01] are usually preferred. Sumii and Pierce adopt bisimulation to reason about abstraction properties in untyped and typed λ-calculi [SP04, SP05].

Grossman, Morrisett & Zdancewic propose a syntactic proof method that traces abstraction boundaries with explicit brackets to show simple abstraction properties [GMZ00]. One such property they call *value abstraction*, which is mostly equivalent to the secrecy half of our opacity property, but without considering type analysis. Their abstraction brackets are basically a static variant of our higher-order coercions, as we will discuss later on in Section 13.6.

A step further into the direction of security go methods for enforcing *non-interference* of information flow. Sabelfeld & Myers give a nice survey of language-related techniques proposed in literature [SM03].

12.11. Summary

- The operational semantics of $\lambda^{\omega}_{\mathrm{SA}\Psi}$ uses a type heap for recording generated types.
- Recursion is derivable for inhabited types by using type analysis.
- The semantics of pickling includes a dynamic term validity check.
- Coercions inhabit abstract types, otherwise, typing is largely standard.
- We give an algorithm for deciding well-typedness.
- The type system is sound with respect to the operational semantics.
- Soundness depends on decidable subtyping (for type analysis) and decidable type checking (for unpickling).
- Reduction preserves *opacity* (secrecy and authentication) for abstract types.

13. Higher-Order Abstraction

In the last two chapters we have developed the basic $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus and its meta-theory. We have shown that it can express abstractions that are safe despite the presence of dynamic type analysis. However, there are some limitations to the system so far:

- It can only generate type names of kind Ω. Thus, higher-order abstract types cannot be expressed. For example, a parametric type like $\mathit{stack}(\alpha)$ cannot be made abstract, because it has kind $\Omega \to \Omega$.
- Abstraction can only be expressed *a priori*, because the implementation has to be scattered with suitable coercions. In contrast, module systems typically allow abstraction to be performed *a posteriori* through a convenient *sealing* construct applied to an "untreated" implementation.
- Sealing also is more expressive because it enables abstraction without access to individual values. For example, a value of type $\mathit{stack}(\mathit{int})$ can be abstracted to $\mathit{stack}(t)$ (if t is defined as int) without having to decompose it and abstract each of its elements individually, like necessary in our calculus. For stateful abstractions, this would actually be crucial, because decomposition implies copying.

As the last part of our technical development, we hence extend the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus with facilities for higher-order abstraction:

- **Higher-order generativity**, to express abstract types of higher kind.
- **Higher-order type coercions**, to express abstraction after the fact.

These extensions are orthogonal to a certain extent, but both are more intricate than one might be lead to believe: because of the interference with dependent kinds (Section 11.2), both extensions require the introduction of another auxiliary concept, namely *kind coercions*. Kind coercions are the type-level analogon to higher-order type coercions. Unlike type coercions, kind coercions can be defined as a derived concept, independent of anything else. However, we feel that the definition will be much easier to understand once we have introduced type coercions. Thus, we chose to first discuss higher-order generativity and type coercions, and come to kind coercions last, even though the former concepts both depend on them.

13.1. Higher-Order Generativity

Superficially, generalising the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus to support higher-order generativity seems straightforward: we just have to extend new to allow arbitrary kinds:

$$\mathsf{new}\,\alpha{:}\kappa \approx \tau\ \mathsf{in}\ e$$

But this raises at least two questions: What is the kind assigned to α? And how does a coercion over α work when it is not ground? As we will see, both these questions can elegantly be answered by a definition of *higher-order abstraction kinds* $\mathrm{A}(\tau : \kappa)$ as a derived concept that closely mirrors the definition of higher-order singletons (Section 11.2.2).

$$\text{expressions} \quad e \quad ::= \quad \cdots \mid \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau\ \mathsf{in}_{\tau}\ e$$

Figure 13.1.: Extension with higher-order generativity

$$(\textsc{Enew'})\ \frac{\Gamma \vdash \tau_1 : \tilde{\kappa} \qquad \Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2}$$

Figure 13.2.: Typing of higher-order type generation

$$\text{heaps} \quad \Delta \quad ::= \quad \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tau : \tilde{\kappa})$$

$$(\textsc{Rnew'}) \quad \Delta; E[\mathsf{new}\,\alpha{:}\tilde{\kappa}{\approx}\tau\ \mathsf{in}\ e] \quad \to \quad (\Delta, \alpha{:}\mathrm{A}(\tau : \tilde{\kappa})); E[e]$$

Figure 13.3.: Reduction of higher-order type generation

13.1.1. Type Generation

Let us first look at the new construct itself. Like suggested, we simply extend the syntax to include a kind annotation; the new syntax is given in Figure 13.1. However, the annotation is restricted to the syntactic subclass $\tilde{\kappa}$ of *concrete kinds*, defined in Figure 13.13, that excludes abstraction kinds. That restriction avoids the need for "nested" abstraction kinds and accompanying complications, as we will discuss in Section 13.3.5.

The typing rule generalises in mostly obvious ways, shown in Figure 13.2. The novelty in this rule is the appearance of a higher-order abstraction kind, which we explain in a minute. Furthermore, we have a separate premise ensuring well-formedness of the representation type τ_1. That is needed because it is not implied by well-formedness of the higher-order abstraction kind $\mathrm{A}(\tau_1 : \tilde{\kappa})$, due to its definition as a derived form (see next section).

The operational semantics of new does not change much in the higher-order version either. However, we have to be more permissive about the syntactic structure of the type heap and allow it to contain higher-order abstraction kinds, accordingly. The updated definitions can be seen in Figure 13.3.

13.1.2. Abstraction Kinds

The new-construct generates types of higher-order kind, so the bound variable has to be assigned respective higher-order abstraction kind. Like higher-order singleton kinds (Section 11.2.2), these do not have to be defined as primitive, but can be derived as sugar. Figure 13.4 gives a definition. Unlike singletons, higher-order abstraction kinds are only defined at concrete kind (Figure 13.13), i.e. there are no kinds $\mathrm{A}(\tau : \mathrm{A}(\tau'))$. Section 13.3.5 will explain the reasoning behind that restriction.

Before we explain the definition further, let us consider the properties required from it. Like with higher-order singletons, higher-order abstraction kinds have to be defined such that we can give appropriate admissible judgement rules that generalise the respective built-in rules for ground abstraction kinds, in order to make their treatment consistent. Figure 13.5 shows the desired validity, equivalence and inclusion rules, which closely mirror those for singletons, except that rule KSabs* is slightly weaker than the corresponding KSsing*, because it does not allow the annotated kind to vary. We will understand why soon.

$$\begin{array}{rcl} \mathrm{A}(\tau : \Omega) & := & \mathrm{A}(\tau) \\ \mathrm{A}(\tau : \mathrm{S}_\Omega(\tau')) & := & \mathrm{S}_\Omega(\tau') \\ \mathrm{A}(\tau : \Pi\alpha{:}\tilde\kappa_1.\tilde\kappa_2) & := & \Pi\alpha{:}\tilde\kappa_1.\mathrm{A}(\tau\,\alpha : \tilde\kappa_2) \\ \mathrm{A}(\tau : \Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2) & := & \Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1 : \tilde\kappa_1).\mathrm{A}(\{\tau{\cdot}2 : \alpha{:}\tilde\kappa_1.\tilde\kappa_2\}_{\alpha/\tau\cdot 1} : \tilde\kappa_2) \end{array}$$

Figure 13.4.: Higher-order abstraction kinds

Kind Validity $\boxed{\Gamma \vdash \kappa : \Box}$

$$(\textsc{Kabs}^*)\ \frac{\Gamma \vdash \tau : \tilde\kappa}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) : \Box}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \Box}$

$$(\textsc{KQabs}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde\kappa \qquad \Gamma \vdash \tilde\kappa \equiv \tilde\kappa' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \equiv \mathrm{A}(\tau' : \tilde\kappa') : \Box}$$

Kind Inclusion $\boxed{\Gamma \vdash \kappa \leq \kappa' : \Box}$

$$(\textsc{KSabs}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde\kappa \qquad \Gamma \vdash \tilde\kappa \equiv \tilde\kappa' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \leq \mathrm{A}(\tau' : \tilde\kappa') : \Box}$$

$$(\textsc{KSabs-left}^*)\ \frac{\Gamma \vdash \tau : \tilde\kappa}{\Gamma \vdash \mathrm{A}(\tau : \tilde\kappa) \leq \tilde\kappa : \Box}$$

Figure 13.5.: Rules for higher-order abstraction kinds

The most critical invariant is stated by rule KSabs-left*. Like with ground abstract types, we want a generated type name to be used as the "key" for a coercion – thus having kind $\mathrm{A}(\tau : \tilde\kappa)$ – and at the same time take the role of the abstract type itself, hence also having kind $\tilde\kappa$. The subkinding rule expresses this requirement.

Now turning back to the definition of higher-order abstraction kinds in Figure 13.4, first note the similarity with the definition of higher-order singletons (Section 11.2.2). They only differ for Σ-kinds: where singletons over pairs can be formed as a simple non-dependent sum by substituting away the dependency (because the variable is equivalent to the first component by the singleton rules), this substitution is not possible with abstraction kinds. Worse, since the first component changes its kind from κ_1 to $\mathrm{A}(\tau{\cdot}1 : \kappa_1)$ when forming the abstraction kind over Σ, the dependency of κ_2 is on the "wrong" type. To see the problem, assume we would simply define

$$\mathrm{A}(\tau : \Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2) \ := \ \Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1 : \tilde\kappa_1).\mathrm{A}(\tau{\cdot}2 : \tilde\kappa_2) \qquad \text{(wrong-1)}$$

The constituent kind $\mathrm{A}(\tau{\cdot}2 : \tilde\kappa_2)$ would be ill-formed in general, because $\tau{\cdot}2$ has kind $\tilde\kappa_2[\tau{\cdot}1/\alpha]$ (according to rule Tsnd) and not $\tilde\kappa_2$. As a concrete example, consider the following higher-order abstraction kind:

$$\kappa = \mathrm{A}(\tau : \Sigma\alpha{:}\Omega.\, \mathrm{S}_\Omega(\alpha) \to \Omega) \quad \text{with} \quad \tau = \langle \mathit{int}, \lambda\beta{:}\mathrm{S}_\Omega(\mathit{int}).\beta \rangle$$

It roughly corresponds to the following use of sealing in ML:

```
struct type t = int; functor F (type u = int) = (type v = u) end
:> sig type t;       functor F (type u = t) : (type v) end
```

Assuming definition (wrong-1), κ would expand to

$$\Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1).\, \mathrm{A}(\tau{\cdot}2 : \mathrm{S}_\Omega(\alpha) \to \Omega)$$

which further expands to

$$\Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1).\, \Pi\beta{:}\mathrm{S}_\Omega(\alpha).\mathrm{A}(\tau{\cdot}2\,\beta)$$

where the type application $\tau{\cdot}2\,\beta$ is ill-formed, because $\tau{\cdot}2$ has argument kind $\mathrm{S}_\Omega(\mathit{int})$, but β is of kind $\mathrm{S}_\Omega(\alpha)$, and $\alpha \not\equiv \mathit{int}$.

On the other hand, defining

$$\mathrm{A}(\tau : \Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2) \;:=\; \Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1 : \tilde\kappa_1).\mathrm{A}(\tau{\cdot}2 : \tilde\kappa_2[\tau{\cdot}1/\alpha]) \qquad \text{(wrong-2)}$$

would produce a well-formed expansion, but would also prevent $\mathrm{A}(\tau : \Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2)$ from being a subkind of the underlying kind $\Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2$ – obviously, it would specialise only $\Sigma\alpha{:}\tilde\kappa_1.\tilde\kappa_2[\tau{\cdot}1/\alpha]$. For our previous example, κ would become

$$\Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1).\, \mathrm{A}(\tau{\cdot}2 : \mathrm{S}_\Omega(\mathit{int}) \to \Omega)$$

Clearly, this cannot be a subkind of $\Sigma\alpha{:}\Omega.\, \mathrm{S}_\Omega(\alpha) \to \Omega$, again because $\alpha \not\equiv \mathit{int}$. Consequently, definition (wrong-2) would break the essential rule KSABS-LEFT*.

The expansion we actually need for κ is a kind that is equivalent to the following:

$$\Sigma\alpha{:}\mathrm{A}(\mathit{int}).\, \mathrm{A}(\lambda\beta{:}\mathrm{S}_\Omega(\alpha).\beta : \mathrm{S}_\Omega(\alpha) \to \Omega)$$

Here, instead of adapting the kind annotation in the second component, the critical occurrence of int in $\tau{\cdot}2$ has been "reverse-substituted" by α in the constitutent type, in order to match the original kind annotation. The expansion is well-formed and obeys the desired subkinding relation. The kind coercion in Figure 13.4 achieves precisely this transformation and thus bridges the critical kind discrepancy.

We hence are left with developing kind coercions. However, kind coercions are somewhat intricate and likely to be more understandable once we have introduced the more earthly notion of *type* coercions. We thus delay the definition of kind coercions until Section 13.3.

Finally, note that well-formedness of a kind $\mathrm{A}(\tau : \tilde\kappa)$ does not necessarily imply that $\tau : \tilde\kappa$. In particular, in the case of $\tilde\kappa = \mathrm{S}_\Omega(\tau')$ the expansion does not contain τ, so any judgement about $\mathrm{A}(\tau : \tilde\kappa)$ is vacuous on τ. In contexts where a well-formed τ is needed that requirement has to be stated separately. This is the same situation as with higher-order singletons [SH06].

13.1.3. Type Coercions

One interesting and nifty consequence of the fine-grained notion of abstraction kinds is that the move to higher-order generativity itself does not require any change to the concept of basic type coercions: because we already allow the abstract type τ in a coercion $\{e\}^\pm_\tau$ to be represented by an arbitrary type expression, and because higher-order abstraction kinds simply decompose into ground ones, coercions already can express values of higher-order abstract type.

Consider an abstract type of stacks:

$$\mathsf{new}\ \mathit{stack} \approx \lambda\alpha{:}\Omega.\mathit{list}\,\alpha\ \mathsf{in}\ \ldots$$

Or, by η-equivalence, simply $\mathit{stack} \approx \mathit{list}$. According to rule ENEW', *stack* will receive higher-order abstraction kind $\mathrm{A}(\mathit{list} : \Omega \to \Omega)$, which decomposes into $\Pi\alpha{:}\Omega.\mathrm{A}(\mathit{list}\,\alpha)$. Consequently, a value of a stack of integers can be formed with a straightforward coercion:

$$s = \{[5,2,7]\}^{+}_{\mathit{stack\ int}}$$

What we can *not* express, however, is taking this value and abstracting it further at a type such as $\mathit{date} \approx \mathit{int}$, e.g.:

$$\{s\}^{+}_{\mathit{stack\ date}}$$

This coercion is ill-formed, because the type *stack date* has kind $\mathrm{A}(\mathit{list\ date})$, while the type of s is *stack int*, which would necessitate $\mathrm{A}(\mathit{stack\ int})$ instead.

The list $[5,2,7]$ *can* be coerced to the desired abstract type *stack date*, but only by performing the coercions the other way round: first abstract *int* to *date* and then *list date* to *stack date*. Doing so would form the value

$$\{[\{5\}^{+}_{\mathit{date}}, \{2\}^{+}_{\mathit{date}}, \{7\}^{+}_{\mathit{date}}]\}^{+}_{\mathit{stack\ date}}$$

Here, every element of the stack is coerced individually. But inversing the coercion order like this generally requires different scoping of the abstractions: we cannot abstract *stack int* to *stack date* this way if we are not in scope of the definition of *stack* as well. Or in other words, if we are given s then we need the ability to decompose and reconstruct it. At the very least, the signature of the abstract type *stack* would have to provide operations to do that. That is relatively easy to provide for stacks, but might not be desirable for other abstract types.

Because such forms of coercion arise frequently when towering abstractions, and because we intend to model the situation in ML, where such abstractions can be formed arbitrarily, we move to a generalised notion of higher-order type coercions in the next section. They enable us to express abstractions of this kind succinctly and without requiring a suitable signature from another abstract type.

13.2. Higher-Order Type Coercions

Higher-order type coercions generalise the notion of basic coercion already introduced with the basic calculus. That is, we replace the basic coercions of the form $\{e\}^{+}_{\tau_+}$ with the more general

$$\{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-}$$

and its inverse (Figure 13.6). Here, $\tilde{\kappa}$ again ranges over the subclass of concrete kinds, while $\tilde{\tau}$ ranges over the syntactic subclass of *concrete types* that likewise excludes abstraction kinds to appear in any quantifier (Figure 13.13). We will explain the restriction in Section 13.3.5.

While the basic form $\{e\}^{+}_{\tau_+}$ of coercion simply coerced e from type τ_- to type τ_+, the higher-order form allows e to take any type $\tilde{\tau}[\tau_-/\alpha]$ and results in type $\tilde{\tau}[\tau_+/\alpha]$. That is, α acts as a placeholder marking those positions in the *residual* type $\tilde{\tau}$ where abstraction ought to be performed. The type τ_+ has to be a (higher-order) abstract type of kind $\mathrm{A}(\tau_- : \tilde{\kappa})$, accordingly.

The negative case is analogous, as is apparent from the typing rules in Figure 13.7. We refer to the type $\tilde{\tau}[\tau_-/\alpha]$ of e as the *inward* type of the coercion, and call the result type $\tilde{\tau}[\tau_+/\alpha]$ its *outward* type – in the negative case they change roles, respectively. Note that we require

$$\text{expressions} \quad e \quad ::= \quad \cdots \mid \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tilde{\tau}\approx\tilde{\tau}} \mid \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tilde{\tau}\approx\tilde{\tau}}$$

Figure 13.6.: Extension with higher-order type coercions

Term Validity $\boxed{\Gamma \vdash e : \tau}$

$$(\textsc{Eup'})\ \frac{\Gamma \vdash e : \tilde{\tau}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- {:} \tilde{\kappa}) \qquad \Gamma \vdash \tau_- : \tilde{\kappa}}{\Gamma \vdash \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_+/\alpha]}$$

$$(\textsc{Edn'})\ \frac{\Gamma \vdash e : \tilde{\tau}[\tau_+/\alpha] \qquad \Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- {:} \tilde{\kappa}) \qquad \Gamma \vdash \tau_- : \tilde{\kappa}}{\Gamma \vdash \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha]}$$

Figure 13.7.: Typing rules for higher-order type coercions

the representation type τ_- to be annotated as well, in order to be able to give simple syntax-directed reduction rules. Also, we need a separate premise to ensure its well-formedness, because well-formedness of $\mathrm{A}(\tau_- : \tilde{\kappa})$ does not imply it (Section 13.1.2).

To fix an economic parlance, we say that a coercion $\{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+\approx\tau_-}$ coerces the expression e *over* type τ_+ *under* kind $\tilde{\kappa}$ *at* type $\tilde{\tau}$. The former coercions $\{e\}^{\pm}_{\tau_+}$ arise as the special cases $\{e : \alpha{:}\Omega.\alpha\}^{\pm}_{\tau_+\approx\tau_-}$. The positive variant of this special case represents the normal form of higher-order coercions. That is, actual *values* of abstract type basically look like before.

The benefit of higher-order coercions is that we can perform arbitrary *a posteriori* abstraction. To take the stack example from the previous section, it is now possible to express the desired coercion of s from *stack int* to *stack date* as follows:

$$\{s : \alpha{:}\Omega.\mathit{stack}\,\alpha\}^{+}_{\mathit{date}\approx\mathit{int}}$$

Note that it only coerces over type *date*, not *stack date* as in the earlier attempt. The reduction rules we are about to present will ultimately reduce this to the 'correct' form of coercion given earlier, where the list elements are coerced individually.

In a similar manner, the abstraction-safe complex number implementation C' from Section 10.6 can be expressed in terms of the transparent implementation C as follows:

$$C' = \mathsf{new}\ c : \Omega \approx \mathit{real} \times \mathit{real}\ \mathsf{in}\ \{C : \mathit{complex}{:}\Omega.\ \mathrm{Dyn}(\mathit{COMPLEX})\}^{+}_{c\approx\mathit{real}\times\mathit{real}}$$

This expression roughly corresponds to sealing in ML, where we can express C' similarly, though without the explicit type generation:

structure C' = C :> COMPLEX

13.2.1. Semantics

Figure 13.8 defines the reduction of higher-order coercions. For the sake of readability, we omit the surrounding context E and the type heap Δ from the rules. Moreover, since upward and downward coercions are dual in most cases, we capture both directions in a single set of rules by writing $\pm$ as a placeholder for either $+$ or $-$. The convention is that all occurrences of

$$\begin{array}{llll} \text{values} & v & ::= & \cdots \mid \{v : \alpha{:}\Omega.\alpha\}^{+}_{\tau_+ \approx \tau_-} \\ \text{contexts} & E & ::= & \cdots \mid \{E : \alpha{:}\tilde\kappa.\tilde\tau\}^{+}_{\tau_+ \approx \tau_-} \mid \{E : \alpha{:}\tilde\kappa.\tilde\tau\}^{-}_{\tau_+ \approx \tau_-} \end{array}$$

(RCOERCE-NORM)	$\{v : \alpha{:}\tilde\kappa.\tilde\tau\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\{v : \alpha{:}\tilde\kappa.\pi\}^{\pm}_{\tau_+ \approx \tau_-}$ where $\pi \neq \tau$ and $\Delta, \alpha{:}\tilde\kappa \triangleright \tilde\tau \Rrightarrow \pi$
(RCOERCE-PSI)	$\{v : \alpha{:}\tilde\kappa.\Psi\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	v
(RCOERCE-ARROW)	$\{v : \alpha{:}\tilde\kappa.\tilde\tau_1 \to \tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\lambda x_1{:}\tilde\tau_1[\tau_\pm/\alpha].\{v\,\{x_1 : \alpha{:}\tilde\kappa.\tilde\tau_1\}^{\mp}_{\tau_+ \approx \tau_-} : \alpha{:}\tilde\kappa.\tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}$
(RCOERCE-TIMES)	$\{v : \alpha{:}\tilde\kappa.\tilde\tau_1 \times \tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\mathsf{let}\langle x_1, x_2\rangle = v \;\mathsf{in}\; \langle \{x_1 : \alpha{:}\tilde\kappa.\tilde\tau_1\}^{\pm}_{\tau_+ \approx \tau_-}, \{x_2 : \alpha{:}\tilde\kappa.\tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}\rangle$
(RCOERCE-UNIV)	$\{v : \alpha{:}\tilde\kappa.\forall\alpha_1{:}\tilde\kappa_1.\tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\lambda\alpha_1{:}\tilde\kappa_1[\tau_\pm/\alpha].\{v\,\{\alpha_1 : \alpha{:}\tilde\kappa.\tilde\kappa_1\}_{\tau_\mp/\tau_\pm} : \alpha{:}\tilde\kappa.\tilde\tau_2'\}^{\pm}_{\tau_+ \approx \tau_-}$ where $\tilde\tau_2' = \tilde\tau_2[\{\alpha_1 : \alpha{:}\tilde\kappa.\tilde\kappa_1\}_{\alpha/\tau_\pm}/\alpha_1]$
(RCOERCE-EXIST)	$\{v : \alpha{:}\tilde\kappa.\exists\alpha_1{:}\tilde\kappa_1.\tilde\tau_2\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\mathsf{let}\langle \alpha_1, x_2\rangle = v \;\mathsf{in}\; \langle \{\alpha_1 : \alpha{:}\tilde\kappa.\tilde\kappa_1\}_{\tau_\pm/\tau_\mp}, \{x_2 : \alpha{:}\tilde\kappa.\tilde\tau_2'\}^{\pm}_{\tau_+ \approx \tau_-}\rangle$ where $\tilde\tau_2' = \tilde\tau_2[\{\alpha_1 : \alpha{:}\tilde\kappa.\tilde\kappa_1\}_{\alpha/\tau_\mp}/\alpha_1]$
(RCOERCE-SWAP)	$\{v : \alpha{:}\tilde\kappa.P[\alpha']\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\{\{e : \alpha{:}\tilde\kappa.P[\tau_-'/\alpha' : \tilde\kappa']\}^{\pm}_{\tau_+ \approx \tau_-} : \alpha'' {:} \tilde\kappa'.P[\alpha''/\alpha' : \tilde\kappa'][\tau_\pm/\alpha]\}^{+}_{\alpha' \approx \tau_-'}$ where $e = \{v : \alpha'' {:} \tilde\kappa'.P[\alpha''/\alpha' : \tilde\kappa'][\tau_\mp/\alpha]\}^{-}_{\alpha' \approx \tau_-'}$ and $\alpha' \neq \alpha$ and $\Delta(\alpha') = \mathrm{A}(\tau_-' : \tilde\kappa')$
(RCOERCE-GROUND)	$\{v : \alpha{:}\tilde\kappa.P[\alpha]\}^{\pm}_{\tau_+ \approx \tau_-}$	$\to$	$\{\{v : \alpha{:}\tilde\kappa.P[\tau_\mp/\alpha : \tilde\kappa]\}^{\pm}_{\tau_+ \approx \tau_-} : \alpha{:}\Omega.\alpha\}^{\pm}_{P[\tau_+/\alpha:\tilde\kappa][\tau_\pm/\alpha] \approx P[\tau_-/\alpha:\tilde\kappa][\tau_\pm/\alpha]}$ where $P \neq \_$
(RCOERCE-CANCEL)	$\{v : \alpha{:}\Omega.\alpha\}^{-}_{\tau_+ \approx \tau_-}$	$\to$	v' where $v = \{v' : \alpha{:}\Omega.\alpha\}^{+}_{\tau_+' \approx \tau_-'}$

Notes: 1. Omitted surrounding $\Delta; E[\_]$ in reduction rules.
2. All RHS variables fresh.

Figure 13.8.: Reduction of higher-order coercions

$\pm$ in a single definition must consistently be subsituted by the same sign. At the same time, occurrences of $\mp$ must be replaced by the respective inverse sign.

Note that some of the reducts reuse the same variable α multiple times, to avoid explicit renaming. That is, the variable convention does not automatically apply, and in some of the later proofs we have to be careful to rename manually.

To understand the rules, first note that they all depend on the structure of the residual type, and most of them expect it to be in path form (Section 11.4.1). To separate concerns we use a separate rule RCOERCE-NORM for normalising the residual type appropriately. For all other rules, we implicitly assume a side condition demanding the residual type actually to be in weak-head normal form. That excludes cases where $\tilde\tau$ is syntactically a path, but kind $\tilde\kappa$ is singleton and enables further reduction.

The remaining rules can be categorised in three groups: monomorphic rules (RCOERCE-PSI, RCOERCE-ARROW and RCOERCE-TIMES), polymorphic rules (RCOERCE-UNIV and RCOERCE-EXIST), and abstract rules (RCOERCE-SPLIT to RCOERCE-CANCEL). We will discuss the respective groups in turn.

13.2.2. Monomorphic Coercions

The rules for simple monomorphic types are straightforward and similar to other canonical uses of higher-order conversions that can be found in literature (Section 13.6).

For constant types like Ψ the coercion is redundant and can be dropped. The same would apply to other primitive monomorphic types in the language. For example, some of our examples silently assume an extension of the calculus with an integer type which is trivially treated like this.

For other types reduction basically works by constructing an η-expansion of the constituent term, and perform the necessary coercions in the expansion. For example, the reduct of coercing a function $f : \tilde{\tau}_1 \to \tilde{\tau}_2$ is the following:

$$\lambda x{:}\tilde{\tau}_1[\tau_+/\alpha].\{f\,\{x : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^-_{\tau_+\approx\tau_-} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^+_{\tau_+\approx\tau_-}$$

This is a function that takes an argument of the desired outward type $\tilde{\tau}_1[\tau_\pm/\alpha]$, coerces it, applies the original function, and then coerces back the result. Since the argument is input, it acts in a contravariant fashion, which means that the coercion must go the other way, indicated by the switch of polarity in the coercion.

13.2.3. Polymorphic Coercions

How does the definition of coercions extend to polymorphic types, i.e. residual types containing quantifiers? In plain λ^ω, reduction could actually happen in a straightforward manner analogous to monomorphic functions and pairs [Ros03a]:

$$\{v : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-} \to \lambda\alpha_1{:}\tilde{\kappa}_1.\{v\,\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-} \qquad \text{(wrong-univ-1)}$$

$$\{v : \alpha{:}\tilde{\kappa}.\exists\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-} \to \mathsf{let}\langle\alpha_1, x_2\rangle = v\ \mathsf{in}\ \langle\alpha_1, \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-}\rangle \qquad \text{(wrong-exist-1)}$$

Unfortunately, these definitions are no longer valid in the presence of dependent kinds, because the placeholder α may occur in the quantifier's classifier $\tilde{\kappa}_1$, rendering the reduct ill-formed! Consider the following example:

$$\{v : \alpha{:}\Omega.\forall\alpha_1{:}\mathrm{S}_\Omega(\alpha).\alpha_1\}^+_{\tau_+\approx\tau_-}$$

The above definition (wrong-univ-1) would produce

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\alpha).\, v\,\alpha_1$$

which is bogus due to the now unbound occurrence of the placeholder α.

We might be tempted to repair this be simply substituting α with τ_+, that is:

$$\{v : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-} \to \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\{v\,\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^\pm_{\tau_+\approx\tau_-} \qquad \text{(wrong-univ-2)}$$

Now we get

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\tau_+).\, v\,\alpha_1$$

which still is ill-typed, but now for slightly more subtle reasons: v is an expression of type $\forall\alpha_1{:}\mathrm{S}_\Omega(\tau_-).\alpha_1$. Since the λ-bound α_1 has kind $\mathrm{S}_\Omega(\tau_+)$, and τ_+ and τ_- are incompatible types, both singleton kinds clash at the polymorphic instantiation $v\,\alpha_1$ – the only type we can legally apply to v is τ_-, not τ_+.

Fortunately, the annotation gives us enough information to make a suitable replacement: wherever the type variable α appears in the kind κ_1 of the argument, we have to "substitute" it by τ_- itself (or by τ_+ in the downward case, respectively). In other words, we need to perform a kind coercion:

$$\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}^\pm_{\tau_+/\tau_-}$$

Given a type τ with inward kind $\tilde{\kappa}[\tau_{\mp}/\alpha]$, it constructs a type τ' with outward kind $\tilde{\kappa}[\tau_{\pm}/\alpha]$.

We defer the definition of kind coercions to Section 13.3. For now, we simply assume it given and use it to refine the coercion rule for universal types as follows:

$$\{v : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} \to \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\{v\,\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}^{-}_{\tau_+/\tau_-} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} \quad \text{(wrong-univ-3)}$$

The expansion of our running example will now be

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\tau_+).v\,\{\alpha_1 : \alpha{:}\Omega.\mathrm{S}_\Omega(\alpha)\}^{-}_{\tau_+/\tau_-}$$

This is much better: the kind coercion produces a type of kind $\mathrm{S}_\Omega(\tau_-)$. That is, it will in fact produce a type equivalent to τ_-, so that we can rewrite this equivalently to

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\tau_+).v\,\tau_-$$

The instantiation is now well-formed.

Are we done? Unfortunately, no. Having successfully adapted the argument to the *domain* of v, there now is a problem with its *codomain*: the type of $v\,\tau_-$ is $\alpha_1[\tau_-/\alpha_1] = \tau_-$, according to typing rule Einst. What we want to have, though, is a result of type τ_+. More precisely, if we look at the latest version of our coercion rule, we notice that the instantiation $v\,\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}^{\mp}_{\tau_+/\tau_-}$ has type $\tilde{\tau}_2[\tau_{\mp}/\alpha][\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}^{\mp}_{\tau_+/\tau_-}/\alpha_1]$ by rule Einst. However, the coercion on the result requires inward type $\tilde{\tau}_2[\tau_{\mp}/\alpha]$.

So the last fix to the rule requires finding a suitable residual type $\tilde{\tau}_2'$ in place of $\tilde{\tau}_2$ in the outer coercion. Looking at the Eup' typing rule, we can observe that it needs to fulfil both of the following equations to match the inward and the outward typing requirements:

$$\begin{aligned}\tilde{\tau}_2'[\tau_{\mp}/\alpha] &= \tilde{\tau}_2[\tau_{\mp}/\alpha][\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}^{\mp}_{\tau_{\mp}/\tau_{\pm}}/\alpha_1]\\ \tilde{\tau}_2'[\tau_{\pm}/\alpha] &= \tilde{\tau}_2[\tau_{\pm}/\alpha]\end{aligned}$$

Does such a type even exist?

Yes, a suitable solution is given in Figure 13.8. It again involves the use of kind coercions. Hence, we explain the idea behind this solution in Section 13.3.

A similar rule is necessary to coerce at existential types. We can prove soundness of these rules after having developed a suitable definition for kind coercions.

13.2.4. Abstract Coercions

Monomorphic and polymorphic coercions cover the cases where the residual type can be reduced to a concrete base type. There remain the cases where the residual type is *abstract*, i.e. a constructor path of the form $P[\alpha]$ (recall the definition of path contexts P from Section 11.4.1). Because reduction happens only for terms that are closed up to the type heap, α must be either the placeholder variable of the coercion, or an abstract type from the heap.

These cases are covered by the last group of reduction rules, respectively. Of this group, the rule Rcoerce-cancel is the obvious generalisation of the original Rcancel rule and replaces it. It eliminates two inverse abstract coercions that are in *ground form*, i.e. where the residual type simply consists of the placeholder variable.

But how do we arrive at ground form? This is achieved by the other two rules.

Let us first look at rule Rcoerce-swap. It treats the case where the root of the path is an(other) abstract type. It thereby implements the actual reduction of coercions over higher-order abstract types. For example, recall the expression

$$\{s : \alpha{:}\Omega.\mathit{stack}\,\alpha\}^{+}_{\mathit{date}\approx\mathit{int}}$$

This is not a value, since the residual type is not plain α. Instead, it has the shape $P[\mathit{stack}]$, with $P = \_\,\alpha$. But how can the coercion be pushed inward?

Well-typedness implies that s is of abstract type $\mathit{stack}\,\mathit{int}$. The operational semantics can hence look up the representation type *list* of *stack* in the type heap and undo the abstraction to *stack* with a respective coercion before performing the actual coercion for *date*, and finally redo the coercion to *stack* with the now abstract argument type. In other words, the above coercion can be split into three simpler ones:

$$\{\{\{s : \alpha'{:}\Omega \to \Omega.\,\alpha'\;\mathit{int}\}^{-}_{\mathit{stack}\approx\mathit{list}} : \alpha{:}\Omega.\,\mathit{list}\,\alpha\}^{+}_{\mathit{date}\approx\mathit{int}} : \alpha'{:}\Omega \to \Omega.\,\alpha'\;\mathit{date}\}^{+}_{\mathit{stack}\approx\mathit{list}}$$

The actual rule RCOERCE-SWAP generalises this idea to arbitrary type paths. Effectively, it uses its unrestricted access to the type heap to "see through" the abstraction barrier of another abstract type and *swap* the order of coercive abstractions. As discussed earlier (Section 13.1.3), this cannot be expressed by the programmer, because she generally does not have access to all of the heap.

Note that the middle coercion at type $\mathit{list}\,\alpha$ in the example is "simpler" than the original one at type $\mathit{stack}\,\alpha$ in the sense that there is a total ordering on the abstract type definitions in the type heap (in particular, there are no cycles), and *list* is necessarily smaller than *stack* with respect to that ordering. This observation can be used to show that reduction of abstract coercions always terminates, even though it temporarily increases the number of nested coercions with no appearant syntactic simplification. We omit the details here, a respective proof for a somewhat simpler system can be found in our previous work [Ros03b].

The remaining rule, RCOERCE-GROUND, is the most intricate one. It applies whenever we coerce at a (non-trivial) path $P[\alpha]$ headed by the placeholder variable itself, i.e. when we do a coercion that is higher-order and higher-kinded at the same time. Intuitively, the rule implements two simultaneous simplifications:

1. *Split* the coercion into two, first coercing the type arguments (occurrences of α in P) and then the head of the residual type.

2. *Ground* the head coercion by lifting path information from the residual type to the abstract type itself.

First, let us consider a very simple example: given an abstract type $\mathit{stack} : \mathrm{A}(\mathit{list} : \Omega \to \Omega)$, the coercion expression

$$\{s : \alpha{:}(\Omega \to \Omega).\,\alpha\;\mathit{int}\}^{+}_{\mathit{stack}\approx\mathit{list}}$$

is reduced by RCOERCE-GROUND to

$$\{\{s : \alpha{:}(\Omega \to \Omega).\,\mathit{list}\;\mathit{int}\}^{+}_{\mathit{stack}\approx\mathit{list}} : \alpha{:}\Omega.\alpha\}^{+}_{\mathit{stack}\;\mathit{int}\approx\mathit{list}\;\mathit{int}}$$

In the reduct of this simple example, the inner coercion actually is an identity coercion, because α does not actually appear in the residual type. Like all identity coercions it is redundant, and will eventually be reduced away. The outer coercion however grounds the residual type by lifting $P[\_] = \_\;\mathit{int}$ to the annotation. Note that the ability to do this follows from the fact that higher-order abstraction kinds are not primitive in our system but decompose into simpler kinds (Figure 13.4). Specifically, the abstract type *stack* has higher-order abstraction kind $\mathrm{A}(\mathit{list} : \Omega \to \Omega)$, which is just short for $\Pi\alpha_1{:}\Omega.\mathrm{A}(\mathit{list}\,\alpha_1)$; thus, rule TAPP simply assigns kind $\mathrm{A}(\mathit{list}\,\alpha_1)[\mathit{int}/\alpha_1]$ to the application $\mathit{stack}\,\mathit{int}$, which is the required kind for making the coercion well-formed, because the typing rule EUP' demands $\mathit{stack}\;\mathit{int} : \mathrm{A}(\mathit{list}\;\mathit{int})$.

So why do we need the inner coercion at all? Would it not be possible to always ground according to following reduction rule?

$$\{v : \alpha{:}\tilde{\kappa}.P[\alpha]\}^{\pm}_{\tau_+ \approx \tau_-} \quad \rightarrow \quad \{v : \alpha{:}\Omega.\alpha\}^{\pm}_{P[\alpha][\tau_+/\alpha] \approx P[\alpha][\tau_-/\alpha]} \qquad \text{(wrong-coerce-ground)}$$

Unfortunately, this does not work. Although the previous example might suggest otherwise, *generally*, the type $P[\alpha][\tau_+/\alpha]$ does not have kind $\mathrm{A}(P[\alpha][\tau_-/\alpha] : \tilde{\kappa})$, as is required by the typing rule EUP'.

For example, let us consider a more complex example where non-trivial splitting occurs:

$$\{\mathit{nil} : \alpha{:}(\Omega \to \Omega).\, \alpha\,(\alpha\ \mathit{int})\}^{+}_{\mathit{stack} \approx \mathit{list}}$$

This expression of type $\mathit{stack}\,(\mathit{stack}\ \mathit{int})$ is split into

$$\{\{\mathit{nil} : \alpha{:}(\Omega \to \Omega).\, \mathit{list}\,(\alpha\ \mathit{int})\}^{+}_{\mathit{stack} \approx \mathit{list}} : \alpha{:}\Omega.\alpha\}^{+}_{\mathit{stack}\,(\mathit{stack}\ \mathit{int}) \approx \mathit{list}\,(\mathit{stack}\ \mathit{int})}$$

Here, the resulting inner coercion is not the identity, yet it still is simpler than the original one because it contains fewer occurrences of α. In particular, the head is now a concrete type (or a simpler abstract type, according to the ordering in the heap). We say that type $\mathit{list}\,(\alpha\ \mathit{int})$ is a *splitting* of the residual $\alpha\,(\alpha\ \mathit{int})$. To see why this is a counter-example for rule (wrong-coerce-ground), observe that $\mathit{stack}\,(\mathit{stack}\ \mathit{int})$ has kind $\mathrm{A}(\mathit{stack}\,(\mathit{list}\ \mathit{int}))$, and *not* $\mathrm{A}(\mathit{stack}\,(\mathit{stack}\ \mathit{int}))$ as the bogus rule would require. This shows that splitting is necessary.

On the other hand, it also is important to note that what consists a valid splitting is dependent on the kind $\tilde{\kappa}$ of the abstract type we coerce over. Because of singleton kinds, it is *not* generally possible to split by simply replacing the head of the path $P[\alpha]$ with the representation type τ_- – even though we know that $\tau_- : \tilde{\kappa}$ the type $P[\tau_-]$ may actually be malformed. As a simple example, consider the kind

$$\kappa_1 = \Sigma\alpha_1{:}\Omega.\Pi\alpha_2{:}\mathrm{S}_\Omega(\alpha_1).\Omega$$

If we have $\alpha : \kappa_1$ then the path $P[\alpha] = \alpha{\cdot}2\,(\alpha{\cdot}1)$ is well-formed. However, forming $P[\tau_-] = \tau_-{\cdot}2\,(\alpha{\cdot}1)$ (with $\tau_- : \kappa_1$) is not: the argument type $\alpha{\cdot}1$ no longer matches the parameter kind of the type constructor $\tau_-{\cdot}2$, which is $\mathrm{S}_\Omega(\tau_-)$. In fact, there does not exist *any* valid non-trivial splitting of $\alpha{\cdot}2\,(\alpha{\cdot}1)$ under $\alpha : \kappa_1$!

To address this problem, both RCOERCE-GROUND and RCOERCE-SWAP use the seemingly strange notation $P[\tau/\alpha : \kappa]$ to express the replacement of the head of a path. In a first approximation, you can think of this *path replacement* as simply meaning $P[\tau]$. However, it is defined such that it adapts occurrences of α in the path P where necessary, such that the resulting type remains well-formed. Since the definition of path replacement involves kind coercions (Section 13.3), we defer its explanation until Section 13.3.4.

For now, just note that $P[\tau_-/\alpha : \kappa_1]$ will yield $\tau_-{\cdot}2\,(\tau_-{\cdot}1)$ for the last example. Hence, only trivial splitting will take place when a respective coercion is reduced, and the residual type is grounded in one step. Unlike in the previous example, this is OK, since given $\tau_+ : \mathrm{A}(\tau_- : \kappa_1)$, the unsplit type $\tau_+{\cdot}2\,(\tau_+{\cdot}1)$ in fact has kind $\mathrm{A}(\tau_-{\cdot}2\,(\tau_-{\cdot}1))$, as is easy to verify using the definition of higher-order abstraction kinds from Figure 13.4.

In summary, it depends on the abstract type's kind how the residual type has to be split and grounded. A more complex example with heterogeneous splitting will be given in Section 13.3.4.

As a final observation, we noted earlier that reduction only occurs for terms that are closed up to the type heap. That in fact is crucial to make reduction of coercions complete. As a counter example, consider the following:

$$\lambda\beta : \Omega \to \Omega.\, \{e : \alpha{:}\Omega.\, \beta\,\alpha\}^{+}_{\tau_+ \approx \tau_-}$$

It is not possible to simplify the coercion before β has been instantiated with a primitive type – reduction of higher-order coercions relies on η-expansion, and no such expansion is possible for a term typed with a parametric type variable (for abstract types, as we saw, the operational semantics knows their representation). That is the fundamental reason why we have to integrate higher-order coercions as an extension to the system, and could not define them merely as syntactic sugar. The latter would only be possible in a system without higher-order type constructors.

13.3. Kind Coercions

In the previous sections we introduced higher-order abstraction kinds and higher-order type coercions, and discovered that both require a notion of coercion on the type level, which we call a *kind coercion*.

13.3.1. Definition and Semantics

As already touched on, a kind coercion is a type expression of the following form:

$$\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-}$$

The expression takes type τ of the kind $\tilde{\kappa}[\tau_-/\alpha]$ and produces a type of kind $\tilde{\kappa}[\tau_+/\alpha]$, where both the *source* type τ_- and the *target* type τ_+ have kind κ. That is, we have a kinding rule as follows:

$$(\textsc{Tcoerce}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} : \tilde{\kappa}[\tau_+/\alpha]}$$

We refer to the kind $\tilde{\kappa}[\tau_-/\alpha]$ of τ as the *inward* kind of the coercion, and call the result kind $\tilde{\kappa}[\tau_+/\alpha]$ its *outward* kind.

In contrast to type coercions, where source and target types τ_- and τ_+ have to be related by a higher-order abstraction kind $\tau_+ : \mathrm{A}(\tau_- : \kappa')$, these types need not bear any relation beyond both having kind κ in a kind coercion. Consequently, no polarity is needed for kind coercions, because the inverse of $\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-}$ is simply $\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_-/\tau_+}$. The reason is that kind coercions need not reduce to a basic form of coercion over an abstract type. Instead, they basically express a generalised form of substitution between two arbitrary types.

Figure 13.10 gives three equivalence rules for kind coercions. The first one, TQcoerce*, is the usual congruence rule. Note that it does not require the residual kinds $\tilde{\kappa}$ and $\tilde{\kappa}'$ to be equivalent – they merely need to be equivalent after substituting the relevant types, which is a weaker requirement. The other two rules express two important observations. Rule TQcoerce-drop* says that a coercion between two equivalent kinds is redundant, i.e. an identity coercion that can be dropped. Rule TQcoerce-cancel* is the type-level analogue to the cancellation reduction rule (Rcoerce-cancel) for type coercions: it describes that nesting two inverse coercions also is an identity transformation. All three equivalence rules are crucial to proving the basic validity rule Tcoerce*, because they are exploited in the definition of quantified coercions.

A definition of kind coercions is given in Figure 13.9. Unlike type coercions, they can be defined as mere syntactic sugar: on the kind level there are no abstract constructors that have to be instantiated before simplification, so the shape of the residual kind is fixed. As we discussed in Section 13.2.4, the potential of free higher-order variables affecting the shape of a type preclude this for type coercions. The definition is otherwise analogous to type coercions. Like the reduction of the latter, it produces η-expansions. Particularly, it uses the same mystic

$$
\begin{array}{rcl}
\{\tau : \alpha{:}\kappa.\Omega\}_{\tau_+/\tau_-} & := & \tau \\
\{\tau : \alpha{:}\kappa.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-} & := & \tau'[\tau_+/\alpha] \\
\{\tau : \alpha{:}\kappa.\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\tau_+/\tau_-} & := & \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\{\tau\,\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+/\tau_-} \\
 & & \text{where } \tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1] \\
\{\tau : \alpha{:}\kappa.\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\tau_+/\tau_-} & := & \langle\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}, \{\tau{\cdot}2 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+/\tau_-}\rangle \\
 & & \text{where } \tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]
\end{array}
$$

Figure 13.9.: Kind coercions

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$
(\textsc{Tcoerce}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} : \tilde{\kappa}[\tau_+/\alpha]}
$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$
(\textsc{TQcoerce}^*)\ \frac{\begin{array}{cc} \Gamma \vdash \tau \equiv \tau' : \tilde{\kappa}[\tau_-/\alpha] & \Gamma \vdash \kappa \equiv \kappa' : \Box \\ \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box & \Gamma, \alpha{:}\kappa' \vdash \tilde{\kappa}' : \Box \\ \Gamma \vdash \tilde{\kappa}[\tau_+/\alpha] \equiv \tilde{\kappa}'[\tau_+'/\alpha] : \Box & \Gamma \vdash \tilde{\kappa}[\tau_-/\alpha] \equiv \tilde{\kappa}'[\tau_-'/\alpha] : \Box \\ \Gamma \vdash \tau_+ \equiv \tau_+' : \kappa & \Gamma \vdash \tau_- \equiv \tau_-' : \kappa \end{array}}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} \equiv \{\tau' : \alpha{:}\kappa'.\tilde{\kappa}'\}_{\tau_+'/\tau_-'} : \tilde{\kappa}[\tau_+/\alpha]}
$$

$$
(\textsc{TQcoerce-drop}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ \equiv \tau_- : \kappa}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}
$$

$$
(\textsc{TQcoerce-cancel}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \{\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_-/\tau_+} \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}
$$

Figure 13.10.: Rules for kind coercions

substitution on the codomain residual in the rules for quantifiers (here Π and Σ) that we have not explained yet.

Take universal quantification. Analogous to coercions at $\forall$-types on the term level (Section 13.2.3), the expansion of coercions at Π-kinds requires finding a residual kind $\tilde{\kappa}_2'$ that fulfils the following equations:

$$\begin{aligned}\tilde{\kappa}_2'[\tau_-/\alpha] &= \tilde{\kappa}_2[\tau_-/\alpha][\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}/\alpha_1] \\ \tilde{\kappa}_2'[\tau_+/\alpha] &= \tilde{\kappa}_2[\tau_+/\alpha]\end{aligned}$$

The first arises from the inward typing requirements of the surrounding coercion, the second from the outward ones.

Why is the kind

$$\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$$

used in Figure 13.9 a solution to these equations? Its trick is that it employs a coercion that is using α as its target type. This is the placeholder of the surrounding coercion. Consequently, when typing the surrounding coercion, α will either be substituted by τ_- when looking inward, or by τ_+ when looking outward. In the former case, the kind coercion will simply result in the right-hand side of the first equation, as required. In the latter case on the other hand, the coercion will become an identity coercion that can be dropped according to rule TQcoerce-drop*, leaving the second of the above equations.

Consider an example analogous to the one from Section 13.2.3 to demonstrate this:

$$\{\tau : \alpha{:}\kappa.\Pi\alpha_1{:}\mathrm{S}_\Omega(\alpha).\mathrm{S}_\Omega(\alpha_1)\}_{\beta/\tau_\beta}$$

According to Figure 13.9, this coercion expands to

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\beta).\{\tau\,\{\alpha_1 : \alpha{:}\kappa.\mathrm{S}_\Omega(\alpha)\}_{\tau_\beta/\beta} : \alpha{:}\kappa.\mathrm{S}_\Omega(\{\alpha_1 : \alpha{:}\kappa.\mathrm{S}_\Omega(\alpha)\}_{\alpha/\beta})\}_{\beta/\tau_\beta}$$

By further expanding the coercions in the argument type and the residual kind we arrive at

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\beta).\,\{\tau\,\tau_\beta : \alpha{:}\kappa.\mathrm{S}_\Omega(\alpha)\}_{\beta/\tau_\beta}$$

which has the correct kind, because $\Pi\alpha_1{:}\mathrm{S}_\Omega(\beta).\mathrm{S}_\Omega(\alpha_1) \equiv \Pi\alpha_1{:}\mathrm{S}_\Omega(\beta).\mathrm{S}_\Omega(\beta)$. In contrast, the same example with a non-singleton quantifier,

$$\{\tau : \alpha{:}\kappa.\Pi\alpha_1{:}\Omega.\mathrm{S}_\Omega(\alpha_1)\}_{\beta/\tau_\beta}$$

yields

$$\lambda\alpha_1{:}\Omega.\,\{\tau\,\alpha_1 : \alpha{:}\kappa.\mathrm{S}_\Omega(\alpha_1)\}_{\beta/\tau_\beta}$$

as required – here, τ is applied to α_1, not τ_β.

For Σ-kinds a similar trick is used. We state correctness of our definitions of higher-order coercions as admissibility of the well-formedness and equivalence rules given in Figure 13.10:

Theorem 38 (Admissibility of Kind Coercion Rules)**.**
The rules Tcoerce*, TQcoerce*, TQcoerce-drop* *and* TQcoerce-cancel* *are admissible.*

The proofs are by obvious induction and not techically difficult, but require a substantial amount of tedious 'computation', due to the size of the rules, as well as some of the expansions (particularly those involving dependencies), which multiply in complexity. See Appendix E.1.

13.3.2. Abstraction Kinds Revisited

After showing correctness of kind coercions we can prove admissibility of the rules from Figure 13.5:

Theorem 39 (Admissibility of Higher-Order Abstraction Kind Rules)**.**
The rules KABS*, KQABS*, KSABS* *and* KSABS-LEFT* *are admissible.*

Interestingly, unlike singletons, the subkinding rule KSABS* does not allow the annotated kind to vary. If we look at the expansions then we see that in the case of Σ-kinds a coercion is required at the constituent kind $\tilde{\kappa}_2$. However, two coerced types are generally not equivalent when going to a subkind in the residual kind, e.g. consider $\{\tau : \alpha{:}\Omega.\Omega\}_{\tau_+/\tau_-}$, which expands directly to τ, versus $\{\tau : \alpha{:}\Omega.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-}$, expanding to the incompatible $\tau'[\tau_+/\alpha]$. Consequently, rule KSABS* would not hold if we allowed $\tilde{\kappa}$ to vary.

13.3.3. Type Coercions Revisited

We can also go back to the reduction of type coercions at polymorphic types now. They employ the very same substitution that is used in the definition of kind coercions at dependent kind.

Recall the example from Section 13.2.3:

$$\{v : \alpha{:}\Omega.\forall\alpha_1{:}\mathrm{S}_\Omega(\alpha).\alpha_1\}^+_{\tau_+\approx\tau_-}$$

According to the rules in Figure 13.8, this coercion reduces to

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\beta).\{v\,\{\alpha_1 : \alpha{:}\Omega.\mathrm{S}_\Omega(\alpha)\}_{\tau_-/\tau_+} : \alpha{:}\Omega.\{\alpha_1 : \alpha{:}\Omega.\mathrm{S}_\Omega(\alpha)\}_{\alpha/\tau_+}\}_{\tau_+\approx\tau_-}$$

Inserting the definitions from Figure 13.9 yields

$$\lambda\alpha_1{:}\mathrm{S}_\Omega(\beta).\,\{v\,\tau_- : \alpha{:}\Omega.\alpha\}^+_{\tau_+\approx\tau_-}$$

which again is correct because $\forall\alpha_1{:}\mathrm{S}_\Omega(\alpha).\alpha_1 \equiv \forall\alpha_1{:}\mathrm{S}_\Omega(\alpha).\alpha$. Again, contrast this to a similar example with an non-singleton quantifier:

$$\{e : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\Omega.\alpha_1\}^+_{\tau_+\approx\tau_-}$$

which reduces to

$$\lambda\alpha_1{:}\Omega.\,\{v\,\alpha_1 : \alpha{:}\Omega.\alpha\}^+_{\tau_+\approx\tau_-}$$

The kind coercions ensure that the polymorphic function v is instantiated with a proper type in both cases.

13.3.4. Path Coercions

Recall the reduction rules RCOERCE-SWAP and RCOERCE-GROUND for coercion at abstract type. As we mentioned on in Section 13.2.4, they require replacing the placeholder in the head of a path $P[\alpha]$ with another type. In a type system without singletons we could show a replacement lemma that states that such a replacement is always possible if the replacement type is the same kind as (or a subkind of) α. However, with singletons, the most specific kind of α is the singleton $\mathrm{S}(\alpha)$. Consequently, the only replacement we can generally make is the trivial one, with an equivalent type, which is not particularly useful.

More specifically, in the aforementioned rules the redex contains a type path of the form $P[\alpha]$ with $\alpha : \tilde{\kappa}$. Even though we know $\tau_- : \tilde{\kappa}$, the type $P[\tau_-]$ that we want to form in the reduct may be malformed. Recall the example from Section 13.2.4:

$$
\begin{array}{lcl}
P[\tau_+/\tau_- : \tilde{\kappa}] & := & P[\alpha{:}\tilde{\kappa}\,.\,\alpha : \tilde{\kappa}]_{\tau_+/\tau_-} \\
\\
\_[\alpha{:}\tilde{\kappa}\,.\,\tau : \Omega]_{\tau_+/\tau_-} & := & \tau[\tau_+/\alpha] \\
\_[\alpha{:}\tilde{\kappa}\,.\,\tau : \mathrm{S}_\Omega(\tau')]_{\tau_+/\tau_-} & := & \tau[\tau_+/\alpha] \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_+/\tau_-}\,\tau') & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau\,\tau'' : \tilde{\kappa}_2[\tau''/\alpha_1]]_{\tau_+/\tau_-} \quad \text{where } \tau'' = \{\tau' : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_-} \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_+/\tau_-}\cdot 1) & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau\cdot 1 : \tilde{\kappa}_1]_{\tau_+/\tau_-} \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_-/\tau_-}\cdot 2) & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau\cdot 2 : \tilde{\kappa}_2[\tau\cdot 1/\alpha_1]]_{\tau_+/\tau_-}
\end{array}
$$

Figure 13.11.: Path coercions and replacement

$$\kappa_1 = \Sigma\alpha_1{:}\Omega.\Pi\alpha_2{:}\mathrm{S}_\Omega(\alpha_1).\Omega$$

Assuming $\alpha : \kappa_1$, the path $P[\alpha] = \alpha{\cdot}2\,(\alpha{\cdot}1)$ is well-formed, but forming $P[\tau_-] = \tau_-{\cdot}2\,(\alpha{\cdot}1)$ produces a kind mismatch with the constructors parameter kind $\mathrm{S}_\Omega(\tau_-)$ in the type application.

In general, to make replacement as used in the rules well-formed, we have to properly coerce the type arguments in the path P to fit the replacement type. That is what the path replacement notation $P[\tau'/\tau : \kappa]$ encapsulates. It is defined in Figure 13.11, and expands to a form of kind coercion on paths. More accurately, we could speak of a "co-coercion", because unlike kind coercions, it does not coerce an inward type to fit outward kinding requirements, but coerces the outward context of a type (the path P) to fit the kind of the inward type. Consequently, its inductive definition proceeds from the inside out, as opposed to kind coercions, which work in the opposite direction.

Consider the path $P[\alpha] = \alpha{\cdot}2\,(\alpha{\cdot}1)$ as before. A replacement of the head α with a type $\tau : \kappa_1$ will unfold as follows:

$$
\begin{array}{lcl}
P[\tau/\alpha : \kappa_1] & = & P[\alpha'{:}\kappa_1.\,\alpha' : \kappa_1]_{\tau/\alpha} \\
& = & [\alpha'{:}\kappa_1.\,\alpha' : \Sigma\alpha_1{:}\Omega.\Pi\alpha_2{:}\mathrm{S}_\Omega(\alpha_1).\Omega]_{\tau/\alpha}\cdot 2\,(\alpha{\cdot}1) \\
& = & [\alpha'{:}\kappa_1.\,\alpha{\cdot}2' : \Pi\alpha_2{:}\mathrm{S}_\Omega(\alpha'{\cdot}1).\Omega]_{\tau/\alpha}\,(\alpha{\cdot}1) \\
& = & [\alpha'{:}\kappa_1.\,\alpha{\cdot}2'\,\{\alpha{\cdot}1 : \alpha'{:}\kappa_1.\mathrm{S}_\Omega(\alpha'{\cdot}1)\}_{\alpha'/\alpha} : \Omega]_{\tau/\alpha} \\
& = & [\alpha'{:}\kappa_1.\,\alpha{\cdot}2'\,(\alpha'{\cdot}1) : \Omega]_{\tau/\alpha} \\
& = & \tau{\cdot}2\,(\tau{\cdot}1)
\end{array}
$$

Given that definition of path replacement, a coercion over a respective type, for example,

$$\{v : \alpha{:}\kappa_1.\,\alpha{\cdot}2\,(\alpha{\cdot}1)\}^+_{\tau_+\approx\tau_-}$$

is reduced by rule Rcoerce-ground to

$$\{\{v : \alpha{:}\kappa_1.\,\tau_-{\cdot}2\,(\tau_-{\cdot}1)\}^+_{\tau_+\approx\tau_-} : \alpha{:}\Omega.\alpha\}^+_{\tau_+{\cdot}2\,(\tau_+{\cdot}1)\approx\tau_-{\cdot}2\,(\tau_-{\cdot}1)}$$

It is easy to check by expansion of the definition of higher-order abstraction kinds that $\tau_+{\cdot}2\,(\tau_+{\cdot}1)$ actually has kind $\mathrm{A}(\tau_-{\cdot}2\,(\tau_-{\cdot}1))$, assuming $\tau_+ : \mathrm{A}(\tau_- : \kappa_1)$.

More interestingly, as noted earlier, the reduction does *not* actually split the residual type $\alpha{\cdot}2\,(\alpha{\cdot}1)$, but instead grounds the type in one step. Indeed, this is the only possible way to reduce the coercion, because their exists no well-formed splitting of the type $\alpha{\cdot}2\,(\alpha{\cdot}1)$ that would separate the two occurrences of α.

To see how the definition of path replacement can cause Rcoerce-ground to split in a heterogeneous manner, consider a more complex example with the following kind:

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\textsc{Treplace}^*)\ \frac{\Gamma \vdash P[\tau_-] : \Omega \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \tilde{\kappa}}{\Gamma \vdash P[\tau_+/\tau_- : \tilde{\kappa}] : \Omega}$$

$$(\textsc{Treplace-abs}^*)\ \frac{\Gamma \vdash P[\tau] : \Omega \qquad \Gamma \vdash \tau : \tilde{\kappa} \qquad \Gamma \vdash \tau_- : \tilde{\kappa} \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})}{\Gamma \vdash P[\tau_+/\tau : \tilde{\kappa}] : \mathrm{A}(P[\tau_-/\tau : \tilde{\kappa}])}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\textsc{TQreplace-subst-up}^*)\ \frac{\Gamma \vdash P[\tau] : \Omega \qquad \Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash P[\alpha/\tau : \tilde{\kappa}][\tau/\alpha] \equiv P[\tau] : \Omega}\ (\alpha \notin \mathrm{FV}(P))$$

$$(\textsc{TQreplace-subst-dn}^*)\ \frac{\Gamma, \alpha{:}\tilde{\kappa} \vdash P[\alpha] : \Omega \qquad \Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash P[\tau/\alpha : \tilde{\kappa}][\tau/\alpha] \equiv P[\alpha][\tau/\alpha] : \Omega}$$

Figure 13.12.: Rules for path coercions

$\kappa_2 = \Pi\alpha_1{:}\Omega.\Sigma\alpha_2{:}\Omega.\Pi\alpha_3{:}\Omega.\Pi\alpha_4{:}\mathrm{S}_\Omega(\alpha_1).\Omega$

Here, given $P[\alpha] = \alpha\ \mathit{int}{\cdot}2\ (\alpha\ \mathit{string}{\cdot}1)\ (\alpha\ \mathit{int}{\cdot}1)$, the path replacement will yield

$$P[\tau_-/\alpha : \kappa_2] \quad = \quad \tau_-\ \mathit{int}{\cdot}2\ (\alpha\ \mathit{string}{\cdot}1)\ (\tau_-\ \mathit{int}{\cdot}1)$$

Note that only the occurrence of α in the latter argument has been replaced (because the corresponding singleton kind necessitates it), while the other has been left untouched (because its choice is independent from the path's head). Applied in an actual instance of rule Rcoerce-ground, this example will yield non-trivial splitting combined with non-trivial grounding:

$$\begin{array}{l}\{v : \alpha{:}\kappa_2.\ \alpha\ \mathit{int}{\cdot}2\ (\alpha\ \mathit{string}{\cdot}1)\ (\alpha\ \mathit{int}{\cdot}1)\}^+_{\tau_+ \approx \tau_-} \quad \rightarrow \\ \{\{v : \alpha{:}\kappa_2.\ \tau_-\ \mathit{int}{\cdot}2\ (\alpha\ \mathit{string}{\cdot}1)\ (\tau_-\ \mathit{int}{\cdot}1)\}^+_{\tau_+ \approx \tau_-} : \alpha{:}\Omega.\alpha\}^+_{\tau'_+ \approx \tau'_-}\end{array}$$

with

$$\begin{array}{lcl}\tau'_+ & = & \tau_+\ \mathit{int}{\cdot}2\ (\tau_+\ \mathit{string}{\cdot}1)\ (\tau_+\ \mathit{int}{\cdot}1) \\ \tau'_- & = & \tau_-\ \mathit{int}{\cdot}2\ (\tau_+\ \mathit{string}{\cdot}1)\ (\tau_-\ \mathit{int}{\cdot}1)\end{array}$$

We leave it as an exercise to the reader to verify that the reduct in fact is well-formed.

In general, the soundness of rule Rcoerce-ground depends on a number of rules for path coercions that are collected in Figure 13.12. These rules are admissible:

Theorem 40 (Admissibility of Path Coercion Rules)**.** *The rules* Treplace*, Treplace-abs*, TQreplace-subst-up* *and* TQreplace-subst-dn* *are admissible.*

Rule Treplace-abs* is the most important one, because it is this rule that allows grounding of higher-order coercions.

$$\begin{array}{lrcl} \text{concrete kinds} & \tilde{\kappa} & ::= & \Omega \mid \mathrm{S}_\Omega(\tau) \mid \Pi\alpha{:}\tilde{\kappa}.\tilde{\kappa} \mid \Sigma\alpha{:}\tilde{\kappa}.\tilde{\kappa} \\ \text{concrete types} & \tilde{\tau} & ::= & \alpha \mid \Psi \mid \tilde{\tau} \to \tilde{\tau} \mid \tilde{\tau} \times \tilde{\tau} \mid \forall\alpha{:}\tilde{\kappa}.\tilde{\tau} \mid \exists\alpha{:}\tilde{\kappa}.\tilde{\tau} \\ & & & \mid \lambda\alpha{:}\tilde{\kappa}.\tilde{\tau} \mid \tilde{\tau}\,\tilde{\tau} \mid \langle\tilde{\tau},\tilde{\tau}\rangle \mid \tilde{\tau}{\cdot}1 \mid \tilde{\tau}{\cdot}2 \end{array}$$

Figure 13.13.: Concrete kinds and types

13.3.5. The Concrete Kind Restriction

We have limited higher-order generativity and coercions to *concrete* kinds and types. They are defined in Figure 13.13 and form a syntactic subclass of kinds and types devoid of occurrences of abstraction kinds.

The fundamental reason for these restrictions is that we could not express kind coercions with a non-concrete residual kind as derived. A coercion like

$$\{\tau : \alpha{:}\kappa.\mathrm{A}(\tau')\}_{\tau_+/\tau_-}$$

cannot be expanded, because there is no extensionality principle for abstraction kinds and hence no η-expansion possible – the same reason for that we had to make singletons over abstraction kind primitive (Section 11.3.1). More specifically, there is no way to define such a coercion by expansion, such that it obeys the rules from Figure 13.10 and allows deriving the following judgement, for instance:

$$\beta{:}\mathrm{A}(1), \gamma{:}\mathrm{A}(\beta \times \beta) \vdash \{\gamma : \alpha{:}\Omega.\mathrm{A}(\beta \times \alpha)\}_{1/\beta} : \mathrm{A}(\beta \times 1)$$

We cannot construct any type expression from γ that would have the desired kind $\mathrm{A}(\beta \times 1)$.

The restriction to concrete kinds propagates to the other extensions discussed in this chapter when combining them:

- Abstraction kinds cannot be formed over non-concrete kinds (even if we made nested abstract kinds primitive like the respective singletons), because the expansion for abstraction kinds over Σ-kinds requires a coercion that uses it as its residual kind.
- Consequently, types cannot be generated at non-concrete kind with new, because their abstraction kind could not be expressed. Neither can type coercions operate under non-concrete kind.
- Type coercions cannot be performed at non-concrete types, because that may produce kind coercions at non-concrete kind in the polymorphic cases (rules RCOERCE-UNIV and RCOERCE-EXIST).
- Finally, type generation may not use non-concrete representation types, because the reduction rules RCOERCE-SPLIT and RCOERCE-SWAP substitute arbitrary representation types from the heap into a residual type.

All these restrictions essentially boil down to the fact that the system simply does not support "nested" abstraction kinds of the form $\mathrm{A}(\tau : \mathrm{A}(\tau'))$. The only possiblity to avoid these restrictions would be to introduce kind coercions, like term-level coercions, as a primitive notion – including the rules TCOERCE*, TQCOERCE*, TQCOERCE-CANCEL*, and TQCOERCE-DROP* as integral parts of the type validity and equivalence judgements. The latter however would vastly complicate the meta-theory of the system, especially with respect to the already complicated type and kind equivalences.

On the other hand, it is not obvious what purpose nested abstraction kinds should serve – they do not appear in the encoding of examples from ML, since abstraction kinds do not exist in the source language. For that reason, the complication did not seem warranted in the context of the current work, and we chose to restrict higher-order abstraction to the concrete types and kinds defined in Figure 13.13. Essentially, this is a simple form of predicativity with respect to abstraction kinds.

To maintain the restriction during reduction, type expressions also have to be restricted to conrete types in other places of the calculus, namely for instantiation and existential formation. All changes to the syntax of the combined system are collected in Appendix A.2.1. To avoid notational clutter, we employ the convention that types named τ_- or τ_+ are always concrete type expressions.

13.4. Properties

After solving all the higher-orderness puzzles, what remains to be done now is to show that soundness and opacity are maintained for the calculus with extensions. For the former, we first need to adapt our type synthesis algorithm to the extended forms of coercions and generativity.

13.4.1. Algorithmic Type Synthesis

The extended expression forms for higher-order type generation and higher-order coercions are richly decorated with explicit type annotations, so adaption of the algorithm for type synthesis is easy. Figure 13.14 shows the changed rules. Showing Correctness is likewise easy:

Theorem 41 (Soundness of Algorithmic Type Synthesis with Higher-Order Abstraction)**.**
Let $\Gamma \vdash \square$.

1. *If* $\Gamma \triangleright e \rightrightarrows \tau$, *then* $\Gamma \vdash e : \tau$.
2. *If* $\Gamma \triangleright e \Rightarrow\!\!\!\!\Rightarrow \pi$, *then* $\Gamma \vdash e : \pi$.
3. *If* $\Gamma \triangleright e \leftleftarrows \tau$ *with* $\Gamma \vdash \tau : \Omega$, *then* $\Gamma \vdash e : \tau$.

Theorem 42 (Completeness of Algorithmic Type Synthesis with Higher-Order Abstraction)**.**

1. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \rightrightarrows \tau'$ *with* $\Gamma \vdash \tau' \leq \tau : \Omega$.
2. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \Rightarrow\!\!\!\!\Rightarrow \pi$ *with* $\Gamma \vdash \pi \leq \tau : \Omega$.
3. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leq \tau' : \Omega$, *then* $\Gamma \triangleright e \leftleftarrows \tau'$.

The decidability result thus extends trivially.

Note that we do not need to adapt any other algorithm, because we did not change the type language – we merely defined higher-order abstraction kinds and kind coercions as derived forms.

13.4.2. Soundness

As a preliminary, we state the obvious fact that the extensions do not compromise the Validity property:

Proposition 43 (Validity with Higher-Order Abstraction)**.**
If $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash \tau : \Omega$.

Preservation can be formulated unchanged:

Type Synthesis

$\Gamma \triangleright \mathsf{new}\, \alpha{:}\tilde{\kappa} \approx \tilde{\tau}_1\ \mathsf{in}_\tau\ e_2 \rightrightarrows \tau$ if $\Gamma \triangleright \tilde{\kappa} : \Box$ and $\Gamma \triangleright \tilde{\tau}_1 \leftleftarrows \tilde{\kappa}$ and $\Gamma \triangleright \tau \leftleftarrows \Omega$ and $\Gamma, \alpha{:}\mathrm{A}(\tau_1) \triangleright e_2 \leftleftarrows \tau$

$\Gamma \triangleright \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-} \rightrightarrows \tilde{\tau}[\tau_+/\alpha]$ if $\Gamma \triangleright \tilde{\kappa} : \Box$ and $\Gamma \triangleright \tau_- \leftleftarrows \tilde{\kappa}$ and $\Gamma \triangleright \tau_+ \leftleftarrows \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \leftleftarrows \Omega$ and $\Gamma \triangleright e \leftleftarrows \tilde{\tau}[\tau_-/\alpha]$

$\Gamma \triangleright \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tau_+ \approx \tau_-} \rightrightarrows \tilde{\tau}[\tau_-/\alpha]$ if $\Gamma \triangleright \tilde{\kappa} : \Box$ and $\Gamma \triangleright \tau_- \leftleftarrows \tilde{\kappa}$ and $\Gamma \triangleright \tau_+ \leftleftarrows \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \leftleftarrows \Omega$ and $\Gamma \triangleright e \leftleftarrows \tilde{\tau}[\tau_+/\alpha]$

Figure 13.14.: Algorithmic type synthesis for higher-order abstraction

---

Theorem 44 (Preservation with Higher-Order Abstraction)**.**

1. *If* $\Delta \vdash e : \tau$ *and* $\Delta; e \to \Delta'; e'$ *with* $E = \_$*, then* $\Delta' \vdash e' : \tau$*.*
2. *If* $\cdot \vdash C : \tau$ *and* $C \to C'$*, then* $\cdot \vdash C' : \tau$*.*

The new cases of the proof are quite involved, due to the complexity of the rules for quantified types. Note however, that the Representation Equivalence property holds unchanged, thanks to higher-order abstraction kinds being derived.

For Progress, the canonical forms lemma has to be adapted to the generalised coercion syntax. Higher-order coercions do not change the form of values otherwise:

Proposition 45 (Canonical Values with Higher-Order Abstraction)**.**
Let $\Gamma \vdash v : \tau$*.*

1. *If* $\Gamma \vdash \tau \leq \tau_+ : \Omega$ *and* $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_-)$*, then* $v = \{v_1 : \alpha{:}\Omega.\alpha\}^{+}_{\tau'_+ \approx \tau'_-}$*.*
2. *If* $\Gamma \vdash \tau \leq \Psi : \Omega$*, then* $v = \psi(v_1)$*.*
3. *If* $\Gamma \vdash \tau \leq \tau_1 \to \tau_2 : \Omega$*, then* $v = \lambda x{:}\tau'_1.e_2$*.*
4. *If* $\Gamma \vdash \tau \leq \tau_1 \times \tau_2 : \Omega$*, then* $v = \langle v_1, v_2 \rangle$*.*
5. *If* $\Gamma \vdash \tau \leq \forall \alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \lambda \alpha{:}\kappa'_1.e_2$*.*
6. *If* $\Gamma \vdash \tau \leq \exists \alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \langle \tau_1, v_2 \rangle$*.*

We can then state Progress as before:

Theorem 46 (Progress with Higher-Order Abstraction)**.**
If $\Delta \vdash e : \tau$*, then either* $e = v$*, or* $(\Delta; e) = (\Delta; E[e_1]) \to (\Delta'; E[e'_1]) = (\Delta'; e')$*.*

The proofs can be found in Appendix E.

13.4.3. Opacity

For the Opacity theorem, we only need to generalise the substitution to an arbitrary path over α_i, everything else stays the same:

Theorem 47 (Opacity with Higher-Order Abstraction)**.**
Let $\Gamma = \alpha{:}\Omega, x{:}\alpha, f{:}\alpha{\to}1$*, and* $\gamma_i = [P[\alpha_i]/\alpha] \cup [v_i/x] \cup [v'_i/f] \cup [\alpha'/\alpha' \mid \alpha' \in \mathrm{Dom}(\Delta')]$ *with* $\alpha \notin \mathrm{Dom}(\Delta, \Delta')$ *and* $\alpha_i \notin \mathrm{Dom}(\Delta)$ *and* $\Delta, \gamma_i(\Delta') \vdash \gamma_i : \Gamma, \Delta'$ *and* $\Delta; v'_i\, v_i \to^* \Delta; \diamond$ *for* $i \in \{1, 2\}$*. Furthermore,* $\Delta(\alpha_1) = \mathrm{A}(\tau_1 : \kappa)$ *and* $\Delta(\alpha_2) = \mathrm{A}(\tau_2 : \kappa)$ *for some* κ*.*

$$
\begin{array}{lcl}
e :> \chi & := & e \\
e :> \Psi & := & e \\
e :> \tilde{\tau}_1 \to \tilde{\tau}_2 & := & e \\
e :> \tilde{\tau}_1 \times \tilde{\tau}_2 & := & e \\
e :> \forall \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 & := & e \\
e :> \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 & := & \mathsf{let}\langle \alpha, x\rangle = e \ \mathsf{in}_{\exists \alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \mathsf{new}\, \alpha'{:}\tilde{\kappa}_1 \approx \alpha \ \mathsf{in}_{\exists \alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \langle \alpha', \{x : \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^{+}_{\alpha' \approx \alpha}\rangle
\end{array}
$$

Figure 13.15.: Sealing

Term Validity $\boxed{\Gamma \vdash e : \tau}$

$$(\textsc{Eseal}^*)\ \frac{\Gamma \vdash e : \tilde{\tau}}{\Gamma \vdash e :> \tilde{\tau} : \tilde{\tau}}$$

Figure 13.16.: Admissible rule for sealing

1. *Let* $\Gamma, \Delta' \vdash \tau : \kappa'$ *and* $\Gamma, \Delta' \vdash \tau' : \kappa'$.
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(\tau) \leq \gamma_1(\tau') : \gamma_1(\kappa')$, *then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) \leq \gamma_2(\tau') : \gamma_2(\kappa')$.
2. *Let* $\Gamma, \Delta' \vdash \square$.
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(e) : \gamma_1(\tau)$, *then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(e) : \gamma_2(\tau)$.
3. *Let* $\Gamma, \Delta' \vdash e : \tau$.
   If and only if $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to^* \Delta, \gamma_1(\Delta'), \Delta_1; v'$, *then* $\Delta_1 = \gamma_1(\Delta'')$ *and* $v' = \gamma_1(v)$ *with* $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to^* \Delta, \gamma_2(\Delta'), \gamma_2(\Delta''); \gamma_2(v)$.

13.5. Sealing

We now have all the necessary ingredients to define a sealing operator $e :> \tau$ in our calculus that mirrors the respective operation in the ML module system and thus enables *a posteriori* abstraction over arbitrary 'signature' types τ.

The definition can be seen in Figure 13.15. The only relevant case is for existential types: the type in the existential is replaced by a newly generated abstract type of the same kind, and the value is wrapped in a suitable coercion to adapt it to the abstract type, such that the overall type is the same as the signature type. The admissibility of the typing rule $\textsc{Eseal}^*$ in Figure 13.16 proves this fact:

Theorem 48 (Admissibility of Sealing Rule)**.**
The rule $\textsc{Eseal}^*$ *is admissible.*

For a concrete example of sealing at work, consider the expression

$$C :> \mathit{COMPLEX}$$

from Section 10.7. It expands to

$$\mathsf{let}\langle \alpha, x\rangle = C' \ \mathsf{in}\ \mathsf{new}\, c{:}\Omega \approx \alpha \ \mathsf{in}\ \langle c, \{x : \alpha{:}\Omega.\, \mathrm{Dyn}(\mathit{COMPLEX})\}^{+}_{c \approx \alpha}\rangle$$

which then reduces to

$$\mathsf{new}\ c{:}\Omega \approx \mathit{real} \times \mathit{real}\ \mathsf{in}\ \langle c, \{\langle \ldots \rangle : \alpha{:}\Omega.\, \mathrm{Dyn}(\mathit{COMPLEX})\}^{+}_{c \approx \mathit{real} \times \mathit{real}} \rangle$$

yielding a result equivalent (up to η-expansion) to the abstraction-safe definition of C' in Section 10.6.

But the definition of sealing can cope with more complex examples. Consider the following functor in ML that creates abstract collection types for sets and maps over a given element type:

```
functor Coll (type elem; . . .) :>
    sig
        type elem = elem
        type set
        type α map
        . . .
    end =
    struct
        type elem = elem
        type set = elem tree
        type α map = (elem × α) tree
        . . .
    end
```

A possible transliteration to $\lambda^{\omega}_{\mathrm{SA}\Psi}$ is the following:

$$\begin{aligned}\lambda \mathit{elem} : \Omega.\, \lambda x : \tau.\, &\langle \langle \mathit{elem}, \mathit{tree}\ \mathit{elem}, \lambda\alpha : \Omega.\, \mathit{tree}\ (\mathit{elem} \times \alpha) \rangle, \langle \ldots \rangle \rangle \\ &:> \exists \alpha : (\mathrm{S}_{\Omega}(\mathit{elem}) \times \Omega \times (\Omega \to \Omega)).\, \tau'\end{aligned}$$

Note the quantification over a triple of types, one of which is singleton, and another is higher-order. Expanding the definition of sealing produces a new-expression of the following form:

$$\begin{aligned}&\mathsf{new}\ \alpha : (\mathrm{S}_{\Omega}(\mathit{elem}) \times \Omega \times (\Omega \to \Omega)) \approx \langle \mathit{elem}, \mathit{tree}\ \mathit{elem}, \lambda\alpha : \Omega.\, \mathit{tree}\ (\mathit{elem} \times \alpha) \rangle \\ &\mathsf{in}\ \langle \alpha, \{x' : \alpha{:}(\mathrm{S}_{\Omega}(\mathit{elem}) \times \Omega \times (\Omega \to \Omega)).\tau'\}^{+}_{\alpha} \rangle\end{aligned}$$

The type name α, which apears free in τ', captures all types defined in the signature: $\alpha{\cdot}1$ is elem and transparent, the abstract set is defined as $\alpha{\cdot}2{\cdot}1$ while map is given by $\alpha{\cdot}2{\cdot}2$.

According to our definition, sealing has no effect on functions and universal types. In ML this would mean that sealing at functor signatures,

$$\mathsf{F} :> \mathbf{fct}\ \mathsf{X} : \mathsf{S}_1 \to \mathsf{S}_2$$

does not have any effect. Less trivial semantics are possible, but would require giving a more elaborate translation scheme for modules. In Appendix B we sketch such a scheme, but a more in-depth treatment is beyond the scope of this thesis.

Also note that for constructor types sealing has no effect either. This corresponds to the fact that in ML, signatures are not higher-order. In Alice ML, abstract signatures (Section 3.3.2) in fact enable limited abstract sealing:

$$\mathbf{functor}\ \mathsf{F}\ (\mathbf{signature}\ \mathsf{S})\ (\mathsf{X} : \mathsf{S}) = \mathsf{X} :> \mathsf{S}$$

Such an example would roughly correspond to sealing with a type variable in the calculus. If we wanted to have a semantics covering this corner of Alice ML, it would be straightforward to realise by making sealing primitive and turning the syntactic definition into reduction rules. However, since this feature is of little practical utility and does not provide any new insight, we omit the details.

13.6. Discussion and Related Work

13.6.1. Design Space

No question: the higher-order extensions we presented in this chapter are complex. Higher-order coercions have surprisingly complicated reduction rules, and even higher-order abstraction kinds are much less obvious than one might expect or like. It took us the better part of a year to solve the puzzle, come up with the definitions and rules as presented in this chapter, and wade through the lengthy proofs – including backtracking on several occasions.

A simpler system clearly would have been more desirable. Unfortunately, we see no way of avoiding the complexity, short of depriving the system of essential expressiveness. There does not appear to be much of a design space regarding coercions, most of the rules were forced upon us by consistency and soundness requirements. We thus believe that the calculus we presented is relatively canonical, despite its complexity.

Higher-order type coercions essentially describe the type transformation happening with sealing. The main reason for their complexity is the presence of dependent kinds, which are due to singletons and abstraction kinds. In particular, dependent kinds necessitate kind coercions, the primary complication of the system. If we eliminated singletons then the calculus could no longer express dynamic type sharing, which is one of the core aspects of the Alice ML type system we wanted to model. If we eliminated abstraction kinds on the other hand, then generativity had to be accompanied by a more ad-hoc mechanism that might not have the desired properties (see the discussion on related work below).

One might consider getting away without coercions altogether. The distinction between abstract types and their representation is needed for the *dynamic* semantics, in order to ensure abstraction safety in the presence of type analysis. In the *static* semantics, scoping and quantification are sufficient. As the Opacity proof shows, coercions themselves are actually inessential for abstraction safety. We could hence contemplate getting rid of coercions by making a distinction between the static and the dynamic semantics, e.g. by simply assigning new-bound type variables singleton kind in the static typing rules, but opaque kind in the heap. This approach does in fact work in a sufficiently conservative type system, as has been shown by Berg [Ber04] (relying on special environment entries instead of singleton kinds). In the presence of dependent kinds it is problematic, however. To see why, consider the following example:

$$\mathsf{new}\,\alpha{:}\Omega \approx \mathit{int}\ \mathsf{in\ case}\ 666 : \alpha\ \mathsf{of}\ x : ((\lambda\beta : \mathrm{S}(\mathit{int}).\,\beta)\,\alpha).\ 0\ \mathsf{else}\ 1$$

Assuming that α is assigned kind $\mathrm{S}(\mathit{int})$ by the static typing rules (i.e. considered equivalent to int), this example – especially the contained type application – would be well-formed. However, if the heap recorded the generated type opaquely as $\alpha{:}\Omega$ like suggested, then the dynamic condition to be checked for the reduction of the type case would become

$$\alpha : \Omega \vdash \alpha \leq (\lambda\beta : \mathrm{S}(\mathit{int}).\,\beta)\,\alpha : \Omega$$

Obviously, the application on the right-hand side of this judgement is not well-formed, because the argument α does not have the kind $\mathrm{S}(\mathit{int})$ required by the type function.

For a slightly more conservative solution, we might think about making coercions *implicit*. Two subsumption-like typing rules aware of abstraction kinds would replace the explicit coercion rules:

$$\frac{\Gamma \vdash e : \tilde\tau[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\tilde\kappa \vdash \tilde\tau : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_-{:}\tilde\kappa) \qquad \Gamma \vdash \tau_- : \tilde\kappa}{\Gamma \vdash e : \tilde\tau[\tau_+/\alpha]}$$

$$\frac{\Gamma \vdash e : \tilde\tau[\tau_+/\alpha] \qquad \Gamma, \alpha{:}\tilde\kappa \vdash \tilde\tau : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_-{:}\tilde\kappa) \qquad \Gamma \vdash \tau_- : \tilde\kappa}{\Gamma \vdash e : \tilde\tau[\tau_-/\alpha]}$$

The problem with these rules is that, for obvious reasons, the *unique* or *least typing* property of the type system would be lost, unless it is reformulated up to isomorphic types. Consequently, it is not clear at all if and how to define a type checking algorithm that can cope with such rules. It seems that the Type Analsis judgement (cf. Section 12.7) would not merely have to check subtyping, but actually guess an arbitrarily long sequence of suitable tuples $(\tilde{\kappa}, \tilde{\tau}, \tau_+, \tau_-)$, possibly interleaved with subtyping, to solve the type constraint. Hence, it is not clear whether such a system would enjoy decidable type checking.

In addition to these considerations, it has to be stressed that, even if we were able to get rid of type coercions, we could still not get rid of kind coercions, because they already appear in the definition of higher-order abstraction kinds. There is no obvious way to avoid them, and as we said, they actually induce most of the complication.

As for their operational cost, coercions are required solely for the purpose of type checking. Operationally, they can be considered identity functions. A type erasing translation of type analysis [CWM98] could reasonably erase all abstractions and their corresponding coercions and collapse the redundant η-expansions. Consequently, coercions do not impose the potential operational overhead that was observed for the similar concept of boxing/unboxing operations [MG98].

13.6.2. Related Work

Notions similar to coercions are a recurring scheme in literature. For example, Leroy uses an analogous tranformation technique to express higher-order boxing/unboxing optimizations [Ler92]. However, we are not aware of any work that uses something like coercions in the context of a type system as expressive as ours, especially with respect to higher-order polymorphism and dependent kinds.

Grossman, Morrisett and Zdancewic proposed *abstraction brackets* as a proof technique for abstraction [GMZ00]. They present a calculus that uses annotated brackets as special syntax for marking abstraction boundaries during reduction. These brackets are very similar to our higher-order coercions over ground types. However, in their system abstraction brackets are not polarised, i.e. it does not distinguish between upward and downward coercions. Instead, all directly nested brackets are collapsed on reduction and annotated with the sequence of *principals* that own the corresponding abstractions. This approach would become very complex in a rich type system like the one we are considering, due to all the annotations involved. Their calculus does not incorporate higher-order abstraction. More importantly, it cannot express dynamic abstraction, but requires identifying a fixed set of principals statically, since technically, the reduction relation has to be extended for each occurring abstraction. In the paper, they prove a simple abstraction property they call *Value Abstraction*, which roughly is equivalent to the secrecy half of our Opacity property, but without considering type analysis.

Crary [Cra00] presents a coercion calculus for eliminating subtyping and bounded quantification. His language is equipped with intersection types, but does not feature higher-order types or dependent kinds. Unlike our work, his calculus expresses higher-order coercions by a separate coercion language, distinguished from terms. A canonicalization step then collapses these coercion expressions into ordinary terms that are roughly equivalent to the reducts for our higher-order coercions. The advantage of this separation is that Crary can immediately prove an erasure result implying that coercions have no impact on the operational behaviour of terms. However, it is not obvious whether his approach carries over to higher-order types. Interestingly, Crary's development requires defining a function map that applies a pair of positive and negative coercions to all occurrences of a type variable in a type. This function thus is roughly equivalent to our notion of kind coercion, but thanks to the absence of higher-order types and dependent

kinds its definition is much more straightforward than what is found in our system.

In earlier work [Ros03a], we already presented a calculus for type generativity that also features higher-order coercions. The system did not have abstraction kinds, so that the normal form of values of abstract type was more ad-hoc, because the abstract type could not be normalised to ground kind – basically, abstract values had the form $\{v : P[\alpha]\}^+_{\alpha\approx\tau\_}$. Coercions in that calculus also differed by not using a placeholder variable. Instead, they always coerced all occurrences of the abstract type name. That resulted in much more complex reduction rules for abstract coercions and the need for a simple built-in notion of kind coercions, which we were able to avoid in $\lambda^{\omega}_{\mathrm{SA}\Psi}$.

Like us, Leifer et al. are concerned with abstraction-safe marshalling and employ type generativity to achieve it [LPSW03], although in a much more limited type system without polymorphism. As already described in Section 12.10.3, generativity is static in their system. Thus, Grossman et al's abstraction brackets are enough for dealing with type conversions over a statically fixed number of types.

Another system very close to ours was devised by Vytiniotis, Washburn & Weirich [VWW05]. They also combine a (richer) type analysis construct with type generativity and higher-order coercions for achieving type abstraction. Their system distinguishes between primitive and higher-order coercions: once its residual type is simplified to path form, a higher-order coercion is reduced to primitive form. Primitive coercions look like the simple coercions from our basic calculus, and 'forget' the residual type annotation: $\{v : \alpha{:}(\Omega \to \Omega).\alpha\,\tau\}^+_{\tau_+\approx\tau\_}$ reduces to $\{v\}^+_{\tau_+\approx\tau\_}$. In our system in contrast, all type information is kept, because it reduces to the equivalent of $\{v\}^+_{\tau_+\tau\approx\tau\_\tau}$ instead. The loss of type information results in a lack of a unique type property. For example, $\{3\}^+_{\alpha\approx\lambda\beta.int}$ could be assigned infinitely many incompatible types in their system, among them $\alpha\,bool$ and $\alpha\,(int{\to}int)$. This probably does not compromise abstraction safety, although it is not entirely clear what effect it has on typing tricks like *phantom types*. Also, it is not obvious that type checking is still decidable, considering higher-order cases like $\{3\}^+_{\alpha\approx\lambda\beta:\Omega\to\Omega.int}$ that may ask for higher-order unification to infer a suitable higher-order argument to α.

None of the aforementioned works incorporates singleton kinds or subtyping, and we are not aware of any other work that combines dynamic type abstraction with full-fledged singletons and thus is able to express Alice ML's dynamic type sharing as explained in Section 4.4. Even higher-order type generation is only considered by our own prior work and by Vytiniotis et al., as discussed above. Both these works do not describe systems with pair kinds and thus cannot express module sealing in the uniform way we have shown here.

13.7. Summary

- Higher-order abstract types require higher-order generativity as primitive.
- Higher-order abstraction kinds are definable, analogous to singletons.
- Higher-order type coercions express *a posteriori* abstraction.
- Dependent abstraction kinds and type coercions at quantified types necessitate kind coercions.
- Kind coercions can be defined as non-trivial derived forms.
- Sealing is expressible with higher-order generativity and coercions.

14. Conclusion and Future Work

14.1. Conclusion

Over the course of the last decade, open programming has become increasingly important as a paradigm for software development: net-oriented applications no longer run in isolation and rarely consist of closed, statically composed programs. Nevertheless, open programming has yet received relatively little attention in programming language design. In particular, there has been little work on satisfactorily reconciling it with strong static typing.

In this dissertation, we have made an attempt at addressing that issue.

- We have presented the design of a dynamic system of components that combines strong typing with open programming. At its core it requires little more than a decent module system with structural interfaces and a generic pickling mechanism, which are sufficient to express type-safe dynamic linking, persistence, and inter-process communication. Because of its economics, the approach potentially is applicable to a large class of languages.

- Specifically, we have designed Alice ML, a language that combines an expressive type and module system in the tradition of ML with fairly advanced open programming concepts, which are realised by a coherent and relatively minimal set of extensions. The language features higher-order modules, light-weight concurrency, a first-class component system with dynamic linking, and type-safe persistence and distribution. All these features have a comparably simple yet flexible semantics and are arranged for practical use. The language guarantees type safety for intra- *and* extra-linguistically created objects, and abstraction safety for intra-linguistic ones.

- We have developed a formal semantics that captures the essentials of Alice ML's type and module system in the context of a polymorphic λ-calculus. In particular, we have given an abstraction-safe operational semantics for sealing based on dynamic type generativity. It incorporates the novel notion of abstraction kinds and includes a definition of higher-order coercions that encompasses dependent kinds. We proved decidability, soundness and a moderate abstraction property for this calculus.

- As part of a larger project, we have implemented the full-fledged language with the aforementioned features and semantics in the Alice Programming System, which is available as open source software.

We believe that our design is solid and represents a significant improvement over the mainstream's state of the art exemplified by Java and related languages. It also improves on previous more research-oriented languages like Oz by reconciling open programming concepts with strong typing.

We hope that future language designers will find some of the presented ideas – and the Alice ML language – inspiring. There is no doubt that they will also find plenty of potential for improving on and extending the presented design.

May types make the Net a better type of place!

14.2. Future Work

Our work has touched many areas of programming language design, theory, and implementation. Naturally, this leaves many paths for future exploration.

Possible future directions in the language design and implementation of Alice ML have already been enumerated in Section 9.4. Here, we only consider the theoretical side represented by the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus.

Theory-wise, the following questions look particularly interesting:

- **Lazy types.** One interesting aspect of Alice ML that we left out of our formal treatment is laziness on the level of types. This seems to be a novel notion that has not been investigated before. Neis recently has formalised lazy types in a higher-order λ-calculus [Nei06], but it remains an open problem how to combine his approach elegantly with singleton kinds and type generation, as present in the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus.

- **Processes and local store.** Our notion of type heap and pickles is simplistic because it does not account for separate processes with distinct local stores. In a more realisitc model, we could not assume the type heap to be global, and had to include parts of it into pickles to transfer it to other processes. This raises interesting questions about consistency of abstract type names and their representations across processes, which had to be checked during unpickling.

  A proper notion of process and local store also would uncover the problem of lack of extra-linguistic abstraction safety (Sections 5.4 and 12.9), which is present in Alice ML but not directly visible in the calculus. A formal model might point out ways to reduce the problem.

- **Strong abstraction property.** We have proven Opacity as a moderate abstraction property guaranteeing secrecy and authentication for abstract types at ground kind. A challenging problem is to find appropriate techniques for proving stronger properties like representation independence for the $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus and higher-order abstractions created with the derived sealing operator.

- **Applicative sealing.** The $\lambda^{\omega}_{\mathrm{SA}\Psi}$-calculus can express the equivalent of applicative functors [Ler95, DCH03] only when they do not perform sealing. Applicative functors with *weak sealing* [DCH03] have no counterpart in the calculus. It is not difficult to add a form of "applicative generativity" by introducing a special form of the new-construct that gets lifted out of functions prior to β-reduction using kind raising to abstract over local type bindings – we devised such a system in earlier work [Ros03a]. Unfortunately, it would give an accurate account only for the *dynamic* semantics of weak sealing, but does not adequately capture its *static* semantics. In order to target it in a phase splitting encoding of modules like sketched in Appendix B, the static semantics would have to treat it by a form of skolemisation.

  Weak sealing would be interesting in the context of the component system we defined. Currently, generativity can become an obstacle when exchanging values of abstract type between different processes. Sharing requires exchanging the definining structure of the abstract type as well, which can become tedious or even impractical on larger scale, particularly if the implementation is not free of resources. Applicative generativity would be a formal basis for supporting different levels of generativity in the language, as available in Acute [SLW+05].

- **Module calculi.** Appendix B sketches an encoding of modules into our system. But ultimately, we would like to be able to model ML modules more directly and apply our approach to a module system like defined by Dreyer, Crary & Harper [DCH03]. In previous work, we already have extended their system with packages and pickling, but without considering type generation [Ros06]. The main complication with integrating type generation into a module calculus is that instead of dependent kinds we would have to deal with a richer system of dependent module *types*. Higher-order cercions would appear on all four language levels: terms, types, modules and signatures.

Obviously, addressing some of these issues would be a prerequisite for a complete formal language specification of Alice ML, like we discussed in Section 9.4.2. Beyond that, they address more universal questions of type system design for open programming that might be interesting to investigate independent from a specific language.

Achknowledgements

Many thanks go to my colleagues and students for coming up with valuable comments and criticism on draft versions of this thesis, for long discussions about language design and theory, and for countless hours of work on the project that once used to be known as 'Stockhausen': Leif Kornstädt, Thorsten Brunklaus, Guido Tack, Didier Le Botlan, Jan Schwinghammer, Cătălin Hriţcu, Marco Kuhlmann, Christian Lindig, Georg Neis, Andi Scharfstein. And of course, I thank my advisor Gert Smolka for giving me the opportunity to do this work in the first place.

A. Calculus Summary

A.1. Basic System

A.1.1. Syntax

$$
\begin{array}{llll}
\text{base kinds} & \hat{\kappa} & ::= & \Omega \mid \mathrm{A}(\tau) \\
\text{kinds} & \kappa & ::= & \hat{\kappa} \mid \mathrm{S}_{\hat{\kappa}}(\tau) \mid \Pi\alpha{:}\kappa.\kappa \mid \Sigma\alpha{:}\kappa.\kappa \\
\text{types} & \tau & ::= & \alpha \mid \Psi \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \\
 & & & \mid \lambda\alpha{:}\kappa.\tau \mid \tau\tau \mid \langle \tau, \tau \rangle \mid \tau{\cdot}1 \mid \tau{\cdot}2 \\
\text{terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \langle e, e \rangle \mid \mathsf{let}\langle x, x \rangle = e \;\mathsf{in}\; e \\
 & & & \mid \lambda\alpha{:}\kappa.e \mid e\,\tau \mid \langle \tau, e \rangle \mid \mathsf{let}\langle \alpha, x \rangle = e \;\mathsf{in}_\tau\; e \\
 & & & \mid \mathsf{new}\,\alpha \approx \tau \;\mathsf{in}_\tau\; e \mid \{e\}^+_\tau \mid \{e\}^-_\tau \mid \mathsf{case}\,e{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}_\tau\; e \\
 & & & \mid \mathsf{pickle}\,e \mid \psi(e) \mid \mathsf{unpickle}\,x \Leftarrow e \;\mathsf{in}\; e \;\mathsf{else}_\tau\; e \\
\text{environments} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\kappa
\end{array}
$$

A.1.2. Derived Forms

Simple Sugar

$$
\begin{array}{lrcl}
\text{kinds} & 1 & := & \Pi\alpha{:}\Omega.\mathrm{S}(\alpha) \\
\text{types} & \diamond & := & \lambda\alpha{:}\Omega.\alpha \\
 & 1 & := & \forall\alpha{:}\Omega.\,\alpha \to \alpha \\
\text{terms} & \diamond & := & \lambda\alpha{:}\Omega.\,\lambda x{:}\alpha.\,x \\
 & \mathsf{let}\,x = e_1 \;\mathsf{in}\; e_2 & := & \mathsf{let}\langle x, \_ \rangle = \langle e_1, \diamond \rangle \;\mathsf{in}\; e_2
\end{array}
$$

Higher-Order Singletons

$\boxed{\mathrm{S}(\tau : \kappa)}$

$$
\begin{array}{rcl}
\mathrm{S}(\tau : \Omega) & := & \mathrm{S}_\Omega(\tau) \\
\mathrm{S}(\tau : \mathrm{A}(\tau')) & := & \mathrm{S}_{\mathrm{A}(\tau')}(\tau) \\
\mathrm{S}(\tau : \mathrm{S}_{\hat{\kappa}}(\tau')) & := & \mathrm{S}_{\hat{\kappa}}(\tau') \\
\mathrm{S}(\tau : \Pi\alpha{:}\kappa_1.\kappa_2) & := & \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2) \\
\mathrm{S}(\tau : \Sigma\alpha{:}\kappa_1.\kappa_2) & := & \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha])
\end{array}
$$

A.1.3. Static Semantics

Environment Validity

$\boxed{\Gamma \vdash \square}$

$$(\textsc{Nempty})\ \frac{}{\cdot \vdash \square}$$

$$(\textsc{Ntype})\ \frac{\Gamma \vdash \kappa : \square}{\Gamma, \alpha{:}\kappa \vdash \square}\ (\alpha \notin \mathrm{Dom}(\Gamma)) \qquad (\textsc{Nterm})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma, x{:}\tau \vdash \square}\ (x \notin \mathrm{Dom}(\Gamma))$$

Kind Validity

$\boxed{\Gamma \vdash \kappa : \square}$

$$(\textsc{Komega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega : \square} \qquad (\textsc{Kabs})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) : \square} \qquad (\textsc{Ksing})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) : \square}$$

$$(\textsc{Kpi})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \square} \qquad (\textsc{Ksigma})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \square}$$

Kind Equivalence

$\boxed{\Gamma \vdash \kappa \equiv \kappa' : \square}$

$$(\mathrm{KQ}\textsc{omega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega \equiv \Omega : \square}$$

$$(\mathrm{KQ}\textsc{abs})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \square} \qquad (\mathrm{KQ}\textsc{sing})\ \frac{\Gamma \vdash \tau \equiv \tau' : \hat{\kappa} \qquad \Gamma \vdash \hat{\kappa} \equiv \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \equiv \mathrm{S}_{\hat{\kappa}'}(\tau') : \square}$$

$$(\mathrm{KQ}\textsc{pi})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \equiv \Pi\alpha{:}\kappa_1'.\kappa_2' : \square}$$

$$(\mathrm{KQ}\textsc{sigma})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \equiv \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}$$

Kind Inclusion

$\boxed{\Gamma \vdash \kappa \leq \kappa' : \square}$

$$(\mathrm{KS}\textsc{omega})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Omega \leq \Omega : \square}$$

$$(\mathrm{KS}\textsc{abs})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \mathrm{A}(\tau') : \square} \qquad (\mathrm{KS}\textsc{sing})\ \frac{\Gamma \vdash \tau \equiv \tau' : \hat{\kappa} \qquad \Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \mathrm{S}_{\hat{\kappa}'}(\tau') : \square}$$

$$(\mathrm{KS}\textsc{abs-left})\ \frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathrm{A}(\tau) \leq \Omega : \square} \qquad (\mathrm{KS}\textsc{sing-left})\ \frac{\Gamma \vdash \tau : \hat{\kappa} \qquad \Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \square}{\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \hat{\kappa}' : \square}$$

$$(\mathrm{KS}\textsc{pi})\ \frac{\Gamma \vdash \kappa_1' \leq \kappa_1 : \square \qquad \Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \square \qquad \Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2' : \square}$$

$$(\mathrm{KS}\textsc{sigma})\ \frac{\Gamma \vdash \kappa_1 \leq \kappa_1' : \square \qquad \Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \leq \kappa_2' : \square \qquad \Gamma \vdash \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}{\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 \leq \Sigma\alpha{:}\kappa_1'.\kappa_2' : \square}$$

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\textsc{Tvar})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \alpha : \Gamma(\alpha)} \qquad (\textsc{Tpsi})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Psi : \Omega}$$

$$(\textsc{Tarrow})\ \frac{\Gamma \vdash \tau_1 : \Omega \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad (\textsc{Ttimes})\ \frac{\Gamma \vdash \tau_1 : \Omega \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 : \Omega}$$

$$(\textsc{Tuniv})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa_1.\tau_2 : \Omega} \qquad (\textsc{Texist})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa_1.\tau_2 : \Omega}$$

$$(\textsc{Tlambda})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2}{\Gamma \vdash \lambda \alpha{:}\kappa_1.\tau_2 : \Pi \alpha{:}\kappa_1.\kappa_2} \qquad (\textsc{Tapp})\ \frac{\Gamma \vdash \tau_1 : \Pi \alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\, \tau_2 : \kappa_2[\tau_2/\alpha]}$$

$$(\textsc{Tpair})\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Tfst})\ \frac{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}1 : \kappa_1} \qquad (\textsc{Tsnd})\ \frac{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]}$$

$$(\textsc{Text-sing})\ \frac{\Gamma \vdash \tau : \hat{\kappa}}{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau)}$$

$$(\textsc{Text-pi})\ \frac{\Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2' \quad \Gamma, \alpha{:}\kappa_1 \vdash \tau\, \alpha : \kappa_2 \quad \Gamma \vdash \Pi \alpha{:}\kappa_1.\kappa_2' : \square}{\Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Text-sigma})\ \frac{\Gamma \vdash \tau{\cdot}1 : \kappa_1 \quad \Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \tau : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\textsc{Tsub})\ \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash \kappa \leq \kappa' : \square}{\Gamma \vdash \tau : \kappa'}$$

Type Inclusion $\boxed{\Gamma \vdash \tau \leq \tau' : \kappa}$

$$(\textsc{TSequiv})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau \leq \tau' : \kappa}$$

$$(\textsc{TSarrow})\ \frac{\Gamma \vdash \tau_1' \leq \tau_1 : \Omega \quad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' : \Omega}$$

$$(\textsc{TStimes})\ \frac{\Gamma \vdash \tau_1 \leq \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \leq \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' : \Omega}$$

$$(\textsc{TSuniv})\ \frac{\Gamma \vdash \kappa' \leq \kappa : \square \quad \Gamma, \alpha{:}\kappa' \vdash \tau \leq \tau' : \Omega \quad \Gamma \vdash \forall \alpha{:}\kappa.\tau : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \leq \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\textsc{TSexist})\ \frac{\Gamma \vdash \kappa \leq \kappa' : \square \quad \Gamma, \alpha{:}\kappa \vdash \tau \leq \tau' : \Omega \quad \Gamma \vdash \exists \alpha{:}\kappa'.\tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \leq \exists \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\textsc{TStrans})\ \frac{\Gamma \vdash \tau \leq \tau' : \kappa \quad \Gamma \vdash \tau' \leq \tau'' : \kappa}{\Gamma \vdash \tau \leq \tau'' : \kappa}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\text{TQ}_{\text{VAR}})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \alpha \equiv \alpha : \Gamma(\alpha)} \qquad (\text{TQ}_{\text{PSI}})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \Psi \equiv \Psi : \Omega}$$

$$(\text{TQ}_{\text{ARROW}})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \Omega}$$

$$(\text{TQ}_{\text{TIMES}})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \Omega}$$

$$(\text{TQ}_{\text{UNIV}})\ \frac{\Gamma \vdash \kappa \equiv \kappa' : \square \quad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \forall \alpha{:}\kappa.\tau \equiv \forall \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TQ}_{\text{EXIST}})\ \frac{\Gamma \vdash \kappa \equiv \kappa' : \square \quad \Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \Omega}{\Gamma \vdash \exists \alpha{:}\kappa.\tau \equiv \exists \alpha{:}\kappa'.\tau' : \Omega}$$

$$(\text{TQ}_{\text{LAMBDA}})\ \frac{\Gamma \vdash \kappa_1 \equiv \kappa_1' : \square \quad \Gamma, \alpha{:}\kappa_1 \vdash \tau \equiv \tau' : \kappa_2}{\Gamma \vdash \lambda \alpha{:}\kappa_1.\tau \equiv \lambda \alpha{:}\kappa_1'.\tau' : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQ}_{\text{APP}})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \Pi \alpha{:}\kappa_1.\kappa_2 \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Gamma \vdash \tau_1\, \tau_2 \equiv \tau_1'\, \tau_2' : \kappa_2[\tau_2/\alpha]}$$

$$(\text{TQ}_{\text{PAIR}})\ \frac{\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1 \quad \Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2[\tau_1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau_1', \tau_2' \rangle : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQ}_{\text{FST}})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1} \qquad (\text{TQ}_{\text{SND}})\ \frac{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]}$$

$$(\text{TQ}_{\text{EXT-SING}})\ \frac{\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau'') \quad \Gamma \vdash \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')}$$

$$(\text{TQ}_{\text{EXT-PI}})\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau\, \alpha \equiv \tau'\, \alpha : \kappa_2 \quad \Gamma \vdash \tau : \Pi \alpha{:}\kappa_1.\kappa_2' \quad \Gamma \vdash \tau' : \Pi \alpha{:}\kappa_1.\kappa_2''}{\Gamma \vdash \tau \equiv \tau' : \Pi \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQ}_{\text{EXT-SIGMA}})\ \frac{\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1 \quad \Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha] \quad \Gamma \vdash \Sigma \alpha{:}\kappa_1.\kappa_2 : \square}{\Gamma \vdash \tau \equiv \tau' : \Sigma \alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQ}_{\text{SYMM}})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa}{\Gamma \vdash \tau' \equiv \tau : \kappa} \qquad (\text{TQ}_{\text{TRANS}})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \quad \Gamma \vdash \tau' \equiv \tau'' : \kappa}{\Gamma \vdash \tau \equiv \tau'' : \kappa}$$

$$(\text{TQ}_{\text{SUB}})\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \quad \Gamma \vdash \kappa \leq \kappa' : \square}{\Gamma \vdash \tau \equiv \tau' : \kappa'}$$

Term Validity $\boxed{\Gamma \vdash e : \tau}$

$$(\textsc{Evar})\ \frac{\Gamma \vdash \square}{\Gamma \vdash x : \Gamma(x)}$$

$$(\textsc{Elambda})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad (\textsc{Eapp})\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$(\textsc{Epair})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$(\textsc{Eproj})\ \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\langle x_1, x_2 \rangle = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$(\textsc{Egen})\ \frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\tau} \qquad (\textsc{Einst})\ \frac{\Gamma \vdash e : \forall \alpha{:}\kappa.\tau \quad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\, \tau_2 : \tau[\tau_2/\alpha]}$$

$$(\textsc{Eclose})\ \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash \langle \tau, e \rangle : \exists \alpha{:}\kappa.\tau_2}$$

$$(\textsc{Eopen})\ \frac{\Gamma \vdash e_1 : \exists \alpha{:}\kappa.\tau_2 \quad \Gamma, \alpha{:}\kappa, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathsf{let}\langle \alpha, x \rangle = e_1\ \mathsf{in}_\tau\ e_2 : \tau}$$

$$(\textsc{Enew})\ \frac{\Gamma, \alpha{:}\mathrm{A}(\tau_1) \vdash e : \tau_2 \quad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\, \alpha \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2}$$

$$(\textsc{Eup})\ \frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \{e\}^+_{\tau_1} : \tau_1} \qquad (\textsc{Edn})\ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)}{\Gamma \vdash \{e\}^-_{\tau_1} : \tau_2}$$

$$(\textsc{Ecase})\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{case}\, e_1{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}_\tau\ e_3 : \tau}$$

$$(\textsc{Epickle})\ \frac{\Gamma \vdash e : \exists \alpha{:}\Omega.\alpha}{\Gamma \vdash \mathsf{pickle}\, e : \Psi} \qquad (\textsc{Epsi})\ \frac{\Gamma \vdash \square}{\Gamma \vdash \psi(v) : \Psi}\ (\mathrm{FV}(v) \subseteq \mathrm{Dom}(\Gamma))$$

$$(\textsc{Eunpickle})\ \frac{\Gamma \vdash e_1 : \Psi \quad \Gamma, x{:}(\exists \alpha{:}\Omega.\alpha) \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{unpickle}\, x \Leftarrow e_1\ \mathsf{in}\ e_2\ \mathsf{else}_\tau\ e_3 : \tau}$$

$$(\textsc{Esub})\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \leq \tau' : \Omega}{\Gamma \vdash e : \tau'}$$

Configuration Validity $\boxed{\Gamma \vdash C : \tau}$

$$(\textsc{Cvalid})\ \frac{\Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \Delta; e : \tau}$$

A.1.4. Derived Rules

Kind Validity $\boxed{\Gamma \vdash \kappa : \Box}$

$$(\text{KSING}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) : \Box}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \Box}$

$$(\text{KQSING}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \equiv \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \equiv \mathrm{S}(\tau' : \kappa') : \Box}$$

Kind Inclusion $\boxed{\Gamma \vdash \kappa \leq \kappa' : \Box}$

$$(\text{KSSING}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \kappa \qquad \Gamma \vdash \kappa \leq \kappa' : \Box}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \mathrm{S}(\tau' : \kappa') : \Box} \qquad (\text{KSSING-LEFT}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \kappa : \Box}$$

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\text{TEXT-SING}^*)\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau : \mathrm{S}(\tau : \kappa)}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\text{TQEXT-SING}^*)\ \frac{\Gamma \vdash \tau : \mathrm{S}(\tau'' : \kappa) \qquad \Gamma \vdash \tau' : \mathrm{S}(\tau'' : \kappa)}{\Gamma \vdash \tau \equiv \tau' : \mathrm{S}(\tau'' : \kappa)}$$

$$(\text{TQAPP-BETA}^*)\ \frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2 \qquad \Gamma \vdash \tau_1 : \kappa_1}{\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 \equiv \tau_2[\tau_1/\alpha] : \kappa_2[\tau_1/\alpha]}$$

$$(\text{TQLAMBDA-ETA}^*)\ \frac{\Gamma \vdash \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2\,\alpha \equiv \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2}$$

$$(\text{TQFST-BETA}^*)\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle\tau_1, \tau_2\rangle{\cdot}1 \equiv \tau_1 : \kappa_1} \qquad (\text{TQSND-BETA}^*)\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle\tau_1, \tau_2\rangle{\cdot}2 \equiv \tau_2 : \kappa_2}$$

$$(\text{TQPAIR-ETA}^*)\ \frac{\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}{\Gamma \vdash \langle\tau{\cdot}1, \tau{\cdot}2\rangle \equiv \tau : \Sigma\alpha{:}\kappa_1.\kappa_2}$$

A.1.5. Dynamic Semantics

Values and Contexts

$$
\begin{array}{lrcl}
\text{values} & v & ::= & \lambda x{:}\tau.e \mid \langle v, v\rangle \mid \lambda\alpha{:}\kappa.e \mid \langle \tau, v\rangle \mid \{v\}^+_\tau \mid \psi(v) \\
\text{contexts} & E & ::= & \_ \mid E\,e \mid v\,E \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathsf{let}\langle x, x\rangle = E \;\mathsf{in}\; e \\
 & & & \quad \mid E\,\tau \qquad \mid \langle \tau, E\rangle \qquad\quad \mid \mathsf{let}\langle \alpha, x\rangle = E \;\mathsf{in}\; e \\
 & & & \quad \mid \{E\}^+_\tau \mid \{E\}^-_\tau \mid \mathsf{case}\,E{:}\tau \;\mathsf{of}\; x{:}\tau.e \;\mathsf{else}\; e \\
 & & & \quad \mid \mathsf{pickle}\,E \mid \mathsf{unpickle}\,x \Leftarrow E \;\mathsf{in}\; e \;\mathsf{else}\; e \\
\text{path contexts} & P & ::= & \_ \mid P\,\tau \mid P{\cdot}1 \mid P{\cdot}2 \\
\text{heaps} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tau) \\
\text{configurations} & C & ::= & \Delta; e
\end{array}
$$

Reduction rules

$$
\begin{array}{lll}
(\textsc{Rapp}) & \Delta; E[(\lambda x{:}\tau.e)\,v] & \to \Delta; E[e[v/x]] \\
(\textsc{Rproj}) & \Delta; E[\mathsf{let}\langle x_1, x_2\rangle{=}\langle v_1, v_2\rangle \;\mathsf{in}\; e] & \to \Delta; E[e[v_1/x_1][v_2/x_2]] \\
(\textsc{Rinst}) & \Delta; E[(\lambda\alpha{:}\kappa.e)\,\tau] & \to \Delta; E[e[\tau/\alpha]] \\
(\textsc{Ropen}) & \Delta; E[\mathsf{let}\langle \alpha, x\rangle{=}\langle \tau, v\rangle \;\mathsf{in}_{\tau'}\; e] & \to \Delta; E[e[\tau/\alpha][v/x]] \\
(\textsc{Rnew}) & \Delta; E[\mathsf{new}\,\alpha{\approx}\tau \;\mathsf{in}_{\tau'}\; e] & \to \Delta, \alpha{:}\mathrm{A}(\tau); E[e] \\
(\textsc{Rcancel}) & \Delta; E[\{\{v\}^+_{\tau_1}\}^-_{\tau_2}] & \to \Delta; E[v] \\
(\textsc{Rcase1}) & \Delta; E[\mathsf{case}\,v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_1[v/x]] \quad \text{if } \Delta \vdash \tau_1 \le \tau_2 : \Omega \\
(\textsc{Rcase2}) & \Delta; E[\mathsf{case}\,v{:}\tau_1 \;\mathsf{of}\; x{:}\tau_2.e_1 \;\mathsf{else}_\tau\; e_2] & \to \Delta; E[e_2] \quad\;\; \text{if } \Delta \not\vdash \tau_1 \le \tau_2 : \Omega \\
(\textsc{Rpickle}) & \Delta; E[\mathsf{pickle}\,v] & \to \Delta; E[\psi(v)] \\
(\textsc{Runpickle1}) & \Delta; E[\mathsf{unpickle}\,x{\Leftarrow}\psi(v) \;\mathsf{in}\; e_1 \;\mathsf{else}\; e_2] & \to \Delta; E[e_1[v/x]] \quad \text{if } \Delta \vdash v : \exists\alpha{:}\Omega.\alpha \\
(\textsc{Runpickle2}) & \Delta; E[\mathsf{unpickle}\,x{\Leftarrow}\psi(v) \;\mathsf{in}\; e_1 \;\mathsf{else}\; e_2] & \to \Delta; E[e_2] \quad\;\; \text{if } \Delta \not\vdash v : \exists\alpha{:}\Omega.\alpha
\end{array}
$$

A.2. Higher-Order Extensions

A.2.1. Syntax

$$
\begin{array}{lrcl}
\text{expressions} & e & ::= & \cdots \mid e\,\tilde\tau \mid \langle \tilde\tau, e\rangle \mid \mathsf{new}\,\alpha{:}\tilde\kappa \approx \tilde\tau \;\mathsf{in}_\tau\; e \mid \{e : \alpha{:}\tilde\kappa.\tilde\tau\}^+_{\tilde\tau\approx\tilde\tau} \mid \{e : \alpha{:}\tilde\kappa.\tilde\tau\}^-_{\tilde\tau\approx\tilde\tau} \\
\text{concrete kinds} & \tilde\kappa & ::= & \Omega \mid \mathrm{S}_\Omega(\tau) \mid \Pi\alpha{:}\tilde\kappa.\tilde\kappa \mid \Sigma\alpha{:}\tilde\kappa.\tilde\kappa \\
\text{concrete types} & \tilde\tau & ::= & \alpha \mid \Psi \mid \tilde\tau \to \tilde\tau \mid \tilde\tau \times \tilde\tau \mid \forall\alpha{:}\tilde\kappa.\tilde\tau \mid \exists\alpha{:}\tilde\kappa.\tilde\tau \\
 & & & \quad \mid \lambda\alpha{:}\tilde\kappa.\tilde\tau \mid \tilde\tau\,\tilde\tau \mid \langle\tilde\tau, \tilde\tau\rangle \mid \tilde\tau{\cdot}1 \mid \tilde\tau{\cdot}2
\end{array}
$$

A.2.2. Derived Forms

Kind Coercions

$\boxed{\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau/\tau}}$

$$
\begin{array}{rcl}
\{\tau : \alpha{:}\kappa.\Omega\}_{\tau_+/\tau_-} & := & \tau \\
\{\tau : \alpha{:}\kappa.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-} & := & \tau'[\tau_+/\alpha] \\
\{\tau : \alpha{:}\kappa.\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\tau_+/\tau_-} & := & \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\{\tau\,\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+/\tau_-} \\
 & & \text{where } \tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1] \\
\{\tau : \alpha{:}\kappa.\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\tau_+/\tau_-} & := & \langle\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}, \{\tau{\cdot}2 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+/\tau_-}\rangle \\
 & & \text{where } \tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]
\end{array}
$$

Higher-Order Abstraction Kinds

$\boxed{\mathrm{A}(\tau : \tilde{\kappa})}$

$$
\begin{array}{rcl}
\mathrm{A}(\tau : \Omega) & := & \mathrm{A}(\tau) \\
\mathrm{A}(\tau : \mathrm{S}_\Omega(\tau')) & := & \mathrm{S}_\Omega(\tau') \\
\mathrm{A}(\tau : \Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Pi\alpha{:}\tilde{\kappa}_1.\mathrm{A}(\tau\,\alpha : \tilde{\kappa}_2) \\
\mathrm{A}(\tau : \Sigma\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2) & := & \Sigma\alpha{:}\mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1).\mathrm{A}(\{\tau{\cdot}2 : \alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\alpha/\tau\cdot 1} : \tilde{\kappa}_2)
\end{array}
$$

Sealing

$\boxed{e :> \tilde{\tau}}$

$$
\begin{array}{lcl}
e :> \chi & := & e \\
e :> \Psi & := & e \\
e :> \tilde{\tau}_1 \to \tilde{\tau}_2 & := & e \\
e :> \tilde{\tau}_1 \times \tilde{\tau}_2 & := & e \\
e :> \forall\alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 & := & e \\
e :> \exists\alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 & := & \mathsf{let}\langle\alpha, x\rangle = e\ \mathsf{in}_{\exists\alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \mathsf{new}\,\alpha'{:}\tilde{\kappa}_1 \approx \alpha\ \mathsf{in}_{\exists\alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \langle\alpha', \{x : \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^+_{\alpha'\approx\alpha}\rangle
\end{array}
$$

A.2.3. Static Semantics

Term Validity

$\boxed{\Gamma \vdash e : \tau}$

$$
(\textsc{New'})\ \frac{\Gamma \vdash \tau_1 : \tilde{\kappa} \qquad \Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}) \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{new}\,\alpha{:}\tilde{\kappa} \approx \tau_1\ \mathsf{in}_{\tau_2}\ e : \tau_2}
$$

$$
(\textsc{Up'})\ \frac{\Gamma \vdash e : \tilde{\tau}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_-{:}\tilde{\kappa}) \qquad \Gamma \vdash \tau_- : \tilde{\kappa}}{\Gamma \vdash \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^+_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_+/\alpha]}
$$

$$
(\textsc{Dn'})\ \frac{\Gamma \vdash e : \tilde{\tau}[\tau_+/\alpha] \qquad \Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega \qquad \Gamma \vdash \tau_+ : \mathrm{A}(\tau_-{:}\tilde{\kappa}) \qquad \Gamma \vdash \tau_- : \tilde{\kappa}}{\Gamma \vdash \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^-_{\tau_+\approx\tau_-} : \tilde{\tau}[\tau_-/\alpha]}
$$

A.2.4. Derived Rules

Kind Validity $\boxed{\Gamma \vdash \kappa : \Box}$

$$(\textsc{Kabs}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) : \Box}$$

Kind Equivalence $\boxed{\Gamma \vdash \kappa \equiv \kappa' : \Box}$

$$(\textsc{KQabs}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa} \qquad \Gamma \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \equiv \mathrm{A}(\tau' : \tilde{\kappa}') : \Box}$$

Kind Inclusion $\boxed{\Gamma \vdash \kappa \leq \kappa' : \Box}$

$$(\textsc{KSabs}^*)\ \frac{\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa} \qquad \Gamma \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \leq \mathrm{A}(\tau' : \tilde{\kappa}') : \Box}$$

$$(\textsc{KSabs-left}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}}{\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \leq \tilde{\kappa} : \Box}$$

Type Validity $\boxed{\Gamma \vdash \tau : \kappa}$

$$(\textsc{Tcoerce}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} : \tilde{\kappa}[\tau_+/\alpha]}$$

Type Equivalence $\boxed{\Gamma \vdash \tau \equiv \tau' : \kappa}$

$$(\textsc{TQcoerce}^*)\ \frac{\begin{array}{cc}\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa}[\tau_-/\alpha] & \Gamma \vdash \kappa \equiv \kappa' : \Box \\ \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box & \Gamma, \alpha{:}\kappa' \vdash \tilde{\kappa}' : \Box \\ \Gamma \vdash \tilde{\kappa}[\tau_+/\alpha] \equiv \tilde{\kappa}'[\tau'_+/\alpha] : \Box & \Gamma \vdash \tilde{\kappa}[\tau_-/\alpha] \equiv \tilde{\kappa}'[\tau'_-/\alpha] : \Box \\ \Gamma \vdash \tau_+ \equiv \tau'_+ : \kappa & \Gamma \vdash \tau_- \equiv \tau'_- : \kappa\end{array}}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} \equiv \{\tau' : \alpha{:}\kappa'.\tilde{\kappa}'\}_{\tau'_+/\tau'_-} : \tilde{\kappa}[\tau_+/\alpha]}$$

$$(\textsc{TQcoerce-drop}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ \equiv \tau_- : \kappa}{\Gamma \vdash \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}$$

$$(\textsc{TQcoerce-cancel}^*)\ \frac{\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha] \qquad \Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box \qquad \Gamma \vdash \tau_+ : \kappa \qquad \Gamma \vdash \tau_- : \kappa}{\Gamma \vdash \{\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_-} : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_-/\tau_+} \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]}$$

Term Validity $\boxed{\Gamma \vdash e : \tau}$

$$(\textsc{Eseal}^*)\ \frac{\Gamma \vdash e : \tilde{\tau}}{\Gamma \vdash e :> \tilde{\tau} : \tilde{\tau}}$$

A.2.5. Dynamic Semantics

Values and Contexts

$$
\begin{array}{lrcl}
\text{(values)} & v & ::= & \cdots \mid \langle \tilde{\tau}, v \rangle \mid \{v : \alpha{:}\Omega.\alpha\}^{+}_{\tilde{\tau}\approx\tilde{\tau}} \\
\text{(contexts)} & E & ::= & \cdots \mid \{E : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tilde{\tau}\approx\tilde{\tau}} \mid \{E : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tilde{\tau}\approx\tilde{\tau}} \\
\text{(heaps)} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\mathrm{A}(\tilde{\tau} : \tilde{\kappa})
\end{array}
$$

Reduction rules

$$
\begin{array}{lll}
\text{(RNEW')} & \Delta; E[\mathsf{new}\,\alpha{:}\tilde{\kappa}{\approx}\tilde{\tau}\;\mathsf{in}\;e] & \rightarrow\; (\Delta, \alpha{:}\mathrm{A}(\tilde{\tau} : \tilde{\kappa})); E[e] \\
\\
\text{(RCOERCE-NORM)} & \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \{v : \alpha{:}\tilde{\kappa}.\pi\}^{\pm}_{\tau_+\approx\tau_-} \\
 & & \quad \text{where } \pi \neq \tau \text{ and } \Delta, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \Rrightarrow \pi \\
\\
\text{(RCOERCE-PSI)} & \{v : \alpha{:}\tilde{\kappa}.\Psi\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; v \\
\text{(RCOERCE-ARROW)} & \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1 \rightarrow \tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \lambda x_1{:}\tilde{\tau}_1[\tau_\pm/\alpha].\{v\,\{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^{\mp}_{\tau_+\approx\tau_-} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} \\
\text{(RCOERCE-TIMES)} & \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1 \times \tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \mathsf{let}\langle x_1, x_2\rangle = v\;\mathsf{in}\;\langle \{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^{\pm}_{\tau_+\approx\tau_-}, \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-}\rangle \\
\text{(RCOERCE-UNIV)} & \{v : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_\pm/\alpha].\{v\,\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_\mp/\tau_\pm} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2'\}^{\pm}_{\tau_+\approx\tau_-} \\
 & & \quad \text{where } \tilde{\tau}_2' = \tilde{\tau}_2[\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_\pm}/\alpha_1] \\
\text{(RCOERCE-EXIST)} & \{v : \alpha{:}\tilde{\kappa}.\exists\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \mathsf{let}\langle \alpha_1, x_2\rangle = v\;\mathsf{in}\;\langle \{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_\pm/\tau_\mp}, \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2'\}^{\pm}_{\tau_+\approx\tau_-}\rangle \\
 & & \quad \text{where } \tilde{\tau}_2' = \tilde{\tau}_2[\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_\mp}/\alpha_1] \\
\text{(RCOERCE-SWAP)} & \{v : \alpha{:}\tilde{\kappa}.P[\alpha']\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \{\{e : \alpha{:}\tilde{\kappa}.P[\tau_-'/\alpha' : \tilde{\kappa}']\}^{\pm}_{\tau_+\approx\tau_-} : \alpha'' {:}\tilde{\kappa}'.P[\alpha''/\alpha' : \tilde{\kappa}'][\tau_\pm/\alpha]\}^{+}_{\alpha'\approx\tau_-'} \\
 & & \quad \text{where } e = \{v : \alpha'' {:}\tilde{\kappa}'.P[\alpha''/\alpha' : \tilde{\kappa}'][\tau_\mp/\alpha]\}^{-}_{\alpha'\approx\tau_-'} \\
 & & \quad \text{and } \alpha' \neq \alpha \text{ and } \Delta(\alpha') = \mathrm{A}(\tau_-' : \tilde{\kappa}') \\
\text{(RCOERCE-GROUND)} & \{v : \alpha{:}\tilde{\kappa}.P[\alpha]\}^{\pm}_{\tau_+\approx\tau_-} & \rightarrow\; \{\{v : \alpha{:}\tilde{\kappa}.P[\tau_\mp/\alpha : \tilde{\kappa}]\}^{\pm}_{\tau_+\approx\tau_-} : \alpha{:}\Omega.\alpha\}^{\pm}_{P[\tau_+/\alpha:\tilde{\kappa}][\tau_\pm/\alpha]\approx P[\tau_-/\alpha:\tilde{\kappa}][\tau_\pm/\alpha]} \\
 & & \quad \text{where } P \neq \_ \\
\text{(RCOERCE-CANCEL)} & \{v : \alpha{:}\Omega.\alpha\}^{-}_{\tau_+\approx\tau_-} & \rightarrow\; v' \\
 & & \quad \text{where } v = \{v' : \alpha{:}\Omega.\alpha\}^{+}_{\tau_+'\approx\tau_-'}
\end{array}
$$

Notes: 1. Omitted surrounding $\Delta; E[\_]$ in reduction rules for coercions.
2. All RHS variables fresh.

Path Replacement

$$
\begin{array}{lcl}
P[\tau_+/\tau_- : \tilde{\kappa}] & := & P[\alpha{:}\tilde{\kappa}\,.\,\alpha : \tilde{\kappa}]_{\tau_+/\tau_-} \\
\\
\_[\alpha{:}\tilde{\kappa}\,.\,\tau : \Omega]_{\tau_+/\tau_-} & := & \tau[\tau_+/\alpha] \\
\_[\alpha{:}\tilde{\kappa}\,.\,\tau : \mathrm{S}_\Omega(\tau')]_{\tau_+/\tau_-} & := & \tau[\tau_+/\alpha] \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_+/\tau_-}\,\tau') & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau\,\tau'' : \tilde{\kappa}_2[\tau''/\alpha_1]]_{\tau_+/\tau_-} \quad \text{where } \tau'' = \{\tau' : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_-} \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_+/\tau_-}\cdot 1) & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau{\cdot}1 : \tilde{\kappa}_1]_{\tau_+/\tau_-} \\
P([\alpha{:}\tilde{\kappa}\,.\,\tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)]_{\tau_-/\tau_-}\cdot 2) & := & P[\alpha{:}\tilde{\kappa}\,.\,\tau{\cdot}2 : \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1]]_{\tau_+/\tau_-}
\end{array}
$$

B. Encoding Modules

In this section we give a brief sketch of a definition of an ML-like module language with packages based on our calculus. Figure B.1 shows the syntax of that language. It closely follows the idealised higher-order module system by Dreyer, Crary & Harper [DCH03], having dependent products and sums to model functors and structures, and basic modules carrying a single term or type. For functors, we distinguish between applicative and generative functors – the latter are marked by a star. For simplicity, the syntax is linearised, i.e. subexpressions have to be let-bound. We omit weak sealing, which does not exist in Alice ML and is not directly expressible in our calculus.

As an example, the module

```
functor F (X : sig type t; val f : t → t; val x : t end) =
struct
    type u = int
    type v = int × t
    structure M = struct type w = int; val g = fn x : int ⇒ x end
    val h = fn y : int ⇒ (3, X.f(X.x))
end
```

can be expressed almost directly as

$$\begin{array}{l}\lambda^* X : (\Sigma t : [\![\Omega]\!].\, \Sigma f : [\![t! \to t!]\!].\, [\![t!]\!]).\\ \quad \langle [int],\\ \quad\ [int \times X{\cdot}1!],\\ \quad\ \langle [int], [\lambda x : int.\, x]\rangle,\\ \quad\ [\lambda y : int.\, \langle 3, X{\cdot}2{\cdot}1!(X{\cdot}2{\cdot}2!)\rangle]\rangle\end{array}$$

where we omit brackets for pairs nested to the right. One possible signature for this module expression is

$$\begin{array}{l}\Pi^* X : (\Sigma t : [\![\Omega]\!].\, \Sigma f : [\![t! \to t!]\!].\, [\![t!]\!]).\\ \quad \Sigma u : [\![\Omega]\!].\\ \quad \Sigma v : [\![\mathrm{S}_\Omega(u! \times X{\cdot}1!)]\!].\\ \quad \Sigma M : (\Sigma w : [\![\Omega]\!].\, [\![u! \to w!]\!]).\\ \quad [\![u! \to v!]\!]\rangle\end{array}$$

which corresponds to the following Alice ML signature expression:

```
fct (X : sig type t; val f : t → t; val x : t end) →
sig
    type u
    type v = u × t
    structure M : sig type w; val g : u → w end
    val h : u → v
end
```

Figure B.2 defines the complete module syntax as syntactic sugar on top of our calculus. The definition is inspired by the *phase splitting* transformation for modules given in literature [HMM90, Sto05, Dre05], which separates every module in its static part of type definitions and its dynamic part of value definitions.

$$
\begin{array}{llll}
\text{(signatures)} & \sigma & ::= & [\![\kappa]\!] \mid [\![\tau]\!] \mid \Pi X{:}\sigma.\sigma \mid \Pi^* X{:}\sigma.\sigma \mid \Sigma X{:}\sigma.\sigma \\
\text{(modules)} & m & ::= & X \mid [\tau] \mid [e] \mid \lambda X{:}\sigma.m \mid \lambda^* X{:}\sigma.m{:}\sigma \mid X_1\, X_2 \mid \langle X_1, X_2\rangle \mid X{\cdot}1 \mid X{\cdot}2 \mid \\
 & & & \mathsf{let}\, X = m\, \mathsf{in}\, m \mid X :> \sigma \mid \mathsf{unpack}\, e : \sigma \\
\text{(types)} & \tau & ::= & \cdots \mid m! \mid \mathsf{package} \\
\text{(terms)} & e & ::= & \cdots \mid X! \mid \mathsf{let}\, X = m\, \mathsf{in}\, e \mid \mathsf{pack}\, X : \sigma
\end{array}
$$

Figure B.1.: Syntax of modules

In our definition, the static part σ^S of a signature σ extracts the kind of its type component, while the dynamic part σ^D reflects the type of its value component, parameterised over the extracted static part.

For modules, the static part m^S accumulates all type definitions, whereas the dynamic part m^D contains the type *and* value definitions. The latter allows pure and impure modules to be treated uniformly. Impure modules are those containing occurrences of sealing or unpacking, and thus not allowing type projections. We employ a simple notion of unit kind, analogous to the unit type (Section 12.1), to encode modules that carry no type information:

$$
\begin{array}{lcl}
1 & := & \Pi\alpha{:}\Omega.\mathrm{S}(\alpha) \\
\diamond & := & \lambda\alpha{:}\Omega.\alpha
\end{array}
$$

Note that generative functors carry no type information, because they are impure and thus cannot appear in a projection.

We take the liberty to ignore type annotations at let and new in the translation where they are intended to be fully transparent. To cope with them, the simple syntax-directed definition had to be refined to be performed by induction on typing *derviations*. Since annotations merely guide type checking, but are otherwise irrelevant, this is a minor issue.

The translation of modules also contains a third type component, which accumulates all types occurring in the module or its signature. This is necessary to deal with local types that are subject to the avoidance problem [DCH03]. In other words, our translation already incorporates the additional *elaboration* phase proposed by Dreyer, Crary & Harper for addressing the avoidance problem by inserting additional quantifiers. In combination, the translation m^D yields an existential triple $\langle \tau_H, \tau_S, e_D\rangle$, with τ_S and e_D representing the dynamic and static parts of the module, and τ_H capturing all (potentially hidden) type names in them. The invariant is that any type name or variable that is bound locally in m^D is free in τ_H, such that its scope can extend. The translation lifts and accumulates all local τ_H types until a generative functor or a term-level let-construct binds a module, where the body has to be annotated with a type not mentioning X, such that the local types cannot escape the scope further. For example, the module expression

$$\mathsf{let}\, X = \langle [int], [3]\rangle :> \Sigma t{:}[\![\Omega]\!].\, [\![t!]\!]\ \mathsf{in}\ [\mathsf{let}\, X_2 = X{\cdot}2\ \mathsf{in}\ X_2!]$$

is – with a few simplifications – represented as

$$
\begin{array}{l}
\mathsf{let}\langle \beta_1, \langle \alpha_X, x\rangle\rangle = \mathsf{new}\, \beta{:}(\Omega{\times}1) \approx \langle int, \diamond\rangle\ \mathsf{in}\ \langle \beta, \langle \beta, \{\langle \diamond, 3\rangle : \alpha{:}(\Omega{\times}1).1{\times}\alpha{\cdot}1\}^+_{\beta\approx\langle int,\diamond\rangle}\rangle\rangle \\
\mathsf{in}\ \langle\langle \beta_1, \diamond\rangle, \langle \diamond, \mathsf{let}\langle \beta_2, \langle \alpha_2, x_2\rangle\rangle = \langle \diamond, \langle \alpha_X{\cdot}2, x{\cdot}2\rangle\rangle\ \mathsf{in}\ x_2\rangle\rangle
\end{array}
$$

The coercion in this expression can be reduced using rules RCOERCE-TIMES and RCOERCE-FST, leaving the equivalent of

$$\mathsf{let}\langle\beta_1, \langle\alpha_X, x\rangle\rangle = \mathsf{new}\,\beta{:}(\Omega \times 1) \approx \langle int, \diamond\rangle \;\mathsf{in}\; \langle\beta, \langle\beta, \langle\diamond, \{3\}^{+}_{\beta\cdot 1 \approx int}\rangle\rangle\rangle$$
$$\mathsf{in}\; \langle\langle\beta_1, \diamond\rangle, \langle\diamond, \mathsf{let}\langle\beta_2, \langle\alpha_2, x_2\rangle\rangle = \langle\diamond, \langle\alpha_X{\cdot}2, x{\cdot}2\rangle\rangle \;\mathsf{in}\; x_2\rangle\rangle$$

where $\{3\}^{+}_{\beta\cdot 1\approx int}$ is short for $\{3 : \alpha{:}\Omega.\alpha\}^{+}_{\beta\cdot 1\approx int}$. Thanks to β_1 being included in the final existential, this expression can be assigned quantified type

$$\exists\beta{:}(\Omega \times 1).\exists\alpha{:}1.\beta{\cdot}1$$

Without the outer quantifier no closed type could be assigned.

Note also that the translation of applicative functors is only well-typed if β does not occur in the (transparent) type of $\langle\alpha', x'\rangle$. This provides a slightly more permissive semantics than applicative functors in Dreyer, Crary & Harper, because it allows (strong) sealing in the body of an applicative functor as long as none of the resulting abstract types is externally visible.

The novel aspects of the translation are with respect to the sealing operator, which dynamically generates fresh type names as investigated in Section 13.5, and the addition of packages, which follows the definitions given in Section 12.3.2. Note however how the pack construct refines the package signature such that its static part becomes transparent with singleton kind. This implements the behaviour discussed in Section 4.4.1.

We conjecture that the translation is sound and complete relative to a module type system as the one given by Dreyer, Crary & Harper and extended with packages [Ros06], but leave a proof for future work.

Signatures

$$
\begin{array}{ll}
[\![\kappa]\!]^S & := \kappa \\
[\![\tau]\!]^S & := 1 \\
(\Pi x{:}\sigma_1.\sigma_2)^S & := \Pi\alpha_X{:}\sigma_1^S.\sigma_2^S \\
(\Pi^* x{:}\sigma_1.\sigma_2)^S & := 1 \\
(\Sigma x{:}\sigma_1.\sigma_2)^S & := \Sigma\alpha_X{:}\sigma_1^S.\sigma_2^S
\end{array}
$$

$$
\begin{array}{ll}
[\![\kappa]\!]^D(\tau^S) & := 1 \\
[\![\tau]\!]^D(\tau^S) & := \tau \\
(\Pi x{:}\sigma_1.\sigma_2)^D(\tau^S) & := \forall\alpha_X{:}\sigma_1^S.\sigma_1^D(\alpha_X) \to \exists\alpha_2{:}\mathrm{S}(\tau^S\,\alpha_X : \sigma_2^S).\sigma_2^D(\alpha_2) \\
(\Pi^* x{:}\sigma_1.\sigma_2)^D(\tau^S) & := \forall\alpha_X{:}\sigma_1^S.\sigma_1^D(\alpha_X) \to \exists\alpha_2{:}\sigma_2^S.\sigma_2^D(\alpha_2) \\
(\Sigma x{:}\sigma_1.\sigma_2)^D(\tau^S) & := \sigma_1^D(\tau^S{\cdot}1) \times \sigma_2^D(\tau^S{\cdot}2)[\tau^S{\cdot}1/\alpha_X]
\end{array}
$$

Modules

$$
\begin{array}{ll}
X^S & := \alpha_X \\
[\tau]^S & := \tau \\
[e]^S & := \diamond \\
(\lambda X{:}\sigma.m)^S & := \lambda\alpha_X{:}\sigma^S.m^S \\
(\lambda^* X{:}\sigma_1.m{:}\sigma_2)^S & := \diamond \\
(X_1\,X_2)^S & := \alpha_{X_1}\,\alpha_{X_2} \\
\langle X_1, X_2\rangle^S & := \langle\alpha_{X_1}, \alpha_{X_2}\rangle \\
(X{\cdot}1)^S & := \alpha_X{\cdot}1 \\
(X{\cdot}2)^S & := \alpha_X{\cdot}2 \\
(\mathsf{let}\,X{=}m_1\ \mathsf{in}\ m_2)^S & := m_2^S[m_1^S/\alpha_X]
\end{array}
$$

$$
\begin{array}{ll}
X^D & := \langle\diamond, \langle\alpha_X, x\rangle\rangle \\
[\tau]^D & := \langle\diamond, \langle\tau, \diamond\rangle\rangle \\
[e]^D & := \langle\diamond, \langle\diamond, e\rangle\rangle \\
(\lambda X{:}\sigma.m)^D & := \langle\diamond, \langle(\lambda\alpha_X{:}\sigma^S.m^S), \lambda\alpha_X{:}\sigma^S.\lambda x{:}\sigma^D(\alpha_X).\,\mathsf{let}\langle\beta, \langle\alpha', x'\rangle\rangle = m^D\ \mathsf{in}\ \langle\alpha', x'\rangle\rangle\rangle \\
(\lambda^* X{:}\sigma_1.m{:}\sigma_2)^D & := \langle\diamond, \langle\diamond, \lambda\alpha_X{:}\sigma_1^S.\lambda x{:}\sigma_1^D(\alpha_X).\,\mathsf{let}\langle\beta, \langle\alpha', x'\rangle\rangle = m^D\ \mathsf{in}_{\exists\alpha:\sigma_2^S.\sigma_2^D(\alpha)}\ \langle\alpha', x'\rangle\rangle\rangle \\
(X_1\,X_2)^D & := \mathsf{let}\langle\alpha', x'\rangle = x_1\,\alpha_{X_2}\,x_2\ \mathsf{in}\ \langle\alpha', \langle\alpha', x'\rangle\rangle \\
\langle X_1, X_2\rangle^D & := \langle\diamond, \langle\langle\alpha_{X_1}, \alpha_{X_2}\rangle, \langle x_1, x_2\rangle\rangle\rangle \\
(X{\cdot}1)^D & := \langle\diamond, \langle\alpha_X{\cdot}1, x{\cdot}1\rangle\rangle \\
(X{\cdot}2)^D & := \langle\diamond, \langle\alpha_X{\cdot}2, x{\cdot}2\rangle\rangle \\
(\mathsf{let}\,X{=}m_1\ \mathsf{in}\ m_2)^D & := \mathsf{let}\langle\beta_1, \langle\alpha_X, x\rangle\rangle = m_1^D\ \mathsf{in}\ \mathsf{let}\langle\beta_2, \langle\alpha', x'\rangle\rangle = m_2^D\ \mathsf{in}\ \langle\langle\beta_1, \beta_2\rangle, \langle\alpha', x'\rangle\rangle \\
(X :> \sigma)^D & := \mathsf{new}\,\alpha'{:}\sigma^S \approx \alpha_X\ \mathsf{in}\ \langle\alpha', \langle\alpha', \{x : \alpha{:}\sigma^S.\sigma^D(\alpha)\}^+_{\alpha'}\rangle\rangle \\
(\mathsf{unpack}\,e : \sigma)^D & := \mathsf{let}\langle\alpha, x\rangle = e\ \mathsf{in} \\
 & \quad \mathsf{case}\,x{:}\alpha\ \mathsf{of}\ x'{:}(\exists\alpha'{:}\sigma^S.\sigma^D(\alpha')).\,\mathsf{let}\langle\alpha'', x''\rangle = x'\ \mathsf{in}\ \langle\alpha'', \langle\alpha'', x''\rangle\rangle\ \mathsf{else}\ \bot
\end{array}
$$

Types

$$
\begin{array}{ll}
m! & := m^S \\
\mathsf{package} & := \exists\alpha{:}\Omega.\alpha
\end{array}
$$

Terms

$$
\begin{array}{ll}
X! & := x \\
\mathsf{let}\,X = m\ \mathsf{in}_\tau\ e & := \mathsf{let}\langle\beta, \langle\alpha_X, x\rangle\rangle = m^D\ \mathsf{in}_\tau\ e \\
\mathsf{pack}\,X : \sigma & := \langle\exists\alpha{:}\mathrm{S}(\alpha_X : \sigma^S).\sigma^D(\alpha), \langle\alpha_X, x\rangle\rangle
\end{array}
$$

Figure B.2.: Translation of modules

C. Proofs of Type Level Properties

This and the following two appendices contain complete proofs for all theorems and propositions stated in the main body of this thesis. Some of them are reformulated or ordered in slightly different ways, though, in order to resolve certain inter-dependencies. An alphabetic list of all propositions can be found in Appendix F for easy reference.

C.1. Declarative Properties

We start with a number of basic properties of the calculus. The presentation closely follows Stone & Harper [SH06]. We discuss deviations where they occur.

C.1.1. Preliminaries

First we present several generic properties that hold for all judgements in the system, including the term validity judgement presented in Section 12.1 (Figure 12.2). To ease formulation, we use $\mathcal{J}$ to range over all judgement forms by defining it as follows:

$$\mathcal{J} \quad ::= \quad \Box \mid \kappa : \Box \mid \kappa \equiv \kappa : \Box \mid \kappa \leq \kappa : \Box \mid \tau : \kappa \mid \tau \equiv \tau : \kappa \mid \tau \leq \tau : \kappa \mid e : \tau$$

The definitions of substitution $\mathcal{J}[t/z]$ and free variables $\mathrm{FV}(\mathcal{J})$ are lifted in the obvious way.

Many proofs will proceed by induction over kinds. Since these proofs often involve substitutions of type variables, we have to induct over the *size* of kinds, which is defined in a manner that is invariant under such substitution:

Definition 1 (Kind Size)**.**
The size of a kind is defined inductively as follows:

$$\begin{aligned}
\mathrm{Size}(\hat{\kappa}) &= 1 \\
\mathrm{Size}(\mathrm{A}(\tau)) &= 2 \\
\mathrm{Size}(\mathrm{S}_{\hat{\kappa}}(\tau)) &= 2 + \mathrm{Size}(\hat{\kappa}) \\
\mathrm{Size}(\Pi\alpha{:}\kappa_1.\kappa_2) &= \mathrm{Size}(\kappa_1) + \mathrm{Size}(\kappa_2) + 1 \\
\mathrm{Size}(\Sigma\alpha{:}\kappa_1.\kappa_2) &= \mathrm{Size}(\kappa_1) + \mathrm{Size}(\kappa_2) + 1
\end{aligned}$$

Proposition 2 (Environment Validity)**.**

1. *Every proof of* $\Gamma \vdash \mathcal{J}$ *contains a subderivation* $\Gamma \vdash \Box$*.*
2. *Every proof of* $\Gamma_1, \alpha{:}\kappa, \Gamma_2 \vdash \mathcal{J}$ *contains a strict subderivation of* $\Gamma_1 \vdash \kappa : \Box$*.*
3. *Every proof of* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathcal{J}$ *contains a strict subderivation of* $\Gamma_1 \vdash \tau : \Omega$*.*

Proof. Each by straightforward induction on the derivation. □

Lemma 3 (Variable Containment)**.**
If $\Gamma \vdash \mathcal{J}$*, then* $\mathrm{FV}(\mathcal{J}) \subseteq \mathrm{Dom}(\Gamma)$*.*

Proof. By straightforward induction on the derivation, using Environment Validity in case of binders and variables. □

Proposition 4 (Reflexivity)**.**

1. *If* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*
2. *If* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \leq \kappa' : \Box$*.*
3. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*
4. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \leq \tau' : \kappa$*.*

Proof. (1)–(3) by straightforward simultaneous induction on the derivation, using Environment Validity in case of binders; (4) trivially by rule TSEQUIV, using (3). □

Lemma 5 (Renaming)**.**

1. *If* $\Gamma_1, \alpha{:}\kappa, \Gamma_2 \vdash \mathcal{J}$*, then there is a derivation of equal structure of* $\Gamma_1, \alpha'{:}\kappa, \Gamma[\alpha'/\alpha] \vdash \mathcal{J}[\alpha'/\alpha]$ *for every* $\alpha' \notin \mathrm{Dom}(\Gamma_1) \cup \mathrm{Dom}(\Gamma_2)$*.*
2. *If* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathcal{J}$*, then there is a derivation of equal structure of* $\Gamma_1, x'{:}\tau, \Gamma[x'/x] \vdash \mathcal{J}[x'/x]$ *for every* $x' \notin \mathrm{Dom}(\Gamma_1) \cup \mathrm{Dom}(\Gamma_2)$*.*

Proof. Each by straightforward induction on the derivation. □

Lemma 6 (Weakening)**.**

1. *If* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma' \vdash \Box$ *such that* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash \mathcal{J}$*.*
2. *If* $\Gamma_1, \alpha{:}\kappa, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \kappa' : \Box$ *such that* $\Gamma_1 \vdash \kappa' \leq \kappa : \Box$*, then* $\Gamma_1, \alpha{:}\kappa', \Gamma_2 \vdash \mathcal{J}$*.*
3. *If* $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \tau' : \Box$ *such that* $\Gamma_1 \vdash \tau' \leq \tau : \Box$*, then* $\Gamma_1, x{:}\tau', \Gamma_2 \vdash \mathcal{J}$*.*

Proof. Each by straightforward induction on the derivation, using Renaming. □

Lemma 7 (Environment Modification)**.**

1. *If* $\Gamma, \alpha{:}\kappa \vdash \Box$ *and* $\Gamma \vdash \kappa' : \Box$*, then* $\Gamma, \alpha{:}\kappa' \vdash \Box$*.*
2. *If* $\Gamma, \alpha{:}\kappa \vdash \mathcal{J}$ *and* $\Gamma \vdash \kappa' : \Box$ *and* $\alpha' \notin \mathrm{Dom}(\Gamma, \alpha{:}\kappa)$*, then* $\Gamma, \alpha{:}\kappa', \alpha'{:}\kappa \vdash \mathcal{J}[\alpha'/\alpha]$*.*

Proof. Straightforward, using Weakening and Renaming. □

We define two auxiliary judgements on substitutions as follows:

Definition 2 (Substitution Validity)**.**
The judgement $\Gamma' \vdash \gamma : \Gamma$ *holds if and only if all of the following conditions hold:*

1. $\Gamma' \vdash \Box$
2. $\forall \alpha \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma(\alpha) : \gamma(\Gamma(\alpha))$ *and* $\gamma(\alpha) = \tau$
3. $\forall x \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma(x) : \gamma(\Gamma(x))$

Definition 3 (Substitution Equivalence)**.**
The judgement $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma$ *holds if and only if all of the following conditions hold:*

1. $\Gamma' \vdash \gamma : \Gamma$

2. $\Gamma' \vdash \gamma' : \Gamma$
3. $\forall \alpha \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma(\alpha) \equiv \gamma'(\alpha) : \gamma(\Gamma(\alpha))$
4. $\forall x \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma(x) \equiv \gamma'(x) : \gamma(\Gamma(x))$

We can now state the following simple facts about substitutions:

Proposition 8 (Substitutability)**.**

1. *If* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma' \vdash \gamma : \Gamma$*, then* $\Gamma' \vdash \gamma(\mathcal{J})$*.*
2. *If* $\Gamma_1, \alpha{:}\kappa, \Gamma_2 \vdash \Box$ *and* $\Gamma_1 \vdash \tau : \kappa$*, then* $\Gamma_1, \Gamma_2[\tau/\alpha] \vdash \Box$*.*

Proof. Both by straightforward induction on the derivation. For rule Epsi note that applying a substitution to a value always produces a value. □

Lemma 9 (Substitution Extensibility)**.** *If* $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma$ *and* $\alpha \notin \mathrm{Dom}(\Gamma')$ *and* $\Gamma' \vdash \gamma(\kappa) \equiv \gamma'(\kappa) : \Box$ *with* $\Gamma' \vdash \gamma(\kappa) : \Box$ *and* $\Gamma' \vdash \gamma'(\kappa) : \Box$*, then:*

1. $\Gamma', \alpha{:}\gamma(\kappa) \vdash \gamma \cup [\alpha/\alpha] \equiv \gamma' \cup [\alpha/\alpha] : \Gamma, \alpha{:}\kappa$
2. $\Gamma', \alpha{:}\gamma'(\kappa) \vdash \gamma \cup [\alpha/\alpha] \equiv \gamma' \cup [\alpha/\alpha] : \Gamma, \alpha{:}\kappa$*.*

Proof. Straightforward, using Ntype and Weakening. □

For some proofs we need inversion principles on subkinding and on the type validity judgement. The proof of the latter relies on the following simple lemma:

Lemma 10 (Subderivations)**.**

1. *Every proof of* $\Gamma \vdash \tau_1\, \tau_2 : \kappa$ *contains a strict subderivation* $\Gamma \vdash \tau_1 : \kappa_1$*.*
2. *Every proof of* $\Gamma \vdash \tau{\cdot}1 : \kappa$ *contains a strict subderivation* $\Gamma \vdash \tau : \kappa_1$*.*
3. *Every proof of* $\Gamma \vdash \tau{\cdot}2 : \kappa$ *contains a strict subderivation* $\Gamma \vdash \tau : \kappa_1$*.*

Proof. By straightforward simultaneous induction on the derivation. □

Proposition 11 (Kind Inclusion Inversion)**.** *Let* $\Gamma \vdash \kappa' \leq \kappa : \Box$*.*

1. *If* $\kappa = \Omega$*, then* $\kappa' = \Omega$ *or* $\kappa' = \mathrm{S}_{\hat{\kappa}}(\tau')$*.*
2. *If* $\kappa = \mathrm{A}(\tau)$*, then* $\kappa' = \mathrm{A}(\tau')$ *with* $\Gamma \vdash \tau' \equiv \tau : \Omega$*, or* $\kappa' = \mathrm{S}_{\mathrm{A}(\tau')}(\tau'')$ *with* $\Gamma \vdash \mathrm{A}(\tau') \leq \kappa : \Omega$ *and* $\Gamma \vdash \tau'' : \mathrm{A}(\tau')$*.*
3. *If* $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$*, then* $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau')$ *with* $\Gamma \vdash \hat{\kappa}' \leq \hat{\kappa} : \Omega$ *and* $\Gamma \vdash \tau' \equiv \tau : \hat{\kappa}'$*.*
4. *If* $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$*, then* $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$ *with* $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ *and* $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \Box$*.*
5. *If* $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$*, then* $\kappa' = \Sigma\alpha{:}\kappa_1'.\kappa_2'$ *with* $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \leq \kappa_2' : \Box$*.*

Proof. By simultaneous induction on the derivation. □

Proposition 12 (Type Validity Inversion)**.** *Let* $\Gamma \vdash \tau : \kappa$*.*

1. *If* $\tau = \alpha$*, then* $\Gamma \vdash \Box$ *and* $\alpha \in \mathrm{Dom}(\Gamma)$*.*
2. *If* $\tau = \Psi$*, then* $\Gamma \vdash \Box$*.*

3. *If* $\tau = \tau_1 \to \tau_2$, *then* $\Gamma \vdash \tau_1 : \Omega$ *and* $\Gamma \vdash \tau_2 : \Omega$.
4. *If* $\tau = \tau_1 \times \tau_2$, *then* $\Gamma \vdash \tau_1 : \Omega$ *and* $\Gamma \vdash \tau_2 : \Omega$.
5. *If* $\tau = \forall\alpha{:}\kappa_1.\tau_2$, *then* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$.
6. *If* $\tau = \exists\alpha{:}\kappa_1.\tau_2$, *then* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$.
7. *If* $\tau = \lambda\alpha{:}\kappa_1.\tau_2$, *then* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2$.
8. *If* $\tau = \tau_1\,\tau_2$, *then* $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_2.\kappa_1$ *and* $\Gamma \vdash \tau_2 : \kappa_2$.
9. *If* $\tau = \langle\tau_1, \tau_2\rangle$, *then* $\Gamma \vdash \tau_1 : \kappa_1$ *and* $\Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha]$.
10. *If* $\tau = \tau_1{\cdot}1$, *then* $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$.
11. *If* $\tau = \tau_1{\cdot}2$, *then* $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$.

Proof. Each by straightforward induction on the derivation, using Subderivations in the case of rule TEXT-SIGMA. □

C.1.2. Validity and Functionality

We next show two central invariants of the type system. *Validity* states that any phrase appearing in a derivable judgement is well-formed. *Functionality* says that applying equivalent substitutions to equivalent phrases yields equivalent phrases. Likewise, any inclusion relation on phrases is maintained by equivalent substitutions.

As in Stone & Harper, the proof has to be broken up by first showing a restricted version of Functionality:

Proposition 13 (Simple Functionality)**.** *Let* $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma$.

1. *If* $\Gamma \vdash \Box$, *then* $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma$.
2. *If* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma' \vdash \gamma(\kappa) \equiv \gamma'(\kappa) : \Box$.
3. *If* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma' \vdash \gamma(\kappa) \leq \gamma'(\kappa) : \Box$.
4. *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau) : \gamma(\kappa)$.
5. *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma' \vdash \gamma(\tau) \leq \gamma'(\tau) : \gamma(\kappa)$.

Proof. By simultaneous induction on the derivations.

1. case NEMPTY: $\Gamma = \cdot$
   - by definition of Substitution Equivalence, $\Gamma' \vdash \gamma : \Gamma$ and $\Gamma' \vdash \gamma' : \Gamma$
   - since $\mathrm{Dom}(\Gamma) =$, trivially $\forall\alpha \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma'(\alpha) \equiv \gamma(\alpha) : \Box$ and $\forall x \in \mathrm{Dom}(\Gamma), \Gamma' \vdash \gamma'(x) \equiv \gamma(x) : \gamma'(\Gamma(x))$
   - hence $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma$

   case NTYPE: $\Gamma = \Gamma_1, \alpha{:}\kappa$
   - by inversion, $\Gamma_1 \vdash \kappa : \Box$
   - by Environment Validity, $\Gamma_1 \vdash \Box$
   - by definition of Substitution Equivalence, $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma_1$
   - by induction, $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma_1$
   - by definition of Substitution Equivalence, $\Gamma' \vdash \gamma(\alpha) \equiv \gamma'(\alpha) : \Box$

- by Symmetry, $\Gamma' \vdash \gamma'(\alpha) \equiv \gamma(\alpha) : \Box$
- hence $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma$

case NTERM: $\Gamma = \Gamma_1, x{:}\tau$

- by inversion, $\Gamma_1 \vdash \tau : \Omega$
- by Environment Validity, $\Gamma_1 \vdash \Box$
- by definition of Substitution Equivalence, $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma_1$
- by induction, $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma_1$
- by definition of Substitution Equivalence, $\Gamma' \vdash \gamma(x) \equiv \gamma'(x) : \gamma(\tau)$
- by Symmetry, $\Gamma' \vdash \gamma'(x) \equiv \gamma(x) : \gamma(\tau)$
- by (a) and induction (3), $\Gamma' \vdash \gamma(\tau) \leq \gamma'(\tau) : \Omega$
- by TSUB, $\Gamma' \vdash \gamma'(x) \equiv \gamma(x) : \gamma'(\tau)$
- hence $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma$

2. case KOMEGA: $\kappa = \Omega$
   - by inversion, $\Gamma \vdash \Box$
   - by KQOMEGA, $\Gamma \vdash \Omega \equiv \Omega : \Box$

   case KABS: $\kappa = \mathrm{A}(\tau)$
   - by inversion, $\Gamma \vdash \tau : \Omega$
   - by induction, $\Gamma \vdash \gamma(\tau) \equiv \gamma'(\tau) : \Omega$
   - by KQABS, $\Gamma \vdash \mathrm{A}(\gamma(\tau)) \equiv \mathrm{A}(\gamma'(\tau)) : \Box$

   case KSING: $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$
   - by inversion, $\Gamma \vdash \tau : \hat{\kappa}$
   - by induction, $\Gamma \vdash \gamma(\tau) \equiv \gamma'(\tau) : \gamma(\hat{\kappa})$ and $\Gamma \vdash \gamma(\hat{\kappa}) \equiv \gamma'(\hat{\kappa}) : \Box$
   - by KQSING, $\Gamma \vdash \mathrm{S}_{\gamma(\hat{\kappa})}(\gamma(\tau)) \equiv \mathrm{S}_{\gamma'(\hat{\kappa})}(\gamma'(\tau)) : \Box$

   case KPI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$
   - by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
   - by induction, $\Gamma' \vdash \gamma(\kappa_1) \equiv \gamma'(\kappa_1) : \Box$
   - by Substitutability, $\Gamma' \vdash \gamma(\kappa_1) : \Box$
   - let $\Gamma_1 = \Gamma, \alpha{:}\kappa_1$ and $\Gamma_1' = \Gamma', \alpha{:}\gamma(\kappa_1)$
   - by NTYPE, $\Gamma_1' \vdash \Box$
   - let $\gamma_1 = \gamma \cup [\alpha/\alpha]$ and $\gamma_1' = \gamma' \cup [\alpha/\alpha]$
   - by Substitution Extensibility, $\Gamma_1' \vdash \gamma_1 \equiv \gamma_1' : \Gamma_1$
   - by induction, $\Gamma', \alpha{:}\gamma(\kappa_1) \vdash \gamma_1(\kappa_2) \equiv \gamma_1'(\kappa_2) : \Box$
   - by KQPI, $\Gamma' \vdash \Pi\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2) \equiv \Pi\alpha{:}\gamma'(\kappa_1).\gamma_1'(\kappa_2) : \Box$
   - hence $\Gamma' \vdash \gamma(\Pi\alpha{:}\kappa_1.\kappa_2) \equiv \gamma'(\Pi\alpha{:}\kappa_1.\kappa_2) : \Box$

   case KSIGMA: analogous

3. Similarly, using (1) in the contravariant case.

4. case TVAR: $\tau = \alpha$ and $\kappa = \Gamma(\alpha)$
   - by inversion, $\Gamma \vdash \Box$
   - by definition of Substitution Equivalence, $\Gamma' \vdash \gamma(\alpha) \equiv \gamma'(\alpha) : \gamma(\Gamma(\alpha))$

   case TPSI: $\tau = \Psi$ and $\kappa = \Omega$
   - by inversion, $\Gamma \vdash \Box$
   - by definition of Substitution Equivalence, $\Gamma' \vdash \Box$
   - by TQPSI, $\Gamma' \vdash \Psi \equiv \Psi : \Omega$

   case TARROW: $\tau = \tau_1 \to \tau_2$ and $\kappa = \Omega$

- by inversion, $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$
- by induction, $\Gamma' \vdash \gamma(\tau_1) \equiv \gamma'(\tau_1) : \Omega$ and $\Gamma' \vdash \gamma(\tau_2) \equiv \gamma'(\tau_2) : \Omega$
- by TQArrow, $\Gamma' \vdash \gamma(\tau_1) \to \gamma(\tau_2) \equiv \gamma'(\tau_1) \to \gamma'(\tau_2) : \Omega$

case Ttimes: analogous

case Tuniv: $\tau = \forall\alpha{:}\kappa_1.\tau_2$ and $\kappa = \Omega$

- by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$
- by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
- by induction, $\Gamma' \vdash \gamma(\kappa_1) \equiv \gamma'(\kappa_1) : \Box$
- by Substitutability, $\Gamma' \vdash \gamma(\kappa_1) : \Box$
- let $\Gamma_1 = \Gamma, \alpha{:}\kappa_1$ and $\Gamma'_1 = \Gamma', \alpha{:}\gamma(\kappa_1)$
- by Ntype, $\Gamma'_1 \vdash \Box$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$ and $\gamma'_1 = \gamma' \cup [\alpha/\alpha]$
- by Substitution Extensibility, $\Gamma'_1 \vdash \gamma_1 \equiv \gamma'_1 : \Gamma_1$
- by induction, $\Gamma', \alpha{:}\gamma(\kappa_1) \vdash \gamma_1(\tau_2) \equiv \gamma'_1(\tau_2) : \Omega$
- by TQuniv, $\Gamma' \vdash \forall\alpha{:}\gamma(\kappa_1).\gamma_1(\tau_2) \equiv \forall\alpha{:}\gamma'(\kappa_1).\gamma'_1(\tau_2) : \Omega$
- hence $\Gamma' \vdash \gamma(\forall\alpha{:}\kappa_1.\tau_2) \equiv \gamma'(\forall\alpha{:}\kappa_1.\tau_2) : \Omega$

case Texist: analogous

case Tlambda: analogous

case Tapp: $\tau = \tau_1\,\tau_2$ and $\kappa = \kappa_2[\tau_2/\alpha]$

- by inversion, $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_1$
- by induction, $\Gamma' \vdash \gamma(\tau_1) \equiv \gamma'(\tau_1) : \gamma(\Pi\alpha{:}\kappa_1.\kappa_2)$ and $\Gamma' \vdash \gamma(\tau_2) \equiv \gamma'(\tau_2) : \gamma(\kappa_1)$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$
- hence $\gamma(\Pi\alpha{:}\kappa_1.\kappa_2) = \Pi\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2)$
- by TQapp, $\Gamma' \vdash \gamma(\tau_1\,\tau_2) \equiv \gamma'(\tau_1\,\tau_2) : \gamma_1(\kappa_2)[\gamma(\tau_2)/\alpha]$
- $\gamma_1(\kappa_2)[\gamma(\tau_2)/\alpha] = \gamma(\kappa_2[\tau_2/\alpha])$

case Tpair: $\tau = \langle\tau_1, \tau_2\rangle$ and $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$

- by inversion, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha]$ and $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by induction, $\Gamma' \vdash \gamma(\tau_1) \equiv \gamma'(\tau_1) : \gamma(\kappa_1)$ and $\Gamma' \vdash \gamma(\tau_2) \equiv \gamma'(\tau_2) : \gamma(\kappa_2[\tau_1/\alpha])$
- by Substitutability, $\Gamma' \vdash \gamma(\Sigma\alpha{:}\kappa_1.\kappa_2) : \Box$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$
- hence $\gamma(\Sigma\alpha{:}\kappa_1.\kappa_2) = \Sigma\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2)$ and $\gamma(\kappa_2[\tau_1/\alpha]) = \gamma_1(\kappa_2)[\gamma(\tau_1)/\alpha]$
- by TQpair, $\Gamma' \vdash \gamma\langle\tau_1, \tau_2\rangle \equiv \gamma'\langle\tau_1, \tau_2\rangle : \gamma(\Sigma\alpha{:}\kappa_1.\kappa_2)$

case Tfst: $\tau = \tau_1{\cdot}1$ and $\kappa = \kappa_1$

- by inversion, $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$
- by induction, $\Gamma' \vdash \gamma(\tau_1) \equiv \gamma'(\tau_1) : \gamma(\Sigma\alpha{:}\kappa_1.\kappa_2)$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$
- hence $\gamma(\Sigma\alpha{:}\kappa_1.\kappa_2) = \Sigma\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2)$
- by TQfst, $\Gamma; \vdash \gamma(\tau_1){\cdot}1 \equiv \gamma'(\tau_1){\cdot}1 : \gamma(\kappa_1)$

case Tsnd: similarly

case Text-sing: $\kappa = \mathrm{S}_{\hat\kappa}(\tau)$

- by inversion, $\Gamma \vdash \tau : \hat\kappa$
- by Substitutability, $\Gamma' \vdash \gamma(\tau) : \gamma(\hat\kappa)$ and $\Gamma' \vdash \gamma'(\tau) : \gamma'(\hat\kappa)$
- by Text-sing, $\Gamma' \vdash \gamma(\tau) : \mathrm{S}_{\gamma(\hat\kappa)}(\gamma(\tau))$ and $\Gamma' \vdash \gamma'(\tau) : \mathrm{S}_{\gamma'(\hat\kappa)}(\gamma'(\tau))$
- by Environment Validity, $\Gamma \vdash \Box$
- by induction, $\Gamma' \vdash \gamma' \equiv \gamma : \Gamma$
- by induction, $\Gamma' \vdash \gamma'(\tau) \equiv \gamma(\tau) : \gamma'(\hat\kappa)$ and $\Gamma' \vdash \gamma'(\hat\kappa) \leq \gamma(\hat\kappa) : \Box$

- by KSSING, $\Gamma' \vdash \mathrm{S}_{\gamma'(\hat{\kappa})}(\gamma'(\tau)) \leq \mathrm{S}_{\gamma(\hat{\kappa})}(\gamma(\tau)) : \Box$
- by TSUB, $\Gamma' \vdash \gamma'(\tau) : \mathrm{S}_{\gamma(\hat{\kappa})}(\gamma(\tau))$
- by TQEXT-SING, $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau) : \mathrm{S}_{\gamma(\hat{\kappa})}(\gamma(\tau))$

case TEXT-PI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$

- by inversion, $\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2'$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \kappa_2$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2' : \Box$
- by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
- by Substitutability, $\Gamma' \vdash \gamma(\kappa_1) : \Box$
- let $\Gamma_1 = \Gamma, \alpha{:}\kappa_1$ and $\Gamma_1' = \Gamma', \alpha{:}\gamma(\kappa_1)$
- by NTYPE, $\Gamma_1' \vdash \Box$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$ and $\gamma_1' = \gamma' \cup [\alpha/\alpha]$
- by Substitution Extensibility, $\Gamma_1' \vdash \gamma_1 \equiv \gamma_1' : \Gamma_1$
- by induction, $\Gamma', \alpha{:}\gamma(\kappa_1) \vdash \gamma_1(\tau)\,\alpha \equiv \gamma_1'(\tau)\,\alpha : \gamma_1(\kappa_2)$
- by Substitutability,
  $\Gamma' \vdash \gamma(\tau) : \Pi\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2)$ and $\Gamma' \vdash \gamma'(\tau) : \Pi\alpha{:}\gamma'(\kappa_1).\gamma_1'(\kappa_2)$
- by induction, $\Gamma' \vdash \gamma(\kappa_1) \leq \gamma'(\kappa_1) : \Box$
- by inverting KPI, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' : \Box$
- by Substitutability, $\Gamma', \alpha{:}\gamma'(\kappa_1) \vdash \gamma_1'(\kappa_2') : \Box$
- by Reflexivity, $\Gamma', \alpha{:}\gamma'(\kappa_1) \vdash \gamma_1'(\kappa_2') \leq \gamma_1'(\kappa_2') : \Box$
- by Weakening, $\Gamma', \alpha{:}\gamma(\kappa_1) \vdash \gamma_1'(\kappa_2') \leq \gamma_1'(\kappa_2') : \Box$
- by Substitutability, $\Gamma' \vdash \Pi\alpha{:}\gamma'(\kappa_1).\gamma_1(\kappa_2') : \Box$
- by KSPI, $\Gamma' \vdash \Pi\alpha{:}\gamma'(\kappa_1).\gamma_1'(\kappa_2') \leq \Pi\alpha{:}\gamma(\kappa_1).\gamma_1'(\kappa_2') : \Box$
- by TQSUB, $\Gamma' \vdash \gamma'(\tau) : \Pi\alpha{:}\gamma(\kappa_1).\gamma_1'(\kappa_2')$
- by TQEXT-PI, $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau) : \Pi\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2)$
- $\Pi\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2) = \gamma(\Pi\alpha{:}\kappa_1.\kappa_2)$

case TEXT-SIGMA: $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$

- by inversion, $\Gamma \vdash \tau{\cdot}1 : \kappa_1$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$ and $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
- by Substitutability, $\Gamma' \vdash \gamma(\kappa_1) : \Box$
- let $\Gamma_1 = \Gamma, \alpha{:}\kappa_1$ and $\Gamma_1' = \Gamma', \alpha{:}\gamma(\kappa_1)$
- by NTYPE, $\Gamma_1' \vdash \Box$
- let $\gamma_1 = \gamma \cup [\alpha/\alpha]$ and $\gamma_1' = \gamma' \cup [\alpha/\alpha]$
- by Substitution Extensibility, $\Gamma_1' \vdash \gamma_1 \equiv \gamma_1' : \Gamma_1$
- by induction,
  $\Gamma' \vdash \gamma(\tau){\cdot}1 \equiv \gamma'(\tau){\cdot}1 : \gamma(\kappa_1)$ and $\Gamma', \alpha{:}\gamma(\kappa_1) \vdash \gamma_1(\tau){\cdot}2 \equiv \gamma_1'(\tau){\cdot}2 : \gamma_1(\kappa_2[\tau{\cdot}1/\alpha])$
- $\gamma_1(\kappa_2[\tau{\cdot}1/\alpha]) = \gamma_1(\kappa_2)[\gamma_1(\tau){\cdot}1/\alpha]$
- by Substitutability, $\Gamma' \vdash \Sigma\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2) : \Box$
- by TQEXT-SIGMA, $\Gamma' \vdash \Sigma\alpha{:}\gamma(\kappa_1).\gamma_1(\kappa_2) \leq \Sigma\alpha{:}\gamma'(\kappa_1).\gamma_1'(\kappa_2) : \Box$

case TSUB:

- by inversion, $\Gamma \vdash \tau : \kappa'$ and $\Gamma \vdash \kappa' \leq \kappa : \Box$
- by induction, $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau) : \gamma(\kappa')$
- by Substitutability, $\Gamma' \vdash \gamma(\kappa') \leq \gamma(\kappa) : \Box$
- by TQSUB, $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau) : \gamma(\kappa)$

5. Similarly, using (1) in the contravariant cases.

□

The proof of Validity itself mutually depends on Reflexivity of subkinding. We hence state the latter together with Validity (item (3)).[1]

[1] The proofs in Stone & Harper contain a minor bug with respect to this.

Proposition 14 (Validity)**.**

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*.*
2. *If* $\Gamma \vdash \kappa \leq \kappa' : \Box$*, then* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*.*
3. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \vdash \kappa \leq \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \leq \kappa : \Box$*.*
4. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \kappa : \Box$*.*
5. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\Gamma \vdash \kappa : \Box$*.*
6. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$*, then* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\Gamma \vdash \kappa : \Box$*.*

Proof. By simultaneous induction on the derivation. We show only the non-trivial cases:

1. case KQSING: $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$ and $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau')$
   - by inversion, $\Gamma \vdash \tau \equiv \tau' : \hat{\kappa}$ and $\Gamma \vdash \hat{\kappa} \equiv \hat{\kappa}' : \Box$
   - by induction, $\Gamma \vdash \tau : \hat{\kappa}$ and $\Gamma \vdash \tau' : \hat{\kappa}$ and $\Gamma \vdash \hat{\kappa} : \Box$ and $\Gamma \vdash \hat{\kappa}' : \Box$ and $\Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \Box$
   - by TSUB, $\Gamma \vdash \tau' : \hat{\kappa}'$
   - by KSING, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) : \Box$ and $\Gamma \vdash \mathrm{S}_{\hat{\kappa}'}(\tau') : \Box$

   case KQPI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$ and $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$
   - by inversion, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box$
   - by induction, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' : \Box$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2' : \Box$
   - by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box$

   case KQSIGMA: similarly

2. case KSSING: analogous to KQSING

   case KSSING-LEFT: analogous to KQSING

   case KSPI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$ and $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$
   - by inversion, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by induction, $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2' : \Box$
   - by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box$

   case KSSIGMA: similarly

3. case KQSING: $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$ and $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau')$
   - by inversion, $\Gamma \vdash \tau \equiv \tau' : \hat{\kappa}$ and $\Gamma \vdash \hat{\kappa} \equiv \hat{\kappa}' : \Box$
   - by induction (3), $\Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \Box$ and $\Gamma \vdash \hat{\kappa}' \leq \hat{\kappa} : \Box$
   - by TQSYMM, $\Gamma \vdash \tau' \equiv \tau : \hat{\kappa}$
   - bz TQSUB, $\Gamma \vdash \tau' \equiv \tau : \hat{\kappa}'$
   - by KSSING, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \mathrm{S}_{\hat{\kappa}'}(\tau') : \Box$ and $\Gamma \vdash \mathrm{S}_{\hat{\kappa}'}(\tau') \leq \mathrm{S}_{\hat{\kappa}}(\tau) : \Box$

   case KQPI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$ and $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$
   - by inversion, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box$
   - by induction, $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \leq \kappa_2' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' \leq \kappa_2 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' : \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \Box$
   - by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box$
   - by KSPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2' \leq \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$

   case KQSIGMA: analogous

4. case TAPP: $\tau = \tau_1\,\tau_2$ and $\kappa = \kappa_2[\tau_2/\alpha]$
   - by inversion, $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_1$
   - by induction, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by inverting KPI, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Substitutability, $\Gamma \vdash \kappa_2[\tau_2/\alpha] : \Box$

   case TFST: $\tau = \tau_1{\cdot}1$ and $\kappa = \kappa_1$
   - by inversion, $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$
   - by induction, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by inverting KSIGMA, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$

   case TSND: $\tau = \tau_1{\cdot}2$ and $\kappa = \kappa_2[\tau_1{\cdot}1/\alpha]$
   - by inversion, $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$
   - by induction, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by inverting KSIGMA, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by TFST, $\Gamma \vdash \tau_1{\cdot}1 : \kappa_1$
   - by Substitutability, $\Gamma \vdash \kappa_2[\tau_1{\cdot}1/\alpha] : \Box$

5. case TQUNIV: $\tau = \forall\alpha{:}\kappa_1.\tau_2$ and $\tau' = \forall\alpha{:}\kappa_1'.\tau_2'$ and $\kappa = \Omega$
   - by inversion, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_2' : \Omega$
   - by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2' : \Omega$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' : \Omega$
   - by TUNIV, $\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 : \Omega$ and $\Gamma \vdash \forall\alpha{:}\kappa_1'.\tau_2' : \Omega$
   - by Environment Validity and KOMEGA, $\Gamma \vdash \Omega : \Box$

   case TQEXIST: analogous

   case TQLAMBDA: $\tau = \lambda\alpha{:}\kappa_1.\tau_2$ and $\tau' = \lambda\alpha{:}\kappa_1'.\tau_2'$ and $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$
   - by inversion, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_2' : \kappa_2$
   - by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2' : \kappa_2$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$
   - by TLAMBDA, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \lambda\alpha{:}\kappa_1'.\tau_2' : \Pi\alpha{:}\kappa_1'.\kappa_2$
   - by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2 : \Box$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1' : \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' : \kappa_2$ and $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 : \Box$
   - by Reflexivity, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \leq \kappa_2 : \Box$
   - by KSPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1'.\kappa_2 \leq \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by TSUB, $\Gamma \vdash \lambda\alpha{:}\kappa_1'.\tau_2' : \Pi\alpha{:}\kappa_1.\kappa_2$

   case TQAPP: $\tau = \tau_1\,\tau_2$ and $\tau' = \tau_1'\,\tau_2'$ and $\kappa = \kappa_2[\tau_2/\alpha]$
   - by inversion, $\Gamma \vdash \tau_1 \equiv \tau_1' : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_1$
   - by induction, $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_1' : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_1$ and $\Gamma \vdash \tau_2' : \kappa_1$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$ and $\Gamma \vdash \kappa_1 : \Box$
   - by TAPP, $\Gamma \vdash \tau_1\,\tau_2 : \kappa_2[\tau_2/\alpha]$ and $\Gamma \vdash \tau_1'\,\tau_2' : \kappa_2[\tau_2'/\alpha]$
   - by inverting KPI, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Substitutability, $\Gamma \vdash \kappa_2[\tau_2/\alpha] : \Box$
   - $\Gamma \vdash [\tau_2/\alpha] \equiv [\tau_2'/\alpha] : \Gamma, \alpha{:}\kappa_1$
   - by Simple Functionality, $\Gamma \vdash \kappa_2[\tau_2'/\alpha] \leq \kappa_2[\tau_2/\alpha] : \Box$
   - by TSUB, $\Gamma \vdash \tau_1'\,\tau_2' : \kappa_2[\tau_2/\alpha]$

   case TQSND: $\tau = \tau_1{\cdot}2$ and $\tau' = \tau_1'{\cdot}2$ and $\kappa = \kappa_2[\tau_1{\cdot}1/\alpha]$
   - by inversion, $\Gamma \vdash \tau_1 \equiv \tau_1' : \Sigma\alpha{:}\kappa_1.\kappa_2$

- by induction, $\Gamma \vdash \tau_1 : \Sigma\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_1' : \Sigma\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by TSND, $\Gamma \vdash \tau_1{\cdot}2 : \kappa_2[\tau_1{\cdot}1/\alpha]$ and $\Gamma \vdash \tau_1'{\cdot}2 : \kappa_2[\tau_1'{\cdot}1/\alpha]$
- by inverting KSIGMA, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by TFST, $\Gamma \vdash \tau_1{\cdot}1 : \kappa_1$ and $\Gamma \vdash \tau_1'{\cdot}1 : \kappa_1$
- by Substitutability, $\Gamma \vdash \kappa_2[\tau_1{\cdot}1/\alpha] : \Box$
- by TQFST, $\Gamma \vdash \tau_1{\cdot}1 \equiv \tau_1'{\cdot}1 : \kappa_1$
- hence $\Gamma \vdash [\tau_1{\cdot}1/\alpha] \equiv [\tau_1'{\cdot}1/\alpha] : \Gamma, \alpha{:}\kappa_1$
- by Simple Functionality, $\Gamma \vdash \kappa_2[\tau_1'/\alpha] \leq \kappa_2[\tau_1/\alpha] : \Box$
- by TSUB, $\Gamma \vdash \tau_1'{\cdot}2 : \kappa_2[\tau_1{\cdot}1/\alpha]$

case TQEXT-PI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$

- by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha \equiv \tau'\,\alpha : \kappa_2$ and $\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2'$ and $\Gamma \vdash \tau' : \Pi\alpha{:}\kappa_1.\kappa_2''$
- by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \kappa_2$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau'\,\alpha : \kappa_2$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2' : \Box$ and $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2'' : \Box$
- by TEXT-PI, $\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau' : \Pi\alpha{:}\kappa_1.\kappa_2$
- by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$

case TQEXT-SIGMA:

- by inversion, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1$ and $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$ and $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by induction, $\Gamma \vdash \tau'{\cdot}1 : \kappa_1$ and $\Gamma \vdash \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$
- by TEXT-SIGMA, $\Gamma \vdash \tau' : \Sigma\alpha{:}\kappa_1.\kappa_2$

6. case TSUNIV: $\tau = \forall\alpha{:}\kappa_1.\tau_2$ and $\tau' = \forall\alpha{:}\kappa_1'.\tau_2'$ and $\kappa = \Omega$
   - by inversion, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2 \leq \tau_2' : \Omega$ and $\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 : \Omega$
   - by induction, $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' : \Omega$
   - by TUNIV, $\Gamma \vdash \forall\alpha{:}\kappa_1'.\tau_2' : \Omega$
   - by Environment Validity and KOMEGA, $\Gamma \vdash \Omega : \Box$

   case TSEXIST: similarly.

□

Given Validity, we can easily prove symmetry and transitivity properties:

Proposition 15 (Antisymmetry of Kind Inclusion)**.** *If and only if* $\Gamma \vdash \kappa \leq \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \leq \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*

Proof. The inverse direction has already been shown as Validity (3). The other is shown by induction on the added size of both derivations. The only cases possible are those where both derivations used the same rule:

case KSOMEGA: $\kappa = \kappa' = \Omega$

- by inversion, $\Gamma \vdash \Box$
- by KQOMEGA, $\Gamma \vdash \Omega \equiv \Omega : \Box$

case KSABS: $\kappa = \mathrm{A}(\tau)$ and $\kappa' = \mathrm{A}(\tau')$

- by inversion, $\Gamma \vdash \tau \equiv \tau' : \Omega$
- by KQABS, $\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \Box$

case KSSING: $\kappa = \mathrm{S}_{\hat\kappa}(\tau)$ and $\kappa' = \mathrm{S}_{\hat\kappa'}(\tau')$

- by inversion, $\Gamma \vdash \tau \equiv \tau' : \hat\kappa$ and $\Gamma \vdash \hat\kappa \leq \hat\kappa' : \Box$ and $\Gamma \vdash \hat\kappa' \leq \hat\kappa : \Box$
- by induction, $\Gamma \vdash \hat\kappa \equiv \hat\kappa' : \Box$

– by KQSING, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \equiv \mathrm{S}_{\hat{\kappa}'}(\tau') : \Box$

case KSPI: $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$ and $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$

– by inversion, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \Box$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' \leq \kappa_2 : \Box$

– by induction, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \equiv \kappa_2' : \Box$

– by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$

– by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box$

– by KQPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 \equiv \Pi\alpha{:}\kappa_1'.\kappa_2' : \Box$

case KSSIGMA: analogous

□

Proposition 16 (Symmetry)**.**

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \vdash \kappa' \equiv \kappa : \Box$*.*
2. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \vdash \tau' \equiv \tau : \kappa$*.*

Proof.

1. By easy induction on the derivation.
2. Trivial, by Rule TQSYMM.

□

Proposition 17 (Transitivity)**.**

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \equiv \kappa'' : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa'' : \Box$*.*
2. *If* $\Gamma \vdash \kappa \leq \kappa' : \Box$ *and* $\Gamma \vdash \kappa' \leq \kappa'' : \Box$*, then* $\Gamma \vdash \kappa \leq \kappa'' : \Box$*.*
3. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$ *and* $\Gamma \vdash \tau' \equiv \tau'' : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau'' : \kappa$*.*
4. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$ *and* $\Gamma \vdash \tau' \leq \tau'' : \kappa$*, then* $\Gamma \vdash \tau \leq \tau'' : \kappa$*.*

Proof.

1. By straightforward induction on $\mathrm{Size}(\kappa)+\mathrm{Size}(\kappa')+\mathrm{Size}(\kappa'')$, using (3) in case of singletons and abstraction types, and Antisymmetry and Weakening in case of binders.
2. Similarly.
3. Trivially by rule TQTRANS.
4. Trivially by rule TSTRANS.

□

This gives us the necessary prerequisites for proving Functionality in its final formulation:

Proposition 18 (Full Functionality)**.** *Let* $\Gamma' \vdash \gamma \equiv \gamma' : \Gamma$*.*

1. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma' \vdash \gamma(\kappa) \equiv \gamma'(\kappa') : \Box$*.*
2. *If* $\Gamma \vdash \kappa \leq \kappa' : \Box$*, then* $\Gamma' \vdash \gamma(\kappa) \leq \gamma'(\kappa') : \Box$*.*

3. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$, *then* $\Gamma' \vdash \gamma(\tau) \equiv \gamma'(\tau') : \gamma(\kappa)$.

4. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$, *then* $\Gamma' \vdash \gamma(\tau) \leq \gamma'(\tau') : \gamma(\kappa)$.

Proof. Each by induction on the derivation:

1.
   - by Substitutability, $\Gamma' \vdash \gamma(\kappa) \equiv \gamma(\kappa') : \Box$
   - by Validity, $\Gamma \vdash \kappa' : \Box$
   - by Simple Functionality, $\Gamma' \vdash \gamma(\kappa') \equiv \gamma'(\kappa') : \Box$
   - by Transitivity, $\Gamma' \vdash \gamma(\kappa) \equiv \gamma'(\kappa') : \Box$

2–4. Likewise.

□

C.2. Admissible Rules

C.2.1. Higher-Order Singletons

With Validity and Functionality proven in the previous section, we are now prepared to show the admissibility of the derived rules for higher-order singletons (Figure 11.9) and for $\beta\eta$-equivalences. We just need another trivial lemma:

Lemma 19 (Kind Subsumption)**.**

1. *If* $\Gamma \vdash \tau : \mathrm{A}(\tau_1)$, *then* $\Gamma \vdash \tau : \Omega$.

2. *If* $\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau_1)$, *then* $\Gamma \vdash \tau : \hat{\kappa}$.

3. *If* $\Gamma \vdash \tau \equiv \tau' : \mathrm{A}(\tau_1)$, *then* $\Gamma \vdash \tau \equiv \tau' : \Omega$.

4. *If* $\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau_1)$, *then* $\Gamma \vdash \tau \equiv \tau' : \hat{\kappa}$.

5. *If* $\Gamma \vdash \tau \leq \tau' : \mathrm{A}(\tau_1)$, *then* $\Gamma \vdash \tau \leq \tau' : \Omega$.

6. *If* $\Gamma \vdash \tau \leq \tau' : \mathrm{S}_{\hat{\kappa}}(\tau_1)$, *then* $\Gamma \vdash \tau \leq \tau' : \hat{\kappa}$.

Proof. Straightforward, using Validity. □

Theorem 20 (Admissibility of Higher-Order Singleton Rules)**.**
The rules KSING*, TEXT-SING, KQSING*, KSSING*, KSSING-LEFT* *and* TQEXT-SING* *are admissible.*

Proof.

1. KQSING*: By induction on the size of κ.

   case $\kappa = \Omega$:
   - by inverting KQOMEGA, $\kappa' = \Omega$
   - by KQSING, $\Gamma \vdash \mathrm{S}_\Omega(\tau) \equiv \mathrm{S}_\Omega(\tau') : \Box$

   case $\kappa = \mathrm{A}(\tau_1)$:
   - by inverting KQABS, $\kappa' = \mathrm{A}(\tau_1')$
   - by KQSING, $\Gamma \vdash \mathrm{S}_{\mathrm{A}(\tau_1)}(\tau) \equiv \mathrm{S}_{\mathrm{A}(\tau_1')}(\tau') : \Box$

   case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau_1)$:

- by inverting KQSING, $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau_1')$
- by premise, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau_1) \equiv \mathrm{S}_{\hat{\kappa}'}(\tau_1') : \Box$

case $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$:

- by inverting KQPI,
  $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$ and $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau \equiv \tau' : \Pi\alpha{:}\kappa_1.\kappa_2$
- by TQVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha \equiv \alpha : \kappa_1$
- by TQAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha \equiv \tau'\,\alpha : \kappa_2$
- by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau\,\alpha : \kappa_2) \equiv \mathrm{S}(\tau'\,\alpha : \kappa_2') : \Box$
- by KQPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2) \equiv \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2') : \Box$

case $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$:

- by Validity, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by inverting KQSIGMA,
  $\kappa' = \Sigma\alpha{:}\kappa_1'.\kappa_2'$ and $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 \equiv \kappa_2' : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by TQFST, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \kappa_1$
- by TQSND, $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$
- obviously, $\Gamma \vdash [\tau{\cdot}1/\alpha] \equiv [\tau'{\cdot}1/\alpha] : \Gamma, \alpha{:}\kappa_1$
- by Full Functionality, $\Gamma \vdash \kappa_2[\tau{\cdot}1/\alpha] \equiv \kappa_2'[\tau'{\cdot}1/\alpha] : \Box$
- by induction, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \equiv \mathrm{S}(\tau'{\cdot}1 : \kappa_1') : \Box$
  and $\Gamma \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \equiv \mathrm{S}(\tau'{\cdot}2 : \kappa_2'[\tau'{\cdot}1/\alpha]) : \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \equiv \mathrm{S}(\tau'{\cdot}2 : \kappa_2'[\tau'{\cdot}1/\alpha]) : \Box$
- by KQSIGMA,
  $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \equiv \mathrm{S}(\tau'{\cdot}1 : \kappa_1') \times \mathrm{S}(\tau'{\cdot}2 : \kappa_2'[\tau'{\cdot}1/\alpha]) : \Box$

2. KSSING*: similarly, but with a case distinction for abstraction kinds and singletons:

case $\kappa = \mathrm{A}(\tau_1)$: case distinction for κ':

subcase $\kappa' = \Omega$ (by KSABS-LEFT):

- by KSSING, $\Gamma \vdash \mathrm{S}_{\mathrm{A}(\tau_1)}(\tau) \leq \mathrm{S}_{\Omega}(\tau') : \Box$

subcase $\kappa' = \mathrm{A}(\tau_1')$ (by KSABS):

- by KSSING, $\Gamma \vdash \mathrm{S}_{\mathrm{A}(\tau_1)}(\tau) \leq \mathrm{S}_{\mathrm{A}(\tau_1')}(\tau') : \Box$

case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau_1)$: case distinction for κ':

subcase $\kappa' = \hat{\kappa}'$ (by KSSING-LEFT):

- by inverting KSSING-LEFT, $\Gamma \vdash \hat{\kappa} \leq \hat{\kappa}' : \Box$
- by inverting KSING, $\Gamma \vdash \tau_1 : \hat{\kappa}$
- by TEXT-SING, $\Gamma \vdash \tau_1 : \mathrm{S}_{\hat{\kappa}}(\tau_1)$
- by Validity, $\Gamma \vdash \tau_1 : \mathrm{S}_{\hat{\kappa}}(\tau_1)$
- by TQEXT-SING, $\Gamma \vdash \tau_1 \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau_1)$
- by Kind Subsumption, $\Gamma \vdash \tau_1 \equiv \tau' : \hat{\kappa}$
- by KSSING, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau_1) \leq \mathrm{S}_{\hat{\kappa}'}(\tau') : \Box$

subcase $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau_1')$ (by KSSING):

- by premise, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau_1) \leq \mathrm{S}_{\hat{\kappa}'}(\tau_1') : \Box$

3. KSSING-LEFT*: by induction on the size of κ:

case $\kappa = \hat{\kappa}$:

- by KSSING-LEFT, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) \leq \hat{\kappa} : \Box$

case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau_1)$:
- by Validity, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau_1) : \Box$
- by Reflexivity, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau_1) \leq \mathrm{S}_{\hat{\kappa}}(\tau_1) : \Box$

case $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$:
- by Validity, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
- by inverting KPI, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by Reflexivity, $\Gamma \vdash \kappa_1 \leq \kappa_1 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2$
- by TVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$
- by TAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \kappa_2$
- by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau\,\alpha : \kappa_2) \leq \kappa_2 : \Box$
- by KSPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2) \leq \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$

case $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$:
- by Validity, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by inverting KSIGMA, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by Reflexivity, $\Gamma \vdash \kappa_1 \leq \kappa_1 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by TFST, $\Gamma \vdash \tau{\cdot}1 : \kappa_1$
- by TSND, $\Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$
- by induction, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \leq \kappa_1 : \Box$ and $\Gamma \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \leq \kappa_2[\tau{\cdot}1/\alpha] : \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \leq \kappa_2[\tau{\cdot}1/\alpha] : \Box$
- by KSSIGMA, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) \leq \kappa_1 \times \kappa_2[\tau{\cdot}1/\alpha] : \Box$

4. TEXT-SING*: by induction on the size of κ:

case $\kappa = \hat{\kappa}$:
- by TEXT-SING, $\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau)$

case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau_1)$:
- by premise, $\Gamma \vdash \tau : \mathrm{S}_{\hat{\kappa}}(\tau_1)$

case $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$:
- by Validity, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
- by inverting KPI, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2$
- by TVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$
- by TAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \kappa_2$
- by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \mathrm{S}(\tau\,\alpha : \kappa_2)$
- by TEXT-PI, $\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\,\alpha : \kappa_2)$

case $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$:
- by Validity, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by inverting KSIGMA, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by TFST, $\Gamma \vdash \tau{\cdot}1 : \kappa_1$
- by TSND, $\Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$
- by induction, $\Gamma \vdash \tau{\cdot}1 : \mathrm{S}(\tau{\cdot}1 : \kappa_1)$ and $\Gamma \vdash \tau{\cdot}2 : \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha])$
- by Validity, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) : \Box$ and $\Gamma \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$

- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau{\cdot}2 : \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha])$ and $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$
- by KSIGMA, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$
- by TEXT-SIGMA, $\Gamma \vdash \tau : \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha])$

5. KSING*:
   - by TEXT-SING*, $\Gamma \vdash \tau : \mathrm{S}(\tau : \kappa)$
   - by Validity, $\Gamma \vdash \mathrm{S}(\tau : \kappa) : \Box$
6. TQEXT-SING*: by induction on the size of κ:

   case $\kappa = \hat{\kappa}$:
   - by TQEXT-SING, $\Gamma \vdash \tau \equiv \tau' : \mathrm{S}_{\hat{\kappa}}(\tau'')$

   case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau_1)$: likewise

   case $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$:
   - by Validity, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau''\,\alpha : \kappa_2) : \Box$
   - by inverting KPI, $\Gamma \vdash \kappa_1 : \Box$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau''\,\alpha : \kappa_2)$
   - by TVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$
   - by TAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha : \mathrm{S}(\tau''\,\alpha : \kappa_2)$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau'\,\alpha : \mathrm{S}(\tau''\,\alpha : \kappa_2)$
   - by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau\,\alpha \equiv \tau'\,\alpha : \mathrm{S}(\tau''\,\alpha : \kappa_2)$
   - by TQEXT-PI, $\Gamma \vdash \tau \equiv \tau' : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau''\,\alpha : \kappa_2)$

   case $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$:
   - w.l.o.g., $\alpha \notin \mathrm{FV}(\tau'')$
   - by Validity, $\Gamma \vdash \mathrm{S}(\tau''{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau''{\cdot}2 : \kappa_2[\tau''{\cdot}1/\alpha]) : \Box$
   - by TFST, $\Gamma \vdash \tau{\cdot}1 : \mathrm{S}(\tau''{\cdot}1 : \kappa_1)$ and $\Gamma \vdash \tau'{\cdot}1 : \mathrm{S}(\tau''{\cdot}1 : \kappa_1)$
   - by TSND, $\Gamma \vdash \tau{\cdot}2 : \mathrm{S}(\tau''{\cdot}2 : \kappa_2[\tau''{\cdot}1/\alpha])$ and $\Gamma \vdash \tau'{\cdot}2 : \mathrm{S}(\tau''{\cdot}2 : \kappa_2[\tau''{\cdot}1/\alpha])$
   - by induction, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \mathrm{S}(\tau''{\cdot}1 : \kappa_1)$ and $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \mathrm{S}(\tau''{\cdot}2 : \kappa_2[\tau''{\cdot}1/\alpha])$
   - by TQEXT-SIGMA, $\Gamma \vdash \tau : \mathrm{S}(\tau''{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau''{\cdot}2 : \kappa_2[\tau''{\cdot}1/\alpha])$

□

Using the singleton rules, we can show that a variable with singleton kind can always be replaced by the equivalent type:

Lemma 21 (Singleton Substitutability)**.** *Let* $\Gamma \vdash \tau : \kappa$

1. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' : \Box$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' \equiv \kappa'[\tau/\alpha] : \Box$*.*
2. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' \equiv \tau'[\tau/\alpha] : \kappa'$*.*
3. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'[\tau/\alpha]$*.*
4. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash e : \tau'$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash e : \tau'[\tau/\alpha]$*.*
5. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'[\tau/\alpha]$ *and* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' : \Box$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'$*.*
6. *If* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash e : \tau'[\tau/\alpha]$ *and* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'$*, then* $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash e : \tau'$*.*

Proof.

1. • by Environment Validity, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \Box$

- by Weakening, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau : \kappa$
- by TEXT-SING*, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau : \mathrm{S}(\tau : \kappa)$
- by TVAR, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \alpha : \mathrm{S}(\tau : \kappa)$
- by TQEXT-SING*, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \alpha \equiv \tau : \mathrm{S}(\tau : \kappa)$
- obviously, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash [\alpha/\alpha] \equiv [\tau/\alpha] : \Gamma, \alpha{:}\mathrm{S}(\tau : \kappa)$
- by Simple Functionality, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' \equiv \kappa'[\tau/\alpha] : \Box$

2. Analogous.

3.
- by Validity, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' : \Box$
- by (1), $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' \equiv \kappa'[\tau/\alpha] : \Box$
- by Antisymmetry, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' \leq \kappa'[\tau/\alpha] : \Box$
- by TSUB, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'[\tau/\alpha]$

4. Analogous.

5.
- by (1), $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa' \equiv \kappa'[\tau/\alpha] : \Box$
- by Antisymmetry, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \kappa'[\tau/\alpha] \leq \kappa' : \Box$
- by TSUB, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau' : \kappa'$

6. Analogous.

□

C.2.2. $\beta\eta$-Equivalence

Proving $\beta\eta$-equivalences is now straightforward:

Theorem 22 (Admissibility of Beta/Eta Rules)**.** *The equivalence rules* TQAPP-BETA*, TQLAMBDA-ETA*, TQFST-BETA*, TQSND-BETA* *and* TQPAIR-ETA* *are admissible.*

Proof.

1. TQAPP-BETA*:
   - by TEXT-SING*, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \mathrm{S}(\tau_2 : \kappa_2)$
   - by TLAMBDA, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau_2 : \kappa_2)$
   - by TAPP, $\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 : \mathrm{S}(\tau_2 : \kappa_2)[\tau_1/\alpha]$
   - obviously, $\Gamma \vdash [\tau_1/\alpha] : \Gamma, \alpha{:}\kappa_1$
   - by Substitutability, $\Gamma \vdash \tau_2[\tau_1/\alpha] : \mathrm{S}(\tau_2 : \kappa_2)[\tau_1/\alpha]$
   - by TQEXT-SING*, $\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 \equiv \tau_2[\tau_1/\alpha] : \mathrm{S}(\tau_2 : \kappa_2)[\tau_1/\alpha]$
   - by Substitutability, $\Gamma \vdash \tau_2[\tau_1/\alpha] : \kappa_2[\tau_1/\alpha]$
   - by KSSING-LEFT*, $\Gamma \vdash \mathrm{S}(\tau_2 : \kappa_2)[\tau_1/\alpha] \leq \kappa_2[\tau_1/\alpha] : \Box$
   - by TQSUB, $\Gamma \vdash (\lambda\alpha{:}\kappa_1.\tau_2)\,\tau_1 \equiv \tau_2[\tau_1/\alpha] : \kappa_2[\tau_1/\alpha]$
2. TQLAMBDA-ETA*:
   - by Validity, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2$
   - by inverting KPI, $\Gamma \vdash \kappa_1 : \Box$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$

- by TVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2$
- by TAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2\,\alpha : \kappa_2$
- by TEXT-SING*, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2\,\alpha : \mathrm{S}(\tau_1\,\alpha : \kappa_2)$
- by TLAMBDA, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2\,\alpha : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau_1\,\alpha : \kappa_2)$
- by TEXT-SING*, $\Gamma \vdash \tau_2 : \mathrm{S}(\tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2)$
- by TQEXT-SING*, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2\,\alpha \equiv \tau_2 : \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau_1\,\alpha : \kappa_2)$

3. TQFST-BETA*:
   - w.l.o.g., $\alpha \notin \mathrm{FV}(\kappa_2)$
   - by TEXT-SING*, $\Gamma \vdash \tau_1 : \mathrm{S}(\tau_1 : \kappa_1)$ and $\Gamma \vdash \tau_2 : \mathrm{S}(\tau_2 : \kappa_2)$
   - by TPAIR, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \mathrm{S}(\tau_1 : \kappa_1) \times \mathrm{S}(\tau_2 : \kappa_2)$
   - by TFST, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \cdot 1 : \mathrm{S}(\tau_1 : \kappa_1)$
   - by TQEXT-SING*, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \cdot 1 \equiv \tau_1 : \mathrm{S}(\tau_1 : \kappa_1)$
4. TQSND-BETA*: analogous.
5. TQPAIR-ETA*:
   - w.l.o.g., $\alpha \notin \mathrm{FV}(\tau)$
   - by Validity, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by TEXT-SING*, $\Gamma \vdash \tau : \mathrm{S}(\tau \cdot 1 : \kappa_1) \times \mathrm{S}(\tau \cdot 2 : \kappa_2[\tau \cdot 1/\alpha])$
   - by TFST, $\Gamma \vdash \tau \cdot 1 : \mathrm{S}(\tau \cdot 1 : \kappa_1)$
   - by TSND, $\Gamma \vdash \tau \cdot 2 : \mathrm{S}(\tau \cdot 2 : \kappa_2[\tau \cdot 1/\alpha])$
   - by TPAIR, $\Gamma \vdash \langle \tau \cdot 1, \tau \cdot 2 \rangle : \mathrm{S}(\tau \cdot 1 : \kappa_1) \times \mathrm{S}(\tau \cdot 2 : \kappa_2[\tau \cdot 1/\alpha])$
   - by TQEXT-SING*, $\Gamma \vdash \langle \tau \cdot 1, \tau \cdot 2 \rangle \equiv \tau : \mathrm{S}(\tau \cdot 1 : \kappa_1) \times \mathrm{S}(\tau \cdot 2 : \kappa_2[\tau \cdot 1/\alpha])$

□

C.3. Algorithmic Formulations

C.3.1. Type Equivalence

Theorem 23 (Soundness of Algorithmic Kind and Type Comparison)**.**

1. *If* $\Gamma \triangleright \tau \Rrightarrow \pi$ *and* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \pi : \kappa$*.*
2. *If* $\Gamma \triangleright \pi \Rrightarrow \pi' \rightrightarrows \kappa'$ *and* $\Gamma \vdash \pi : \kappa$*, then* $\Gamma \vdash \pi \equiv \pi' : \kappa$*.*
3. *If* $\Gamma \triangleright \tau \Rrightarrow \tau' \lhd\!\!\lhd \kappa$ *and* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*
4. *If* $\Gamma \triangleright \tau \equiv \tau' \lhd\!\!\lhd \kappa$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*
5. *If* $\Gamma \triangleright \kappa \Rrightarrow \kappa'$ *and* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*
6. *If* $\Gamma \triangleright \kappa \equiv \kappa'$ *and* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$*, then* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*.*

Proof. See Stone & Harper [SH06], Corollary 4.7, Proposition 4.8, and Corollary 4.9. The type syntax of their language is extended as follows:

$$A, \sigma \quad ::= \quad \cdots \mid \mathrm{A}(M) \mid \mathrm{S}_{\mathrm{A}(N)}(M)$$

where our $\mathrm{S}_\Omega(\tau)$ maps to the existing $\mathrm{S}(M)$. Our type language can be encoded pragmatically by variables prebound in an initial environment Γ_0 that is contained in every environment we consider:

$$\begin{array}{rcl} \Psi & : & \Omega \\ (\rightarrow) & : & \Omega \rightarrow \Omega \rightarrow \Omega \\ (\times) & : & \Omega \rightarrow \Omega \rightarrow \Omega \\ \forall_\kappa & : & (\kappa \rightarrow \Omega) \rightarrow \Omega \\ \exists_\kappa & : & (\kappa \rightarrow \Omega) \rightarrow \Omega \end{array}$$

All relevant lemmata and propositions are easily adapted to cover the additional cases in the algorithms. In particular, abstraction kinds act like base kinds with respect to normalization. □

Theorem 24 (Completeness of Algorithmic Kind and Type Comparison)**.**

1. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \triangleright \tau \Rrightarrow \tau'' \leftleftarrows \kappa$ *and* $\Gamma \triangleright \tau' \Rrightarrow \tau'' \leftleftarrows \kappa$ *for some* τ''*.*
2. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$*.*
3. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \triangleright \kappa \Rrightarrow \kappa''$ *and* $\Gamma \triangleright \kappa' \Rrightarrow \kappa''$ *for some* κ''*.*
4. *If* $\Gamma \vdash \kappa \equiv \kappa' : \Box$*, then* $\Gamma \triangleright \kappa \equiv \kappa'$*.*

Proof. See Stone & Harper [SH06], Corollary 4.18. The proof builds upon most of the preceding definitions and lemmata in the paper, which are adapted as for the previous proof. The key to the completeness proof are the logical relations in Figure 10 of the paper, to which we have to add cases as follows:

- $\mathcal{A}$ ok $[\Delta]$ if
  - ...
  - Or, $\mathcal{A} = \mathrm{A}(\mathcal{N})$, and $\mathcal{N}$ in $\{b\}$ $[\Delta]$
  - Or, $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$, and $\mathcal{N}$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
- $\mathcal{M}$ in $\mathcal{A}$ $[\Delta]$ if

(2)
  - ...
  - Or, $\mathcal{A} = \mathrm{A}(\mathcal{N})$, and $\mathcal{M}$ in $\{b\}$ $[\Delta]$
  - Or, $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$, and $(\mathcal{M} \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$

Here, $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$ is pattern matching notation defined as for quantifiers in the paper, i.e. meaning $\mathcal{L} = \{L_i \mid i \in I\}$ and $\mathcal{N} = \{N_i \mid i \in I\}$ if $\mathcal{A} = \{\mathrm{S}_{\mathrm{A}(L_i)}(N_i) \mid i \in I\}$.

All respective lemmata and theorems proved by induction over the size of a type or a derivation have to be extended with new cases as follows.

Lemma 4.12

(1)
  - Case: $\mathcal{A}_2 = \mathrm{A}(\mathcal{N}_2)$ and $\mathcal{A}_1 = \mathrm{A}(\mathcal{N}_1)$ with $\mathcal{N}_1 \subseteq \mathcal{N}_2$.

  - by definition, $\mathcal{N}_2$ in $\{b\}$ $[\Delta]$
  - by induction (2), $\mathcal{N}_1$ in $\{b\}$ $[\Delta]$
  - by definition, $\mathcal{A}_1$ ok $[\Delta]$
- Case: $\mathcal{A}_2 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_2)}(\mathcal{N}_2)$ and $\mathcal{A}_1 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_1)}(\mathcal{N}_1)$ with $\mathcal{L}_1 \subseteq \mathcal{L}_2$ and $\mathcal{N}_1 \subseteq \mathcal{N}_2$.
  - by definition, $\mathcal{N}_2$ in $\mathrm{A}(\mathcal{L}_2)$ $[\Delta]$
  - by induction (2), $\mathcal{N}_1$ in $\mathrm{A}(\mathcal{L}_1)$ $[\Delta]$
  - by definition, $\mathcal{A}_1$ ok $[\Delta]$

(2)
- Case: $\mathcal{A}_2 = \mathrm{A}(\mathcal{N}_2)$ and $\mathcal{A}_1 = \mathrm{A}(\mathcal{N}_1)$ with $\mathcal{N}_1 \subseteq \mathcal{N}_2$.
  - by definition, $\mathcal{A}_2$ ok $[\Delta]$ and $\mathcal{M}_2$ in $\{b\}$ $[\Delta]$
  - by definition, $\mathcal{N}_2$ in $\{b\}$ $[\Delta]$
  - by induction, $\mathcal{M}_1$ in $\{b\}$ $[\Delta]$ and $\mathcal{N}_1$ in $\{b\}$ $[\Delta]$
  - by definition, $\mathcal{A}_1$ ok $[\Delta]$
  - by definition, $\mathcal{M}_1$ in $\mathcal{A}_1$ $[\Delta]$
- Case: $\mathcal{A}_2 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_2)}(\mathcal{N}_2)$ and $\mathcal{A}_1 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_1)}(\mathcal{N}_1)$ with $\mathcal{L}_1 \subseteq \mathcal{L}_2$ and $\mathcal{N}_1 \subseteq \mathcal{N}_2$.
  - by definition, $(\mathcal{M}_2 \cup \mathcal{N}_2)$ in $\mathrm{A}(\mathcal{L}_2)$ $[\Delta]$
  - by induction, $(\mathcal{M}_1 \cup \mathcal{N}_1)$ in $\mathrm{A}(\mathcal{L}_1)$ $[\Delta]$
  - by definition, $\mathcal{M}_1$ in $\mathcal{A}_1$ $[\Delta]$

(3)
- Case: $\mathcal{A}_1 = \mathrm{A}(\mathcal{N}_1)$ and $\mathcal{A}_2 = \mathrm{A}(\mathcal{N}_2)$ with $\mathcal{N}_1 \cup \mathcal{N}_2 \neq \emptyset$.
  - by definition, $\mathcal{N}_1$ in $\{b\}$ $[\Delta]$ and $\mathcal{N}_2$ in $\{b\}$ $[\Delta]$
  - by induction (5), $(\mathcal{N}_1 \cup \mathcal{N}_2)$ in $\{b\}$ $[\Delta]$
  - by definition, $(\mathcal{A}_1 \cup \mathcal{A}_2)$ ok $[\Delta]$
- Case: $\mathcal{A}_1 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_1)}(\mathcal{N}_1)$ and $\mathcal{A}_2 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_2)}(\mathcal{N}_2)$ with $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$ and $\mathcal{N}_1 \cap \mathcal{N}_2 \neq \emptyset$.
  - by definition, $\mathcal{N}_1$ in $\mathrm{A}(\mathcal{L}_1)$ $[\Delta]$ and $\mathcal{N}_2$ in $\mathrm{A}(\mathcal{L}_2)$ $[\Delta]$
  - by definition, $\mathrm{A}(\mathcal{L}_1)$ ok $[\Delta]$ and $\mathrm{A}(\mathcal{L}_2)$ ok $[\Delta]$
  - by induction, $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ ok $[\Delta]$
  - by induction (4), $\mathcal{N}_1$ in $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ $[\Delta]$ and $\mathcal{N}_2$ in $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ $[\Delta]$
  - by induction (5), $(\mathcal{N}_1 \cup \mathcal{N}_1)$ in $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ $[\Delta]$
  - by definition, $(\mathcal{A}_1 \cup \mathcal{A}_2)$ ok $[\Delta]$

(4)
- Case: $\mathcal{A}_1 = \mathrm{A}(\mathcal{N}_1)$ and $\mathcal{A}_2 = \mathrm{A}(\mathcal{N}_2)$ with $\mathcal{N}_1 \cup \mathcal{N}_2 \neq \emptyset$.
  - by definition, $\mathcal{A}_1$ ok $[\Delta]$ and $\mathcal{M}$ in $\{b\}$ $[\Delta]$
  - by definition, $\mathcal{N}_1$ in $\{b\}$ $[\Delta]$ and $\mathcal{N}_2$ in $\{b\}$ $[\Delta]$
  - by induction, $(\mathcal{N}_1 \cup \mathcal{N}_2)$ in $\{b\}$ $[\Delta]$
  - by induction (3), $(\mathcal{A}_1 \cup \mathcal{A}_2)$ ok $[\Delta]$
  - by definition, $\mathcal{M}$ in $(\mathcal{A}_1 \cup \mathcal{A}_2)$ $[\Delta]$
- Case: $\mathcal{A}_1 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_1)}(\mathcal{N}_1)$ and $\mathcal{A}_2 = \mathrm{S}_{\mathrm{A}(\mathcal{L}_2)}(\mathcal{N}_2)$ with $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$ and $\mathcal{N}_1 \cap \mathcal{N}_2 \neq \emptyset$.
  - by definition, $\mathcal{A}_1$ ok $[\Delta]$ and $(\mathcal{M} \cup \mathcal{N}_1)$ in $\mathrm{A}(\mathcal{L}_1)$ $[\Delta]$
  - by induction (3), $(\mathcal{A}_1 \cup \mathcal{A}_2)$ ok $[\Delta]$
  - by definition, $(\mathcal{N}_1 \cup \mathcal{N}_2)$ in $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ $[\Delta]$
  - by definition, $(\mathcal{M} \cup \mathcal{N}_1)$ in $\{b\}$ $[\Delta]$ and $(\mathcal{N}_1 \cup \mathcal{N}_2)$ in $\{b\}$ $[\Delta]$
  - by definition, all $M \in \mathcal{M} \cup \mathcal{N}_1 \cup \mathcal{N}_2$ have the same normal form
  - by definition, $(\mathcal{M} \cup \mathcal{N}_1 \cup \mathcal{N}_2)$ in $\{b\}$ $[\Delta]$
  - by definition, $(\mathcal{M} \cup \mathcal{N}_1 \cup \mathcal{N}_2)$ in $\mathrm{A}(\mathcal{L}_1 \cup \mathcal{L}_2)$ $[\Delta]$
  - by definition, $\mathcal{M}$ in $(\mathcal{A}_1 \cup \mathcal{A}_2)$ $[\Delta]$

(5)
- Case: $\mathcal{A} = \mathrm{A}(\mathcal{N})$.

  - by definition, $\mathcal{A}$ ok $[\Delta]$ and $\mathcal{M}_1$ in $\{b\}$ $[\Delta]$ and $\mathcal{M}_2$ in $\{b\}$ $[\Delta]$ and $\mathcal{N}$ in $\{b\}$ $[\Delta]$
  - by induction, $\mathcal{M}_1 \cup \mathcal{M}_2$ in $\{b\}$ $[\Delta]$
  - by definition, $\mathcal{M}_1 \cup \mathcal{M}_2$ in $\mathcal{A}$ $[\Delta]$
- Case: $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$.
  - by definition, $\mathcal{A}$ ok $[\Delta]$ and $(\mathcal{M}_1 \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$ and $(\mathcal{M}_2 \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - by definition, $(\mathcal{M}_1 \cup \mathcal{N})$ in $\{b\}$ $[\Delta]$ and $(\mathcal{M}_2 \cup \mathcal{N})$ in $\{b\}$ $[\Delta]$
  - by definition, all $M \in \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{N}$ have the same normal form
  - by definition, $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{N})$ in $\{b\}$ $[\Delta]$
  - by definition, $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - by definition, $(\mathcal{M}_1 \cup \mathcal{M}_2)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{N})}(\mathcal{L})$ $[\Delta]$

Lemma 4.13 (Head Expansion)

(2)
- Case: $\mathcal{A} = \mathrm{A}(\mathcal{N})$.
  - by definition, $\mathcal{A}$ ok $[\Delta]$ and $\mathcal{M}_2$ in $\{b\}$ $[\Delta]$
  - by induction, $(\mathcal{M}_1 \cup \mathcal{M}_2)$ in $\{b\}$ $[\Delta]$
  - by definition, $(\mathcal{M}_1 \cup \mathcal{M}_2)$ in $\mathcal{A}$ $[\Delta]$
- Case: $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$.
  - by definition, $(\mathcal{M}_2 \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - obviously, $\forall \theta \in \Delta, M_1 \in \mathcal{M}_1. \exists M_2 \in \mathcal{M}_2. \theta \triangleright M_1 \rightsquigarrow M_2$
  - by induction, $(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - by definition, $(\mathcal{M}_1 \cup \mathcal{M}_2)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$ $[\Delta]$

Lemma 4.14

(1)
- Case: $\mathcal{A} = \mathrm{A}(\mathcal{N})$.
  - by definition, $\mathcal{A}$ ok $[\Delta]$
  - by definition, $\mathcal{N}$ in $\{b\}$ $[\Delta]$
  - by induction (2), there exists unique normal form N of all terms in $\mathcal{N}$
  - put $B := \mathrm{A}(N)$
- Case: $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$.
  - by definition, $(\mathcal{M} \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - by induction (2), there exists unique normal form M of all terms in $\mathcal{M} \cup \mathcal{N}$
  - by definition, $\mathrm{A}(\mathcal{L})$ ok $[\Delta]$
  - by induction, there exists unique normal form $\mathrm{A}(L)$ of all terms in $\mathrm{A}(\mathcal{L})$
  - put $B := \mathrm{S}_{\mathrm{A}(L)}(N)$

(2)
- Case: $\mathcal{A} = \mathrm{A}(\mathcal{N})$.
  - by definition, $\mathcal{M}$ in $\{b\}$ $[\Delta]$
  - by induction, there exists unique normal form M of all terms in $\mathcal{M}$
  - by definition of normalization, normalization at abstraction type is the same as normalization at base type
- Case: $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$.
  - by definition, $(\mathcal{M} \cup \mathcal{N})$ in $\mathrm{A}(\mathcal{L})$ $[\Delta]$
  - by induction, there exists unique normal form M of all terms in $\mathcal{M} \cup \mathcal{N}$
  - by definition of normalization, normalization at singleton type is the same as normalization at abstraction type

(3)
- Case: $\mathcal{A} = \mathrm{A}(\mathcal{N})$. Analogous to the case $A = \{b\}$.
- Case: $\mathcal{A} = \mathrm{S}_{\mathrm{A}(\mathcal{L})}(\mathcal{N})$. Analogous to the case $A = \mathrm{S}(\mathcal{N})$.

Theorem 4.16 (Fundamental Theorem)

- Type Well-Formedness Rules: $\Gamma \vdash A$
  - Case: $\frac{\Gamma \vdash M : b}{\Gamma \vdash \mathrm{A}(M)}$ (cf. rule KABS)
    - by induction (4), $\mathcal{G}(M)$ in $\{b\}$ $[\Delta]$
    - by definition, $\mathcal{G}(\mathrm{S}(M))$ ok $[\Delta]$
  - Case: $\frac{\Gamma \vdash M : \mathrm{A}(N)}{\Gamma \vdash \mathrm{S}_{\mathrm{A}(N)}(M)}$ (cf. rule KSING)
    - by induction (4), $\mathcal{G}(M)$ in $\mathrm{A}(N)$ $[\Delta]$
    - by definition, $\mathcal{G}(\mathrm{A}(M))$ ok $[\Delta]$
- Subtyping Rules: $\Gamma \vdash A_1 \leq A_2$
  - Case: $\frac{\Gamma \vdash M : b}{\Gamma \vdash \mathrm{A}(M) \leq b}$ (cf. rule KSABS-LEFT)
    - by induction (4), $\mathcal{G}(M)$ in $\{b\}$ $[\Delta]$
  - Case: $\frac{\Gamma \vdash M_1 \equiv M_2 : b}{\Gamma \vdash \mathrm{A}(M_1) \leq \mathrm{A}(M_2)}$ (cf. rule KSABS)
    - by definition, $\mathcal{M}$ in $\{b\}$ $[\Delta]$
    - by induction (5), $(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\{b\}$ $[\Delta]$
    - by Lemma 4.12 (2), $\mathcal{G}(M_2)$ in $\{b\}$ $[\Delta]$
    - by definition, $\mathrm{A}(\mathcal{G}(M_2))$ ok $[\Delta]$
    - by definition, $\mathcal{M}$ in $\mathrm{A}(\mathcal{G}(M_2))$ $[\Delta]$
  - Case: $\frac{\Gamma \vdash M : \mathrm{A}(N)}{\Gamma \vdash \mathrm{S}_{\mathrm{A}(N)}(M) \leq \mathrm{A}(N)}$ (cf. rule KSSING-LEFT)
    - by induction (4), $\mathcal{G}(M)$ in $\mathrm{A}(N)$ $[\Delta]$
  - Case: $\frac{\Gamma \vdash M_1 \equiv M_2 : \mathrm{A}(N_1) \quad \Gamma \vdash \mathrm{A}(N_1) \equiv \mathrm{A}(N_2)}{\Gamma \vdash \mathrm{S}_{\mathrm{A}(N_1)}(M_1) \leq \mathrm{S}_{\mathrm{A}(N_2)}(M_2)}$ (cf. rule KSSING)
    - by definition, $(\mathcal{M} \cup \mathcal{G}(M_1))$ in $\mathrm{A}(\mathcal{G}(N_1))$ $[\Delta]$
    - by definition, $(\mathcal{M} \cup \mathcal{G}(M_1))$ in $\{b\}$ $[\Delta]$ and $\mathrm{A}(\mathcal{G}(N_1))$ ok $[\Delta]$
    - by inversion, $\Gamma \vdash N_1 \equiv N_2 : b$
    - by induction (5), $(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_1))$ $[\Delta]$ and $(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ in $\{b\}$ $[\Delta]$
    - by Lemma 4.12 (5), $(\mathcal{M} \cup \mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_1))$ $[\Delta]$
    - by definition, $\mathrm{A}(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ ok $[\Delta]$
    - by Lemma 4.12 (4), $(\mathcal{M} \cup \mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ $[\Delta]$
    - by Lemma 4.12 (2), $(\mathcal{M} \cup \cup\mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_2))$ $[\Delta]$
    - by definition, $\mathcal{M}$ in $\mathrm{S}_{\mathrm{A}(\mathcal{G}(N_2))}(\mathcal{G}(M_2))$ $[\Delta]$
- Type Equivalence Rules: $\Gamma \vdash A_1 \equiv A_2$
  - Case: $\frac{\Gamma \vdash M_1 \equiv M_2 : b}{\Gamma \vdash \mathrm{A}(M_1) \equiv \mathrm{A}(M_2)}$ (cf. rule KQABS)
    - by induction (5), $(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\{b\}$ $[\Delta]$
    - by definition, $\mathrm{A}(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ ok $[\Delta]$
  - Case: $\frac{\Gamma \vdash M_1 \equiv M_2 : \mathrm{A}(N_1) \quad \Gamma \vdash \mathrm{A}(N_1) \equiv \mathrm{A}(N_2)}{\Gamma \vdash \mathrm{S}_{\mathrm{A}(N_1)}(M_1) \equiv \mathrm{S}_{\mathrm{A}(N_2)}(M_2)}$ (cf. rule KQSING)
    - by inversion, $\Gamma \vdash N_1 \equiv N_2 : b$
    - by induction (5), $(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_1))$ $[\Delta]$ and $(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ in $\{b\}$ $[\Delta]$
    - by definition, $\mathrm{A}(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ ok $[\Delta]$
    - by Lemma 4.12 (4), $(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ in $\mathrm{A}(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))$ $[\Delta]$
    - by definition, $\mathrm{S}_{\mathrm{A}(\mathcal{G}(N_1) \cup \mathcal{G}(N_2))}(\mathcal{G}(M_1) \cup \mathcal{G}(M_2))$ ok $[\Delta]$

- Term Validity Rules: $\Gamma \vdash M : A$
  - Case: $\frac{\Gamma \vdash M : \mathrm{A}(N)}{\Gamma \vdash M : \mathrm{S}_{\mathrm{A}(N)}(M)}$ (cf. rule TEXT-SING)
    - by induction, $\mathcal{G}(M)$ in $\mathrm{A}(\mathcal{G}(N))$ $[\Delta]$
    - by definition, $\mathrm{A}(\mathcal{G}(N))$ ok $[\Delta]$
    - by definition, $\mathcal{G}(M)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{G}(N))}(\mathcal{G}(M))$ $[\Delta]$
- Term Equivalence Rules: $\Gamma \vdash M_1 \equiv M_2 : A$
  - Case: $\frac{\Gamma \vdash M_1 : \mathrm{S}_{\mathrm{A}(L)}(N) \quad \Gamma \vdash M_2 : \mathrm{S}_{\mathrm{A}(L)}(N)}{\Gamma \vdash M_1 \equiv M_2 : \mathrm{S}_{\mathrm{A}(L)}(N)}$ (cf. rule TQEXT-SING)
    - by induction (4), $\mathcal{G}(M_1)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{G}(L))}(\mathcal{G}(N))$ $[\Delta]$ and $\mathcal{G}(M_2)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{G}(L))}(\mathcal{G}(N))$ $[\Delta]$
    - by definition, $\mathcal{G}(M_1) \cup \mathcal{G}(N)$ in $\mathrm{A}(\mathcal{G}(L))$ $[\Delta]$ and $\mathcal{G}(M_2) \cup \mathcal{G}(N)$ in $\mathrm{A}(\mathcal{G}(L))$ $[\Delta]$
    - by Lemma 4.12 (5), $\mathcal{G}(M_1) \cup \mathcal{G}(M_2) \cup \mathcal{G}(N)$ in $\mathrm{A}(\mathcal{G}(L))$ $[\Delta]$
    - by definition, $\mathcal{G}(M_1) \cup \mathcal{G}(M_2)$ in $\mathrm{S}_{\mathrm{A}(\mathcal{G}(L))}(\mathcal{G}(N))$ $[\Delta]$

□

With soundness and completeness, we can establish important inversion principles and statements about the shape of types, which are needed for later proofs:

Proposition 25 (Type Equivalence Inversion)**.**

1. *If* $\Gamma \vdash \alpha \equiv \alpha : \kappa$, *then* $\Gamma \vdash \alpha : \Gamma(\alpha)$.
2. *If* $\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2' : \kappa$, *then* $\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega$ *and* $\Gamma \vdash \tau_2 \equiv \tau_2' : \Omega$.
3. *If* $\Gamma \vdash \tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2' : \kappa$, *then* $\Gamma \vdash \tau_1 \equiv \tau_1' : \Omega$ *and* $\Gamma \vdash \tau_2 \equiv \tau_2' : \Omega$.
4. *If* $\Gamma \vdash \forall \alpha{:}\kappa_1.\tau_2 \equiv \forall \alpha{:}\kappa_1'.\tau_2' : \kappa$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_2' : \Omega$.
5. *If* $\Gamma \vdash \exists \alpha{:}\kappa_1.\tau_2 \equiv \exists \alpha{:}\kappa_1'.\tau_2' : \kappa$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_2' : \Omega$.
6. *If* $\Gamma \vdash \lambda \alpha{:}\kappa_1.\tau_2 \equiv \lambda \alpha{:}\kappa_1'.\tau_2' : \kappa$, *then* $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_2' : \kappa_2$.
7. *If* $\Gamma \vdash \tau_1\, \tau_2 \equiv \tau_1'\, \tau_2' : \kappa$, *then* $\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1$ *and* $\Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2$.
8. *If* $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau_1', \tau_2' \rangle : \kappa$, *then* $\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1$ *and* $\Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2$.
9. *If* $\Gamma \vdash \tau \cdot 1 \equiv \tau' \cdot 1 : \kappa$, *then* $\Gamma \vdash \tau \equiv \tau' : \kappa'$.
10. *If* $\Gamma \vdash \tau \cdot 2 \equiv \tau' \cdot 2 : \kappa$, *then* $\Gamma \vdash \tau \equiv \tau' : \kappa'$.

Proof. By Completeness of Type Comparison and induction of the Normalization algorithm. □

Proposition 26 (Shape Consistency of Type Equivalence)**.**
Let $\Gamma \vdash \pi \equiv \pi' : \Omega$ *and* $\Gamma \rhd \pi \Rrightarrow \pi$ *and* $\Gamma \rhd \pi' \Rrightarrow \pi'$.

1. *If* $\pi = \alpha$, *then* $\pi' = \alpha$.
2. *If* $\pi = \Psi$, *then* $\pi' = \Psi$.
3. *If* $\pi = \tau_1 \to \tau_2$, *then* $\pi' = \tau_1' \to \tau_2'$.
4. *If* $\pi = \tau_1 \times \tau_2$, *then* $\pi' = \tau_1' \times \tau_2'$.
5. *If* $\pi = \forall \alpha{:}\kappa_1.\tau_2$, *then* $\pi' = \forall \alpha{:}\kappa_1'.\tau_2'$.

6. *If* $\pi = \exists\alpha{:}\kappa_1.\tau_2$, *then* $\pi' = \exists\alpha{:}\kappa_1'.\tau_2'$.

Proof. By Completeness of Type Comparison and induction on the Normalization algorithm. □

Proposition 27 (Canonical Types)**.** *Let* $\Gamma \triangleright \tau \Rrightarrow \tau$ *and* $\Gamma \vdash \tau : \kappa'$ *and* $\Gamma \vdash \kappa' \leq \kappa : \Box$.

1. *If* $\kappa = \Omega$, *then* $\tau = \pi$.
2. *If* $\kappa = \mathrm{A}(\tau')$, *then* $\tau = \pi$ *and* $\tau = \chi$.
3. *If* $\kappa = \mathrm{S}_\Omega(\tau')$, *then* $\tau = \pi$.
4. *If* $\kappa = \mathrm{S}_{\mathrm{A}(\tau')}(\tau'')$, *then* $\tau = \pi$ *and* $\tau = \chi$.
5. *If* $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$, *then* $\tau = \chi$.
6. *If* $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$, *then* $\tau = \chi$.

Proof. By simultaneous induction on the derivation. □

C.3.2. Subkinding

Theorem 28 (Soundness of Algorithmic Kind Matching)**.**
If $\Gamma \triangleright \kappa \leq \kappa'$ *and* $\Gamma \vdash \kappa : \Box$ *and* $\Gamma \vdash \kappa' : \Box$, *then* $\Gamma \vdash \kappa \leq \kappa' : \Box$.

Proof. By easy induction on the derivation of the first premise, using Soundness of Type Comparison. □

The inverse direction of correctness proceeds similarly:

Theorem 29 (Completeness of Algorithmic Kind Matching)**.**
If $\Gamma \vdash \kappa \leq \kappa' : \Box$, *then* $\Gamma \triangleright \kappa \leq \kappa'$.

Proof. By easy induction on the derivation, using Completeness of Type Comparison.

case KSOMEGA:

- by rule, $\Gamma \triangleright \Omega \leq \Omega$

case KSABS-LEFT:

- by rule, $\Gamma \triangleright \mathrm{A}(\tau) \leq \Omega$

case KSSING-LEFT:

- by inversion, $\Gamma \vdash \hat\kappa \leq \hat\kappa' : \Box$
- by induction, $\Gamma \triangleright \hat\kappa \leq \hat\kappa'$
- by rule, $\Gamma \triangleright \mathrm{S}_{\hat\kappa}(\tau) \leq \hat\kappa'$

case KSABS:

- by inversion, $\Gamma \vdash \tau \equiv \tau' : \Omega$
- by Validity, $\Gamma \vdash \Omega : \Box$
- by Completeness of Type Comparison, $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \Omega$
- by rule, $\Gamma \triangleright \mathrm{A}(\tau) \leq \mathrm{A}(\tau')$

case KSSING:

- by inversion, $\Gamma \vdash \tau \equiv \tau' : \hat\kappa'$ and $\Gamma \vdash \hat\kappa \leq \hat\kappa' : \Box$

- by Validity, $\Gamma \vdash \hat{\kappa} : \Box$
- by Completeness of Type Comparison, $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \hat{\kappa}$
- by induction, $\Gamma \triangleright \hat{\kappa} \leq \hat{\kappa}'$
- by rule, $\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) \leq \mathrm{S}_{\hat{\kappa}'}(\tau')$

case KSPI:

- by inversion, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2 \leq \kappa_2' : \Box$
- by induction, $\Gamma \triangleright \kappa_1' \leq \kappa_1$ and $\Gamma, \alpha{:}\kappa_1' \triangleright \kappa_2 \leq \kappa_2'$
- by rule, $\Gamma \triangleright \Pi\alpha{:}\kappa_1.\kappa_2 \leq \Pi\alpha{:}\kappa_1'.\kappa_2'$

case KSSIGMA: analogous

□

A simple consequence is the following:

Proposition 30 (Shape Consistency of Kind Inclusion)**.** *Let* $\Gamma \vdash \kappa \leq \kappa' : \Box$.

1. *If* $\kappa' = \Omega$, *then* $\kappa = \Omega$ *or* $\kappa = \mathrm{A}(\tau)$ *or* $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$.
2. *If* $\kappa' = \mathrm{A}(\tau')$, *then* $\kappa = \mathrm{A}(\tau)$ *or* $\kappa = \mathrm{S}_{\mathrm{A}(\tau'')}(\tau)$.
3. *If* $\kappa' = \mathrm{S}_{\hat{\kappa}'}(\tau')$, *then* $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$.
4. *If* $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$, *then* $\kappa' = \Pi\alpha{:}\kappa_1.\kappa_2$.
5. *If* $\kappa' = \Sigma\alpha{:}\kappa_1'.\kappa_2'$, *then* $\kappa' = \Sigma\alpha{:}\kappa_1.\kappa_2$.

Proof. By Completeness of Kind Matching and induction on the matching algorithm. □

C.3.3. Kind Synthesis

Theorem 31 (Soundness of Algorithmic Kind Synthesis)**.**

1. *If* $\Gamma \triangleright \kappa : \Box$ *and* $\Gamma \vdash \Box$, *then* $\Gamma \vdash \kappa : \Box$.
2. *If* $\Gamma \triangleright \tau \rightrightarrows \kappa$ *and* $\Gamma \vdash \Box$, *then* $\Gamma \vdash \tau : \kappa$.
3. *If* $\Gamma \triangleright \tau \leftleftarrows \kappa$ *and* $\Gamma \vdash \kappa : \Box$, *then* $\Gamma \vdash \tau : \kappa$.

Proof. By simultaneous induction on the derivation of the first premise:

1. case $\kappa = \Omega$:
   - by KOMEGA, $\Gamma \vdash \Omega : \Box$

   case $\kappa = \mathrm{A}(\tau)$:
   - by inversion, $\Gamma \triangleright \tau \leftleftarrows \Omega$
   - by induction (3), $\Gamma \vdash \tau : \Omega$
   - by KABS, $\Gamma \vdash \mathrm{A}(\tau) : \Box$

   case $\kappa = \mathrm{S}_{\hat{\kappa}}(\tau)$:
   - by inversion, $\Gamma \triangleright \hat{\kappa} : \Box$ and $\Gamma \triangleright \tau \leftleftarrows \hat{\kappa}$
   - by induction, $\Gamma \vdash \hat{\kappa} : \Box$
   - by induction (3), $\Gamma \vdash \tau : \hat{\kappa}$
   - by KSING, $\Gamma \vdash \mathrm{S}_{\hat{\kappa}}(\tau) : \Box$

   case $\kappa = \Pi\alpha{:}\kappa_1.\kappa_2$:

- by inversion, $\Gamma \triangleright \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 : \Box$
- by induction, $\Gamma \vdash \kappa_1 : \Box$
- by NTYPE, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$

case $\kappa = \Sigma\alpha{:}\kappa_1.\kappa_2$: analogous

2. case $\tau = \alpha$:
   - by inversion, $\alpha \in \mathrm{Dom}(\Gamma)$
   - by TVAR, $\Gamma \vdash \alpha : \Gamma(\alpha)$
   - by TEXT-SING*, $\Gamma \vdash \alpha : \mathrm{S}(\alpha : \Gamma(\alpha))$

   case $\tau = \Psi$:
   - by TPSI, $\Gamma \vdash \Psi : \Omega$
   - by TEXT-SING, $\Gamma \vdash \Psi : \mathrm{S}_\Omega(\Psi)$

   case $\tau = \tau_1 \to \tau_2$:
   - by inversion, $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$
   - by KOMEGA, $\Gamma \vdash \Omega : \Box$
   - by induction (3), $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$
   - by TARROW, $\Gamma \vdash \tau_1 \to \tau_2 : \Omega$
   - by TEXT-SING, $\Gamma \vdash \tau_1 \to \tau_2 : \mathrm{S}_\Omega(\tau_1 \to \tau_2)$

   case $\tau = \tau_1 \times \tau_2$: analogous

   case $\tau = \forall\alpha{:}\kappa_1.\tau_2$:
   - by inversion, $\Gamma \triangleright \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \leftleftarrows \Omega$
   - by induction (1), $\Gamma \vdash \kappa_1 : \Box$
   - by NTYPE, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
   - by KOMEGA, $\Gamma, \alpha{:}\kappa_1 \vdash \Omega : \Box$
   - by induction (3), $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$
   - by TUNIV, $\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 : \Omega$
   - by TEXT-SING, $\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 : \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2)$

   case $\tau = \exists\alpha{:}\kappa_1.\tau_2$: analogous

   case $\tau = \lambda\alpha{:}\kappa_1.\tau_2$:
   - by inversion, $\Gamma \triangleright \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \rightrightarrows \kappa_2$
   - by induction (1), $\Gamma \vdash \kappa_1 : \Box$
   - by NTYPE, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
   - by induction, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2$
   - by TLAMBDA, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 : \Pi\alpha{:}\kappa_1.\kappa_2$

   case $\tau = \tau_1\,\tau_2$:
   - by inversion, $\Gamma \triangleright \tau_1 \rightrightarrows \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \triangleright \tau_2 \leftleftarrows \kappa_1$
   - by induction, $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$
   - by Validity, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$
   - by inverting KPI, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
   - by induction (3), $\Gamma \vdash \tau_2 : \kappa_1$
   - by TAPP, $\Gamma \vdash \tau_1\,\tau_2 : \kappa_2[\tau_2/\alpha]$

   case $\tau = \langle\tau_1, \tau_2\rangle$:
   - by inversion, $\Gamma \triangleright \tau_1 \rightrightarrows \kappa_1$ and $\Gamma \triangleright \tau_2 \rightrightarrows \kappa_2$

- by induction, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2$
- by Validity, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma \vdash \kappa_2 : \Box$
- w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma)$
- by Ntype, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
- by Ksigma, $\Gamma \vdash \Sigma\alpha{:}\kappa_1.\kappa_2 : \Box$
- by Tpair, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \Sigma\alpha{:}\kappa_1.\kappa_2$

case $\tau = \tau{\cdot}1$:

- by inversion, $\Gamma \triangleright \tau \rightrightarrows \Sigma\alpha{:}\kappa_1.\kappa_2$
- by induction, $\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2$
- by Tfst, $\Gamma \vdash \tau{\cdot}1 : \kappa_1$

case $\tau = \tau{\cdot}2$: analogous

3. case $\Gamma \triangleright \tau \leftleftarrows \kappa$:
   - by inversion, $\Gamma \triangleright \tau \rightrightarrows \kappa'$ and $\Gamma \triangleright \kappa' \leq \kappa$
   - by Environment Validity, $\Gamma \vdash \Box$
   - by induction (2), $\Gamma \vdash \tau : \kappa'$
   - by Validity, $\Gamma \vdash \kappa' : \Box$
   - by Soundness of Kind Matching, $\Gamma \vdash \kappa' \leq \kappa : \Box$
   - by Tsub, $\Gamma \vdash \tau : \kappa$

□

Theorem 32 (Completeness of Algorithmic Kind Synthesis)**.**

1. *If* $\Gamma \vdash \kappa : \Box$*, then* $\Gamma \triangleright \kappa : \Box$*.*
2. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \triangleright \tau \rightrightarrows \kappa'$ *and* $\Gamma \vdash \kappa' \leq \mathrm{S}(\tau : \kappa) : \Box$*.*
3. *If* $\Gamma \vdash \tau : \kappa$*, then* $\Gamma \triangleright \tau \leftleftarrows \kappa$*.*

Proof. By simultaneous induction on the derivation:

1. case Komega: directly

   case Kabs:
   - by inversion, $\Gamma \vdash \tau : \Omega$
   - by induction (3), $\Gamma \triangleright \tau \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \mathrm{A}(\tau) : \Box$

   case Ksing:
   - by inversion, $\Gamma \vdash \tau : \hat{\kappa}$
   - by Validity, $\Gamma \vdash \hat{\kappa} : \Box$
   - by induction, $\Gamma \triangleright \hat{\kappa} : \Box$
   - by induction (3), $\Gamma \triangleright \tau \leftleftarrows \hat{\kappa}$
   - by rule, $\Gamma \triangleright \mathrm{S}_{\hat{\kappa}}(\tau) : \Box$

   case Kpi:
   - by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2 : \Box$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
   - by induction, $\Gamma \triangleright \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \kappa_2 : \Box$
   - by rule, $\Gamma \triangleright \Pi\alpha{:}\kappa_1.\kappa_2 : \Box$

   case Ksigma: analogous

2. case TVAR:
   - by inversion, $\kappa = \Gamma(\alpha)$
   - by rule, $\Gamma \triangleright \alpha \rightrightarrows \mathrm{S}(\alpha : \Gamma(\alpha))$
   - by Reflexivity, $\Gamma \vdash \alpha \equiv \alpha : \Gamma(\alpha)$
   - by Validity, $\Gamma \vdash \Gamma(\alpha) : \Box$
   - by Reflexivity, $\Gamma \vdash \Gamma(\alpha) \leq \Gamma(\alpha) : \Box$
   - by KSSING*, $\Gamma \vdash \mathrm{S}(\alpha : \Gamma(\alpha)) \leq \mathrm{S}(\alpha : \Gamma(\alpha)) : \Box$

   case TPSI:
   - by rule, $\Gamma \triangleright \Psi \rightrightarrows \mathrm{S}_\Omega(\Psi)$
   - by TEXT-SING, $\Gamma \vdash \mathrm{S}_\Omega(\Psi) : \Box$
   - by Reflexivity, $\Gamma \vdash \mathrm{S}_\Omega(\Psi) \leq \mathrm{S}_\Omega(\Psi) : \Box$

   case TARROW:
   - by inversion, $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_1 : \Omega$
   - by induction (3), $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \tau_1 \to \tau_2 \rightrightarrows \mathrm{S}_\Omega(\tau_1 \to \tau_2)$
   - by TEXT-SING, $\Gamma \vdash \mathrm{S}_\Omega(\tau_1 \to \tau_2) : \Box$
   - by Reflexivity, $\Gamma \vdash \mathrm{S}_\Omega(\tau_1 \to \tau_2) \leq \mathrm{S}_\Omega(\tau_1 \to \tau_2) : \Box$

   case TTIMES: analogous

   case TUNIV:
   - by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
   - by induction (1), $\Gamma \triangleright \kappa_1 : \Box$
   - by induction (3), $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \forall\alpha{:}\kappa_1.\tau_2 \rightrightarrows \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2)$
   - by TEXT-SING, $\Gamma \vdash \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2) : \Box$
   - by Reflexivity, $\Gamma \vdash \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2) \leq \mathrm{S}_\Omega(\forall\alpha{:}\kappa_1.\tau_2) : \Box$

   case TEXIST: analogous

   case TLAMBDA:
   - by inversion, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \kappa_2$
   - by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$
   - by induction (1), $\Gamma \triangleright \kappa_1 : \Box$
   - by induction, $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \rightrightarrows \kappa_2'$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' \leq \mathrm{S}(\tau_2 : \kappa_2) : \Box$
   - by rule, $\Gamma \triangleright \lambda\alpha{:}\kappa_1.\tau_2 \rightrightarrows \Pi\alpha{:}\kappa_1.\kappa_2'$
   - by Reflexivity, $\Gamma \vdash \kappa_1 \leq \kappa_1 : \Box$
   - by Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' : \Box$
   - by KPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2' : \Box$
   - by KSPI, $\Gamma \vdash \Pi\alpha{:}\kappa_1.\kappa_2' \leq \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau_2 : \kappa_2) : \Box$

   case TAPP:
   - by inversion, $\Gamma \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_1$ and $\kappa = \kappa_2[\tau_2/\alpha]$
   - by induction, $\Gamma \triangleright \tau_1 \rightrightarrows \kappa'$ and $\Gamma \vdash \kappa' \leq \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau_1\,\alpha : \kappa_2) : \Box$
   - by Kind Inclusion Inversion, $\kappa' = \Pi\alpha{:}\kappa_1'.\kappa_2'$ and $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2' \leq \mathrm{S}(\tau_1\,\alpha : \kappa_2) : \Box$
   - by TSUB, $\Gamma \vdash \tau_2 : \kappa_1'$
   - by induction (3), $\Gamma \triangleright \tau_2 \leftleftarrows \kappa_1'$
   - by rule, $\Gamma \triangleright \tau_1\,\tau_2 \rightrightarrows \kappa_2'[\tau_2/\alpha]$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa_1 \vdash \Box$

- obviously, $\Gamma \vdash [\tau_2/\alpha] : \Gamma, \alpha{:}\kappa_1$
- by Substitutability, $\Gamma \vdash \kappa_2'[\tau_2/\alpha] \leq \mathrm{S}(\tau_1\ \alpha : \kappa_2)[\tau_2/\alpha] : \Box$
- by TVAR, $\Gamma, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$
- by Weakening, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_1 : \Pi\alpha{:}\kappa_1.\kappa_2$
- by TAPP, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_1\ \alpha : \kappa_2$
- by KSSING-LEFT*, $\Gamma, \alpha{:}\kappa_1 \vdash \mathrm{S}(\tau_1\ \alpha : \kappa_2) \leq \kappa_2 : \Box$
- by Substitutability, $\Gamma \vdash \mathrm{S}(\tau_1\ \alpha : \kappa_2)[\tau_2/\alpha] \leq \kappa_2[\tau_2/\alpha] : \Box$
- by Transitivity, $\Gamma \vdash \kappa_2'[\tau_2/\alpha] \leq \kappa_2[\tau_2/\alpha] : \Box$

case TPAIR:

- by inversion, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2[\tau_1/\alpha]$
- by induction, $\Gamma \triangleright \tau_1 \rightrightarrows \kappa_1'$ and $\Gamma \vdash \kappa_1' \leq \mathrm{S}(\tau_1 : \kappa_1) : \Box$
  and $\Gamma \triangleright \tau_2 \rightrightarrows \kappa_2'$ and $\Gamma \vdash \kappa_2' \leq \mathrm{S}(\tau_2 : \kappa_2[\tau_1/\alpha]) : \Box$
- by rule, $\Gamma \triangleright \langle \tau_1, \tau_2 \rangle \rightrightarrows \kappa_1' \times \kappa_2'$
- by Validity, $\Gamma \vdash \kappa_1' : \Box$
- w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(\tau)$
- by NTYPE, $\Gamma, \alpha{:}\kappa_1' \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2' \leq \mathrm{S}(\tau_2 : \kappa_2[\tau_1/\alpha]) : \Box$
- by Validity and KSIGMA, $\Gamma \vdash \mathrm{S}(\tau_1 : \kappa_1) \times \mathrm{S}(\tau_2 : \kappa_2[\tau_1/\alpha]) : \Box$
- by KSSIGMA, $\Gamma \vdash \kappa_1' \times \kappa_2' \leq \mathrm{S}(\tau_1 : \kappa_1) \times \mathrm{S}(\tau_2 : \kappa_2[\tau_1/\alpha]) : \Box$

case TFST:

- by inversion, $\Gamma \vdash \tau : \Sigma\alpha{:}\kappa_1.\kappa_2$
- by induction, $\Gamma \triangleright \tau \rightrightarrows \kappa$ and $\Gamma \vdash \kappa \leq \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$
- by Kind Inclusion Inversion, $\kappa = \Sigma\alpha{:}\kappa_1'.\kappa_2'$ and $\Gamma \vdash \kappa_1' \leq \mathrm{S}(\tau{\cdot}1 : \kappa_1) : \Box$
- by rule, $\Gamma \triangleright \tau{\cdot}1 \rightrightarrows \kappa_1'$

case TSND: similarly

case TEXT-SING:

- by inversion, $\Gamma \vdash \tau : \hat{\kappa}$
- by induction, $\Gamma \triangleright \tau \rightrightarrows \kappa$ and $\Gamma \vdash \kappa \leq \mathrm{S}_{\hat{\kappa}}(\tau) : \Box$

case TEXT-PI:

- by inversion, $\Gamma \vdash \tau : \Pi\alpha{:}\kappa_1.\kappa_2'$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau\ \alpha : \kappa_2$
- by induction, $\Gamma \triangleright \tau \rightrightarrows \kappa$ and $\Gamma \vdash \kappa \leq \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\ \alpha : \kappa_2') : \Box$
  and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau\ \alpha \rightrightarrows \kappa_2''$ and $\Gamma, \alpha{:}\kappa_1 \vdash \kappa_2'' \leq \mathrm{S}(\tau\ \alpha : \kappa_2) : \Box$
- by inverting $\Gamma, \alpha{:}\kappa_1 \triangleright \tau\ \alpha \rightrightarrows \kappa_2''$, we know that $\kappa = \Pi\alpha{:}\kappa_1'.\kappa_2''$
- by inverting KSPI, $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$
- by KSPI, $\Gamma \vdash \kappa \leq \Pi\alpha{:}\kappa_1.\mathrm{S}(\tau\ \alpha : \kappa_2) : \Box$

case TEXT-SIGMA:

- by inversion, $\Gamma \vdash \tau{\cdot}1 : \kappa_1$ and $\Gamma \vdash \tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]$
- by induction, $\Gamma \triangleright \tau{\cdot}1 \rightrightarrows \kappa_1'$ and $\Gamma \vdash \kappa_1' \leq \mathrm{S}(\tau{\cdot}1 : \kappa_1) : \Box$
  and $\Gamma \triangleright \tau{\cdot}2 \rightrightarrows \kappa_2'$ and $\Gamma \vdash \kappa_2' \leq \mathrm{S}(\tau{\cdot}2 : \kappa_2)[\tau{\cdot}1/\alpha] : \Box$
- by inverting $\Gamma \triangleright \tau{\cdot}1 \rightrightarrows \kappa_1'$, we know that $\Gamma \triangleright \tau \rightrightarrows \Sigma\alpha{:}\kappa_1'.\kappa_2''$
- by inverting $\Gamma \triangleright \tau{\cdot}2 \rightrightarrows \kappa_2'$, we further know that $\kappa_2' = \kappa_2''[\tau{\cdot}1/\alpha]$
- by Validity, $\Gamma \vdash \kappa_1' : \Box$
- w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(\tau)$
- by NTYPE, $\Gamma, \alpha{:}\kappa_1' \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \kappa_2' \leq \mathrm{S}(\tau{\cdot}2 : \kappa_2)[\tau{\cdot}1/\alpha] : \Box$
- by Validity and KSIGMA, $\Gamma \vdash \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$
- by KSSIGMA, $\Gamma \vdash \Sigma\alpha{:}\kappa_1'.\kappa_2'' \leq \mathrm{S}(\tau{\cdot}1 : \kappa_1) \times \mathrm{S}(\tau{\cdot}2 : \kappa_2[\tau{\cdot}1/\alpha]) : \Box$

case TSUB:
- by inversion, $\Gamma \vdash \tau : \kappa'$ and $\Gamma \vdash \kappa' \leq \kappa : \Box$
- by induction, $\Gamma \triangleright \tau \rightrightarrows \kappa''$ and $\Gamma \vdash \kappa'' \leq \mathrm{S}(\tau : \kappa') : \Box$
- by Reflexivity, $\Gamma \vdash \tau \equiv \tau : \kappa'$
- by KSSING*, $\Gamma \vdash \mathrm{S}(\tau : \kappa') \leq \mathrm{S}(\tau : \kappa) : \Box$
- by Transitivity, $\Gamma \vdash \kappa'' \leq \mathrm{S}(\tau : \kappa) : \Box$

3.
- by induction (2), $\Gamma \triangleright \tau \rightrightarrows \kappa'$ and $\Gamma \vdash \kappa' \leq \mathrm{S}(\tau : \kappa) : \Box$
- by KSSING-LEFT, $\Gamma \vdash \mathrm{S}(\tau : \kappa) \leq \kappa : \Box$
- by Transitivity, $\Gamma \vdash \kappa' \leq \kappa : \Box$
- by Completeness of Kind Matching, $\Gamma \triangleright \kappa' \leq \kappa$
- by rule, $\Gamma \triangleright \tau \leftleftarrows \kappa$

□

C.3.4. Subtyping

Theorem 33 (Soundness of Algorithmic Type Matching)**.**

1. *If* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, then* $\Gamma \vdash \tau \leq \tau' : \kappa$*.*
2. *If* $\Gamma \triangleright \pi \sqsubseteq \pi'$ *and* $\Gamma \vdash \pi : \Omega$ *and* $\Gamma \vdash \pi' : \Omega$*, then* $\Gamma \vdash \pi \leq \pi' : \Omega$*.*

Proof. By simultaneous induction on the derivation:

1. case $\kappa = \Omega$:
   - bz inversion, $\Gamma \triangleright \tau \Rrightarrow \pi \leftleftarrows \Omega$ and $\Gamma \triangleright \tau' \Rrightarrow \pi' \leftleftarrows \Omega$ and $\Gamma \triangleright \pi \sqsubseteq \pi'$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$ and $\Gamma \vdash \tau' \equiv \pi' : \Omega$
   - by Validity, $\Gamma \vdash \pi : \Omega$ and $\Gamma \vdash \pi' : \Omega$
   - by induction (2), $\Gamma \vdash \pi \leq \pi' : \Omega$
   - by Symmetry, TSEQUIV, and Transitivity, $\Gamma \vdash \tau \leq \tau' : \Omega$

   case $\kappa \neq \Omega$:
   - by inversion, $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \tau' : \kappa$

2. case $\pi = \chi$ and $\pi' = \chi'$:
   - bz inversion, $\Gamma \triangleright \chi \equiv \chi' \leftleftarrows \Omega$
   - by Soundness of Type Comparison, $\Gamma \vdash \chi \equiv \chi' : \Omega$

   case $\pi = \tau_1 \to \tau_2$ and $\pi' = \tau_1' \to \tau_2'$:
   - by inversion, $\Gamma \triangleright \tau_1' \leq \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leq \tau_2' \leftleftarrows \Omega$
   - by Type Validity Inversion, $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma \vdash \tau_1' : \Omega$ and $\Gamma \vdash \tau_2' : \Omega$
   - by induction (1), $\Gamma \vdash \tau_1' \leq \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 \leq \tau_2' : \Omega$
   - by TSARROW, $\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' : \Omega$

   case $\pi = \tau_1 \times \tau_2$ and $\pi' = \tau_1' \times \tau_2'$: analogous

   case $\pi = \forall \alpha{:}\kappa.\tau$ and $\pi' = \forall \alpha{:}\kappa'.\tau'$:
   - by inversion, $\Gamma \triangleright \kappa' \leq \kappa$ and $\Gamma, \alpha{:}\kappa' \triangleright \tau \leq \tau' \leftleftarrows \Omega$
   - by Type Validity Inversion, $\Gamma, \alpha{:}\kappa \vdash \tau : \Omega$ and $\Gamma, \alpha{:}\kappa' \vdash \tau' : \Omega$
   - by Environment Validity, $\Gamma \vdash \kappa : \Box$ and $\Gamma \vdash \kappa' : \Box$
   - by Soundness of Kind Matching, $\Gamma \vdash \kappa' \leq \kappa : \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa' \vdash \tau : \Omega$

- by induction (1), $\Gamma, \alpha{:}\kappa' \vdash \tau \leq \tau' : \Omega$
- by TSUNIV, $\Gamma \vdash \forall\alpha{:}\kappa.\tau \leq \forall\alpha{:}\kappa'.\tau' : \Omega$

case $\pi = \exists\alpha{:}\kappa.\tau$ and $\pi' = \exists\alpha{:}\kappa'.\tau'$: analogous

□

For completeness, we first show that Type Matching is complete for equivalent types, which can be proved directly:

Theorem 34 (Completeness of Algorithmic Type Matching with respect to Equivalence)**.**

1. *If* $\Gamma \triangleright \tau \Rrightarrow \tau'' \leftleftarrows \kappa$ *and* $\Gamma \triangleright \tau' \Rrightarrow \tau'' \leftleftarrows \kappa$ *with* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$*.*
2. *If* $\Gamma \triangleright \pi \Rrightarrow \pi'' \rightrightarrows \kappa$ *and* $\Gamma \triangleright \pi' \Rrightarrow \pi'' \rightrightarrows \kappa'$ *with* $\Gamma \vdash \pi : \Omega$ *and* $\Gamma \vdash \pi' : \Omega$*, then* $\Gamma \triangleright \pi \sqsubseteq \pi'$*.*
3. *If* $\Gamma \vdash \tau \equiv \tau' : \kappa$*, then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$*.*
4. *If* $\Gamma \vdash \pi \equiv \pi' : \Omega$ *where* π, π' *in weak-head normal form with respect to* Γ*, then* $\Gamma \triangleright \pi \sqsubseteq \pi'$*.*

Proof.

1. By induction on the combined length of the reduction sequences, simultaneously with (2):

   case $\kappa = \Omega$:
   - by inverting Normalization, $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$ and $\Gamma \triangleright \pi \Rrightarrow \tau'' \rightrightarrows \kappa$ and $\Gamma \triangleright \pi \Rrightarrow \tau'' \rightrightarrows \kappa'$ and $\tau'' = \pi''$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$ and $\Gamma \vdash \tau' \equiv \pi' : \Omega$
   - by Validity, $\Gamma \vdash \pi : \Omega$ and $\Gamma \vdash \pi' : \Omega$
   - by induction (2), $\Gamma \triangleright \pi \sqsubseteq \pi'$
   - by rule, $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$

   case $\kappa \neq \Omega$:
   - by definition of Type Comparison, $\Gamma \triangleright \tau \equiv \tau' \leftleftarrows \kappa$
   - by rule, $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$

2. by Completeness of Type Comparison, π, π' have the same shape

   case $\pi = \chi$ and $\pi' = \chi'$
   - by definition of Type Comparison, $\Gamma \triangleright \chi \equiv \chi' \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \chi \sqsubseteq \chi'$

   case $\pi = \tau_1 \to \tau_2$ and $\pi' = \tau_1' \to \tau_2'$:
   - by inversion, $\Gamma \triangleright \tau_1 \Rrightarrow \pi_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_1' \Rrightarrow \pi_1' \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2' \Rrightarrow \pi_2' \leftleftarrows \Omega$
   - by Completeness of Type Comparison, $\pi_1 = \pi_1'$ and $\pi_2 = \pi_2'$
   - by inverting TARROW, $\Gamma \vdash \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 : \Omega$ and $\Gamma \vdash \tau_1' : \Omega$ and $\Gamma \vdash \tau_2' : \Omega$
   - by induction (1), $\Gamma \triangleright \tau_1' \leq \tau_1 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_2 \leq \tau_2' \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'$

   case $\pi = \tau_1 \times \tau_2$ and $\pi' = \tau_1' \times \tau_2'$: analogous

   case $\pi = \forall\alpha{:}\kappa_1.\tau_2$ and $\pi' = \forall\alpha{:}\kappa_1'.\tau_2'$:
   - by inversion, $\Gamma \triangleright \kappa_1 \Rrightarrow \kappa_1''$ and $\Gamma, \alpha{:}\kappa_1 \triangleright \tau_2 \Rrightarrow \pi_2 \leftleftarrows \Omega$ and $\Gamma \triangleright \kappa_1' \Rrightarrow \kappa_1''$ and $\Gamma, \alpha{:}\kappa_1' \triangleright \tau_2' \Rrightarrow \pi_2 \leftleftarrows \Omega$
   - by inverting TUNIV, $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 : \Omega$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' : \Omega$

– by Environment Validity, $\Gamma \vdash \kappa_1 : \Box$ and $\Gamma \vdash \kappa_1' : \Box$
– by Soundness of Type Comparison, $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$
– by Antisymmetry, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$
– by Completeness of Kind Matching, $\Gamma \triangleright \kappa_1' \leq \kappa_1$
– by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2 : \Omega$
– by induction (1), $\Gamma, \alpha{:}\kappa_1' \triangleright \tau_2 \leq \tau_2' \leftleftarrows \Omega$
– by rule, $\Gamma \triangleright \forall\alpha{:}\kappa_1.\tau_2 \sqsubseteq \forall\alpha{:}\kappa_1'.\tau_2'$

case $\pi = \exists\alpha{:}\kappa.\tau$ and $\pi' = \exists\alpha{:}\kappa'.\tau'$: analogous

3. • by Validity, $\Gamma \vdash \tau : \kappa$ and $\Gamma \vdash \tau' : \kappa$
   • by Completeness of Type Comparison, $\Gamma \triangleright \tau \Rrightarrow \tau'' \leftleftarrows \kappa$ and $\Gamma \triangleright \tau' \Rrightarrow \tau'' \leftleftarrows \kappa$
   • by (1), $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$ and $\Gamma \triangleright \tau' \leq \tau \leftleftarrows \kappa$
4. • by Validity, $\Gamma \vdash \pi : \kappa$ and $\Gamma \vdash \pi' : \kappa$
   • by Completeness of Type Comparison, $\Gamma \triangleright \pi \Rrightarrow \pi'' \leftleftarrows \Omega$ and $\Gamma \triangleright \pi' \Rrightarrow \pi'' \leftleftarrows \Omega$
   • by inverting these, $\Gamma \triangleright \pi \Rrightarrow \pi'' \rightrightarrows \kappa$ and $\Gamma \triangleright \pi' \Rrightarrow \pi'' \rightrightarrows \kappa'$
   • by (2), $\Gamma \triangleright \pi \sqsubseteq \pi'$ and $\Gamma \triangleright \pi' \sqsubseteq \pi$

□

Figure 11.18 gave a transitive formulation of the subtyping judgement, whose soundness is easy to show:

Proposition 35 (Soundness of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \vdash \tau \leq \tau' : \Omega$*.*

Proof. By straightforward induction on the derivation, using Symmetry, Transitivity, and Validity. □

To show that the new formulation is actually transitive, we need a Weakening lemma:

Lemma 36 (Weakening of Transitive Type Inclusion)**.**
If $\Gamma = \Gamma_1, \alpha{:}\kappa, \Gamma_2$ *and* $\Gamma' = \Gamma_1, \alpha{:}\kappa', \Gamma_2$ *with* $\Gamma \vdash \Box$ *and* $\Gamma' \vdash \Box$ *and* $\Gamma_1 \vdash \kappa' \leq \kappa : \Box$*, and* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*.*

Proof. By straightforward induction on the derivation of $\Gamma \vdash \tau \leq^* \tau' : \Omega$, using Weakening. □

Let the *effective size* of a Transitive Type Inclusion derivation be the number of respective type inclusion rules used. To perform induction in the following proofs, it is important to know that a derivation for Transitive Type Inclusion of the same effective size can be performed for any equivalent pair of types:

Lemma 37 (Size Invariance of Transitive Type Inclusion Derivation)**.**
If $\Gamma \vdash \tau \leq^* \tau' : \Omega$ *and* $\Gamma \vdash \tau'' \equiv \tau : \Omega$ *and* $\Gamma \vdash \tau' \equiv \tau''' : \Omega$*, then there is a derivation for* $\Gamma \vdash \tau'' \leq^* \tau''' : \Omega$ *with the same effective size.*

Proof. By straightforward induction on the derivation, using Transitivity of Type Equivalence. □

Using this lemma, transitivity and completeness are easy:

Proposition 38 (Transitivity of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq^* \tau' : \Omega$ *and* $\Gamma \vdash \tau' \leq^* \tau'' : \Omega$*, then* $\Gamma \vdash \tau \leq^* \tau'' : \Omega$*.*

Proof. By induction on the combined effective size of the derivations. Arrow types are paths, so by Transitivity and Shape Consistency of Equivalence, only the following cases apply:

case at least one rule uses TSEQUIV*:

- by Reflexivity and Size Invariance of Transitive Type Inclusion Derivation, $\Gamma \vdash \tau \leq^* \tau'' : \Omega$

case TSARROW* / TSARROW*:

- by inversion, $\Gamma \vdash \tau \equiv \tau_1 \to \tau_2 : \Omega$ and $\Gamma \vdash \tau' \equiv \tau_1' \to \tau_2' : \Omega$ and $\Gamma \vdash \tau' \equiv \tau_1''' \to \tau_2''' : \Omega$ and $\Gamma \vdash \tau'' \equiv \tau_1'' \to \tau_2'' : \Omega$ and $\Gamma \vdash \tau_1' \leq^* \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 \leq^* \tau_2' : \Omega$ and $\Gamma \vdash \tau_1'' \leq^* \tau_1''' : \Omega$ and $\Gamma \vdash \tau_2''' \leq^* \tau_2'' : \Omega$
- by Transitivity, $\Gamma \vdash \tau_1' \to \tau_2' \equiv \tau_1''' \to \tau_2''' : \Omega$
- by Type Equivalence Inversion, $\Gamma \vdash \tau_1' \equiv \tau_1''' : \Omega$ and $\Gamma \vdash \tau_2' \equiv \tau_2''' : \Omega$
- by Symmetry and Size Invariance of Transitive Type Inclusion Derivation, $\Gamma \vdash \tau_1''' \leq^* \tau_1 : \Omega$ and $\Gamma \vdash \tau_2' \leq^* \tau_2'' : \Omega$
- by induction, $\Gamma \vdash \tau_1'' \leq^* \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 \leq^* \tau_2'' : \Omega$
- by TSARROW*, $\Gamma \vdash \tau \leq^* \tau'' : \Omega$

case TSTIMES* / TSTIMES*: analogous

case TSUNIV* / TSUNIV*:

- by inversion, $\Gamma \vdash \tau \equiv \forall \alpha{:}\kappa_1.\tau_2 : \Omega$ and $\Gamma \vdash \tau' \equiv \forall \alpha{:}\kappa_1'.\tau_2' : \Omega$ and $\Gamma \vdash \tau' \equiv \forall \alpha{:}\kappa_1'''.\tau_2''' : \Omega$ and $\Gamma \vdash \tau'' \equiv \forall \alpha{:}\kappa_1''.\tau_2'' : \Omega$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2 \leq^* \tau_2' : \Omega$ and $\Gamma \vdash \kappa_1'' \leq \kappa_1''' : \Box$ and $\Gamma, \alpha{:}\kappa_1'' \vdash \tau_2''' \leq^* \tau_2'' : \Omega$
- by Transitivity, $\Gamma \vdash \forall \alpha{:}\kappa_1'.\tau_2' \equiv \forall \alpha{:}\kappa_1'''.\tau_2''' : \Omega$
- by Type Equivalence Inversion, $\Gamma \vdash \kappa_1' \equiv \kappa_1''' : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' \equiv \tau_2''' : \Omega$
- by Transitivity and KSEQUIV, $\Gamma \vdash \kappa_1'' \leq \kappa_1' : \Box$ and $\Gamma \vdash \kappa_1'' \leq \kappa_1 : \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1'' \vdash \tau_2' \equiv \tau_2''' : \Omega$
- by Weakening for Transitive Type Inclusion, $\Gamma, \alpha{:}\kappa_1'' \vdash \tau_2 \leq^* \tau_2' : \Omega$
- by Size Invariance of Transitive Type Inclusion Derivation, $\Gamma, \alpha{:}\kappa_1'' \vdash \tau_2 \leq^* \tau_2''' : \Omega$
- by induction, $\Gamma, \alpha{:}\kappa_1'' \vdash \tau_2 \leq^* \tau_2'' : \Omega$
- by TSUNIV*, $\Gamma \vdash \tau \leq^* \tau'' : \Omega$

case TSEXIST* / TSEXIST*: analogous

□

Proposition 39 (Completeness of Transitive Type Inclusion)**.**
If $\Gamma \vdash \tau \leq \tau' : \Omega$*, then* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*.*

Proof. By straightforward induction on the derivation, using Reflexivity and applying Transitivity of Transitive Type Inclusion in the case of rule TSTRANS. □

Finally, we show completeness of the type matching algorithm by relating it to the transitive judgement:

Theorem 40 (Completeness of Algorithmic Type Matching)**.**

1. *If* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$*.*
2. *If* $\Gamma \vdash \tau \leq^* \tau' : \Omega$*, then* $\Gamma \triangleright \tau \Rrightarrow \pi$ *and* $\Gamma \triangleright \tau' \Rrightarrow \pi'$ *and* $\Gamma \triangleright \pi \sqsubseteq \pi'$*.*

3. *If* $\Gamma \vdash \tau \leq \tau' : \kappa$, *then* $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$.

Proof.

1. By simultaneous induction on the effective size of the derivation, simultaneously with (2):

   case TSequiv*:
   - by inversion, $\Gamma \vdash \tau \equiv \tau' : \Omega$
   - by Completeness of Type Matching wrt. Equivalence, $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$

   case TSarrow*:
   - by (2), $\Gamma \triangleright \pi \sqsubseteq \pi'$ with $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$
   - by rule, $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$

   case TStimes*: analogous

   case TSuniv*: analogous

   case TSexist*: analogous

2. case TSequiv*:
   - by inversion, $\Gamma \vdash \tau \equiv \tau' : \Omega$
   - by Validity, $\Gamma \vdash \tau : \Omega$ and $\Gamma \vdash \tau' : \Omega$
   - by Completeness of Type Comparison, $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$ and $\Gamma \vdash \tau' \equiv \pi' : \Omega$
   - by Symmetry and Transitivity, $\Gamma \vdash \pi \equiv \pi' : \Omega$
   - by Completeness of Type Matching wrt. Equivalence, $\Gamma \triangleright \pi \sqsubseteq \pi'$

   case TSarrow*:
   - by inversion, $\Gamma \vdash \tau \equiv \tau_1 \to \tau_2 : \Omega$ and $\Gamma \vdash \tau' \equiv \tau_1' \to \tau_2' : \Omega$ and $\Gamma \vdash \tau_1' \leq^* \tau_1 : \Omega$ and $\Gamma \vdash \tau_2 \leq^* \tau_2' : \Omega$
   - by Completeness of Type Comparison, $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$ and $\Gamma \triangleright \pi \Rrightarrow \pi'' \rightrightarrows \kappa$ and $\Gamma \triangleright \pi' \Rrightarrow \pi'' \rightrightarrows \kappa'$ and $\Gamma \triangleright \tau_1 \to \tau_2 \Rrightarrow \pi'' \rightrightarrows \kappa''$ and $\Gamma \triangleright \tau_1' \to \tau_2' \Rrightarrow \pi'' \rightrightarrows \kappa'''$
   - by definition of Path Normalization, $\pi'' = \tau_1'' \to \tau_2''$
   - by inversion of Path Normalization, $\pi = \tau_3 \to \tau_4$ and $\pi' = \tau_3' \to \tau_4'$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$ and $\Gamma \vdash \tau' \equiv \pi' : \Omega$
   - by Symmetry and Transitivity, $\Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_3 \to \tau_4 : \Omega$ and $\Gamma \vdash \tau_1' \to \tau_2' \equiv \tau_3' \to \tau_4' : \Omega$
   - by Type Equivalence Inversion, $\Gamma \vdash \tau_1 \equiv \tau_3 : \Omega$ and $\Gamma \vdash \tau_2 \equiv \tau_4 : \Omega$ and $\Gamma \vdash \tau_1' \equiv \tau_3' : \Omega$ and $\Gamma \vdash \tau_2' \equiv \tau_4' : \Omega$
   - by Symmetry and Size Invariance of Transitive Type Inclusion Derivation, $\Gamma \vdash \tau_3' \leq^* \tau_3 : \Omega$ and $\Gamma \vdash \tau_4 \leq^* \tau_4' : \Omega$
   - by induction (1), $\Gamma \triangleright \tau_3' \leq \tau_3 \leftleftarrows \Omega$ and $\Gamma \triangleright \tau_4 \leq \tau_4' \leftleftarrows \Omega$
   - by rule, $\Gamma \triangleright \tau_3 \to \tau_4 \sqsubseteq \tau_3' \to \tau_4'$

   case TStimes*: analogous

   case TSuniv*:
   - by inversion, $\Gamma \vdash \tau \equiv \forall \alpha{:}\kappa_1.\tau_2 : \Omega$ and $\Gamma \vdash \tau' \equiv \forall \alpha{:}\kappa_1'.\tau_2' : \Omega$ and $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2 \leq^* \tau_2' : \Omega$
   - by Completeness of Type Comparison, $\Gamma \triangleright \tau \Rrightarrow \pi$ and $\Gamma \triangleright \tau' \Rrightarrow \pi'$ and $\Gamma \triangleright \pi \Rrightarrow \pi'' \rightrightarrows \kappa$ and $\Gamma \triangleright \pi' \Rrightarrow \pi'' \rightrightarrows \kappa'$ and $\Gamma \triangleright \forall \alpha{:}\kappa_1.\tau_2 \Rrightarrow \pi'' \rightrightarrows \kappa'$ and $\Gamma \triangleright \forall \alpha{:}\kappa_1'.\tau_2' \Rrightarrow \pi'' \rightrightarrows \kappa'$
   - by definition of Path Normalization, $\pi'' = \forall \alpha{:}\kappa_1''.\tau_2''$
   - by inversion of Path Normalization, $\pi = \forall \alpha{:}\kappa_3.\tau_4$ and $\pi' = \forall \alpha{:}\kappa_3'.\tau_4'$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$ and $\Gamma \vdash \tau' \equiv \pi' : \Omega$
   - by Symmetry and Transitivity, $\Gamma \vdash \forall \alpha{:}\kappa_1.\tau_2 \equiv \forall \alpha{:}\kappa_3.\tau_4 : \Omega$ and $\Gamma \vdash \forall \alpha{:}\kappa_1'.\tau_2' \equiv \forall \alpha{:}\kappa_3'.\tau_4' : \Omega$

- by Type Equivalence Inversion, $\Gamma \vdash \kappa_1 \equiv \kappa_3 : \Box$ and $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \equiv \tau_4 : \Omega$ and $\Gamma \vdash \kappa_1' \equiv \kappa_3' : \Box$ and $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' \equiv \tau_4' : \Omega$
- by Symmetry, KSEQUIV, and Transitivity, $\Gamma \vdash \kappa_3' \leq \kappa_1 : \Box$ and $\Gamma \vdash \kappa_3' \leq \kappa_1' : \Box$ and $\Gamma \vdash \kappa_3' \leq \kappa_3 : \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_3' \vdash \tau_2 \leq^* \tau_2' : \Omega$ and $\Gamma, \alpha{:}\kappa_3' \vdash \tau_2 \equiv \tau_4 : \Omega$ and $\Gamma, \alpha{:}\kappa_3' \vdash \tau_2' \equiv \tau_4' : \Omega$
- by Symmetry and Size Invariance of Transitive Type Inclusion Derivation, $\Gamma, \alpha{:}\kappa_3' \vdash \tau_4 \leq^* \tau_4' : \Omega$
- by induction (1), $\Gamma, \kappa_3' \triangleright \tau_4 \leq \tau_4' \leftleftarrows \Omega$
- by rule, $\Gamma \triangleright \forall\alpha{:}\kappa_3.\tau_4 \sqsubseteq \forall\alpha{:}\kappa_3'.\tau_4'$

case TSEXIST*: analogous

3. case $\kappa = \Omega$:
   - by Completeness of Transitive Type Inclusion, $\Gamma \vdash \tau \leq^* \tau' : \Omega$
   - by (1), $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \Omega$

   case $\kappa \neq \Omega$:
   - by straightforward induction on the derivation, $\Gamma \vdash \tau \equiv \tau' : \kappa$
   - by Completeness of Type Matching wrt. Equivalence, $\Gamma \triangleright \tau \leq \tau' \leftleftarrows \kappa$

□

Theorem 41 (Decidability of Type Inclusion)**.**
Given $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$*, it is decidable whether* $\Gamma \vdash \tau \leq \tau' : \kappa$ *holds.*

Proof. By Completeness of Kind Matching and Type Matching. □

The transitive formulation of subtyping can also be used to show some non-trivial properties about the subtyping relation:

Proposition 42 (Antisymmetry of Type Inclusion)**.** *If and only if* $\Gamma \vdash \tau \leq \tau' : \kappa$ *and* $\Gamma \vdash \tau' \leq \tau : \kappa$*, then* $\Gamma \vdash \tau \equiv \tau' : \kappa$*.*

Proof. The inverse direction is trivial by rule TSEQUIV. The other follows by straightforward induction on the effective size of the respective Transitive Type Inclusion Derivation, its Soundness and Completeness, and Antisymmetry of Kind Inclusion. □

As for equivalence, we can derive handy inversion principles:

Proposition 43 (Type Inclusion Inversion)**.**

1. *If* $\Gamma \vdash \alpha \leq \alpha : \kappa$*, then* $\Gamma \vdash \alpha : \Gamma(\alpha)$ *and* $\Gamma \vdash \Gamma(\alpha) \leq \kappa : \Box$*.*
2. *If* $\Gamma \vdash \Psi \leq \Psi : \kappa$*, then* $\Gamma \vdash \Box$*.*
3. *If* $\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' : \kappa$*, then* $\Gamma \vdash \tau_1' \leq \tau_1 : \Omega$ *and* $\Gamma \vdash \tau_2 \leq \tau_2' : \Omega$*.*
4. *If* $\Gamma \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2' : \kappa$*, then* $\Gamma \vdash \tau_1 \leq \tau_1' : \Omega$ *and* $\Gamma \vdash \tau_2 \leq \tau_2' : \Omega$*.*
5. *If* $\Gamma \vdash \forall\alpha{:}\kappa_1.\tau_2 \leq \forall\alpha{:}\kappa_1'.\tau_2' : \kappa$*, then* $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \leq \tau_2' : \Omega$*.*
6. *If* $\Gamma \vdash \exists\alpha{:}\kappa_1.\tau_2 \leq \exists\alpha{:}\kappa_1'.\tau_2' : \kappa$*, then* $\Gamma \vdash \kappa_1 \leq \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \leq \tau_2' : \Omega$*.*
7. *If* $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 \leq \lambda\alpha{:}\kappa_1'.\tau_2' : \kappa$*, then* $\Gamma \vdash \kappa_1 \equiv \kappa_1' : \Box$ *and* $\Gamma, \alpha{:}\kappa_1 \vdash \tau_2 \leq \tau_2' : \kappa_2$*.*
8. *If* $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \leq \langle \tau_1', \tau_2' \rangle : \kappa$*, then* $\Gamma \vdash \tau_1 \equiv \tau_1' : \kappa_1$ *and* $\Gamma \vdash \tau_2 \equiv \tau_2' : \kappa_2$*.*

9. *If* $\Gamma \vdash \chi \leq \tau : \kappa$ *or* $\Gamma \vdash \tau \leq \chi : \kappa$, *and* $\chi = \pi$, *then* $\Gamma \vdash \chi \equiv \tau : \kappa$.

Proof. By Completeness of Type Matching and induction on the Matching algorithm, using Type Equivalence Inversion. □

Proposition 44 (Shape Consistency of Type Inclusion)**.**
Let $\Gamma \vdash \pi' \leq \pi : \Omega$ *and* $\Gamma \triangleright \pi \Rrightarrow \pi$ *and* $\Gamma \triangleright \pi' \Rrightarrow \pi'$.

1. *If* $\pi = \alpha$, *then* $\pi' = \alpha$.
2. *If* $\pi = \Psi$, *then* $\pi' = \Psi$.
3. *If* $\pi = \tau_1 \to \tau_2$, *then* $\pi' = \tau_1' \to \tau_2'$.
4. *If* $\pi = \tau_1 \times \tau_2$, *then* $\pi' = \tau_1' \times \tau_2'$.
5. *If* $\pi = \forall \alpha{:}\kappa_1.\tau_2$, *then* $\pi' = \forall \alpha{:}\kappa_1'.\tau_2'$.
6. *If* $\pi = \exists \alpha{:}\kappa_1.\tau_2$, *then* $\pi' = \exists \alpha{:}\kappa_1'.\tau_2'$.

Proof. By Completeness of Type Matching and induction on the Matching algorithm, using Shape Consistency of Type Equivalence. □

D. Proofs of Term Level Properties

D.1. Declarative Properties

Proposition 45 (Validity of Term Validity Rules)**.**
If $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \tau : \Omega$*.*

Proof. By induction on the derivation. We show only the non-trivial cases:

case EAPP:

- by inversion, $\tau = \tau_2$ and $\Gamma \vdash e_1 : \tau_1 \to \tau_2$
- by induction, $\Gamma \vdash \tau_1 \to \tau_2 : \Omega$
- by Type Validity Inversion, $\Gamma \vdash \tau_2 : \Omega$

case EINST:

- by inversion, $\tau = \tau_1[\tau_2/\alpha]$ and $\Gamma \vdash e_1 : \forall \alpha{:}\kappa_2.\tau_1$ and $\Gamma \vdash \tau_2 : \kappa_2$
- by induction, $\Gamma \vdash \forall \alpha{:}\kappa_2.\tau_1 : \Omega$
- by Type Validity Inversion, $\Gamma, \alpha{:}\kappa_2 \vdash \tau_1 : \Omega$
- by Substitutability, $\Gamma \vdash \tau_1[\tau_2/\alpha] : \Omega$

case ECLOSE:

- by inversion, $\tau = \exists \alpha{:}\mathrm{S}(\tau : \kappa).\tau_2$ and $\Gamma \vdash \tau : \kappa$ and $\Gamma \vdash e : \tau_2$
- by induction, $\Gamma \vdash \tau_2 : \Omega$
- by KSING*, $\Gamma \vdash \mathrm{S}(\tau : \kappa) : \Box$
- w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(\tau_2)$
- by NTYPE, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\mathrm{S}(\tau : \kappa) \vdash \tau_2 : \Omega$
- by TEXIST, $\Gamma \vdash \exists \alpha{:}\mathrm{S}(\tau : \kappa).\tau_2 : \Omega$

□

For later proofs we need an inversion principle on term validity, which is stated up to subtyping:

Proposition 46 (Inversion)**.** *Let* $\Gamma \vdash e : \tau$*.*

1. *If* $e = x$*, then* $\Gamma \vdash \Gamma(x) \leq \tau : \Omega$*.*
2. *If* $e = \lambda x{:}\tau_1.e_2$*, then* $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$ *with* $\Gamma \vdash \tau_1 \to \tau_2 \leq \tau : \Omega$
3. *If* $e = e_1\, e_2$*, then* $\Gamma \vdash e_1 : \tau_2 \to \tau$ *with* $\Gamma \vdash e_2 : \tau_2$*.*
4. *If* $e = \langle e_1, e_2 \rangle$*, then* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$ *with* $\Gamma \vdash \tau_1 \times \tau_2 \leq \tau : \Omega$*.*
5. *If* $e = \mathsf{let}\langle x_1, x_2 \rangle = e_1$ in e_2*, then* $\Gamma \vdash e_1 : \tau_1 \times \tau_2$ *with* $\Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau$*.*
6. *If* $e = \lambda \alpha{:}\kappa_1.e_2$*, then* $\Gamma, \alpha{:}\kappa_1 \vdash e_2 : \tau_2$ *with* $\Gamma \vdash \forall \alpha{:}\kappa_1.\tau_2 \leq \tau : \Omega$*.*

7. *If* $e = e_1\,\tau_2$, *then* $\Gamma \vdash e_1 : \forall\alpha{:}\kappa_2.\tau_1$ *with* $\Gamma \vdash \tau_2 : \kappa_2$ *and* $\Gamma \vdash \tau_1[\tau_2/\alpha] \leq \tau : \Omega$.
8. *If* $e = \langle \tau_1, e_2 \rangle$, *then* $\Gamma \vdash \tau_1 : \kappa_1$ *and* $\Gamma \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \exists\alpha{:}\mathrm{S}(\tau_1 : \kappa_1).\tau_2 \leq \tau : \Omega$.
9. *If* $e = \mathsf{let}\langle\alpha, x\rangle = e_1\ \mathsf{in}_{\tau_2}\ e_2$, *then* $\Gamma \vdash e_1 : \exists\alpha{:}\kappa_1.\tau_2$ *and* $\Gamma, \alpha{:}\kappa_1, x{:}\tau_2 \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.
10. *If* $e = \mathsf{new}\,\alpha \approx \tau_1\ \mathsf{in}_{\tau_2}\ e_2$, *then* $\Gamma, \alpha{:}\mathrm{A}(\tau_1) \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.
11. *If* $e = \{e_1\}^+_{\tau_2}$, *then* $\Gamma \vdash e_1 : \tau_1$ *with* $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_1)$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.
12. *If* $e = \{e_1\}^-_{\tau_1}$, *then* $\Gamma \vdash e_1 : \tau_1$ *with* $\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.
13. *If* $e = \mathsf{case}\,e_1{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}_{\tau_2}\ e_3$, *then* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_2 \vdash e_2 : \tau_2$ *and* $\Gamma \vdash e_3 : \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.
14. *If* $e = \mathsf{pickle}\,e_1$, *then* $\Gamma \vdash e_1 : \exists\alpha{:}\Omega.\alpha$ *and* $\Gamma \vdash \Psi \leq \tau : \Omega$.
15. *If* $e = \psi(v)$, *then* $\mathrm{FV}(v) \subseteq \mathrm{Dom}(\Gamma)$ *and* $\Gamma \vdash \Psi \leq \tau : \Omega$.
16. *If* $e = \mathsf{unpickle}\,x \Leftarrow e_1\ \mathsf{in}\ e_2\ \mathsf{else}_{\tau_2}\ e_3$, *then* $\Gamma \vdash e_1 : \Psi$ *and* $\Gamma, x{:}\exists\alpha{:}\Omega.\alpha \vdash e_2 : \tau_2$ *and* $\Gamma \vdash e_3 : \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.

Proof. Each by easy induction on the derivation. We show only a few representative cases:

1. $e = x$

   case EVAR: $\tau = \Gamma(x)$:
   - by inversion, $\Gamma \vdash \Box$
   - by Environment Validity, $\Gamma_1 \vdash \Gamma(x) : \Omega$ with $\Gamma = \Gamma_1, x{:}\Gamma(x), \Gamma_2$
   - by Weakening, $\Gamma \vdash \Gamma(x) : \Omega$
   - by Reflexivity, $\Gamma \vdash \Gamma(x) \leq \tau : \Omega$

   case ESUB:
   - by inversion, $\Gamma \vdash e : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
   - by induction, $\Gamma \vdash \Gamma(x) \leq \tau' : \Omega$
   - by Transitivity, $\Gamma \vdash \Gamma(x) \leq \tau : \Omega$

3. $e = e_1\,e_2$

   case EAPP:
   - by inversion, $\Gamma \vdash e_1 : \tau_2 \to \tau$ and $\Gamma \vdash e_2 : \tau_2$ directly

   case ESUB:
   - by inversion, $\Gamma \vdash e : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
   - by induction, $\Gamma \vdash e_1 : \tau_2 \to \tau'$ and $\Gamma \vdash e_2 : \tau_2$
   - by Validity and Reflexivity, $\Gamma \vdash \tau_2 \leq \tau_2 : \Omega$
   - by TSARROW, $\Gamma \vdash \tau_2 \to \tau' \leq \tau_2 \to \tau : \Omega$
   - by ESUB, $\Gamma \vdash e_1 : \tau_2 \to \tau$

11. $e = \{e_1\}^+_\tau$

    case EUP: $\tau = \tau_2$
    - by inversion, $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_1)$
    - by Validity, $\Gamma \vdash \tau_1 : \Omega$
    - by KSABS, $\Gamma \vdash \mathrm{A}(\tau_1) \leq \Omega : \Box$

- by TSUB, $\Gamma \vdash \tau_2 : \Omega$
- by Validity and Reflexivity, $\Gamma \vdash \tau_2 \leq \tau : \Omega$

case ESUB:

- by inversion, $\Gamma \vdash e : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
- by induction, $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_1)$ and $\Gamma \vdash \tau_2 \leq \tau' : \Omega$
- by Transitivity, $\Gamma \vdash \tau_2 \leq \tau : \Omega$

16. $e = \mathsf{unpickle}\, x \Leftarrow e_1 \;\mathsf{in}\; e_2 \;\mathsf{else}_{\tau_2}\; e_3$

case EUNPICKLE: $\tau = \tau_2$

- by inversion, $\Gamma \vdash e_1 : \Psi$ and $\Gamma, x{:}\exists\alpha{:}\Omega.\alpha \vdash e_2 : \tau_2$ and $\Gamma \vdash e_3 : \tau_2$
- by Reflexivity, $\Gamma \vdash \tau_2 \leq \tau_2 : \Omega$

case ESUB:

- by inversion, $\Gamma \vdash e : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
- by induction, $\Gamma \vdash e_1 : \Psi$ and $\Gamma, x{:}\exists\alpha{:}\Omega.\alpha \vdash e_2 : \tau_2$ and $\Gamma \vdash e_3 : \tau_2$ and $\Gamma \vdash \tau_2 \leq \tau' : \Omega$
- by Transitivity, $\Gamma \vdash \tau_2 \leq \tau : \Omega$

□

D.2. Algorithmic Type Checking

Theorem 47 (Soundness of Algorithmic Type Synthesis)**.**
Let $\Gamma \vdash \square$.

1. *If* $\Gamma \triangleright e \rightrightarrows \tau$, *then* $\Gamma \vdash e : \tau$.
2. *If* $\Gamma \triangleright e \rightrightarrows\!\!\!\!\!\!\Rightarrow \pi$, *then* $\Gamma \vdash e : \pi$.
3. *If* $\Gamma \triangleright e \leftleftarrows \tau$ *with* $\Gamma \vdash \tau : \Omega$, *then* $\Gamma \vdash e : \tau$.

Proof. By easy simultaneous induction on the derivation, using Soundness of Algorithmic Kind Synthesis. We show only a few representative cases:

1. case $e = \lambda x{:}\tau_1.e_2$
   - by inversion, $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$ and $\Gamma, x{:}\tau_1 \triangleright e_2 \rightrightarrows \tau_2$
   - by Soundness of Kind Synthesis, $\Gamma \vdash \tau_1 : \Omega$
   - w.l.o.g., $x \notin \mathrm{Dom}(\Gamma)$
   - by NTERM, $\Gamma, x{:}\tau_1 \vdash \square$
   - by induction, $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$
   - by ELAMBDA, $\Gamma \vdash \lambda x{:}\tau_1.e_2 : \tau_1 \to \tau_2$

   case $e = e_1\, e_2$
   - by inversion, $\Gamma \triangleright e_1 \rightrightarrows\!\!\!\!\!\!\Rightarrow \tau_2 \to \tau_1$ and $\Gamma \triangleright e_2 \leftleftarrows \tau_2$
   - by induction (2), $\Gamma \vdash e_1 : \tau_2 \to \tau_1$
   - by Validity, $\Gamma \vdash \tau_2 \to \tau_1 : \Omega$
   - by Type Validity Inversion, $\Gamma \vdash \tau_2 : \Omega$
   - by induction (3), $\Gamma \vdash e_2 : \tau_2$
   - by EAPP, $\Gamma \vdash e_1\, e_2 : \tau_1$

   case $e = \{e\}^{+}_{\tau_1}$
   - by inversion, $\Gamma \triangleright \tau_1 \rightrightarrows \mathrm{S}_{\mathrm{A}(\tau_2)}(\tau_3)$ and $\Gamma \triangleright e_2 \leftleftarrows \tau_2$
   - by Soundness of Kind Synthesis, $\Gamma \vdash \tau_1 : \mathrm{S}_{\mathrm{A}(\tau_2)}(\tau_3)$

   - by Kind Subsumption, $\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)$
   - by Validity, $\Gamma \vdash \mathrm{A}(\tau_2) : \Box$
   - by inverting KABS, $\Gamma \vdash \tau_2 : \Omega$
   - by induction (3), $\Gamma \vdash e : \tau_2$
   - by EUP, $\Gamma \vdash \{e\}^+_{\tau_1} : \tau_1$

2. - by inversion, $\Gamma \triangleright e \rightrightarrows \tau$ and $\Gamma \triangleright \tau \Rrightarrow \pi$
   - by induction (1), $\Gamma \vdash e : \tau$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau \equiv \pi : \Omega$
   - by TSEQUIV, $\Gamma \vdash \tau \leq \pi : \Omega$
   - by ESUB, $\Gamma \vdash e : \pi$

3. - by inversion, $\Gamma \triangleright e \rightrightarrows \tau'$ and $\Gamma \triangleright \tau' \leq \tau \leftleftarrows \Omega$
   - by induction (1), $\Gamma \vdash e : \tau'$
   - by Validity, $\Gamma \vdash \tau' : \Omega$ and $\Gamma \vdash \Omega : \Box$
   - by Soundness of Type Matching, $\Gamma \vdash \tau' \leq \tau : \Omega$
   - by ESUB, $\Gamma \vdash e : \tau$

□

Theorem 48 (Completeness of Algorithmic Type Synthesis)**.**

1. *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \triangleright e \rightrightarrows \tau'$ *with* $\Gamma \vdash \tau' \leq \tau : \Omega$*.*
2. *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \triangleright e \Rrightarrow \pi$ *with* $\Gamma \vdash \pi \leq \tau : \Omega$*.*
3. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leq \tau' : \Omega$*, then* $\Gamma \triangleright e \leftleftarrows \tau'$*.*

Proof. By simultaneous induction on the derivation, using Completeness of Algorithmic Kind Synthesis. We show only a few representative cases:

1. case ELAMBDA:
   - by inversion, $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$
   - by Environment Validity, $\Gamma, x{:}\tau_1 \vdash \Box$ and $\Gamma \vdash \tau_1 : \Omega$
   - by Completeness of Kind Synthesis, $\Gamma \triangleright \tau_1 \leftleftarrows \Omega$
   - by induction, $\Gamma, x{:}\tau_1 \triangleright e_2 \rightrightarrows \tau_2'$ with $\Gamma \vdash \tau_2' \leq \tau_2 : \Omega$
   - by rule, $\Gamma \triangleright \lambda x{:}\tau_1.e_2 : \tau_1 \to \tau_2'$
   - by Reflexivity, $\Gamma \vdash \tau_1 \leq \tau_1 : \Omega$
   - by TSARROW, $\Gamma \vdash \tau_1 \to \tau_2' \leq \tau_1 \to \tau_2 : \Omega$

   case EAPP:
   - by inversion, $\Gamma \vdash e_1 : \tau_2 \to \tau$ and $\Gamma \vdash e_2 : \tau_2$
   - by induction (2), $\Gamma \triangleright e_1 \Rrightarrow \pi$ with $\Gamma \vdash \pi \leq \tau_2 \to \tau : \Omega$
   - by Shape Consistency of Type Inclusion, $\pi = \tau_2' \to \tau'$
   - by Type Inclusion Inversion, $\Gamma \vdash \tau_2 \leq \tau_2' : \Omega$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
   - by induction (3), $\Gamma \triangleright e_2 \leftleftarrows \tau_2'$
   - by rule, $\Gamma \triangleright e_1\, e_2 \rightrightarrows \tau'$

   case ECLOSE:
   - by inversion, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash e_2 : \tau_2$
   - by Completeness of Algorithmic Kind Synthesis, $\Gamma \triangleright \tau_1 \rightrightarrows \kappa_1'$ with $\Gamma \vdash \kappa_1' \leq \mathrm{S}(\tau_1 : \kappa_1) : \Box$
   - by KSSING-LEFT*, $\Gamma \vdash \mathrm{S}(\tau_1 : \kappa_1) \leq \kappa_1 : \Box$

- by Transitivity, $\Gamma \vdash \kappa_1' \leq \kappa_1 : \Box$
- by induction, $\Gamma \vdash e_2 \rightrightarrows \tau_2'$ with $\Gamma \vdash \tau_2' \leq \tau_2 : \Omega$
- by rule, $\Gamma \triangleright e \rightrightarrows \exists \alpha{:}\kappa_1'.\tau_2'$
- by Validity, $\Gamma \vdash \kappa_1' : \Box$
- by NTYPE, $\Gamma, \alpha{:}\kappa_1' \vdash \Box$
- by Weakening, $\Gamma, \alpha{:}\kappa_1' \vdash \tau_2' \leq \tau_2 : \Omega$
- by Validity, $\Gamma \vdash \exists \alpha{:}\kappa_1'.\tau_2'$
- by TSEXIST, $\Gamma \vdash \exists \alpha{:}\kappa_1'.\tau_2' \leq \exists \alpha{:}\kappa_1.\tau_2 : \Omega$

case EUP:

- by inversion, $\Gamma \vdash \tau_1 : \mathrm{A}(\tau_2)$ and $\Gamma \vdash e : \tau_2$
- by inspection of Kind Inclusion, the only proper subkinds of $\mathrm{A}(\tau_2)$ are of the form $\mathrm{S}_{\mathrm{A}(\tau_2')}(\tau_3)$
- by Completeness of Kind Synthesis, $\Gamma \triangleright \tau_1 \rightrightarrows \mathrm{S}_{\mathrm{A}(\tau_2')}(\tau_3)$ and $\Gamma \vdash \mathrm{S}_{\mathrm{A}(\tau_2')}(\tau_3) \leq \mathrm{A}(\tau_2) : \Box$
- by inverting KSSING-LEFT, $\Gamma \vdash \mathrm{A}(\tau_2') \leq \mathrm{A}(\tau_2) : \Box$
- by inverting KSABS, $\Gamma \vdash \tau_2' \equiv \tau_2 : \Box$
- by Antisymmetry, $\Gamma \vdash \tau_2 \leq \tau_2' : \Omega$
- by induction (3), $\Gamma \triangleright e \leftleftarrows \tau_2'$
- by rule, $\Gamma \triangleright \{e\}^+_{\tau_1} \rightrightarrows \tau_1$
- by Reflexivity, $\Gamma \vdash \tau_1 \leq \tau_1 : \Omega$

2.
- by induction (1), $\Gamma \triangleright e \rightrightarrows \tau'$ with $\Gamma \vdash \tau' \leq \tau : \Omega$
- by Validity, $\Gamma \vdash \tau' : \Omega$ and $\Gamma \vdash \tau : \Omega$
- by Completeness of Type Comparison, $\Gamma \triangleright \tau' \Rrightarrow \pi$
- by rule, $\Gamma \triangleright e \rightrightrightarrows \pi$
- by Soundness of Type Comparison, $\Gamma \vdash \tau' \equiv \pi : \Omega$
- by Antisymmetry and Transitivity, $\Gamma \vdash \pi \leq \tau : \Omega$

3.
- by induction (1), $\Gamma \triangleright e \rightrightarrows \tau''$ with $\Gamma \vdash \tau'' \leq \tau : \Omega$
- by Transitivity, $\Gamma \vdash \tau'' \leq \tau' : \Omega$
- by Validity, $\Gamma \vdash \tau'' : \Omega$ and $\Gamma \vdash \tau' : \Omega$
- by Completeness of Type Matching, $\Gamma \triangleright \tau'' \leq \tau' \leftleftarrows \Omega$
- by rule, $\Gamma \triangleright e \leftleftarrows \tau'$

□

D.3. Soundness

D.3.1. Preservation

For showing preservation of cancelled coercions, it is important to know that two abstract types that are known to be equal at kind Ω are actually the same abstract type, with the same representation:

Proposition 49 (Representation Equivalence)**.**
Let $\Gamma \vdash \tau_1 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_2')$.

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2 : \Omega$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_1' \equiv \tau_2' : \Omega$.
2. *If* $\Gamma \vdash \tau_1 \leq \tau_2 : \Omega$, *then* $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1')$ *and* $\Gamma \vdash \tau_1' \equiv \tau_2' : \Omega$.

Proof.

1. - by Completeness of Type Comparison, $\Gamma \vdash \tau_1 \Rrightarrow \tau$ and $\Gamma \vdash \tau_2 \Rrightarrow \tau$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau_1 \equiv \tau : \mathrm{A}(\tau_1')$ and $\Gamma \vdash \tau_2 \equiv \tau : \mathrm{A}(\tau_2')$
   - by Validity, $\Gamma \vdash \tau : \mathrm{A}(\tau_1')$ and $\Gamma \vdash \tau : \mathrm{A}(\tau_2')$
   - by Completeness of Kind Synthesis, $\Gamma \vdash \tau : \mathrm{A}(\tau')$ with $\Gamma \vdash \mathrm{A}(\tau') \leq \mathrm{A}(\tau_1') : \Box$ and $\Gamma \vdash \mathrm{A}(\tau') \leq \mathrm{A}(\tau_2') : \Box$
   - by inverting KSABS, $\Gamma \vdash \tau' \equiv \tau_1' : \Omega$ and $\Gamma \vdash \tau' \equiv \tau_2' : \Omega$
   - by Symmetry and Transitivity, $\Gamma \vdash \tau_2' \equiv \tau_1' : \Omega$
   - by KSABS, $\Gamma \vdash \mathrm{A}(\tau_2') \leq \mathrm{A}(\tau_1') : \Box$
   - by TSUB, $\Gamma \vdash \tau_2 : \mathrm{A}(\tau_1')$
   - by definition of Type Comparison, $\Gamma \triangleright \tau_1 \equiv \tau_2 \leftleftarrows \mathrm{A}(\tau_1)$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1)$
2. - by Completeness of Type Comparison, $\Gamma \triangleright \tau_1 \Rrightarrow \pi_1$ and $\Gamma \triangleright \tau_2 \Rrightarrow \pi_2$
   - by Soundness of Type Comparison, $\Gamma \vdash \tau_1 \equiv \pi_1 : \mathrm{A}(\tau_1')$ and $\Gamma \vdash \tau_2 \equiv \pi_2 : \mathrm{A}(\tau_2')$
   - by Validity, $\Gamma \vdash \pi_1 : \mathrm{A}(\tau_1')$ and $\Gamma \vdash \pi_2 : \mathrm{A}(\tau_2')$
   - by Canonical Types, $\pi_1 = \chi_1$ and $\pi_2 = \chi_2$
   - by Kind Subsumption, $\Gamma \vdash \tau_1 \equiv \pi_1 : \Omega$ and $\Gamma \vdash \tau_2 \equiv \pi_2 : \Omega$
   - by Symmetry, TSEQUIV, and Transitivity, $\Gamma \vdash \pi_1 \leq \pi_2 : \Omega$
   - by Type Inclusion Inversion, $\Gamma \vdash \pi_1 \equiv \pi_2 : \Omega$
   - by Transitivity, $\Gamma \vdash \tau_1 \equiv \tau_2 : \Omega$
   - by (1), $\Gamma \vdash \tau_1 \equiv \tau_2 : \mathrm{A}(\tau_1')$ and $\Gamma \vdash \tau_1' \equiv \tau_2' : \Omega$

□

Lemma 50 (Decomposition and Replacement)**.** *If* $\Gamma \vdash E[e] : \tau$*, then* $\Gamma \vdash e : \tau'$*, and if* $\Gamma' \vdash e' : \tau'$ *with* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash E[e'] : \tau$*.*

Proof. By straightforward induction on the structure of the context, using Inversion. □

Theorem 51 (Preservation)**.**

1. *If* $\Delta \vdash e : \tau$ *and* $\Delta; e \to \Delta'; e'$ *with* $E = \_$*, then* $\Delta' \vdash e' : \tau$*.*
2. *If* $\cdot \vdash C : \tau$ *and* $C \to C'$*, then* $\cdot \vdash C' : \tau$*.*

Proof. 1. Case analysis:

case RAPP: $e = (\lambda x{:}\tau_1.e_2)\, v$ and $e' = e_2[v/x]$
- by Inversion (3), $\Delta \vdash \lambda x{:}\tau_1.e_2 : \tau_1' \to \tau''$ and $\Delta \vdash v : \tau_1'$
- by Inversion (2), $\Delta, x{:}\tau_1 \vdash e_2 : \tau_2$ and $\Delta \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau : \Omega$
- by Type Inclusion Inversion, $\Delta \vdash \tau_1' \leq \tau_1 : \Omega$ and $\Delta \vdash \tau_2 \leq \tau : \Omega$
- by Weakening, $\Delta, x{:}\tau_1 \vdash \tau_2 \leq \tau : \Omega$
- by ESUB, $\Delta \vdash v : \tau_1$ and $\Delta, x{:}\tau_1 \vdash e_2 : \tau$
- by Substitutability, $\Delta \vdash e_2[v/x] : \tau$

case RPROJ: $e = \mathsf{let}\langle x_1, x_2\rangle{=}\langle v_1, v_2\rangle\ \mathsf{in}\ e_2$ and $e' = e_2[v_1/x_1][v_2/x_2]$
- by Inversion (5), $\Delta \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2$ and $\Delta, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau$
- by Inversion (4), $\Delta \vdash v_1 : \tau_1'$ and $\Delta \vdash v_2 : \tau_2'$ and $\Delta \vdash \tau_1' \times \tau_2' \leq \tau_1 \times \tau_2 : \Omega$

- by Type Inclusion Inversion, $\Delta \vdash \tau_1' \leq \tau_1 : \Omega$ and $\Delta \vdash \tau_2' \leq \tau_2 : \Omega$
- by ESUB, $\Delta \vdash v_1 : \tau_1$ and $\Delta \vdash v_2 : \tau_2$
- by Substitutability, $\Delta \vdash e_2[v_1/x_1][v_2/x_2] : \tau$

case RINST: $e = (\lambda\alpha{:}\kappa_1.e_2)\,\tau_3$ and $e' = e_2[\tau_3/\alpha]$

- by Inversion (7), $\Delta \vdash \lambda\alpha{:}\kappa_1.e_2 : \forall\alpha{:}\kappa_1'.\tau_2'$ and $\Delta \vdash \tau_3 : \kappa_1'$ and $\Delta \vdash \tau_2'[\tau_3/\alpha] \leq \tau : \Omega$
- by Inversion (6), $\Delta, \alpha{:}\kappa_1 \vdash e_2 : \tau_2$ and $\Delta \vdash \forall\alpha{:}\kappa_1.\tau_2 \leq \forall\alpha{:}\kappa_1'.\tau_2' : \Omega$
- by Type Inclusion Inversion, $\Delta \vdash \kappa_1' \leq \kappa_1 : \Omega$ and $\Delta, \alpha{:}\kappa_1' \vdash \tau_2 \leq \tau_2' : \Omega$
- by Substitutability, $\Delta \vdash \tau_2[\tau_3/\alpha] \leq \tau_2'[\tau_3/\alpha] : \Omega$
- by TSUB, $\Delta \vdash \tau_3 : \kappa_1$
- by Substitutability, $\Delta \vdash e_2[\tau_3/\alpha] : \tau_2[\tau_3/\alpha]$
- by ESUB, $\Delta \vdash e_2[\tau_3/\alpha] : \tau_2'[\tau_3/\alpha]$
- by ESUB, $\Delta \vdash e_2[\tau_3/\alpha] : \tau$

case ROPEN: $e = \mathsf{let}\langle\alpha, x\rangle{=}\langle\tau_1, v_2\rangle\ \mathsf{in}_{\tau'}\ e_2$ and $e' = e_2[\tau_1/\alpha][v_2/x]$

- w.l.o.g., $\alpha \notin \mathrm{Dom}(\Delta)$
- by Inversion (9), $\Delta \vdash \langle\tau_1, v_2\rangle : \exists\alpha{:}\kappa_1.\tau_2$ and $\Delta, \alpha{:}\kappa_1, x{:}\tau_2 \vdash e_2 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by Inversion (8), $\Delta \vdash \tau_1 : \kappa_1'$ and $\Delta \vdash v_2 : \tau_2'$ and $\Delta \vdash \exists\alpha{:}\mathrm{S}(\tau_1 : \kappa_1').\tau_2' \leq \exists\alpha{:}\kappa_1.\tau_2 : \Omega$
- by Type Inclusion Inversion, $\Delta \vdash \mathrm{S}(\tau_1 : \kappa_1') \leq \kappa_1 : \Omega$ and $\Delta, \alpha{:}\mathrm{S}(\tau_1 : \kappa_1') \vdash \tau_2' \leq \tau_2 : \Omega$
- by TEXT-SING*, $\Delta \vdash \tau_1 : \mathrm{S}(\tau_1 : \kappa_1')$
- by Substitutability, $\Delta \vdash \tau_2'[\tau_1/\alpha] \leq \tau_2[\tau_1/\alpha] : \Omega$
- by ESUB, $\Delta \vdash v_2 : \tau_2[\tau_1/\alpha]$
- by TSUB, $\Delta \vdash \tau_1 : \kappa_1$
- by Substitutability, $\Delta \vdash e_2[\tau_1/\alpha][v_2/x] : \tau'[\tau_1/\alpha]$
- by Validity, $\Delta \vdash \tau' : \Omega$
- by Variable Containment, $\tau'[\tau_1/\alpha] = \tau'$
- by ESUB, $\Delta \vdash e_2[\tau_1/\alpha][v_2/x] : \tau$

case RNEW: $e = \mathsf{new}\,\alpha \approx \tau_1\ \mathsf{in}_{\tau'}\ e_2$ and $e' = e_2$ and $\Delta' = \Delta, \alpha{:}\mathrm{A}(\tau_1)$

- by Inversion (10), $\Delta, \alpha{:}\mathrm{A}(\tau_1) \vdash e_2 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by ESUB, $\Delta, \alpha{:}\mathrm{A}(\tau_1) \vdash e_2 : \tau$

case RCANCEL: $e = \{\{v\}^+_{\tau_2}\}^-_{\tau_1}$ and $e' = v$

- by Inversion (12), $\Delta \vdash \{v\}^+_{\tau_2} : \tau_1$ and $\Delta \vdash \tau_1 : \mathrm{A}(\tau_1')$ and $\Delta \vdash \tau_1' \leq \tau : \Omega$
- by Inversion (11), $\Delta \vdash v : \tau_2'$ and $\Delta \vdash \tau_2 : \mathrm{A}(\tau_2')$ and $\Delta \vdash \tau_2 \leq \tau_1 : \Omega$
- by Representation Equivalence, $\Delta \vdash \tau_2' \equiv \tau_1' : \Omega$
- by Antisymmetry, $\Delta \vdash \tau_2' \leq \tau_1' : \Omega$
- by Transitivity, $\Delta \vdash \tau_2' \leq \tau : \Omega$
- by ESUB, $\Delta \vdash v : \tau$

case RCASE1: $e = \mathsf{case}\,v{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_3\ \mathsf{else}_{\tau'}\ e_4$ and $e' = e_3[v/x]$ and $\Delta \vdash \tau_1 \equiv \tau_2 : \Omega$

- by Inversion (13), $\Delta \vdash v : \tau_1$ and $\Delta, x{:}\tau_2 \vdash e_3 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by Weakening, $\Delta \vdash \tau_1 \equiv \tau_2 : \Omega$
- by TSEQUIV and ESUB, $\Delta \vdash v : \tau_2$
- by Substitutability, $\Delta \vdash e_3[v/x] : \tau'$
- by ESUB, $\Delta \vdash e_3[v/x] : \tau$

case RCASE2: $e = \mathsf{case}\,v{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_3\ \mathsf{else}_{\tau'}\ e_4$ and $e' = e_4$

- by Inversion (13), $\Delta \vdash e_4 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by ESUB, $\Delta \vdash e_4 : \tau$

case RPICKLE: $e = \mathsf{pickle}\,v$ and $e' = \psi(v)$

- by Inversion (15), $\Delta \vdash v : \exists\alpha{:}\Omega.\alpha$ and $\Delta \vdash \Psi \leq \tau : \Omega$
- by Variable Containment, $\mathrm{FV}(v) \subseteq \mathrm{Dom}(\Delta)$
- by Epsi, $\Delta \vdash \psi(v) : \Psi$
- by Esub, $\Delta \vdash \psi(v) : \tau$

case Runpickle1: $e = \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}_{\tau'}\ e_2$ and $e' = e_1[v/x]$ and $\Delta \vdash v : \exists\alpha{:}\Omega.\alpha$

- by Inversion (16), $\Delta, x{:}\exists\alpha{:}\Omega.\alpha \vdash e_1 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by Weakening, $\Delta \vdash v : \exists\alpha{:}\Omega.\alpha$
- by Substitutability, $\Delta \vdash e_1[v/x] : \tau'$
- by Esub, $\Delta \vdash e_1[v/x] : \tau$

case Runpickle2: $e = \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}_{\tau'}\ e_2$ and $e' = e_2$

- by Inversion (16), $\Delta \vdash e_2 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
- by Esub, $\Delta \vdash e_2 : \tau$

2.
- let $C = \Delta; E[e]$ such that E maximal
- by inverting Cvalid, $\Delta \vdash E[e] : \tau$
- by Decomposition, $\Delta \vdash e : \tau'$
- by (1), $\Delta' \vdash e' : \tau'$
- by Replacement, $\Delta' \vdash E[e'] : \tau$
- by Cvalid, $\cdot \vdash \Delta'; E[e'] : \tau$

□

D.3.2. Progress

The progress proof requires the usual canonical forms lemma, but here respecting the heap:

Lemma 52 (Canonical Values)**.**
Let $\Gamma \vdash v : \tau$.

1. *If* $\Gamma \vdash \tau \leq \tau' : \Omega$ *and* $\Gamma \vdash \tau' : \mathrm{A}(\tau_1)$*, then* $v = \{v_1\}^+_{\tau_2}$.
2. *If* $\Gamma \vdash \tau \leq \Psi : \Omega$*, then* $v = \psi(v_1)$.
3. *If* $\Gamma \vdash \tau \leq \tau_1 \to \tau_2 : \Omega$*, then* $v = \lambda x{:}\tau_1'.e_2$.
4. *If* $\Gamma \vdash \tau \leq \tau_1 \times \tau_2 : \Omega$*, then* $v = \langle v_1, v_2\rangle$.
5. *If* $\Gamma \vdash \tau \leq \forall\alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \lambda\alpha{:}\kappa_1'.e_2$.
6. *If* $\Gamma \vdash \tau \leq \exists\alpha{:}\kappa_1.\tau_2 : \Omega$*, then* $v = \langle \tau_1, v_2\rangle$.

Proof. Each by induction on the derivation of $\Gamma \vdash v : \tau$, using normalisation and Shape Consistency to exclude impossible cases.

1. case Eup: $v = \{v_1\}^+_{\tau_2}$

   case Esub:
   - by inversion, $\Gamma \vdash v : \tau''$ and $\Gamma \vdash \tau'' \leq \tau : \Omega$
   - by Transitivity, $\Gamma \vdash \tau'' \leq \tau' : \Omega$
   - by induction, $v = \{v_1\}^+_{\tau_2}$

2. case Epsi: $v = \psi(v_1)$

case ESUB:

- by inversion, $\Gamma \vdash v : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
- by Transitivity, $\Gamma \vdash \tau' \leq \Psi : \Omega$
- by induction, $v = \psi(v_1)$

3–6. Analogous.

□

Lemma 53 (Embedding)**.**
If $\Delta; E[e] \to \Delta, \Delta'; E[e']$*, then* $\Delta_1, \Delta, \Delta_2; E_1 E E_2[e] \to \Delta_1, \Delta, \Delta_2, \Delta'; E_1 E E_2[e']$ *for any contexts* E_1, E_2 *and heaps* Δ_1, Δ_2*.*

Proof. By induction on the structure of the additional contexts and case analysis for the reduction rule applied. Note that $E_1[E[E_2]]$ is itself a context. □

Theorem 54 (Progress)**.**
If $\Delta \vdash e : \tau$*, then either* $e = v$*, or* $(\Delta; e) = (\Delta; E[e_1]) \to (\Delta'; E[e_1']) = (\Delta'; e')$*.*

Proof. By induction on the structure of e. We show a few representative cases:

case $e = x$: impossible, due to Variable Containment

case $e = e_1\, e_2$ with $e_1 \neq v_1$:

- by Inversion, $\Delta \vdash e_1 : \tau_2 \to \tau$
- by induction, $\Delta; e_1 = \Delta; E'[e_3] \to \Delta'; E'[e_3']$
- let $E = E'\, e_2$
- by Embedding, $\Delta; E[e_3] \to \Delta'; E[e_3']$

case $e = v_1\, e_2$ with $e_2 \neq v_2$: Analogous.

case $e = v_1\, v_2$:

- by Inversion, $\Delta \vdash v_1 : \tau_2 \to \tau$
- by Validity and Reflexivity, $\Delta \vdash \tau_2 \to \tau \leq \tau_2 \to \tau : \Omega$
- by Canonical Values, $v_1 = \lambda x{:}\tau_2'.e_1$
- let $E = \_$
- by RAPP, $\Delta; e = \Delta; E[v_1\, v_2] \to \Delta; E[e_1[v_2/x]]$

case $e = (\mathsf{new}\, \alpha \approx \tau_1\ \mathsf{in}_{\tau_2}\ e_2)$:

- let $\Delta' = \Delta, \alpha{:}\mathrm{A}(\tau_1)$
- by RNEW, $\Delta; e = \Delta; e \to \Delta'; e_2$

case $e = \{v\}^-_{\tau_1}$:

- by Inversion, $\Delta \vdash v : \tau_1$ and $\Delta \vdash \tau_1 : \mathrm{A}(\tau_2)$
- by Canonical Values, $v = \{v_1\}^+_{\tau_1'}$
- let $E = \_$
- by RCANCEL, $\Delta; e = \Delta; E[\{v\}^-_{\tau_1}] \to \Delta; E[v_1]$

case $e = (\mathsf{case}\, v{:}\tau_1\ \mathsf{of}\ x{:}\tau_2.e_2\ \mathsf{else}_{\tau_2}\ e_3)$:

- let $E = \_$

- by Decidability of Type Inclusion, it is decidable whether $\Delta \vdash \tau_1 \leq \tau_2 : \Omega$
- if $\Delta \vdash \tau_1 \leq \tau_2 : \Omega$, then by RCASE1, $\Delta; e = \Delta; E[e] \to \Delta; E[e_2[v/x]]$
- otherwise, by RCASE2, $\Delta; e = \Delta; E[e] \to \Delta; E[e_3]$

case $e = (\mathsf{unpickle}\, x \Leftarrow v \;\mathsf{in}\; e_1 \;\mathsf{else}_{\tau_2}\; e_2)$:

- by Inversion, $\Delta \vdash v : \Psi$
- by Validity and Reflexivity, $\Delta \vdash \Psi \leq \Psi : \Omega$
- by Canonical Values, $v = \psi(v_1)$
- let $E = \_$
- by Decidability of Term Validity, it is decidable whether $\Delta \vdash v_1 : \exists \alpha{:}\Omega.\alpha$
- if $\Delta \vdash v_1 : \exists \alpha{:}\Omega.\alpha$, then by RUNPICKLE1, $\Delta; e = \Delta; E[e] \to \Delta; E[e_1[v_1/x]]$
- otherwise, by RUNPICKLE2, $\Delta; e = \Delta; E[e] \to \Delta; E[e_2]$

□

D.4. Opacity

The core of the authentication half of the proof actually is a straightforward lemma showing that we cannot construct a value of variable type α, where $\alpha : \Omega$ (note that the kind assumption is crucial, as for a type of kind $\mathrm{S}_{\hat{\kappa}}(\tau)$ or $\mathrm{A}(\tau)$ we often *could* write down a value).

Proposition 55 (Abstractness)**.** *If* $\Gamma = \Gamma_1, \alpha{:}\Omega, x{:}\alpha, \Gamma_2$ *and* $\Gamma \vdash e : \tau$ *with* $\Gamma \vdash \tau \leq \alpha$*, then* e *is not a value.*

Proof. By easy induction on the derivation. All rules that apply to values can be excluded using Type Shape Consistency, except for ESUB, which is trivial by Transitivity and induction. □

Theorem 56 (Opacity)**.**
Let $\Gamma = \alpha{:}\Omega, x{:}\alpha, f{:}\alpha{\to}1$*, and* $\gamma_i = [\alpha_i/\alpha] \cup [v_i/x] \cup [v_i'/f] \cup [\alpha'/\alpha' \mid \alpha' \in \mathrm{Dom}(\Delta')]$ *with* $\alpha \notin \mathrm{Dom}(\Delta, \Delta')$ *and* $\alpha_i \notin \mathrm{Dom}(\Delta)$ *and* $\Delta, \gamma_i(\Delta') \vdash \gamma_i : \Gamma, \Delta'$ *and* $\Delta; v_i'\, v_i \to^* \Delta; \diamond$ *for* $i \in \{1, 2\}$*.*

1. *Let* $\Gamma, \Delta' \vdash \tau : \kappa$ *and* $\Gamma, \Delta' \vdash \tau' : \kappa$*.*
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(\tau) \leq \gamma_1(\tau') : \gamma_1(\kappa)$*, then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) \leq \gamma_2(\tau') : \gamma_2(\kappa)$*.*
2. *Let* $\Gamma, \Delta' \vdash \square$*.*
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(e) : \gamma_1(\tau)$*, then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(e) : \gamma_2(\tau)$*.*
3. *Let* $\Gamma, \Delta' \vdash e : \tau$*.*
   If and only if $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to^* \Delta, \gamma_1(\Delta'), \Delta_1; v'$*, then* $\Delta_1 = \gamma_1(\Delta'')$ *and* $v' = \gamma_1(v)$ *with* $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to^* \Delta, \gamma_2(\Delta'), \gamma_2(\Delta''); \gamma_2(v)$*.*

Proof. For all parts it suffices to show only one direction, since the other is symmetric.

1. By Soundness and Completeness of Algorithmic Type Comparison and Matching, and induction on the derivation of $\Delta, \gamma_1(\Delta') \triangleright \gamma_1(\tau) \leq \gamma_1(\tau') \leftleftarrows \gamma_1(\kappa)$.
2. By Soundness and Completeness of Algorithmic Type Synthesis and induction on the derivation of $\Delta, \gamma_1(\Delta') \triangleright \gamma_1(e) \leftleftarrows \gamma_1(\tau)$.
3. By induction on the reduction sequence. Note that by Substitutability, $\Delta, \gamma_i(\Delta') \vdash \gamma_i(e) : \gamma_i(\tau)$. Moreover, if $\gamma_1(e')$ is a value for some e', then $\gamma_2(e')$ obviously is a value as well.

   case $\gamma_1(e) = v'$ and $\Delta_1 = \cdot$:

- obviously, $\Delta_1 = \gamma_1(\cdot) = \gamma_2(\cdot)$
- obviously, $\gamma_2(e)$ is also a value

case RAPP: $\Delta_1 = \cdot$

subcase $e = E[(\lambda x'{:}\tau_2.e_3)\, e_4]$ with $\gamma_1(e_4)$ value:

* by rule, $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to \Delta, \gamma_1(\Delta'); e'$ with $e' = \gamma_1(E)[\gamma_1(e_3)[\gamma_1(e_4)/x']]$
* by Variable Convention, $e' = \gamma_1(E[e_3[e_4/x']])$
* obviously, $\gamma_2(e_4)$ is also a value
* obviously, $\Delta_1 = \gamma_1(\cdot) = \gamma_2(\cdot)$
* hence likewise, $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to \Delta, \gamma_2(\Delta'); \gamma_2(E[e_3[e_4/x']])$
* the claim follows by induction

subcase $e = E[f\, e_4]$ with $\gamma_1(e_4)$ value:

* by Decomposition, $\Gamma \vdash f\, e_4 : \tau'$
* by Inversion, $\Gamma \vdash f : \tau_4 \to \tau'$ and $\Gamma \vdash e_4 : \tau_4$
* by Inversion, $\Gamma \vdash \Gamma(f) \leq \tau_4 \to \tau' : \Omega$
* by Type Inclusion Inversion, $\Gamma \vdash \tau_4 \leq \alpha : \Omega$
* by Abstractness, e_4 not a value
* since, $\gamma_1(e_4)$ value, $e_4 = x$ or $e_4 = f$
* by Inversion, $\Gamma \vdash \Gamma(e_4) \leq \alpha : \Omega$
* by Type Inclusion Inversion, $\Gamma \vdash \Gamma(e_4) \equiv \alpha : \Omega$
* hence, $e_4 \neq f$ and $e_4 = x$
* hence, $\gamma_1(f\, e_4) = v_1'\, v_1$
* by assumption, $\Delta; v_1'\, v_1 \to^* \Delta; \diamond$
* by Embedding, $\Delta, \gamma_1(\Delta'); \gamma_1(E)[v_1'\, v_1] \to^* \Delta, \gamma_1(\Delta'); \gamma_1(E)[\diamond]$
* likewise, $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to^* \Delta, \gamma_2(\Delta'); \gamma_2(E[\diamond])$
* the claim follows by induction

case RNEW: $e = E[\mathsf{new}\, \alpha' \approx \tau_1\ \mathsf{in}_\tau\ e_2]$:

- by rule, $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to \Delta, \gamma_1(\Delta'), \alpha'{:}\gamma_1(\tau_1); e'$ with $e' = \gamma_1(E)[\gamma_1(e_2)]$
- by Variable Convention, $e' = \gamma_1(E[e_2])$
- hence likewise, $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to \Delta, \gamma_2(\Delta'), \alpha'{:}\gamma_2(\tau_1); \gamma_2(E[e_2])$
- the claim follows by induction

case RCASE1: $e = E[\mathsf{case}\, e_1{:}\tau_1\ \mathsf{of}\ x'{:}\tau_2.e_2\ \mathsf{else}_\tau\ e_3]$ with $\gamma_1(e_1)$ value:

- by rule, $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to \Delta, \gamma_1(\Delta'); e'$ with $e' = \gamma_1(E)[\gamma_1(e_2)[\gamma(e_1)/x']]$
- hence, $\Delta_1 = \cdot$
- by Variable Convention, $e' = \gamma_1(E[e_2[e_1/x']])$
- by side condition, $\Delta, \gamma_1(\Delta') \vdash \gamma_1(\tau) \leq \gamma_1(\tau') : \Omega$
- by (1), $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) \leq \gamma_2(\tau') : \Omega$
- obviously, $\gamma_2(e_1)$ is also a value
- hence likewise, $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to \Delta, \gamma_2(\Delta'); \gamma_2(E[e_2[e_1/x']])$
- the claim follows by induction

case RCASE2: similarly

case RUNPICKLE1: $e = E[\mathsf{unpickle}\, x' \Leftarrow \psi(e_1)\ \mathsf{in}\ e_2\ \mathsf{else}_\tau\ e_3]$ with $\gamma_1(e_1)$ value:

- by rule, $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to \Delta, \gamma_1(\Delta'); e'$ with $e' = \gamma_1(E)[\gamma_1(e_2)[\gamma(e_1)/x']]$
- hence, $\Delta_1 = \cdot$
- by Variable Convention, $e' = \gamma_1(E[e_2[e_1/x']])$
- by side condition, $\Delta, \gamma_1(\Delta') \vdash \gamma_1(e_1) : \exists\alpha{:}\Omega.\alpha$
- by (2), $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) : \exists\alpha{:}\Omega.\alpha$
- obviously, $\gamma_2(e_1)$ is also a value

- hence likewise, $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to \Delta, \gamma_2(\Delta'); \gamma_2(E[e_2[e_1/x']])$
- the claim follows by induction

case RUNPICKLE2: similarly

The remaining cases are straightforward.

□

E. Proofs for Higher-Order Abstraction

E.1. Kind Coercions

We start with kind coercions, because all other extensions depend on them. To ease some of the proofs, we factor out the following technical lemmata, which state that the substitutions in the residual kind of a dependent coercions do 'the right thing', i.e. yield the right equivalence inward and outward. They depend on the actual correctness of the judgement rules for kind coercions for the types involved, so we have to take that as an assumption. Since we invoke the lemma only on subterms during induction of the actual proofs, this approach is well-founded.

Lemma 57 (Kind Adaptive Substitutions)**.** *Provided the rules* TCOERCE*, TQCOERCE*, TQCOERCE-DROP* *and* TQCOERCE-CANCEL* *hold for kinds of size* $\mathrm{Size}(\tilde{\kappa})$*, and* $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} : \Box$ *and* $\Gamma \vdash \tau_+ : \kappa$ *and* $\Gamma \vdash \tau_- : \kappa$ *and* $\alpha' \notin \mathrm{Dom}(\Gamma, \alpha{:}\kappa)$*, then:*

1. *If* $\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha]$*, then* $\Gamma, \alpha{:}\kappa \vdash [\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_-}/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$.

2. *If* $\Gamma \vdash \tau : \tilde{\kappa}[\tau_-/\alpha]$*, then* $\Gamma \vdash [\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_-/\tau_-}/\alpha'] \equiv [\tau/\alpha'] : \Gamma, \alpha'{:}\tilde{\kappa}[\tau_-/\alpha]$.

3. *If* $\Gamma \vdash \tau \equiv \tau' : \tilde{\kappa}[\tau_-/\alpha]$ *and* $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box$ *and* $\Gamma \vdash \tau_- \equiv \tau'_- : \kappa$*,*
   then $\Gamma, \alpha{:}\kappa \vdash [\{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_-}/\alpha'] \equiv [\{\tau' : \alpha{:}\kappa.\tilde{\kappa}'\}_{\alpha/\tau'_-}/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$.

4. $\Gamma, \alpha'{:}\tilde{\kappa}[\tau_+/\alpha], \alpha{:}\kappa \vdash [\{\alpha' : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_+}/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$.

5. $\Gamma, \alpha'{:}\tilde{\kappa}[\tau_+/\alpha] \vdash [\{\alpha' : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_+/\tau_+}/\alpha'] \equiv [\alpha'/\alpha'] : \Gamma, \alpha'{:}\tilde{\kappa}[\tau_+/\alpha]$.

6. *If* $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa} \equiv \tilde{\kappa}' : \Box$ *and* $\Gamma \vdash \tau_+ \equiv \tau'_+ : \kappa$ *and* $\Gamma \vdash \tau_- \equiv \tau'_- : \kappa$*, then* $\Gamma, \alpha'{:}\tilde{\kappa}[\tau_+/\alpha], \alpha{:}\kappa \vdash$
   $[\{\alpha' : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_+}/\alpha'] \equiv [\{\alpha' : \alpha{:}\kappa.\tilde{\kappa}'\}_{\alpha/\tau'_+}/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$.

Proof.

1. - let $\tau' = \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_-}$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa \vdash \tau : \tilde{\kappa}[\tau_-/\alpha]$ and $\Gamma, \alpha{:}\kappa \vdash \tau_- : \kappa$
   - by TVAR, $\Gamma, \alpha{:}\kappa \vdash \alpha : \kappa$
   - by TCOERCE*, $\Gamma, \alpha{:}\kappa \vdash \tau' : \tilde{\kappa}$
   - by definition of Substitution Validity, $\Gamma, \alpha{:}\kappa \vdash [\tau'/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$
2. - let $\tau' = \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\tau_-/\tau_-}$
   - by Reflexivity, $\Gamma \vdash \tau_- \equiv \tau_- : \kappa$
   - by TQCOERCE-DROP*, $\Gamma \vdash \tau' \equiv \tau : \tilde{\kappa}[\tau_-/\alpha]$
   - by Validity, $\Gamma \vdash \tilde{\kappa}[\tau_-/\alpha] : \Box$ and $\Gamma \vdash \tau' : \tilde{\kappa}[\tau_-/\alpha]$
   - by NTYPE, $\Gamma, \alpha'{:}\tilde{\kappa}[\tau_-/\alpha] \vdash \Box$
   - by TVAR, $\Gamma, \alpha'{:}\tilde{\kappa}[\tau_-/\alpha] \vdash \alpha' : \tilde{\kappa}[\tau_-/\alpha]$

- by definition of Substitution Equivalence, $\Gamma \vdash [\tau'/\alpha'] \equiv [\tau/\alpha'] : \Gamma, \alpha'{:}\tilde{\kappa}[\tau_-/\alpha]$

3. - let $\tau_1 = \{\tau : \alpha{:}\kappa.\tilde{\kappa}\}_{\alpha/\tau_-}$ and $\tau_1' = \{\tau' : \alpha{:}\kappa'.\tilde{\kappa}'\}_{\alpha/\tau_-'}$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \Box$
   - by Weakening, $\Gamma, \alpha{:}\kappa \vdash \tau \equiv \tau' : \tilde{\kappa}[\tau_-/\alpha]$ and $\Gamma, \alpha{:}\kappa \vdash \tau_- \equiv \tau_-' : \kappa$
   - by TQVAR, $\Gamma, \alpha{:}\kappa \vdash \alpha \equiv \alpha : \kappa$
   - obviously, $\Gamma \vdash [\tau_+/\alpha] \equiv [\tau_+'/\alpha] : \Gamma, \alpha{:}\kappa$ and $\Gamma \vdash [\tau_-/\alpha] \equiv [\tau_-'/\alpha] : \Gamma, \alpha{:}\kappa$
   - by Full Functionality, $\Gamma \vdash \tilde{\kappa}[\tau_+/\alpha] \equiv \tilde{\kappa}'[\tau_+'/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}[\tau_-/\alpha] \equiv \tilde{\kappa}'[\tau_-'/\alpha] : \Box$
   - by TQCOERCE*, $\Gamma, \alpha{:}\kappa \vdash \tau_1 \equiv \tau_1' : \tilde{\kappa}$
   - by definition of Substitution Equivalence, $\Gamma, \alpha{:}\kappa \vdash [\tau_1/\alpha'] \equiv [\tau_1'/\alpha'] : \Gamma, \alpha{:}\kappa, \alpha'{:}\tilde{\kappa}$

4–6. Analogously.

□

Lemma 58 (Kind Adaption)**.** *Provided the rules* TCOERCE*, TQCOERCE*, TQCOERCE-DROP* *and* TQCOERCE-CANCEL* *hold for kinds of size* $\mathrm{Size}(\tilde{\kappa}_1)$ *and* $\mathrm{Size}(\tilde{\kappa}_2)$*, and* $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$ *and* $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$ *and* $\Gamma \vdash \tau_+ : \kappa$ *and* $\Gamma \vdash \tau_- : \kappa$*, then:*

1. *Let* $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$ *and* $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$*. Then:*
   a) $\Gamma_1 \vdash \Box$ *and* $\Gamma_1, \alpha{:}\kappa \vdash \Box$
   b) $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
   c) $\Gamma_1 \vdash \tilde{\kappa}_2'[\tau_+/\alpha] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
2. *Let* $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha]$*. If* $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$*, then:*
   a) $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
   b) $\Gamma_1 \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \leq \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$

Proof.

1. - by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \Box$
   - by inverting NTYPE, $\alpha \notin \mathrm{Dom}(\Gamma)$
   - obviously, $\Gamma \vdash [\tau_+/\alpha] : \Gamma, \alpha{:}\kappa$
   - by Substitutability, $\Gamma \vdash \tilde{\kappa}_1[\tau_+/\alpha] : \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha] \vdash \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
   - by NTYPE, $\Gamma_1 \vdash \Box$
   - by Weakening, $\Gamma_1 \vdash \tau_+ : \kappa$
   - by Validity, $\Gamma_1 \vdash \kappa : \Box$
   - by NTYPE, $\Gamma_1, \alpha{:}\kappa \vdash \Box$
   - by Weakening, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
   - by TVAR, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
   - let $\tau_1 = \{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}$
   - by Kind Adaptive Substitution (4), $\Gamma_1, \alpha{:}\kappa \vdash [\tau_1/\alpha_1] : \Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1$
   - by Substitutability, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
   - obviously, $\tilde{\kappa}_2'[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+}/\alpha_1]$
   - by Kind Adaptive Substitution (5), $\Gamma_1 \vdash [\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+}/\alpha_1] \equiv [\alpha_1/\alpha_1] : \Gamma_1$

- by Simple Functionality, $\Gamma_1 \vdash \tilde{\kappa}_2[\tau_+/\alpha][\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+}/\alpha_1] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Box$

2.
   - by TCOERCE*, $\Gamma \vdash \tau_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
   - let $\tau_1' = \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}$
   - by Kind Adaptive Substitution (1), $\Gamma, \alpha{:}\kappa \vdash [\tau_1'/\alpha_1] : \Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1$
   - by Substitutability, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
   - obviously, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash [\tau_-/\alpha] : \Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1$
   - by Substitutability, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash \tilde{\kappa}_2[\tau_-/\alpha] : \Box$
   - by Kind Adaptive Substitution (2),
     $\Gamma \vdash [\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-}/\alpha_1] \equiv [\tau{\cdot}1/\alpha_1] : \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha]$
   - by Simple Functionality,
     $\Gamma \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_2[\tau_-/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-}/\alpha_1] : \Box$
   - obviously, $\tilde{\kappa}_2'[\tau_-/\alpha] = \tilde{\kappa}_2[\tau_-/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-}/\alpha_1]$

□

With these lemmas we can prove actual correctness of the kind coercion rules, and thus the definition of kind coercions themselves:

Theorem 59 (Admissibility of Kind Coercion Rules)**.**
The rules TCOERCE*, TQCOERCE*, TQCOERCE-DROP* *and* TQCOERCE-CANCEL* *are admissible.*

Proof. By simultaneous induction on Size($\tilde{\kappa}$):

1. TCOERCE*:

   case $\alpha \notin \mathrm{FV}(\tilde{\kappa})$: trivial

   case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau')$:
   - by inverting KSING, $\Gamma, \alpha{:}\kappa \vdash \tau' : \Omega$
   - obviously, $\Gamma \vdash [\tau_+/\alpha] : \Gamma, \alpha{:}\kappa$
   - by Substitutability, $\Gamma \vdash \tau'[\tau_+/\alpha] : \Omega$
   - by TEXT-SING, $\Gamma \vdash \tau'[\tau_+/\alpha] : \mathrm{S}_\Omega(\tau'[\tau_+/\alpha])$

   case $\tilde{\kappa} = \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$
   - by inverting KPI, $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
   - by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
   - let $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
   - by Kind Adaption (1a) with induction, $\Gamma_1 \vdash \Box$ and $\Gamma_1, \alpha{:}\kappa \vdash \Box$
   - by Weakening, $\Gamma_1 \vdash \tau_+ : \kappa$ and $\Gamma_1 \vdash \tau_- : \kappa$ and $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
   - let $\tau_1 = \{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}$
   - by TVAR, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
   - by induction, $\Gamma_1 \vdash \tau_1 : \tilde{\kappa}_1[\tau_-/\alpha]$
   - by Weakening, $\Gamma_1 \vdash \tau : (\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
   - by TAPP, $\Gamma_1 \vdash \tau\,\tau_1 : \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1]$
   - let $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$
   - obviously, $\tilde{\kappa}_2'[\tau_-/\alpha] = \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1]$
   - by Reflexivity, $\Gamma_1 \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1] \equiv \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$
   - by Antisymmetry and TSUB, $\Gamma_1 \vdash \tau\,\tau_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$
   - let $\tau_2 = \{\tau\,\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+/\tau_-}$

- by Kind Adaption (1b) with induction, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
- by induction, $\Gamma_1 \vdash \tau_2 : \tilde{\kappa}_2'[\tau_+/\alpha]$
- by Kind Adaption (1c) with induction, $\Gamma_1 \vdash \tilde{\kappa}_2'[\tau_+/\alpha] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
- by Antisymmetry and Tsub, $\Gamma_1 \vdash \tau_2 : \tilde{\kappa}_2[\tau_+/\alpha]$
- by Tlambda, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau_2 : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$

case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- by Tfst, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by inverting Ksigma, $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- let $\tau_1 = \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}$
- by induction, $\Gamma \vdash \tau_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by Tsnd, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- let $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$
- by Kind Adaption (2b) with induction, $\Gamma \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$
- by Antisymmetry and Tsub, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- let $\tau_2 = \{\tau{\cdot}2 : \tilde{\kappa}_2'\}_{\tau_+/\tau_-}$
- by Kind Adaption (2a) with induction, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
- by induction, $\Gamma \vdash \tau_2 : \tilde{\kappa}_2'[\tau_+/\alpha]$
- obviously, $\tilde{\kappa}_2'[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}/\alpha_1] = \tilde{\kappa}_2[\tau_+/\alpha][\tau_1/\alpha_1]$
- by Tpair, $\Gamma \vdash \langle\tau_1, \tau_2\rangle : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$

2. TQcoerce*:

case $\alpha \notin \mathrm{FV}(\tilde{\kappa})$: trivial

case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau'')$:

- by inverting KQsing, $\tilde{\kappa}' = \mathrm{S}_\Omega(\tau''')$ and $\Gamma \vdash \tau''[\tau_+/\alpha] \equiv \tau'''[\tau_+/\alpha] : \Omega$
- by Validity, $\Gamma \vdash \tau''[\tau_+/\alpha] : \Omega$ and $\Gamma \vdash \tau'''[\tau_+/\alpha] : \Omega$
- by Text-sing, $\Gamma \vdash \tau''[\tau_+/\alpha] : \mathrm{S}_\Omega(\tau''[\tau_+/\alpha])$ and $\Gamma \vdash \tau'''[\tau_+/\alpha] : \mathrm{S}_\Omega(\tau'''[\tau_+/\alpha])$
- by Symmetry and KSsing, $\Gamma \vdash \mathrm{S}_\Omega(\tau'''[\tau_+/\alpha]) \leq \mathrm{S}_\Omega(\tau''[\tau_+/\alpha]) : \Box$
- by Tsub, $\Gamma \vdash \tau'''[\tau_+/\alpha] : \mathrm{S}(\tau''[\tau_+/\alpha])$
- by TQext-sing, $\Gamma \vdash \tau''[\tau_+/\alpha] \equiv \tau'''[\tau_+/\alpha] : \mathrm{S}_\Omega(\tau''[\tau_+/\alpha])$

case $\tilde{\kappa} = \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- by inverting KQpi, $\tilde{\kappa}' = \Pi\alpha_1{:}\tilde{\kappa}_1'.\tilde{\kappa}_2'$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_+/\alpha] \equiv \tilde{\kappa}_1'[\tau_+'/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_-/\alpha] \equiv \tilde{\kappa}_1'[\tau_-'/\alpha] : \Box$
- by inverting Kpi, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- let $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
- by Kind Adaption (1a) with induction, $\Gamma_1 \vdash \Box$ and $\Gamma_1, \alpha{:}\kappa \vdash \Box$
- by Weakening, $\Gamma_1 \vdash \tau_+ : \kappa$ and $\Gamma_1 \vdash \tau_- : \kappa$ and $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- let $\tau_1 = \{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}$ and $\tau_1' = \{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1'\}_{\tau_-'/\tau_+'}$
- by TQvar, $\Gamma_1 \vdash \alpha_1 \equiv \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by induction, $\Gamma_1 \vdash \tau_1 \equiv \tau_1' : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Weakening, $\Gamma_1 \vdash \tau \equiv \tau' : (\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by TQapp, $\Gamma_1 \vdash \tau\,\tau_1 \equiv \tau'\,\tau_1' : \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1]$
- let $\tilde{\kappa}_3 = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$ and $\tilde{\kappa}_3' = \tilde{\kappa}_2'[\{\alpha_1 : \alpha{:}\kappa'.\tilde{\kappa}_1'\}_{\alpha/\tau_+'}/\alpha_1]$
- by Kind Adaptive Substitution (6) with induction, $\Gamma_1 \vdash [\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1] \equiv [\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1'\}_{\alpha/\tau_+'}/\alpha_1] : \Gamma_1$
- by Full Functionality, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_3 \equiv \tilde{\kappa}_3' : \Box$

- by Validity, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_3 : \Box$ and $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}'_3 : \Box$
- obviously, $\Gamma_1 \vdash [\tau_+/\alpha] \equiv [\tau'_+/\alpha] : \Gamma_1, \alpha{:}\kappa$ and $\Gamma_1 \vdash [\tau_-/\alpha] \equiv [\tau'_-/\alpha] : \Gamma_1, \alpha{:}\kappa$
- by Full Functionality, $\Gamma_1 \vdash \tilde{\kappa}_3[\tau_+/\alpha] \equiv \tilde{\kappa}'_3[\tau'_+/\alpha] : \Box$ and $\Gamma_1 \vdash \tilde{\kappa}_3[\tau_-/\alpha] \equiv \tilde{\kappa}'_3[\tau'_-/\alpha] : \Box$
- by Reflexivity, $\Gamma_1 \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1] \equiv \tilde{\kappa}_3[\tau_-/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma_1 \vdash \tau\,\tau_1 \equiv \tau'\,\tau'_1 : \tilde{\kappa}_3[\tau_-/\alpha]$
- let $\tau_2 = \{\tau\,\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_3\}_{\tau_+/\tau_-}$ and $\tau'_2 = \{\tau'\,\tau'_1 : \alpha{:}\kappa'.\tilde{\kappa}'_3\}_{\tau_+/\tau_-}$
- by Kind Adaption (1b) with induction, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_3 : \Box$ and $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}'_3 : \Box$
- by induction, $\Gamma_1 \vdash \tau_2 \equiv \tau'_2 : \tilde{\kappa}_3[\tau_+/\alpha]$
- by Kind Adaption (1c) with induction, $\Gamma_1 \vdash \tilde{\kappa}_3[\tau_+/\alpha] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma_1 \vdash \tau_2 \equiv \tau'_2 : \tilde{\kappa}_2[\tau_+/\alpha]$
- by TQLAMBDA, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau_2 \equiv \lambda\alpha{:}\tilde{\kappa}'_1[\tau'_+/\alpha].\tau'_2 : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$

case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- by inverting KQSIGMA, $\tilde{\kappa}' = \Sigma\alpha_1{:}\tilde{\kappa}'_1.\tilde{\kappa}'_2$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_+/\alpha] \equiv \tilde{\kappa}'_1[\tau'_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_-/\alpha] \equiv \tilde{\kappa}'_1[\tau'_-/\alpha] : \Box$
- by TQFST, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by inverting KSIGMA, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- let $\tau_1 = \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}$ and $\tau'_1 = \{\tau'{\cdot}1 : \alpha{:}\kappa'.\tilde{\kappa}'_1\}_{\tau'_+/\tau'_-}$
- by induction, $\Gamma \vdash \tau_1 \equiv \tau'_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by TQSND, $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- let $\tilde{\kappa}_3 = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$ and $\tilde{\kappa}'_3 = \tilde{\kappa}'_2[\{\tau'{\cdot}1 : \alpha{:}\kappa'.\tilde{\kappa}'_1\}_{\alpha/\tau'_-}/\alpha_1]$
- by Kind Adaptive Substitution (3) with induction, $\Gamma_1 \vdash [\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1] \equiv [\{\tau'{\cdot}1 : \alpha{:}\kappa'.\tilde{\kappa}'_1\}_{\alpha/\tau'_-}/\alpha_1] : \Gamma_1$
- by Full Functionality, $\Gamma_1 \vdash \tilde{\kappa}_3 \equiv \tilde{\kappa}'_3 : \Box$
- by Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_3 : \Box$ and $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}'_3 : \Box$
- obviously, $\Gamma \vdash [\tau_+/\alpha] \equiv [\tau'_+/\alpha] : \Gamma, \alpha{:}\kappa$ and $\Gamma \vdash [\tau_-/\alpha] \equiv [\tau'_-/\alpha] : \Gamma, \alpha{:}\kappa$
- by Full Functionality, $\Gamma \vdash \tilde{\kappa}_3[\tau_+/\alpha] \equiv \tilde{\kappa}'_3[\tau'_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_3[\tau_-/\alpha] \equiv \tilde{\kappa}'_3[\tau'_-/\alpha] : \Box$
- by Kind Adaption (2b) with induction, $\Gamma \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_3[\tau_-/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \tilde{\kappa}'_2[\tau_-/\alpha]$
- let $\tau_2 = \{\tau{\cdot}2 : \alpha{:}\kappa.\tilde{\kappa}_3\}_{\tau_+/\tau_-}$ and $\tau'_2 = \{\tau'{\cdot}2 : \alpha{:}\kappa'.\tilde{\kappa}'_3\}_{\tau'_+/\tau'_-}$
- by induction, $\Gamma \vdash \tau_2 \equiv \tau'_2 : \tilde{\kappa}_3[\tau_+/\alpha]$
- obviously, $\tilde{\kappa}_3[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}/\alpha_1] = \tilde{\kappa}_2[\tau_+/\alpha][\tau_1/\alpha_1]$
- by TQPAIR, $\Gamma \vdash \langle\tau_1, \tau_2\rangle \equiv \langle\tau'_1, \tau'_2\rangle : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$

3. TQCOERCE-DROP*:

case $\alpha \notin \mathrm{FV}(\tilde{\kappa})$:

- by Reflexivity, $\Gamma \vdash \tau \equiv \tau : \tilde{\kappa}$

case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau')$:

- by Validity, $\Gamma \vdash \tau_+ : \kappa$ and $\Gamma \vdash \tau_- : \kappa$
- by TCOERCE*, $\Gamma \vdash \{\tau : \alpha{:}\kappa.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-} : \mathrm{S}_\Omega(\tau')[\tau_+/\alpha]$
- by Environment Validity, $\Gamma \vdash \Box$
- obviously, $\Gamma \vdash [\tau_+/\alpha] \equiv [\tau_-/\alpha] : \Gamma, \alpha{:}\kappa$
- by Simple Functionality, $\Gamma \vdash \mathrm{S}_\Omega(\tau')[\tau_+/\alpha] \leq \mathrm{S}_\Omega(\tau')[\tau_-/\alpha] : \Box$
- by TSUB, $\Gamma \vdash \{\tau : \alpha{:}\kappa.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-} : \mathrm{S}_\Omega(\tau')[\tau_-/\alpha]$
- by TQEXT-SING, $\Gamma \vdash \{\tau : \alpha{:}\kappa.\mathrm{S}_\Omega(\tau')\}_{\tau_+/\tau_-} \equiv \tau : \mathrm{S}_\Omega(\tau')[\tau_-/\alpha]$

case $\tilde{\kappa} = \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- by Validity, $\Gamma \vdash \tau_+ : \kappa$ and $\Gamma \vdash \tau_- : \kappa$
- by inverting KPI, $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- let $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
- by Kind Adaption (1a) with induction, $\Gamma_1 \vdash \Box$ and $\Gamma_1, \alpha{:}\kappa \vdash \Box$
- by Weakening, $\Gamma_1 \vdash \tau_+ \equiv \tau_- : \kappa$ and $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- by Symmetry, $\Gamma_1 \vdash \tau_- \equiv \tau_+ : \kappa$
- let $\tau_1 = \{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}$
- by TVAR, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by induction, $\Gamma_1 \vdash \tau_1 \equiv \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- obviously, $\Gamma_1 \vdash [\tau_+/\alpha] \equiv [\tau_-/\alpha] : \Gamma_1, \alpha{:}\kappa$
- by Simple Functionality, $\Gamma_1 \vdash \tilde{\kappa}_1[\tau_+/\alpha] \leq \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- by TQSUB, $\Gamma_1 \vdash \tau_1 \equiv \alpha_1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Weakening, $\Gamma_1 \vdash \tau : (\Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by Reflexivity, $\Gamma_1 \vdash \tau \equiv \tau : \Pi\alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha].\tilde{\kappa}_2[\tau_-/\alpha]$
- by TQAPP, $\Gamma_1 \vdash \tau\,\tau_1 \equiv \tau\,\alpha_1 : \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1]$
- let $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$
- by Reflexivity, $\Gamma_1 \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1] \equiv \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma_1 \vdash \tau\,\tau_1 \equiv \tau\,\alpha_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- let $\tau_2 = \{\tau\,\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+\approx\tau_-}$ and $\tau_2' = \{\tau\,\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_+\approx\tau_-}$
- by Validity, $\Gamma_1 \vdash \tau\,\tau_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$ and $\Gamma_1 \vdash \tau\,\alpha_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Kind Adaption (1b) with induction, $\Gamma_1, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
- by induction, $\Gamma_1 \vdash \tau_2 \equiv \tau\,\tau_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$ and $\Gamma_1 \vdash \tau_2' \equiv \tau\,\alpha_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Symmetry and Transitivity, $\Gamma_1 \vdash \tau_2 \equiv \tau\,\alpha_1 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Simple Functionality, $\Gamma_1 \vdash \tilde{\kappa}_2'[\tau_-/\alpha] \leq \tilde{\kappa}_2'[\tau_+/\alpha] : \Box$
- by TQSUB, $\Gamma_1 \vdash \tau_2 \equiv \tau\,\alpha_1 : \tilde{\kappa}_2'[\tau_+/\alpha]$
- by Kind Adaption (1d) with induction, $\Gamma_1 \vdash \tilde{\kappa}_2'[\tau_+/\alpha] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma_1 \vdash \tau_2 \equiv \tau\,\alpha_1 : \tilde{\kappa}_2[\tau_+/\alpha]$
- by TQLAMBDA, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau_2 \equiv \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau\,\alpha_1 : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$
- by Simple Functionality, $\Gamma \vdash (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha] \leq (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha] : \Box$
- by TQSUB, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau_2 \equiv \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau\,\alpha_1 : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by TQLAMBDA-ETA*, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau\,\alpha_1 \equiv \tau : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by Transitivity, $\Gamma \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].\tau_2 \equiv \tau : (\Pi\alpha{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$

case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- by Validity, $\Gamma \vdash \tau_+ : \kappa$ and $\Gamma \vdash \tau_- : \kappa$
- obviously, $\Gamma \vdash [\tau_+/\alpha] \equiv [\tau_-/\alpha] : \Gamma, \alpha{:}\kappa$
- by TFST, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by inverting KSIGMA, $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- let $\tau_1 = \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}$
- by induction, $\Gamma \vdash \tau_1 \equiv \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Validity, $\Gamma \vdash \tau_1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by TSND, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- let $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$
- by Kind Adaption (2b) with induction, $\Gamma \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$
- by Antisymmetry and TSUB, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- let $\tau_2 = \{\tau{\cdot}2 : \tilde{\kappa}_2'\}_{\tau_+/\tau_-}$

- by Kind Adaption (2a) with induction, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_2' : \Box$
- by induction, $\Gamma \vdash \tau_2 \equiv \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Substitutability, $\Gamma \vdash \tilde{\kappa}_2'[\tau_-/\alpha] \leq \tilde{\kappa}_2'[\tau_+/\alpha] : \Box$
- obviously, $\tilde{\kappa}_2'[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}/\alpha_1] = \tilde{\kappa}_2[\tau_+/\alpha][\tau_1/\alpha_1]$
- obviously, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash [\tau_-/\alpha] \equiv \tilde{\kappa}_2[\tau_+/\alpha] : \Gamma, \alpha{:}\kappa, \alpha{:}\tilde{\kappa}_1$
- by Simple Functionality, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash \tilde{\kappa}_2[\tau_+/\alpha] \leq \tilde{\kappa}_2[\tau_-/\alpha] : \Box$
- obviously, $\Gamma \vdash [\tau_1/\alpha_1] : \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha]$
- by Substitutability, $\Gamma \vdash \tilde{\kappa}_2[\tau_+/\alpha][\tau_1/\alpha_1] \leq \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1] : \Box$
- by Transitivity, $\Gamma \vdash \tilde{\kappa}_2'[\tau_-/\alpha] \leq \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1] : \Box$
- TQSUB, $\Gamma \vdash \tau_2 \equiv \tau{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau_1/\alpha_1]$
- by TQPAIR, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \equiv \langle \tau{\cdot}1, \tau{\cdot}2 \rangle : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by TQPAIR-ETA*, $\Gamma \vdash \langle \tau{\cdot}1, \tau{\cdot}2 \rangle \equiv \tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by Transitivity, $\Gamma \vdash \langle \tau_1, \tau_2 \rangle \equiv \tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$

4. TQCOERCE-CANCEL*:

case $\alpha \notin \mathrm{FV}(\tilde{\kappa})$:

- by Reflexivity, $\Gamma \vdash \tau \equiv \tau : \tilde{\kappa}$

case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau')$:

- by Validity, $\Gamma \vdash \mathrm{S}_\Omega(\tau')[\tau_-/\alpha] : \Box$
- by inverting KSING, $\Gamma \vdash \tau'[\tau_-/\alpha] : \Omega$
- by TEXT-SING, $\Gamma \vdash \tau'[\tau_-/\alpha] : \mathrm{S}_\Omega(\tau')[\tau_-/\alpha]$
- by TQEXT-SING, $\Gamma \vdash \tau'[\tau_-/\alpha] \equiv \tau : \mathrm{S}_\Omega(\tau')[\tau_-/\alpha]$

case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$

- let $\tau_1 = \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_-}$ and $\tau_2 = \{\tau{\cdot}2 : \tilde{\kappa}_2'\}_{\tau_+/\tau_-}$ and $\tau' = \langle \tau_1, \tau_2 \rangle$ with $\tilde{\kappa}_2' = \tilde{\kappa}_2[\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$ and
- let $\tau_1' = \{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}$ and $\tau_2' = \{\tau'{\cdot}2 : \tilde{\kappa}_2''\}_{\tau_-/\tau_+}$ and $\tau'' = \langle \tau_1', \tau_2' \rangle$ with $\tilde{\kappa}_2'' = \tilde{\kappa}_2[\{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$
- by TCOERCE*, $\Gamma \vdash \tau' : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_+/\alpha]$
- by TQFST-BETA*, $\Gamma \vdash \tau'{\cdot}1 \equiv \tau_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by TQSND-BETA*, $\Gamma \vdash \tau'{\cdot}2 \equiv \tau_2 : \tilde{\kappa}_2[\tau_+/\alpha][\tau'{\cdot}1/\alpha_1]$
- by inverting KSIGMA, $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- by Environment Validity, $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$
- obviously, $\Gamma \vdash [\tau_+/\alpha] : \Gamma, \alpha{:}\kappa$ and $\Gamma \vdash [\tau_-/\alpha] : \Gamma, \alpha{:}\kappa$
- by Substitutability, $\Gamma \vdash \tilde{\kappa}_1[\tau_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- by Reflexivity, $\Gamma \vdash \tilde{\kappa}_1[\tau_+/\alpha] \equiv \tilde{\kappa}_1[\tau_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_1[\tau_-/\alpha] \equiv \tilde{\kappa}_1[\tau_-/\alpha] : \Box$ and $\Gamma \vdash \tau_+ \equiv \tau_+ : \kappa$ and $\Gamma \vdash \tau_- \equiv \tau_- : \kappa$
- by TQCOERCE*, $\Gamma \vdash \tau_1' \equiv \{\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \tilde{\kappa}_1[\tau_-/\alpha]$
- by TFST, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by induction, $\Gamma \vdash \{\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} \equiv \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Transitivity, $\Gamma \vdash \tau_1' \equiv \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Validity, $\Gamma \vdash \tau'{\cdot}1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by Kind Adaption (2b) with induction (inverted signs), $\Gamma \vdash \tilde{\kappa}_2[\tau_+/\alpha][\tau'{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_2''[\tau_+/\alpha] : \Box$
- by Antisymmetry and TQSUB, $\Gamma \vdash \tau'{\cdot}2 \equiv \tau_2 : \tilde{\kappa}_2''[\tau_+/\alpha]$
- by Kind Adaption (2a) with induction (inverted signs), $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_2'' : \Box$
- by Substitutability, $\Gamma \vdash \tilde{\kappa}_2''[\tau_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_2''[\tau_-/\alpha] : \Box$

- by Reflexivity, $\Gamma \vdash \tilde{\kappa}_2''[\tau_+/\alpha] \equiv \tilde{\kappa}_2''[\tau_+/\alpha] : \Box$ and $\Gamma \vdash \tilde{\kappa}_2''[\tau_-/\alpha] \equiv \tilde{\kappa}_2''[\tau_-/\alpha] : \Box$
- by TQCOERCE*, $\Gamma \vdash \tau_2' \equiv \{\tau_2 : \alpha{:}\kappa.\tilde{\kappa}_2''\}_{\tau_-/\tau_+} : \tilde{\kappa}_2''[\tau_-/\alpha]$
- obviously, $\tilde{\kappa}_2'[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\tau_1/\alpha_1]$ and $\tilde{\kappa}_2''[\tau_+/\alpha] = \tilde{\kappa}_2[\tau_+/\alpha][\{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+}/\alpha_1]$
- obviously, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha] \vdash [\tau_+/\alpha] : \Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1$
- by Substitutability, $\Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha] \vdash \tilde{\kappa}_2[\tau_+/\alpha] : \Box$
- by TQCOERCE-DROP*, $\Gamma \vdash \{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+} \equiv \tau'{\cdot}1 : \tilde{\kappa}_1[\tau_+/\alpha] : \Box$
- by Transitivity, $\Gamma \vdash \{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+} \equiv \tau_1 : \tilde{\kappa}_1[\tau_+/\alpha] : \Box$
- obviously, $\Gamma \vdash [\{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_+/\tau_+}/\alpha_1] \equiv [\tau_1/\alpha_1] : \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
- by Simple Functionality, $\Gamma \vdash \tilde{\kappa}_2'[\tau_+/\alpha] \equiv \tilde{\kappa}_2''[\tau_+/\alpha] : \Box$
- obviously, $\tilde{\kappa}_2'[\tau_-/\alpha] = \tilde{\kappa}_2[\tau_-/\alpha][\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-}/\alpha_1]$ and $\tilde{\kappa}_2''[\tau_-/\alpha] = \tilde{\kappa}_2[\tau_-/\alpha][\{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}/\alpha_1]$
- by TQCOERCE-DROP*, $\Gamma \vdash \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-} \equiv \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- by TQCOERCE*, $\Gamma \vdash \{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} \equiv \{\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- by induction, $\Gamma \vdash \{\tau_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} \equiv \tau{\cdot}1 : \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- by Symmetry and Transitivity, $\Gamma \vdash \{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-} \equiv \{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \tilde{\kappa}_1[\tau_-/\alpha] : \Box$
- obviously, $\Gamma \vdash [\{\tau{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_-+}/\alpha_1] \equiv [\{\tau'{\cdot}1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\tau_-/\tau_+}/\alpha_1] : \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
- by Simple Functionality, $\Gamma \vdash \tilde{\kappa}_2'[\tau_-/\alpha] \equiv \tilde{\kappa}_2''[\tau_-/\alpha] : \Box$
- by TSND, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- by Kind Adaption (2b) with induction, $\Gamma \vdash \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}_2'[\tau_-/\alpha] : \Box$
- by Antisymmetry and TSUB, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by induction, $\Gamma \vdash \{\tau_2 : \tilde{\kappa}_2'\}_{\tau_-/\tau_+} \equiv \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by TCOERCE*, $\Gamma \vdash \tau_2 : \tilde{\kappa}_2'[\tau_+/\alpha]$
- by Reflexivity, $\Gamma \vdash \tau_2 \equiv \tau_2 : \tilde{\kappa}_2'[\tau_+/\alpha]$
- by TQCOERCE*, $\Gamma \vdash \{\tau_2 : \alpha{:}\kappa.\tilde{\kappa}_2'\}_{\tau_-/\tau_+} \equiv \tau_2' : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Symmetry and Transitivity, $\Gamma \vdash \tau_2' \equiv \tau{\cdot}2 : \tilde{\kappa}_2'[\tau_-/\alpha]$
- by Antisymmetry and TQSUB, $\Gamma \vdash \tau_2' \equiv \tau{\cdot}2 : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- by Symmetry, $\Gamma \vdash \tau{\cdot}1 \equiv \tau_1' : \tilde{\kappa}_1[\tau_-/\alpha]$ and $\Gamma \vdash \tau{\cdot}2 \equiv \tau_2' : \tilde{\kappa}_2[\tau_-/\alpha][\tau{\cdot}1/\alpha_1]$
- by TQPAIR, $\Gamma \vdash \langle\tau{\cdot}1, \tau{\cdot}2\rangle \equiv \tau'' : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by TQPAIR-ETA*, $\Gamma \vdash \langle\tau{\cdot}1, \tau{\cdot}2\rangle \equiv \tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$
- by Symmetry and Transitivity, $\Gamma \vdash \tau'' \equiv \tau : (\Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2)[\tau_-/\alpha]$

□

E.2. Abstraction Kinds

The definition of higher-order abstraction kinds mostly follows higher-order singleton kinds, so the proofs of admissibility are similar as well. However, to deal with the coercion in the Σ-case we state a technical lemma:

Lemma 60 (Second Component Conversion)**.**

1. *If* $\Gamma \vdash \tau : \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$*, then* $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \{\tau{\cdot}2 : \alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\alpha_1/\tau{\cdot}1} : \tilde{\kappa}_2$.

2. *If* $\Gamma \vdash \tau \equiv \tau' : \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$ *and* $\Gamma \vdash \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 \equiv \Sigma\alpha_1{:}\tilde{\kappa}'_1.\tilde{\kappa}'_2 : \Box$,
then $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \{\tau{\cdot}2 : \alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\alpha_1/\tau\cdot 1} \equiv \{\tau'{\cdot}2 : \alpha_1{:}\tilde{\kappa}'_1.\tilde{\kappa}'_2\}_{\alpha_1/\tau'\cdot 1} : \tilde{\kappa}_2$.

Proof.

1.
- by Validity, $\Gamma \vdash \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$
- by inverting KSIGMA, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- let $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1$
- by Environment Validity, $\Gamma_1 \vdash \Box$ and $\Gamma \vdash \tilde{\kappa}_1 \vdash \Box$ and $\Gamma \vdash \Box$
- by TFST, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1$
- by TSND, $\Gamma \vdash \tau{\cdot}2 : \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1]$
- by Validity, $\Gamma \vdash \tilde{\kappa}_1 : \Box$
- by Weakening, $\Gamma_1 \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma_1 \vdash \tau{\cdot}1 : \tilde{\kappa}_1$ and $\Gamma_1 \vdash \tau{\cdot}2 : \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1]$
- w.l.o.g., $\alpha'_1 \notin \mathrm{Dom}(\Gamma_1)$
- by Renaming, $\Gamma, \alpha'_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2[\alpha'_1/\alpha_1] : \Box$
- by Environment Validity, $\Gamma_1, \alpha'_1{:}\tilde{\kappa}_1 \vdash \Box$
- by Weakening, $\Gamma_1, \alpha'_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2[\alpha'_1/\alpha_1] : \Box$
- obviously, $\tilde{\kappa}_2[\tau{\cdot}1/\alpha_1] = \tilde{\kappa}_2[\alpha'_1/\alpha_1][\tau{\cdot}1/\alpha'_1]$
- by TVAR, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1$
- by TCOERCE*, $\Gamma_1 \vdash \{\tau{\cdot}2 : \alpha'_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2[\alpha'_1/\alpha_1]\}_{\alpha_1/\tau\cdot 1} : \tilde{\kappa}_2[\alpha'_1/\alpha_1][\alpha_1/\alpha'_1]$

2.
- by inverting KQSIGMA, $\Gamma \vdash \tilde{\kappa}_1 \equiv \tilde{\kappa}'_1 \vdash \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 \equiv \tilde{\kappa}'_2 : \Box$
- let $\Gamma_1 = \Gamma, \alpha_1{:}\mathrm{K}(\tau{\cdot}1 : \tilde{\kappa}_1)$
- by Environment Validity, $\Gamma \vdash \Box$ and $\Gamma_1 \vdash \Box$
- by TQFST, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \tilde{\kappa}_1$
- by TQSND, $\Gamma \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1]$
- by Validity, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}'_2 : \Box$
- by Weakening, $\Gamma_1 \vdash \tilde{\kappa}_1 \equiv \tilde{\kappa}'_1 : \Box$ and $\Gamma_1 \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \tilde{\kappa}_1$ and $\Gamma_1 \vdash \tau{\cdot}2 \equiv \tau'{\cdot}2 : \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1]$
- w.l.o.g., $\alpha'_1 \notin \mathrm{Dom}(\Gamma_1)$
- by Renaming,
$\Gamma, \alpha'_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2[\alpha'_1/\alpha_1] : \Box$ and $\Gamma, \alpha'_1{:}\tilde{\kappa}'_1 \vdash \tilde{\kappa}'_2[\alpha'_1/\alpha_1] : \Box$
- by Environment Validity, $\Gamma_1, \alpha'_1{:}\tilde{\kappa}_1 \vdash \Box$ and $\Gamma_1, \alpha'_1{:}\tilde{\kappa}'_1 \vdash \Box$
- by Weakening, $\Gamma_1, \alpha'_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2[\alpha'_1/\alpha_1] : \Box$ and $\Gamma_1, \alpha'_1{:}\tilde{\kappa}'_1 \vdash \tilde{\kappa}'_2[\alpha'_1/\alpha_1] : \Box$
- obviously, $\tilde{\kappa}_2[\tau{\cdot}1/\alpha_1] = \tilde{\kappa}_2[\alpha'_1/\alpha_1][\tau{\cdot}1/\alpha'_1]$ and $\tilde{\kappa}'_2[\tau'{\cdot}1/\alpha_1] = \tilde{\kappa}'_2[\alpha'_1/\alpha_1][\tau'{\cdot}1/\alpha'_1]$
- obviously, $\Gamma \vdash [\tau{\cdot}1/\alpha_1] \equiv [\tau'{\cdot}1/\alpha_1] : \Gamma, \alpha_1{:}\tilde{\kappa}_1$
- by Full Functionality, $\Gamma \vdash \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}'_2[\tau'{\cdot}1/\alpha_1] : \Box$
- by Weakening, $\Gamma_1 \vdash \tilde{\kappa}_2[\tau{\cdot}1/\alpha_1] \equiv \tilde{\kappa}'_2[\tau'{\cdot}1/\alpha_1] : \Box$
- by TQVAR, $\Gamma_1 \vdash \alpha_1 \equiv \alpha_1 : \tilde{\kappa}_1$
- by TQCOERCE*,
$\Gamma_1 \vdash \{\tau{\cdot}2 : \alpha'_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2[\alpha'_1/\alpha_1]\}_{\alpha_1/\tau\cdot 1} \equiv \{\tau'{\cdot}2 : \alpha'_1{:}\tilde{\kappa}'_1.\tilde{\kappa}'_2[\alpha'_1/\alpha_1]\}_{\alpha_1/\tau'\cdot 1} : \tilde{\kappa}_2[\alpha'_1/\alpha_1][\alpha_1/\alpha'_1]$

□

Theorem 61 (Admissibility of Higher-Order Abstraction Kind Rules)**.**
The rules KABS*, KQABS*, KSABS* *and* KSABS-LEFT* *are admissible.*

Proof. By simultaneous induction on $\mathrm{Size}(\tilde{\kappa})$.

1. KQABS*:

   case $\tilde{\kappa} = \Omega$:
   - by inverting KQOMEGA, $\tilde{\kappa}' = \Omega$
   - by KQABS, $\Gamma \vdash \mathrm{A}(\tau) \equiv \mathrm{A}(\tau') : \Box$

   case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau_1)$:
   - by inverting KQSING, $\tilde{\kappa}' = \mathrm{S}_\Omega(\tau_1')$

   case $\tilde{\kappa} = \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$:
   - by Validity, $\Gamma \vdash \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$
   - by inverting KQPI, $\tilde{\kappa}' = \Pi\alpha_1{:}\tilde{\kappa}_1'.\tilde{\kappa}_2'$ and $\Gamma \vdash \tilde{\kappa}_1 \equiv \tilde{\kappa}_1' : \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 \equiv \tilde{\kappa}_2' : \Box$
   - by Environment Validity, $\Gamma \vdash \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \Box$
   - by Weakening, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau \equiv \tau' : \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$
   - by TQVAR, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \alpha_1 \equiv \alpha_1 : \tilde{\kappa}_1$
   - by TQAPP, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau\,\alpha_1 \equiv \tau'\,\alpha_1 : \tilde{\kappa}_2$
   - by induction, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \mathrm{A}(\tau\,\alpha_1 : \tilde{\kappa}_2) \equiv \mathrm{A}(\tau'\,\alpha_1 : \tilde{\kappa}_2') : \Box$
   - by KQPI, $\Gamma \vdash \Pi\alpha_1{:}\tilde{\kappa}_1.\mathrm{A}(\tau\,\alpha_1 : \tilde{\kappa}_2) \equiv \Pi\alpha_1{:}\tilde{\kappa}_1'.\mathrm{A}(\tau'\,\alpha_1 : \tilde{\kappa}_2') : \Box$

   case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$:
   - by Validity, $\Gamma \vdash \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$
   - by inverting KQSIGMA,
     $\tilde{\kappa}' = \Sigma\alpha_1{:}\tilde{\kappa}_1'.\tilde{\kappa}_2'$ and $\Gamma \vdash \tilde{\kappa}_1 \equiv \tilde{\kappa}_1' : \Box$ and $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 \equiv \tilde{\kappa}_2' : \Box$
   - by TQFST, $\Gamma \vdash \tau{\cdot}1 \equiv \tau'{\cdot}1 : \tilde{\kappa}_1$
   - by induction, $\Gamma \vdash \mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1) \equiv \mathrm{A}(\tau'{\cdot}1 : \tilde{\kappa}_1') : \Box$
   - let $\tau_2 = \{\tau{\cdot}2 : \alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\alpha_1/\tau{\cdot}1}$ and $\tau_2' = \{\tau'{\cdot}2 : \alpha_1{:}\tilde{\kappa}_1'.\tilde{\kappa}_2'\}_{\alpha_1/\tau'{\cdot}1}$
   - by Second Component Conversion, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau_2 \equiv \tau_2' : \tilde{\kappa}_2$
   - by Validity, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1$ and $\Gamma \vdash \mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1) : \Box$
   - by induction (KSABS-LEFT*), $\Gamma \vdash \mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1) \leq \tilde{\kappa}_1 : \Box$
   - let $\Gamma_1 = \Gamma, \alpha_1{:}\mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1)$
   - by NTYPE, $\Gamma_1 \vdash \Box$
   - by Weakening, $\Gamma_1 \vdash \tau_2 \equiv \tau_2' : \tilde{\kappa}_2$ and $\Gamma_1 \vdash \tilde{\kappa}_2 \equiv \tilde{\kappa}_2' : \Box$
   - by induction, $\Gamma_1 \vdash \mathrm{A}(\tau_2 : \tilde{\kappa}_2) \equiv \mathrm{A}(\tau_2' : \tilde{\kappa}_2') : \Box$
   - by KQSIGMA, $\Gamma \vdash \Sigma\alpha_1{:}\mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1).\mathrm{A}(\tau_2 : \tilde{\kappa}_2) \equiv \Sigma\alpha_1{:}\mathrm{A}(\tau'{\cdot}1 : \tilde{\kappa}_1').\mathrm{A}(\tau_2' : \tilde{\kappa}_2') : \Box$

2. KABS*:

   - by Reflexivity, $\Gamma \vdash \tau \equiv \tau : \tilde{\kappa}$ and $\Gamma \vdash \tilde{\kappa} \equiv \tilde{\kappa} : \Box$
   - by KQABS*, $\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \equiv \mathrm{A}(\tau : \tilde{\kappa}) : \Box$
   - by Validity, $\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) : \Box$

3. KSABS*:

   - by KQABS*, $\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \equiv \mathrm{A}(\tau' : \tilde{\kappa}') : \Box$
   - by Antisymmetry, $\Gamma \vdash \mathrm{A}(\tau : \tilde{\kappa}) \leq \mathrm{A}(\tau' : \tilde{\kappa}') : \Box$

4. KSABS-LEFT*:

   case $\tilde{\kappa} = \Omega$:
   - by KSABS-LEFT, $\Gamma \vdash \mathrm{A}(\tau) \leq \Omega : \Box$

   case $\tilde{\kappa} = \mathrm{S}_\Omega(\tau_1)$:
   - by Validity, $\Gamma \vdash \mathrm{S}_\Omega(\tau_1) : \Box$
   - by Reflexivity, $\Gamma \vdash \mathrm{S}_\Omega(\tau_1) \leq \mathrm{S}_\Omega(\tau_1) : \Box$

case $\tilde{\kappa} = \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$:

- by Validity, $\Gamma \vdash \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$
- by inverting KPI, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\kappa}_2 : \Box$
- by Environment Validity, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \Box$ and $\Gamma \vdash \tilde{\kappa}_1 : \Box$
- by Weakening, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau : \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$
- by TVAR, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \alpha_1 : \tilde{\kappa}_1$
- by TAPP, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau\,\alpha_1 : \tilde{\kappa}_2$
- by induction, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \mathrm{A}(\tau\,\alpha_1 : \tilde{\kappa}_2) \leq \tilde{\kappa}_2 : \Box$
- by Reflexivity, $\Gamma \vdash \tilde{\kappa}_1 \leq \tilde{\kappa}_1 : \Box$
- by KSPI, $\Gamma \vdash \Pi\alpha_1{:}\tilde{\kappa}_1.\mathrm{A}(\tau\,\alpha_1 : \tilde{\kappa}_2) \leq \Pi\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$

case $\tilde{\kappa} = \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2$:

- by TFST, $\Gamma \vdash \tau{\cdot}1 : \tilde{\kappa}_1$
- by induction, $\Gamma \vdash \mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1) \leq \tilde{\kappa}_1 : \Box$
- let $\tau_2 = \{\tau{\cdot}2 : \alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2\}_{\alpha_1/\tau\cdot 1}$
- by Second Component Conversion, $\Gamma, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau_2 : \tilde{\kappa}_2$
- let $\Gamma_1 = \Gamma, \alpha_1{:}\mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1)$
- by NTYPE, $\Gamma_1 \vdash \Box$
- by Weakening, $\Gamma_1 \vdash \tau_2 : \tilde{\kappa}_2$
- by induction, $\Gamma_1 \vdash \mathrm{A}(\tau_2 : \tilde{\kappa}_2) \leq \tilde{\kappa}_2 : \Box$
- by KSSIGMA, $\Gamma \vdash \Sigma\alpha_1{:}\mathrm{A}(\tau{\cdot}1 : \tilde{\kappa}_1).\mathrm{A}(\tau_2 : \tilde{\kappa}_2) \leq \Sigma\alpha_1{:}\tilde{\kappa}_1.\tilde{\kappa}_2 : \Box$

□

E.3. Higher-Order Generativity and Type Coercions

E.3.1. Declarative Properties

Unsurprisingly, all interesting properties still hold for the extended calculus:

Proposition 62 (Declarative Properties with Higher-Order Abstraction)**.**
All declarative properties from Section C.1.1 still hold in the presence of rules ENEW', EUP' *and* EDN'.

Proof. Straightforward extensions of the respective proofs with the new cases. □

Proposition 63 (Validity with Higher-Order Abstraction)**.**
If $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \tau : \Omega$.

Proof. By straightforward extension of the original proof. The new cases are easy. For example:

case EDN':

- by inversion, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ and $\Gamma \vdash \tau_{\_} : \tilde{\kappa}$
- obviously, $\Gamma \vdash [\tau_{\_}/\alpha] : \Gamma, \alpha{:}\tilde{\kappa}$
- by Substitutability, $\Gamma \vdash \tilde{\tau}[\tau_{\_}/\alpha] : \Omega$

□

The inversion principle on Term Validity has to be adapted to the changed rules:

Proposition 64 (Inversion for Higher-Order Abstraction)**.** *Let* $\Gamma \vdash e : \tau$.

10. *If* $e = \mathsf{new}\,\alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}_{\tau_2}\ e_2$, *then* $\Gamma \vdash \tilde{\tau}_1 : \tilde{\kappa}_1$ *and* $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \tau_2 \leq \tau : \Omega$.

11. *If* $e = \{e_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-}$, *then* $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ *and* $\Gamma \vdash e_1 : \tilde{\tau}[\tau_-/\alpha]$ *with* $\Gamma \vdash \tau_- : \tilde{\kappa}$ *and* $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ *and* $\Gamma \vdash \tau_2[\tau_+/\alpha] \leq \tau : \Omega$.

12. *If* $e = \{e_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tau_+ \approx \tau_-}$, *then* $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ *and* $\Gamma \vdash e_1 : \tilde{\tau}[\tau_+/\alpha]$ *with* $\Gamma \vdash \tau_- : \tilde{\kappa}$ *and* $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ *and* $\Gamma \vdash \tau_2[\tau_-/\alpha] \leq \tau : \Omega$.

Proof. Each by easy induction on the derivation. For example:

11. $e = \{e_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-}$

    case EUP': $\tau = \tilde{\tau}[\tau_+/\alpha]$
    - by inversion, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ and $\Gamma \vdash e_1 : \tilde{\tau}[\tau_-/\alpha]$ with $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma \vdash \tau_- : \tilde{\kappa}$
    - by KSABS-LEFT*, $\Gamma \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) \leq \tilde{\kappa} : \Box$
    - by TSUB, $\Gamma \vdash \tau_+ : \tilde{\kappa}$
    - obviously, $\Gamma \vdash [\tau_+/\alpha] : \Gamma, \alpha{:}\tilde{\kappa}$
    - by Substitutability, $\Gamma \vdash \tilde{\tau}[\tau_+/\alpha] : \Omega$
    - by Reflexivity, $\Gamma \vdash \tilde{\tau}[\tau_+/\alpha] \leq \tilde{\tau}[\tau_+/\alpha] : \Omega$

    case ESUB:
    - by inversion, $\Gamma \vdash e : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
    - by induction, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ and $\Gamma \vdash e_1 : \tilde{\tau}[\tau_-/\alpha]$ with $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma \vdash \tau_- : \tilde{\kappa}$ and $\Gamma \vdash \tilde{\tau}[\tau_+/\alpha] \leq \tau' : \Omega$
    - by Transititvity, $\Gamma \vdash \tilde{\tau}[\tau_+/\alpha] \leq \tau : \Omega$

□

E.3.2. Algorithmic Type Synthesis

Treatment of the algorithm for type synthesis is straightforward.

Theorem 65 (Soundness of Algorithmic Type Synthesis with Higher-Order Abstraction)**.** *Let* $\Gamma \vdash \Box$.

1. *If* $\Gamma \triangleright e \rightrightarrows \tau$, *then* $\Gamma \vdash e : \tau$.
2. *If* $\Gamma \triangleright e \rightrightarrows \pi$, *then* $\Gamma \vdash e : \pi$.
3. *If* $\Gamma \triangleright e \leftleftarrows \tau$ *with* $\Gamma \vdash \tau : \Omega$, *then* $\Gamma \vdash e : \tau$.

Proof. As before. The modified cases are:

1. case $e = \mathsf{new}\,\alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}_{\tau_2}\ e_2$
   - by inversion, $\Gamma \triangleright \tilde{\kappa}_1 : \Box$ and $\Gamma \triangleright \tilde{\tau}_1 \leftleftarrows \tilde{\kappa}_1$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$ and $\Gamma, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1) \triangleright e \leftleftarrows \tau_2$
   - by Soundness of Kind Checking, $\Gamma \vdash \tilde{\kappa}_1 : \Box$
   - by Soundness of Kind Analysis, $\Gamma \vdash \tilde{\tau}_1 : \tilde{\kappa}_1$ and $\Gamma \vdash \tau_2 : \Omega$
   - by KABS*, $\Gamma \vdash \mathrm{A}(\tau_1 : \tilde{\kappa}_1) : \Box$
   - w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma)$
   - by NTYPE, $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash \Box$
   - by Weakening, $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash \tau_2 : \Omega$
   - by induction (3), $\Gamma, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1) \vdash e : \tau_2$

– by ENEW', $\Gamma \vdash (\mathsf{new}\,\alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}_{\tau_2}\ e_2) : \tau_2$

case $e = \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-}$

– by inversion, $\Gamma \triangleright \tilde{\kappa} : \Box$ and $\Gamma \triangleright \tau_- \leftleftarrows \tilde{\kappa}$ and $\Gamma \triangleright \tau_+ \leftleftarrows \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \leftleftarrows \Omega$ and $\Gamma \triangleright e \leftleftarrows \tilde{\tau}[\tau_-/\alpha]$

– by Soundness of Kind Checking, $\Gamma \vdash \tilde{\kappa} : \Box$

– by Soundness of Kind Analysis, $\Gamma \vdash \tau_- : \tilde{\kappa}$

– by KABS*, $\Gamma \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) : \Box$

– by Soundness of Kind Analysis, $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$

– w.l.o.g., $\alpha \notin \mathrm{Dom}(\Gamma)$

– by NTYPE, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \Box$

– by KOMEGA, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \Omega : \Box$

– by Soundness of Kind Analysis, $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$

– obviously, $\Gamma \vdash [\tau_-/\alpha] : \Gamma, \alpha{:}\tilde{\kappa}$

– by Substitutability, $\Gamma \vdash \tilde{\tau}[\tau_-/\alpha] : \Omega$

– by induction (3), $\Gamma \vdash e : \tilde{\tau}[\tau_-/\alpha]$

– by EUP', $\Gamma \vdash \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{+}_{\tau_+ \approx \tau_-} : \tilde{\tau}[\tau_+/\alpha]$

case $e = \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{-}_{\tau_+ \approx \tau_-}$: analogous

□

Theorem 66 (Completeness of Algorithmic Type Synthesis with Higher-Order Abstraction)**.**

1. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \rightrightarrows \tau'$ *with* $\Gamma \vdash \tau' \leq \tau : \Omega$.
2. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \triangleright e \rightrightarrows\!\!\!\!\!\!\rightarrow\ \pi$ *with* $\Gamma \vdash \pi \leq \tau : \Omega$.
3. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leq \tau' : \Omega$, *then* $\Gamma \triangleright e \leftleftarrows \tau'$.

Proof. As before. The modified cases are:

1. case ENEW':

– by inversion, $\Gamma \vdash \tilde{\tau}_1 : \tilde{\kappa}_1$ and $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash e : \tau_2$ and $\Gamma \vdash \tau_2 : \Omega$

– by Environment Validity, $\Gamma \vdash \Box$ and $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash \Box$

– by Validity, $\Gamma \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma \vdash \Omega : \Box$

– by Completeness of Kind Checking, $\Gamma \triangleright \tilde{\kappa}_1 : \Box$

– by Completeness of Kind Analysis, $\Gamma \triangleright \tilde{\tau}_1 \leftleftarrows \tilde{\kappa}_1$ and $\Gamma \triangleright \tau_2 \leftleftarrows \Omega$

– by Weakening, $\Gamma, \alpha{:}\mathrm{A}(\tau_1 : \tilde{\kappa}_1) \vdash \tau_2 : \Omega$

– by induction (3), $\Gamma, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1) \triangleright e \leftleftarrows \tau_2$

– by rule, $\Gamma \triangleright (\mathsf{new}\,\alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}_{\tau_2}\ e_2) \rightrightarrows \tau_2$

case EUP':

– by inversion, $\Gamma \vdash e : \tilde{\tau}[\tau_-/\alpha]$ and $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ and $\Gamma \vdash \tau_- : \tilde{\kappa}$ and $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$

– by Environment Validity, $\Gamma \vdash \Box$ and $\Gamma, \alpha{:}\tilde{\kappa} \vdash \Box$

– by Validity, $\Gamma \vdash \tilde{\tau}[\tau_-/\alpha] : \Box$ and $\Gamma, \alpha{:}\tilde{\kappa} \vdash \Omega : \Box$ and $\Gamma \vdash \tilde{\kappa} : \Box$ and $\Gamma \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) : \Box$

– by Completeness of Kind Checking, $\Gamma \triangleright \tilde{\kappa} : \Box$

- by Completeness of Kind Analysis, $\Gamma \triangleright \tau_- \leftleftarrows \tilde{\kappa}$ and $\Gamma \triangleright \tau_+ \leftleftarrows \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \leftleftarrows \Omega$
- by induction (3), $\Gamma \triangleright e \leftleftarrows \tilde{\tau}[\tau_-/\alpha]$
- by rule, $\Gamma \triangleright \{e : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^+_{\tau_+ \approx \tau_-} \rightrightarrows \tilde{\tau}[\tau_+/\alpha]$

case Edn': analogous

□

E.3.3. Path Replacement

Before we can show soundness, we have to show admissibility of the path replacement rules from Figure 13.12. We start with two simple lemmas.

Lemma 67 (Path Decomposition)**.**
If $\Gamma \vdash P[\tau] : \kappa$*, then* $\Gamma \vdash \tau : \kappa'$*.*

Proof. By straightforward induction on the structure of P. □

Lemma 68 (Path Replacement Substitutability)**.**
$\gamma(P[\tau_+/\tau_- : \kappa]) = (\gamma P)[\gamma\tau_+/\gamma\tau_- : \gamma\kappa]$

Proof. By straightforward induction on the structure of P and κ. □

The actual rules have to be proved by induction following the definition of path coercions. That requires strenghtened induction hypotheses. The following three lemmata capture the essentials.

Lemma 69 (Inductive Path Replacement Reversal)**.**
Let $\Gamma \vdash \tau : \tilde{\kappa}$*, and w.l.o.g., assume* $\alpha' \notin \mathrm{Dom}(\Gamma)$ *(and hence,* $\alpha' \notin \mathrm{FV}(\tau, P)$ *for* P *below).*

1. *If the following conditions hold:*

   a) $\Gamma, \alpha'{:}\tilde{\kappa} \triangleright \tau' \rightrightarrows \kappa''$ *with* $\Gamma, \alpha'{:}\tilde{\kappa} \vdash \kappa'' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau/\alpha] : \Box$

   b) $\Gamma \triangleright \tau'[\tau/\alpha'] \rightrightarrows \kappa'''$ *with* $\Gamma \vdash \kappa''' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau/\alpha'][\tau/\alpha] : \Box$

   c) $\Gamma \vdash P[\tau'][\tau/\alpha'] : \Omega$ *and* $\alpha \notin \mathrm{FV}(P, \tau')$

   Then $\Gamma \vdash P[\alpha'{:}\tilde{\kappa}.\,\tau' : \tilde{\kappa}']_{\alpha/\tau}[\tau/\alpha] \equiv P[\tau'][\tau/\alpha'] : \Omega$*.*

2. *If the following conditions hold:*

   a) $\Gamma, \alpha'{:}\tilde{\kappa} \triangleright \tau'[\tau/\alpha] \rightrightarrows \kappa''$ *with* $\Gamma, \alpha'{:}\tilde{\kappa} \vdash \kappa'' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau/\alpha] : \Box$

   b) $\Gamma \triangleright \tau'[\tau/\alpha'][\tau/\alpha] \rightrightarrows \kappa'''$ *with* $\Gamma \vdash \kappa''' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau/\alpha'][\tau/\alpha] : \Box$

   c) $\Gamma \vdash P[\tau'][\tau/\alpha'][\tau/\alpha] : \Omega$ *and* $\alpha \notin \mathrm{FV}(\tau')$

   Then $\Gamma \vdash P[\alpha'{:}\tilde{\kappa}.\,\tau' : \tilde{\kappa}']_{\tau/\alpha}[\tau/\alpha] \equiv P[\tau'][\tau/\alpha'][\tau/\alpha] : \Omega$*.*

Proof. Each by induction on the structure of P, inside out. Note that the premises use algorithmic kind synthesis to require that $\tilde{\kappa}'$ is the most specific kind that can be assigned to τ' modulo selfification. This is needed in the case of application, to show that the argument type matches the parameter kind in $\tilde{\kappa}'$. Also note that we repeat the same premises with α' substituted. This is necessary because we need both statements, but the algorithmic judgements are not closed under substitution. □

Lemma 70 (Inductive Path Grounding)**.**
Let $\Gamma \vdash \tau_- : \tilde{\kappa}$ *and* $\Gamma \vdash \tau_+ : \tilde{\kappa}$*. W.l.o.g., assume* $\alpha \notin \mathrm{Dom}(\Gamma)$ *(and hence,* $\alpha \notin \mathrm{FV}(\tau_-, \tau_+, P)$*). If the following conditions hold:*

1. $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tau' \rightrightarrows \kappa''$ *with* $\Gamma, \alpha{:}\tilde{\kappa} \vdash \kappa'' \equiv \mathrm{S}(\tau' : \tilde{\kappa}') : \Box$
2. $\Gamma \triangleright \tau'[\tau_+/\alpha] \rightrightarrows \kappa'''$ *with* $\Gamma \vdash \kappa''' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau_+/\alpha] : \Box$
3. $\Gamma \vdash P[\tau_-] : \Omega$
4. $\Gamma, \alpha{:}\tilde{\kappa} \vdash \tau' : \tilde{\kappa}'$

Then $\Gamma \vdash P[\alpha{:}\tilde{\kappa}.\, \tau' : \tilde{\kappa}']_{\tau_+/\tau_-} : \Omega$*.*

Proof. By induction on the structure of P, inside out. □

Lemma 71 (Inductive Path Abstraction)**.**
Let $\Gamma \vdash \tau : \tilde{\kappa}$ *and* $\Gamma \vdash \tau_- : \tilde{\kappa}$ *and* $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$*. W.l.o.g., assume* $\alpha \notin \mathrm{Dom}(\Gamma)$ *(and hence,* $\alpha \notin \mathrm{FV}(\tau, \tau_-, \tau_+, P)$*). If the following conditions hold:*

1. $\Gamma, \alpha{:}\tilde{\kappa} \triangleright \tau' \rightrightarrows \kappa''$ *with* $\Gamma, \alpha{:}\tilde{\kappa} \vdash \kappa'' \equiv \mathrm{S}(\tau' : \tilde{\kappa}') : \Box$
2. $\Gamma \triangleright \tau'[\tau_-/\alpha] \rightrightarrows \kappa'''$ *with* $\Gamma \vdash \kappa''' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau_-/\alpha] : \Box$
3. $\Gamma \triangleright \tau'[\tau_+/\alpha] \rightrightarrows \kappa'''$ *with* $\Gamma \vdash \kappa''' \equiv \mathrm{S}(\tau' : \tilde{\kappa}')[\tau_+/\alpha] : \Box$
4. $\Gamma \vdash P[\tau] : \Omega$
5. $\Gamma \vdash \tau'[\tau_+/\alpha] : \mathrm{A}(\{\tau'[\tau_-/\alpha] : \alpha{:}\tilde{\kappa}.\tilde{\kappa}'\}_{\tau_+ \approx \tau_-} : \tilde{\kappa}'[\tau_+/\alpha])$

Then $\Gamma \vdash P[\alpha{:}\tilde{\kappa}.\, \tau' : \tilde{\kappa}']_{\tau_+/\tau} : \mathrm{A}(P[\alpha{:}\tilde{\kappa}.\, \tau' : \tilde{\kappa}']_{\tau_-/\tau})$*.*

Proof. By induction on the structure of P, inside out. □

Theorem 72 (Admissibility of Path Coercion Rules)**.**
The rules TREPLACE*, TREPLACE-ABS*, TQREPLACE-SUBST-DN* *and* TQREPLACE-SUBST-DN* *are admissible.*

Proof.

1. TREPLACE*: corollary of Inductive Path Grounding.
2. TREPLACE-ABS*: corollary of Inductive Path Abstraction.
3. TREPLACE-SUBST-UP*: corollary of Inductive Path Replacement Reversal (1).
4. TREPLACE-SUBST-DN*: corollary of Inductive Path Replacement Reversal (2).

□

E.3.4. Preservation

For term-level coercions we use a technical lemma similar to the one employed for type-level coercions (Section E.1), that treats the substitution in the residual type of coercions at quantified types:

Lemma 73 (Type Adaption)**.**
If $\Gamma, \alpha{:}\kappa \vdash \tilde{\kappa}_1 : \Box$ *and* $\Gamma, \alpha{:}\kappa, \alpha_1{:}\tilde{\kappa}_1 \vdash \tau_2 : \Omega$ *and* $\Gamma \vdash \tau_+ : \kappa$ *and* $\Gamma \vdash \tau_- : \kappa$*, then:*

1. *Let* $\tau_2' = \tau_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha]$ *and* $\Gamma_1 = \Gamma, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$*. Then:*
   a) $\Gamma_1 \vdash \Box$ *and* $\Gamma_1, \alpha{:}\kappa \vdash \Box$
   b) $\Gamma_1, \alpha{:}\kappa \vdash \tau_2' : \Omega$
   c) $\Gamma_1 \vdash \tau_2'[\tau_+/\alpha] \equiv \tau_2[\tau_+/\alpha] : \Omega$
2. *Let* $\tau_2' = \tau_2[\{\alpha_1 : \alpha{:}\kappa.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha]$*. If* $\Gamma \vdash \alpha_1 : \tilde{\kappa}_1[\tau_-/\alpha]$*, then:*
   a) $\Gamma_1, \alpha{:}\kappa \vdash \tau_2' : \Omega$
   b) $\Gamma_1 \vdash \tau_2[\tau_-/\alpha] \leq \tau_2'[\tau_-/\alpha] : \Omega$

Proof. Analogous to Kind Adaption. □

Decomposition has to be adapted to the change in syntax:

Lemma 74 (Decomposition and Replacement with Higher-Order Abstraction)**.**
If $\Gamma \vdash E[e] : \tau$*, then* $\Gamma \vdash e : \tau'$*, and if* $\Gamma' \vdash e' : \tau'$ *with* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash E[e'] : \tau$*.*

Proof. By straightforward extension of the original induction, using Inversion for Higher-Order Abstraction. □

Theorem 75 (Preservation with Higher-Order Abstraction)**.**

1. *If* $\Delta \vdash e : \tau$ *and* $\Delta; e \to \Delta'; e'$ *with* $E = \_$*, then* $\Delta' \vdash e' : \tau$*.*
2. *If* $\cdot \vdash C : \tau$ *and* $C \to C'$*, then* $\cdot \vdash C' : \tau$*.*

Proof. The new cases are as follows. For coercions, we only show the positive case, the negative always is analogous.

1. Case analysis:

   case RNEW': $e = \mathsf{new}\,\alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}_{\tau'}\ e_2$ and $e' = e_2$ and $\Delta' = \Delta, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1)$
   - by Inversion (10), $\Delta, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1) \vdash e_2 : \tau'$ and $\Delta \vdash \tau' \leq \tau : \Omega$
   - by ESUB, $\Delta, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1) \vdash e_2 : \tau$

   case RCOERCE-NORM: $e = \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^+_{\tau_+ \approx \tau_-}$ and $e' = \{v : \alpha{:}\tilde{\kappa}.\tilde{\pi}\}^+_{\tau_+ \approx \tau_-}$ and $\Delta, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \Rrightarrow \tilde{\pi}$
   - by Inversion (11), $\Delta \vdash \tilde{\tau}[\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$ and $\Delta \vdash v : \tilde{\tau}[\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
   - by Soundness of Type Comparison, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} \equiv \tilde{\pi} : \Omega$
   - by Validity, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\pi} : \Omega$
   - by KSABS-LEFT*, $\Delta \vdash \mathrm{A}(\tau_- : \kappa) \leq \kappa : \Box$
   - by TSUB, $\Delta \vdash \tau_+ : \kappa$
   - obviously, $\Delta \vdash [\tau_+/\alpha] : \Delta, \alpha{:}\tilde{\kappa}$ and $\Delta \vdash [\tau_-/\alpha] : \Delta, \alpha{:}\tilde{\kappa}$

- by Substitutability, $\Delta \vdash \tilde{\tau}[\tau_-/\alpha] \equiv \tilde{\pi}[\tau_-/\alpha] : \Omega$ and $\Delta \vdash \tilde{\tau}[\tau_+/\alpha] \equiv \tilde{\pi}[\tau_+/\alpha] : \Omega$
- by Antisymmetry, $\Delta \vdash \tilde{\tau}[\tau_-/\alpha] \leq \tilde{\pi}[\tau_-/\alpha] : \Omega$ and $\Delta \vdash \tilde{\pi}[\tau_+/\alpha] \leq \tilde{\tau}[\tau_+/\alpha] : \Omega$
- by Tsub, $\Delta \vdash v : \tilde{\pi}[\tau_-/\alpha]$
- by Eup', $\Delta \vdash \{v : \alpha{:}\tilde{\kappa}.\tilde{\pi}\}^+_{\tau_+ \approx \tau_-} : \tilde{\pi}[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \{v : \alpha{:}\tilde{\kappa}.\tilde{\pi}\}^+_{\tau_+ \approx \tau_-} : \tilde{\tau}[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \{v : \alpha{:}\tilde{\kappa}.\tilde{\pi}\}^+_{\tau_+ \approx \tau_-} : \tau$

case Rcoerce-psi: $e = \{v : \alpha{:}\tilde{\kappa}.\Psi\}^+_{\tau_+ \approx \tau_-}$ and $e' = v$

- by Inversion (11), $\Delta \vdash \Psi \leq \tau : \Omega$ and $\Delta \vdash v : \Psi$
- by Tsub, $\Delta \vdash v : \tau$

case Rcoerce-arrow: $e = \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1 \to \tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
and $e' = \lambda x_1{:}\tilde{\tau}_1[\tau_+/\alpha].\{v\,\{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^-_{\tau_+ \approx \tau_-} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$

- by Inversion (11), $\Delta \vdash (\tilde{\tau}_1 \to \tilde{\tau}_2)[\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_1 \to \tilde{\tau}_2 : \Omega$ and $\Delta \vdash v : (\tilde{\tau}_1 \to \tilde{\tau}_2)[\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Type Validity Inversion, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_1 : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_2 : \Omega$
- by KSabs-left*, $\Delta \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) \leq \tilde{\kappa} : \Box$
- by Tsub, $\Delta \vdash \tau_+ : \tilde{\kappa}$
- obviously, $\Delta \vdash [\tau_+/\alpha] : \Delta, \alpha{:}\tilde{\kappa}$
- by Substitutability, $\Delta \vdash \tilde{\tau}_1[\tau_+/\alpha] : \Omega$
- let $\Gamma_1 = \Delta, x_1{:}\tilde{\tau}_1[\tau_+/\alpha]$
- by Nterm, $\Gamma_1 \vdash \Box$
- by Evar, $\Gamma_1 \vdash x_1 : \tilde{\tau}_1[\tau_+/\alpha]$
- let $e_1 = \{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^-_{\tau_+ \approx \tau_-}$
- by Edn', $\Gamma_1 \vdash e_1 : \tilde{\tau}_1[\tau_-/\alpha]$
- by Weakening, $\Gamma_1 \vdash v : (\tilde{\tau}_1 \to \tilde{\tau}_2)[\tau_-/\alpha]$
- by Eapp, $\Gamma_1 \vdash v\,e_1 : \tilde{\tau}_2[\tau_-/\alpha]$
- let $e_2 = \{v\,e_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
- by Eup', $\Gamma_1 \vdash e_2 : \tilde{\tau}_2[\tau_+/\alpha]$
- by Elambda, $\Delta \vdash \lambda x_1{:}\tilde{\tau}_1[\tau_+/\alpha].e_2 : (\tilde{\tau}_1 \to \tilde{\tau}_2)[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \lambda x_1{:}\tilde{\tau}_1[\tau_+/\alpha].e_2 : \tau$

case Rcoerce-times: $e = \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1 \times \tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
and $e' = \mathsf{let}\langle x_1, x_2\rangle = v\ \mathsf{in}\ \langle\{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^+_{\tau_+ \approx \tau_-}, \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}\rangle$

- by Inversion (11), $\Delta \vdash (\tilde{\tau}_1 \times \tilde{\tau}_2)[\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_1 \times \tilde{\tau}_2 : \Omega$ and $\Delta \vdash v : (\tilde{\tau}_1 \times \tilde{\tau}_2)[\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Type Validity Inversion, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_1 : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_2 : \Omega$
- obviously, $\Delta \vdash [\tau_-/\alpha] : \Delta, \alpha{:}\tilde{\kappa}$
- by Substitutability, $\Delta \vdash \tilde{\tau}_1[\tau_-/\alpha] : \Omega$ and $\Delta \vdash \tilde{\tau}_2[\tau_-/\alpha] : \Omega$
- let $\Gamma_1 = \Delta, x_1{:}\tilde{\tau}_1[\tau_-/\alpha], x_2{:}\tilde{\tau}_2[\tau_-/\alpha]$
- by Nterm, $\Gamma_1 \vdash \Box$
- by Evar, $\Gamma_1 \vdash x_1 : \tilde{\tau}_1[\tau_-/\alpha]$ and $\Gamma_1 \vdash x_2 : \tilde{\tau}_2[\tau_-/\alpha]$
- let $e_1 = \{x_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_1\}^+_{\tau_+ \approx \tau_-}$ and $e_2 = \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
- by Eup', $\Gamma_1 \vdash e_1 : \tilde{\tau}_1[\tau_+/\alpha]$ and $\Gamma_1 \vdash e_2 : \tilde{\tau}_2[\tau_+/\alpha]$
- by Eproj, $\Delta \vdash \mathsf{let}\langle x_1, x_2\rangle = v\ \mathsf{in}\ \langle e_1, e_2\rangle : (\tilde{\tau}_1 \times \tilde{\tau}_2)[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \mathsf{let}\langle x_1, x_2\rangle = v\ \mathsf{in}\ \langle e_1, e_2\rangle : \tau$

case Rcoerce-univ: $e = \{v : \alpha{:}\tilde{\kappa}.\forall\alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
and $e' = \lambda\alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha].\{v\,\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_-/\tau_+} : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2'\}^+_{\tau_+ \approx \tau_-}$
with $\tilde{\tau}_2' = \tilde{\tau}_2[\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_+}/\alpha_1]$

- by Inversion (11), $\Delta \vdash (\forall \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \forall \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2 : \Omega$ and $\Delta \vdash v : (\forall \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Type Validity Inversion, $\Delta, \alpha{:}\tilde{\kappa}, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\tau}_2 : \Box$
- by Environment Validity, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\kappa}_1 : \Box$
- by KSabs-left*, $\Delta \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) \leq \tilde{\kappa} : \Box$
- by Tsub, $\Delta \vdash \tau_+ : \tilde{\kappa}$
- let $\Gamma_1 = \Delta, \alpha_1{:}\tilde{\kappa}_1[\tau_+/\alpha]$
- by Type Adaption (1a), $\Gamma_1 \vdash \Box$ and $\Gamma_1, \alpha{:}\tilde{\kappa} \vdash \Box$
- by Weakening, $\Gamma_1, \alpha{:}\tilde{\kappa} \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma_1 \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Gamma_1 \vdash \tau_+ : \tilde{\kappa}$
- let $\tilde{\tau}_1 = \{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_-/\tau_+}$
- by Tvar, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by Tcoerce*, $\Gamma_1 \vdash \tilde{\tau}_1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Weakening, $\Gamma_1 \vdash v : (\forall \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_-/\alpha]$
- by Einst, $\Gamma_1 \vdash v\,\tilde{\tau}_1 : \tilde{\tau}_2[\tau_-/\alpha][\tilde{\tau}_1/\alpha_1]$
- obviously, $\tilde{\tau}_2'[\tau_-/\alpha] = \tilde{\tau}_2[\tau_-/\alpha][\tilde{\tau}_1/\alpha_1]$
- by Reflexivity, $\Gamma_1 \vdash \tilde{\tau}_2[\tau_-/\alpha][\tilde{\tau}_1/\alpha_1] \equiv \tilde{\tau}_2'[\tau_-/\alpha] : \Omega$
- by Antisymmetry and Esub, $\Gamma_1 \vdash v\,\tilde{\tau}_1 : \tilde{\tau}_2'[\tau_-/\alpha]$
- let $e_2 = \{v\,\tilde{\tau}_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_2'\}^+_{\tau_+ \approx \tau_-}$
- by Type Adaption (1b), $\Gamma_1, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_2' : \Omega$
- by induction, $\Gamma_1 \vdash e_2 : \tilde{\tau}_2'[\tau_+/\alpha]$
- by Type Adaption (1c), $\Gamma_1 \vdash \tilde{\tau}_2'[\tau_+/\alpha] \equiv \tilde{\tau}_2[\tau_+/\alpha] : \Omega$
- by Antisymmetry and Esub, $\Gamma_1 \vdash e_2 : \tilde{\tau}_2[\tau_+/\alpha]$
- by Egen, $\Delta \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].e_2 : (\forall \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \lambda\alpha{:}\tilde{\kappa}_1[\tau_+/\alpha].e_2 : \tau$

case Rcoerce-exist: $e = \{v : \alpha{:}\tilde{\kappa}.\exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2\}^+_{\tau_+ \approx \tau_-}$
and $e' = \mathsf{let}\langle \alpha_1, x_2\rangle = v \;\mathsf{in}\; \langle \{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_+/\tau_-}, \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2'\}^+_{\tau_+ \approx \tau_-}\rangle$
with $\tilde{\tau}_2' = \tilde{\tau}_2[\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$

- by Inversion (11), $\Delta \vdash (\exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2 : \Omega$ and $\Delta \vdash v : (\exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Type Validity Inversion, $\Delta, \alpha{:}\tilde{\kappa}, \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\tau}_2 : \Box$
- by Environment Validity, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\kappa}_1 : \Box$
- obviously, $\Delta, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash [\tau_-/\alpha] : \Delta, \alpha{:}\tilde{\kappa}, \alpha_1{:}\tilde{\kappa}_1$
- by Substitutability, $\Delta, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha] \vdash \tilde{\tau}_2[\tau_-/\alpha] : \Omega$
- by Environment Validity and inverting Nterm, $\alpha \notin \mathrm{Dom}(\Delta)$
- w.l.o.g., $x_2 \notin \mathrm{Dom}(\Delta)$
- let $\Gamma_1 = \Delta, \alpha_1{:}\tilde{\kappa}_1[\tau_-/\alpha], x_2{:}\tilde{\tau}_2$
- by Nterm, $\Gamma_1 \vdash \Box$
- by Weakening, $\Gamma_1, \alpha{:}\tilde{\kappa} \vdash \tilde{\kappa}_1 : \Box$ and $\Gamma_1 \vdash \tau_+ : \mathrm{A}(\tau_-)$
- by KSabs-left*, $\Gamma_1 \vdash \mathrm{A}(\tau_- : \tilde{\kappa}) \leq \tilde{\kappa} : \Box$
- by Tsub, $\Gamma_1 \vdash \tau_+ : \tilde{\kappa}$
- let $\tilde{\tau}_1 = \{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_+^!/\tau_-}$
- by Tvar, $\Gamma_1 \vdash \alpha_1 : \tilde{\kappa}_1[\tau_-/\alpha]$
- by Tcoerce*, $\Gamma_1 \vdash \tilde{\tau}_1 : \tilde{\kappa}_1[\tau_+/\alpha]$
- by Evar, $\Gamma_1 \vdash x_2 : \tilde{\tau}_2[\tau_-/\alpha]$
- let $\tilde{\tau}_2' = \tilde{\tau}_2[\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\alpha/\tau_-}/\alpha_1]$
- by Type Adaption (2b), $\Gamma_1 \vdash \tilde{\tau}_2[\tau_-/\alpha] \equiv \tilde{\tau}_2'[\tau_-/\alpha] : \Omega$
- by Antisymmetry of Type Inclusion and Esub, $\Gamma_1 \vdash x_2 : \tilde{\tau}_2'[\tau_-/\alpha]$

- let $e_2 = \{x_2 : \alpha{:}\tilde{\kappa}.\tilde{\tau}_2'\}^{+}_{\tau_+ \approx \tau_-}$
- by Type Adaption (2a), $\Gamma_1, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau}_2' : \Omega$
- by Eup', $\Gamma_1 \vdash e_2 : \tilde{\tau}_2'[\tau_+/\alpha]$
- obviously, $\tilde{\tau}_2'[\tau_+/\alpha] = \tilde{\tau}_2[\tau_+/\alpha][\{\alpha_1 : \alpha{:}\tilde{\kappa}.\tilde{\kappa}_1\}_{\tau_+ !/\tau_-}/\alpha_1] = \tilde{\tau}_2[\tau_+/\alpha][\tilde{\tau}_1/\alpha_1]$
- by Eclose, $\Gamma_1 \vdash \langle \tilde{\tau}_1, e_2 \rangle : (\exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_+/\alpha]$
- by Eopen, $\Delta \vdash \mathsf{let}\langle \alpha_1, x_2 \rangle = v \ \mathsf{in} \ \langle \tilde{\tau}_1, e_2 \rangle : (\exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2)[\tau_+/\alpha]$
- by Tsub, $\Delta \vdash \mathsf{let}\langle \alpha_1, x_2 \rangle = v \ \mathsf{in} \ \langle \tilde{\tau}_1, e_2 \rangle : \tau$

case Rcoerce-swap: $e = \{v : \alpha{:}\tilde{\kappa}.P[\alpha']\}^{+}_{\tau_+ \approx \tau_-}$ with $\Delta(\alpha') = \mathrm{A}(\tau_-' : \tilde{\kappa}')$,
and $e' = \{e'' : \alpha''{:}\tilde{\kappa}'.P[\alpha''/\alpha' : \tilde{\kappa}'][\tau_+/\alpha]\}^{+}_{\alpha' \approx \tau_-'}$
with $e'' = \{e''' : \alpha{:}\tilde{\kappa}.P[\tau_-'/\alpha' : \tilde{\kappa}']\}^{+}_{\tau_+ \approx \tau_-}$
and $e''' = \{v : \alpha''{:}\tilde{\kappa}'.P[\alpha''/\alpha' : \tilde{\kappa}'][\tau_-/\alpha]\}^{-}_{\alpha' \approx \tau_-'}$

- by Inversion (11), $\Delta \vdash P[\alpha'][\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash P[\alpha'] : \Omega$ and $\Delta \vdash v : P[\alpha'][\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Validity, $\Delta \vdash P[\alpha'][\tau_-/\alpha] : \Omega$
- by Environment Validity, $\Delta, \alpha{:}\tilde{\kappa} \vdash \Box$
- by inverting Ntype, $\Delta \vdash \tilde{\kappa} : \ \Box$
- by side condition, $\alpha'' \notin \mathrm{Dom}(\Delta) \cup \{\alpha\}$
- by Ntype, $\Delta, \alpha''{:}\tilde{\kappa} \vdash \Box$
- by Weakening, $\Delta, \alpha''{:}\tilde{\kappa} \vdash P[\alpha'][\tau_-/\alpha] : \Omega$
- by Tvar, $\Delta, \alpha''{:}\tilde{\kappa} \vdash \alpha'' : \tilde{\kappa}$
- by Treplace-subst-up*, $\Delta \vdash P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-/\alpha][\alpha'/\alpha''] \equiv P[\alpha'][\tau_-/\alpha] : \Omega$
- by TQsymm, TSequiv & Esub, $\Delta \vdash v : P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-/\alpha][\alpha'/\alpha'']$
- by Edn', $\Delta \vdash e''' : P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-/\alpha][\tau_-'/\alpha'']$
- obviously, $P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-/\alpha][\tau_-'/\alpha''] = P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-'/\alpha''][\tau_-/\alpha] = P[\tau_-'/\alpha' : \tilde{\kappa}][\tau_-/\alpha]$
- by Eup', $\Delta \vdash e'' : P[\tau_-'/\alpha' : \tilde{\kappa}][\tau_+/\alpha]$
- obviously, $P[\tau_-'/\alpha' : \tilde{\kappa}][\tau_+/\alpha] = P[\alpha''/\alpha' : \tilde{\kappa}][\tau_-'/\alpha''][\tau_+/\alpha] = P[\alpha''/\alpha' : \tilde{\kappa}][\tau_+/\alpha][\tau_-'/\alpha'']$
- by Eup', $\Delta \vdash e' : P[\alpha''/\alpha' : \tilde{\kappa}][\tau_+/\alpha][\tau_-'/\alpha'']$
- by Treplace-subst-up*, $\Delta \vdash P[\alpha''/\alpha' : \tilde{\kappa}][\tau_+/\alpha][\tau_-'/\alpha''] \equiv P[\alpha'][\tau_+/\alpha] : \Omega$
- by TSequiv, TStrans & Esub, $\Delta \vdash e' : \tau$

case Rcoerce-ground: $e = \{v : \alpha{:}\tilde{\kappa}.P[\alpha]\}^{+}_{\tau_+ \approx \tau_-}$,
and $e' = \{e'' : \alpha{:}\Omega.\alpha\}^{+}_{P[\tau_+/\alpha:\tilde{\kappa}][\tau_+/\alpha] \approx P[\tau_-/\alpha:\tilde{\kappa}][\tau_+/\alpha]}$
with $e'' = \{v : \alpha{:}\tilde{\kappa}.P[\tau_-/\alpha : \tilde{\kappa}]\}^{+}_{\tau_+ \approx \tau_-}$

- by Inversion (11), $\Delta \vdash P[\alpha][\tau_+/\alpha] \leq \tau : \Omega$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash P[\alpha] : \Omega$ and $\Delta \vdash v : P[\alpha][\tau_-/\alpha]$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$ and $\Delta \vdash \tau_- : \tilde{\kappa}$
- by Treplace-subst-dn*, $\Delta \vdash P[\tau_-/\alpha : \tilde{\kappa}][\tau_-/\alpha] \equiv P[\alpha][\tau_-/\alpha] : \Omega$
- by TQsymm, TSequiv & Esub, $\Delta \vdash v : P[\tau_-/\alpha : \tilde{\kappa}][\tau_-/\alpha]$
- by Eup', $\Delta \vdash e'' : P[\tau_-/\alpha : \tilde{\kappa}][\tau_+/\alpha]$
- by Environment Validity, $\Delta, \alpha{:}\tilde{\kappa} \vdash \Box$
- by Weakening, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tau_- : \tilde{\kappa}$ and $\Delta, \alpha{:}\tilde{\kappa} \vdash \tau_+ : \mathrm{A}(\tau_- : \tilde{\kappa})$
- by Tvar, $\Delta, \alpha{:}\tilde{\kappa} \vdash \alpha : \tilde{\kappa}$
- by Treplace-abs*, $\Delta, \alpha{:}\tilde{\kappa} \vdash P[\tau_+/\alpha : \tilde{\kappa}] : \mathrm{A}([\tau_-/\alpha : \tilde{\kappa}])$
- by Substitutability, $\Delta \vdash P[\tau_+/\alpha : \tilde{\kappa}][\tau_+/\alpha] : \mathrm{A}([\tau_-[\tau_+/\alpha]/\alpha : \tilde{\kappa}])$
- by Eup', $\Delta \vdash e' : P[\tau_+/\alpha : \tilde{\kappa}][\tau_+/\alpha]$
- by Treplace-subst-dn*, $\Delta \vdash P[\tau_+/\alpha : \tilde{\kappa}][\tau_+/\alpha] \equiv P[\alpha][\tau_+/\alpha] : \Omega$
- by TSequiv, TStrans & Esub, $\Delta \vdash e' : \tau$

case Rcoerce-cancel: $e = \{v : \alpha{:}\Omega.\alpha\}^{-}_{\tau_+ \approx \tau_-}$ with $v = \{v' : \alpha{:}\Omega.\alpha\}^{+}_{\tau_+' \approx \tau_-'}$, and $e' = v'$

- by Inversion (12), $\Delta \vdash \tau_- \leq \tau : \Omega$ and $\Delta, \alpha{:}\Omega \vdash \alpha : \Omega$ and $\Delta \vdash v : \tau_+$ and $\Delta \vdash \tau_+ : \mathrm{A}(\tau_-)$ and $\Delta \vdash \tau_- : \Omega$
- by Inversion (11), $\Delta \vdash \tau'_+ \leq \tau_+ : \Omega$ and $\Delta \vdash v' : \tau'_-$ and $\Delta \vdash \tau'_+ : \mathrm{A}(\tau'_-)$ and $\Delta \vdash \tau'_- : \Omega$
- by Representation Equivalence, $\Delta \vdash \tau'_- \equiv \tau_- : \Omega$
- by Antisymmetry of Type Inclusion, $\Delta \vdash \tau'_- \leq \tau_- : \Omega$
- by Transitivity, $\Delta \vdash \tau'_- \leq \tau : \Omega$
- by Esub, $\Delta \vdash v' : \tau$

□

E.3.5. Progress

Lemma 76 (Canonical Values with Higher-Order Abstraction)**.**
Let $\Gamma \vdash v : \tau$. *If* $\Gamma \vdash \tau \leq \tau_+ : \Omega$ *and* $\Gamma \vdash \tau_+ : \mathrm{A}(\tau_-)$, *then* $v = \{v_1 : \alpha{:}\Omega.\alpha\}^+_{\tau'_- \approx \tau'_+}$.

Proof. As before by induction on the derivation of $\Gamma \vdash v : \tau$, using normalisation and Shape Consistency to exclude impossible cases:

case Eup: $v = \{v_1 : \alpha{:}\Omega.\alpha\}^+_{\tau_+ \approx \tau_-}$

case Esub:

- by inversion, $\Gamma \vdash v : \tau'$ and $\Gamma \vdash \tau' \leq \tau : \Omega$
- by Transitivity, $\Gamma \vdash \tau' \leq \tau_+ : \Omega$
- by induction, $v = \{v_1 : \alpha{:}\Omega.\alpha\}^+_{\tau'_+ \approx \tau'_-}$

□

The formulation of the Embedding property does not change, but its proof has to consider the change in syntax:

Lemma 77 (Embedding with Higher-Order Abstraction)**.**
If $\Delta; E[e] \to \Delta, \Delta'; E[e']$, *then* $\Delta_1, \Delta, \Delta_2; E_1 E E_2[e] \to \Delta_1, \Delta, \Delta_2, \Delta'; E_1 E E_2[e']$ *for any contexts* E_1, E_2 *and heaps* Δ_1, Δ_2.

Proof. As before; the changes are trivial. □

Theorem 78 (Progress with Higher-Order Abstraction)**.**
If $\Delta \vdash e : \tau$, *then either* $e = v$, *or* $(\Delta; e) = (\Delta; E[e_1]) \to (\Delta'; E[e'_1]) = (\Delta'; e')$.

Proof. By induction on the structure of e. The new cases are:

case $e = (\mathsf{new}\, \alpha{:}\tilde{\kappa}_1 \approx \tilde{\tau}_1\ \mathsf{in}\ e_2)$:

- let $\Delta' = \Delta, \alpha{:}\mathrm{A}(\tilde{\tau}_1 : \tilde{\kappa}_1)$
- by Rnew', $\Delta; e = \Delta; e \to \Delta'; e_2$

case $e = \{e_1 : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^\pm_{\tau_+ \approx \tau_-}$ with $e \neq v$:

- by Inversion, $\Delta \vdash e_1 : \tilde{\tau}[\tau_\mp/\alpha]$
- by induction, $\Delta; e_1 = \Delta; E'[e_2] \to \Delta; E'[e'_2]$
- let $E = \{E' : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^\pm_{\tau_+ \approx \tau_-}$
- by Embedding, $\Delta; E[e_2] \to \Delta; E[e'_2]$

case $e = \{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{\pm}_{\tau_+ \approx \tau_-}$:

subcase $\tilde{\tau} \neq \tilde{\pi}$, or $\tilde{\tau} = \tilde{\pi}$ and $\Delta \triangleright \tilde{\pi} \Rrightarrow \tilde{\pi}'$ with $\tilde{\pi}' \neq \tilde{\pi}$:

- by Inversion, $\Delta, \alpha{:}\tilde{\kappa} \vdash \tilde{\tau} : \Omega$
- by Completeness of Type Comparison, $\Delta, \alpha{:}\tilde{\kappa} \triangleright \tilde{\tau} \Rrightarrow \tilde{\pi}$
- let $E = \_$
- by RCOERCE-NORM, $\Delta; e = \Delta; E[\{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{\pm}_{\tau_+ \approx \tau_-}] \to \Delta; E[\{v : \alpha{:}\tilde{\kappa}.\tilde{\pi}\}^{\pm}_{\tau_+ \approx \tau_-}]$

subcase $\tilde{\tau} = \Psi$:

- let $E = \_$
- by RCOERCE-PSI, $\Delta; e = \Delta; E[\{v : \alpha{:}\tilde{\kappa}.\tilde{\tau}\}^{\pm}_{\tau_+ \approx \tau_-}] \to \Delta; E[v]$

subcase $\tilde{\tau} = \tilde{\tau}_1 \to \tilde{\tau}_2$: analogous, by RCOERCE-ARROW

subcase $\tilde{\tau} = \tilde{\tau}_1 \times \tilde{\tau}_2$: analogous, by RCOERCE-TIMES

subcase $\tilde{\tau} = \forall \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2$: analogous, by RCOERCE-UNIV

subcase $\tilde{\tau} = \exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2$: analogous, by RCOERCE-EXIST

subcase $\tilde{\tau} = P[\alpha']$ with $\alpha' \neq \alpha$ and and $\Delta \triangleright P[\alpha'] \Rrightarrow P[\alpha']$:

- by Variable Containment, $\alpha' \in \mathrm{Dom}(\Delta)$
- by definition of Δ, $\Delta(\alpha') = \mathrm{A}(\tau'_- : \tilde{\kappa}')$
- let $E = \_$
- reduce by RCOERCE-SWAP

subcase $\tilde{\tau} = P[\alpha]$ and $\Delta \triangleright P[\alpha] \Rrightarrow P[\alpha]$:

subsubcase $P = \_$:

- by Type Synthesis, $\Delta, \alpha{:}\tilde{\kappa} \triangleright \alpha \rightrightarrows \mathrm{S}(\alpha : \tilde{\kappa})$
- by Principality of Type Synthesis, $\Delta, \alpha{:}\Omega \vdash \mathrm{S}(\alpha : \tilde{\kappa}) \leq \Omega : \Box$
- by Shape Consistency for Kind Inclusion, $\tilde{\kappa} = \Omega$ or $\tilde{\kappa} = \mathrm{S}_\Omega(\tau)$
- by assumption, $\Delta \triangleright P[\alpha] \Rrightarrow P[\alpha]$
- hence, $\tilde{\kappa} \neq \mathrm{S}_\Omega(\tau)$
- since e not a value, $\pm = -$
- by Inversion, $\Delta \vdash v : \tau_+$
- by Canonical Values, $v = \{v' : \alpha{:}\Omega.\alpha\}^{+}_{\tau'_+ \approx \tau'_-}$
- let $E = \_$
- by RCOERCE-CANCEL, $\Delta; E[\{v : \alpha{:}\Omega.\alpha\}^{-}_{\tau_+ \approx \tau_-}] \to \Delta; E[v']$

subsubcase $P \neq \_$:

- let $E = \_$
- reduce by RCOERCE-GROUND

□

E.3.6. Opacity

Proposition 79 (Abstractness with Higher-Order Abstraction)**.** *If* $\Gamma = \Gamma_1, \alpha{:}\Omega, x{:}\alpha, \Gamma_2$ *and* $\Gamma \vdash e : \tau$ *with* $\Gamma \vdash \tau \leq \alpha$*, then* e *is not a value.*

Proof. As before. □

Theorem 80 (Opacity with Higher-Order Abstraction)**.**
Let $\Gamma = \alpha{:}\Omega, x{:}\alpha, f{:}\alpha{\to}1$*, and* $\gamma_i = [P[\alpha_i]/\alpha] \cup [v_i/x] \cup [v'_i/f] \cup [\alpha'/\alpha' \mid \alpha' \in \mathrm{Dom}(\Delta')]$ *with* $\alpha \notin \mathrm{Dom}(\Delta, \Delta')$ *and* $\alpha_i \notin \mathrm{Dom}(\Delta)$ *and* $\Delta, \gamma_i(\Delta') \vdash \gamma_i : \Gamma, \Delta'$ *and* $\Delta; v'_i\, v_i \to^* \Delta; \diamond$ *for* $i \in \{1, 2\}$*. Furthermore,* $\Delta(\alpha_1) = \mathrm{A}(\tau_1 : \kappa)$ *and* $\Delta(\alpha_2) = \mathrm{A}(\tau_2 : \kappa)$ *for some* κ*.*

1. *Let* $\Gamma, \Delta' \vdash \tau : \kappa'$ *and* $\Gamma, \Delta' \vdash \tau' : \kappa'$.
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(\tau) \leq \gamma_1(\tau') : \gamma_1(\kappa')$, *then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(\tau) \leq \gamma_2(\tau') : \gamma_2(\kappa')$.
2. *Let* $\Gamma, \Delta' \vdash e : \tau$.
   If and only if $\Delta, \gamma_1(\Delta') \vdash \gamma_1(e) : \gamma_1(\tau)$, *then* $\Delta, \gamma_2(\Delta') \vdash \gamma_2(e) : \gamma_2(\tau)$.
3. *Let* $\Gamma, \Delta' \vdash e : \tau$.
   If and only if $\Delta, \gamma_1(\Delta'); \gamma_1(e) \to^* \Delta, \gamma_1(\Delta'), \Delta_1; v'$, *then* $\Delta_1 = \gamma_1(\Delta'')$ *and* $v' = \gamma_1(v)$ *with* $\Delta, \gamma_2(\Delta'); \gamma_2(e) \to^* \Delta, \gamma_2(\Delta'), \gamma_2(\Delta''); \gamma_2(v)$.

Proof. As before, the only difference being additional cases in (1) in the type normalization algorithm that deal with $\pi = P[\alpha_i]$. Note again that coercions actually are inessential to the proof. □

E.4. Sealing

Theorem 81 (Admissibility of Sealing Rule)**.**
The rule ESEAL* *is admissible.*

Proof. By induction on $\tilde{\tau}$. The only non-trivial case is the following:

case $\tilde{\tau} = \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2$:

- by Validity, $\Gamma \vdash \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 : \Omega$
- by Type Validity Inversion, $\Gamma, \alpha{:}\tilde{\kappa}_1 \vdash \tilde{\tau}_2 : \Omega$
- by Environment Validity, $\Gamma, \alpha{:}\tilde{\kappa}_1 \vdash \Box$ and $\Gamma \vdash \tilde{\kappa}_1 : \Box$
- w.l.o.g., $x \notin \mathrm{Dom}(\Gamma, \alpha{:}\tilde{\kappa}_1)$
- let $\Gamma' = \Gamma, \alpha{:}\tilde{\kappa}_1, x{:}\tilde{\tau}_2$
- by NTERM, $\Gamma' \vdash \Box$
- by TVAR, $\Gamma' \vdash \alpha : \tilde{\kappa}_1$
- by KABS*, $\Gamma' \vdash \mathrm{A}(\alpha : \tilde{\kappa}_1) : \Box$
- w.l.o.g., $\alpha' \notin \mathrm{Dom}(\Gamma')$
- let $\Gamma'' = \Gamma', \alpha'{:}\mathrm{A}(\alpha : \tilde{\kappa}_1)$
- by NTYPE, $\Gamma'' \vdash \Box$
- by EVAR, $\Gamma'' \vdash x : \tilde{\tau}_2$
- by TVAR, $\Gamma'' \vdash \alpha : \tilde{\kappa}_1$ and $\Gamma'' \vdash \alpha' : \mathrm{A}(\alpha : \tilde{\kappa}_1)$
- w.l.o.g., $\alpha_1 \notin \mathrm{Dom}(\Gamma'')$
- by Weakening, $\Gamma'' \vdash \exists \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2[\alpha_1/\alpha] : \Omega$
- by Type Validity Inversion, $\Gamma'', \alpha_1{:}\tilde{\kappa}_1 \vdash \tilde{\tau}_2 : \Omega$
- let $e_1 = \{x : \alpha_1{:}\tilde{\kappa}_1.\tilde{\tau}_2[\alpha_1/\alpha]\}^+_{\alpha' \approx \alpha}$
- by EUP', $\Gamma'' \vdash e_1 : \tilde{\tau}_2[\alpha'/\alpha]$
- by ECLOSE, $\Gamma'' \vdash \langle \alpha', e_1 \rangle : \exists \alpha_1{:}\mathrm{S}(\alpha' : \tilde{\kappa}_1).\tilde{\tau}_2[\alpha'/\alpha]$
- by Validity, $\Gamma'' \vdash \exists \alpha_1{:}\mathrm{S}(\alpha' : \tilde{\kappa}_1).\tilde{\tau}_2[\alpha'/\alpha] : \Omega$
- by Type Validity Inversion, $\Gamma'', \alpha_1{:}\mathrm{S}(\alpha' : \tilde{\kappa}_1) \vdash \tilde{\tau}_2[\alpha'/\alpha] : \Omega$
- by KSSING-LEFT*, $\Gamma'' \vdash \mathrm{S}(\alpha' : \tilde{\kappa}_1) \leq \tilde{\kappa}_1 : \Box$
- by TSEXIST and some renaming, $\Gamma'' \vdash \exists \alpha_1{:}\mathrm{S}(\alpha' : \tilde{\kappa}_1).\tilde{\tau}_2[\alpha'/\alpha] \leq \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2 : \Box$
- by ESUB, $\Gamma'' \vdash \langle \alpha', e_1 \rangle : \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2$
- let $e_2 = \mathsf{new}\, \alpha'{:}\tilde{\kappa}_1 \approx \alpha\ \mathsf{in}_{\exists \alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ \langle \alpha', e_1 \rangle$
- by ENEW', $\Gamma' \vdash e_2 : \exists \alpha{:}\tilde{\kappa}_1.\tilde{\tau}_2$
- by EOPEN, $\Gamma \vdash \mathsf{let} \langle \alpha, x \rangle = e\ \mathsf{in}_{\exists \alpha:\tilde{\kappa}_1.\tilde{\tau}_2}\ e_2 : \exists \alpha{:}\kappa_1.\tau_2$

□

F. Index of Propositions

Bibliography

[ACC93] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121:9–58, 1993.

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[ACPP89] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *16th Symposium on Principles of Programming Languages*, pages 213–227, Austin, USA, January 1989.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[AFZ04] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A calculus with lazy module operators. In Jean-Jacques Levy, Ernst Mayr, and John Mitchell, editors, *3rd International Conference on Theoretical Computer Science*, pages 423–436. Kluwer Academic Publishers, 2004.

[Ala04] Lauri Alanko. Types and reflection. Master's thesis, Helsingin Yliopisto, Helsinki, Finland, November 2004.

[Ali03] Alice Team. *The Alice System*. Programming Systems Lab, Universität des Saarlandes, `http://www.ps.un-sb.de/alice/`, 2003.

[ANP89] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-structures: data structures for parallel computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[Apt03] Krysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctoral dissertation, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[Asp95] David Aspinall. Subtyping with singleton types. In *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Asp97] David Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, Edinburgh University, Edinburgh, UK, December 1997.

[AZ02] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2), 2002.

[BA99] Matthias Blume and Andrew Appel. Hierarchical modularity. *Transactions on Programming Languages and Systems*, 21(4):813–847, July 1999.

[Bar92] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[Ber04] Matthias Berg. Polymorphic lambda calculus with dynamic types. Fopra thesis, Universität des Saarlandes, Saarbrücken, Germany, October 2004.

[BFSS89] E.S. Bainbridge, Peter Freyd, Andre Scedrov, and Philip Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1989. Corrigendum in 71(3):431, 1990.

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *8th International Conference on Functional Programming*, Uppsala, Sweden, September 2003.

[BJW87] Andrew Birrell, Michael Jones, and Edwar Wobber. A simple and efficient implementation for small databases. In *11th ACM Symposium on Operating System Principles*, volume 21(5) of *Operating Systems Review*, pages 149–154. ACM Press, November 1987.

[BK02] Thorsten Brunklaus and Leif Kornstaedt. A virtual machine for multi-language execution. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, November 2002.

[BK03] Thorsten Brunklaus and Leif Kornstaedt. Open programming services for virtual machines: The design of Mozart and SEAM. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, March 2003.

[BN84] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *Transactions on Computer Systems*, 2(1):39–59, 1984.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobbler. Network objects. *Software – Practice and Experience*, 25(S4), December 1995.

[BNSS94] Peter Bailey, Malcolm Newey, David Sitsky, and Robin Stanton. Supporting coarse and fine grain parallelism in an extension of ML. In *Conference on Algorithms and Hardware for Parallel Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 693–704. Springer-Verlag, 1994.

[Bur84] Rod Burstall. Programming with modules as typed functional programming. In *International Conference on 5th Generation Computing Systems*, Tokyo, Japan, 1984.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, Germany, 1991.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[Car97a] Luca Cardelli. Program fragments, linking, and modularization. In *24th Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997.

[Car97b] Luca Cardelli. Type systems. In Allen Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CDG+91] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. In Greg Nelson, editor, *System Programming with Modula-3*, chapter 2, pages 11–66. Prentice Hall, 1991.

[CK92] Robert Cooper and Clifford Krumvieda. Distributed programming with asynchronous ordered channels in distributed ML. In Peter Lee, editor, *Workshop on ML and its Applications*, pages 134–148, June 1992.

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, March 1990.

[CL99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: mobile agents for objective-caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, pages 22–29, Palm Springs, USA, October 1999.

[Cra00] Karl Crary. Typed compilation of inclusive subtyping. In *5th International Conference on Functional Programming*, Montreal, Canada, September 2000.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *5th International Conference on Functional Programming*, pages 233–248, Paris, France, October 1999.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morisett. Intensional polymorphism in type-erasure semantics. In *3rd International Conference on Functional Programming*, Baltimore, USA, September 1998.

[CWM02] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *30th Symposium on Principles of Programming Languages*, New Orleans, USA, January 2003.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *4th Conference on Computer and Communications Security*, pages 18–27, 1997.

[Den85] Peter Denning, editor. *Special Issue on Prolog*, volume 28(12) of *Communications of the ACM*. ACM Press, December 1985.

[DFW96] Drew Dean, Edward Felten, and Dan Wallach. Java security: from HotJava to Netscape and beyond. In *Symposium on Security and Privacy*, pages 190–200, Oakland, USA, May 1996. IEEE Computer Society Press.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, San Francisco, USA, 2004.

[DHC01] Derek Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. Technical Report CMU-CS-01-112, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, March 2001.

[DKSS98] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, 1998. Draft, `http://www.ps.uni-sb.de/papers`.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In Richard DeMillo, editor, *7th Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, USA, January 1982. ACM Press.

[Dre04] Derek Dreyer. A type system for well-founded recursion. In *31st Symposium on Principles of Programming Languages*, Venice, Italy, January 2004.

[Dre05] Derek Dreyer. *Understanding and Evolving the ML Module System*. Phd thesis, Carnegie Mellon University, 2005.

[Dre07] Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, Forthcoming 2007.

[DRW95] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages*, San Francisco, USA, January 1995.

[Dug99] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.

[Dug02] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *Transaction on Programming Languages and Systems*, 24(6):711–804, November 2002.

[Els99] Martin Elsman. Static interpretation of modules. In *Fourth International Conference on Functional Programming*, pages 208–219, Paris, France, September 1999. ACM Press.

[FF95] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *22nd Symposium on Principles of Programming Languages*, pages 209–220, San Francisco, USA, January 1995.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. On distributed programming: A calculus of mobile agents. In *7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

[FMS01] Cédric Fournet, Luc Maranget, and Alan Schmitt. *The JoCaml Language beta release*. INRIA, `http://pauillac.inria.fr/jocaml/htmlman/`, January 2001.

[FW76] Daniel Friedman and David Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, July 1976.

[Gec05] Gecode Team. Generic constraint development environment, 2005. `http://www.gecode.org/`.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analysis, et son application à l'élimination des coupures dans l'analysis et la théorie des types. In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symposium*. North-Holland, 1971.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification*. Addison–Wesley, 1996.

[Gle99] Neal Glew. Type dispatch for named hierarchical types. In *5th International Conference on Functional Programming*, Paris, France, October 1999.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *26th Symposium on Principles of Programming Languages*, pages 250–261, January 1999.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

[GP98] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75–96, February 1998.

[GR04] Emden Gansner and John Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Har06] Robert Harper. *Programming in Standard ML*. Draft, 2006.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[HCS+01] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jefferey, Peter Schachte, Simon Taylor, and Chris Speirs. *The Mercury Language Reference Manual*. `http://www.cs.mu.oz.au/research/mercury/information/documentation.html`, 2001.

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. In *Transactions of AMS*, pages 146:29–60, 1969.

[HL82] Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137, Portland, USA, January 1994.

[HM93] Robert Harper and John Mitchell. On the type structure of Standard ML. *Theory of Programming Languages and Systems*, 15(2):211–252, April 1993.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, USA, January 1995.

[HM99] Robert Harper and John Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, 1999.

[HMM90] Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, USA, January 1990.

[HMPH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005.

[HP05] Robert Harper and Benjamin Pierce. *Design Considerations for ML-Style Module Systems*, chapter 8, pages 293–345. The MIT Press, 2005.

[HR99] Michael Hansen and Hans Rischel. *Introduction to Programming using SML*. Addison-Wesley, 1999.

[HRB+99] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[HRBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HS00] Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[Ken04] Andrew Kennedy. Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, November 2004.

[Kor06] Leif Kornstaedt. *Design and Implementation of a Programmable Middleware*. Doctoral dissertation, Universität des Saarlandes, Saarbrücken, Germany, December 2006.

[LAB+79] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Robert Scheifler, and Alan Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, 1979.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Object-Oriented Programming, Systems, Languages and Applications*, October 1998.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th Symposium on Principles of Programming Languages*, pages 177–188, New York, USA, January 1992. ACM Press.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium on Principles of Programming Languages*, pages 109–122, Portland, USA, January 1994. ACM.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, USA, January 1995. ACM.

[Ler03] Xavier Leroy. *The Objective Caml System*. INRIA, 2003. `http://pauillac.inria.fr/ocaml/htmlman/`.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, May 1997.

[LM93] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[LPSW03] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *8th International Conference on Functional Programming*, Uppsala, Sweden, September 2003.

[LS88] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Programming Language Design and Implementation*, pages 260–267. ACM Press, 1988.

[Luc00] Lucent Technologies. *Standard ML of New Jersey User's Guide*. `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/`, 2000.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison–Wesley, 1996.

[Mac84] David MacQueen. Modules for Standard ML. In *Symposium on LISP and Functional Programming*, pages 198–207, Austin, USA, 1984.

[Mac86] David MacQueen. Using dependent types to express modular structure. In *13th Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg, USA, January 1986.

[Mac93] David MacQueen. Reflections on standard ML. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 32–46, 1993.

[MG98] Yasuhiko Minamide and Jacque Garrigue. On the runtime complexity of type-directed unboxing. In *3rd International Conference on Functional Programming*, Baltimore, USA, September 1998.

[Mic03] Microsoft Corporation. *Microsoft .NET*. `http://www.microsoft.com/net/`, 2003.

[Mil78] Robin Milner. A theory of type polymorphism. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[Mil06] Mark Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Doctoral dissertation, Johns Hopkins University, Baltimore, USA, May 2006.

[Mit91] John Mitchell. On the equivalence of data representations. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[ML71] Per Martin-Löf. A theory of types, 1971. Unpublished manuscript.

[MM91] John Mitchell and Eugenion Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991.

[Mor73a] James Morris. Protection in programming language. *Communications of the ACM*, 16(1), January 1973.

[Mor73b] James Morris. Types are not sets. In *1st Symposium on Principles of Programming Languages*, pages 120–124, Boston, USA, October 1973.

[Moz04] Mozart Consortium. The Mozart programming system, 2004. `http://www.mozart-oz.org`.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *12th Symposium on Principles of Programming Languages*, 1985.

[MPO02] Simon Marlow, Simon Peyton Jones, and Others. *The Glasgow Haskell Compiler*. University of Glasgow, `http://www.haskell.org/ghc/`, 2002.

[MR86] Albert Meyer and Mark Reinhold. 'Type' is not a type. In *13th Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg Beach, Florida, USA, 1986.

[MT94] David MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423, Edinburgh, UK, April 1994. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Mül06] Christian Müller. Run-time byte code compilation, optimization, and interpretation for Alice. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, March 2006.

[Myc83] Alan Mycroft. Dynamic types in ML, 1983.

[Nei06] Georg Neis. A semantics for lazy types. Bachelor's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2006.

[NSS05] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In *5th International Workshop on Frontiers of Combining Systems*, Vienna, Austria, September 2005.

[NSS06] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, November 2006.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *European Conference on Object-oriented Programming*, Darmstadt, Germany, July 2003.

[Ode05] Martin Odersky. *Programming in Scala*. École Polytechnique Fédérale de Lausanne, 2005.

[OK93] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *20th Symposium on Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[PA93] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Beeze and Jan Friso Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, Berlin, 1993.

[Pal05] Niko Paltzer. Efficient representation of dynamic types. Bachelor's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2005.

[Par72] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

[Pau96] Larry Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[PH99] Simon Peyton Jones and John Hughes. *Haskell 98: A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport`, 1999.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, February 2002.

[Pil96] Marco Pil. First class file I/O. In Werner Kluge, editor, *8th International Workshop on Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 233–246. Springer-Verlag, 1996.

[Pit98] Andrew Pitts. Existential types: Logical relations and operational equivalence. In *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.

[Pit00] Andrew Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[Pit05] Andrew Pitts. *Typed Operational Reasoning*, chapter 7, pages 245–289. The MIT Press, 2005.

[PS93] Andrew Pitts and Ian Stark. On the observable properties of higher order functions that dynamically create local names. In Paul Hudak, editor, *Workshop on State in Programming Languages*, pages 31–45, Copenhagen, Denmark, 1993.

[Pv00] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report*. `http://www.cs.kun.nl/~clean/Manuals/manuals.html`, 2000.

[Ram05] Norman Ramsey. ML module mania: A type-safe, separately compiled, extensible interpreter. In Nick Benton and Xavier Leroy, editors, *Workshop on ML*, Tallinn, Estonia, September 2005.

[Rep99] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[Rey83] John Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing*, pages 513–523, Amsterdam, 1983. North Holland.

[RLT+06] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice ML through the looking glass. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5, Munich, Germany, 2006. Intellect.

[Ros01] Andreas Rossberg. Defects in the Revised Definition of Standard ML. Technical report, Universität des Saarlandes, Saarbrücken, Germany, October 2001. `http://www.ps.uni-sb.de/Papers/`.

[Ros03a] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003.

[Ros03b] Andreas Rossberg. Generativity and dynamic opacity for abstract types (Extended Version). Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, August 2003. `http://www.ps.uni-sb.de/Papers/`.

[Ros05] Andreas Rossberg. The definition of Standard ML with packages. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, 2005. http://www.ps.uni-sb.de/Papers/.

[Ros06] Andreas Rossberg. The missing link – dynamic components for ML. In *11th International Conference on Functional Programming*, Portland, Oregon, USA, September 2006. ACM Press.

[RRS00] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Language Overview*. `ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.pdf`, 2000.

[Rus98] Claudio Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[Rus99] Claudio Russo. Non-dependent types for Standard ML modules. In *International Conference on Principles and Practice of Declarative Programming*, Paris, France, September 1999.

[Rus00] Claudio Russo. First-class structures for Standard ML. In Gert Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, Berlin, Germany, March 2000. Springer-Verlag.

[Rus01] Claudio Russo. Recursive structures for Standard ML. In *Sixth International Conference on Functional Programming*, pages 50–61, Florence, Italy, September 2001. ACM Press.

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon – Steps beyond Pascal and Modula*. ACM Press, 1992.

[RWWB96] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the java system. *Computing Systems*, 9(4):291–312, 1996.

[Sch02] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.

[Sch06] Andi Scharfstein. A sandboxing infrastructure for Alice ML. Bachelor's thesis, Universität des Saarlandes, Saarbrücken, Germany, October 2006.

[Sea04] Seam Team. Simple extensible abstract machine, 2004. `http://www.ps.uni-sb.de/seam/`.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *28th Symposium on Principles of Programming Languages*, London, UK, January 2001.

[SH06] Christopher Stone and Robert Harper. Extensional equivalence and singleton types. *Transactions on Computational Logic*, 7(4), October 2006.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Survey*, 21(3):413–511, 1989.

[SLW+05] Peter Sewell, James Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *10th International Conference on Functional Programming*, pages 15–26, Tallinn, Estonia, September 2005.

[SM03] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SMCH06] David Swasey, Tom Murphy, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. Technical Report CMU-CS-06-104, Carnegie Mellon University School of Computer Science, January 2006.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, Germany, 1995.

[Smo99] Gert Smolka. From concurrent constraint programming to concurrent functional programming with transients, 1999. `http://www.ps.uni-sb.de/~smolka/ccl99.ps`.

[SP03] Eijiro Sumii and Benjamin Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.

[SP04] Eijiro Sumii and Benjamin Pierce. A bisimulation for dynamic sealing. In *31st Symposium on Principles of Programming Languages*, Venice, Italy, January 2004.

[SP05] Eijiro Sumii and Benjamin Pierce. A bisimulation for type abstraction and recursion. In *32nd Symposium on Principles of Programming Languages*, pages 63–74, Long Beach, USA, January 2005. ACM Press.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 2nd edition, 1994.

[ST05] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In Mats Carlsson, Francois Fages, Brahim Hnich, and Francesca Rossi, editors, *Recent Advances in Constraints*, volume 3978 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 2005.

[Sto00] Christopher Stone. *Singleton Types and Singleton Kinds.* Dissertation, Carnegie Mellon University, August 2000.

[Sto05] Christopher Stone. *Type Definitions*, chapter 9, pages 347–385. The MIT Press, 2005.

[Sun97] Sun Microsystems. *JavaBeans Specifications 1.01*, August 1997.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, December 2001.

[Tac03] Guido Tack. Linearisation, minimisation and transformation of data graphs with transients. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, May 2003.

[TKS06] Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. In *Workshop on ML*, volume 148(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–103. Elsevier, March 2006.

[TL04] Guido Tack and Didier Le Botlan. Compositional abstractions for search factories. In Peter Van Roy, editor, *Second International Mozart/Oz Conference*, volume 3389 of *Lecture Notes in Computer Science*, Charleroi, Belgium, October 2004. Springer-Verlag.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A facile tutorial. In *7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.

[Tof94] Mads Tofte. Principal signatures for higher-order program modules. *Journal of Functional Programming*, 4(3):285–335, July 1994.

[TSS00] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *5th International Conference on Functional Programming*, pages 82–93, Montreal, Canada, September 2000.

[Tur76] David Turner. The SASL language manual. Technical report, University of St. Andrews, 1976.

[Tur85] David Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[Tv88] Anne Troelstra and Dirk van Dalen. *Constructivism in Mathematics*, volume 2. North-Holland, 1988.

[Ull97] Jeffrey Ullman. *Elements of ML Programming*. Prentice-Hall, second edition, 1997.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[VWW05] Dimtrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Types in Language Design and Implementation*, Long Beach, USA, January 2005.

[Wad89] Philip Wadler. Theorems for free! In Dave MacQueen, editor, *4th International Conference on Functional Programming and Computer Architecture*, pages 347–359, London, UK, September 1989. ACM Press.

[WAM99] Ken Wakita, Takashi Asano, and Sassa Masataka. D'caml: A native distributed ML compiler for heterogeneous environment. In *European Parallel Processing Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 914–924, Toulouse, France, September 1999. Springer-Verlag.

[Wei00] Stephanie Weirich. Type-safe cast. In *5th International Conference on Functional Programming*, pages 58–67, Montreal, Canada, September 2000.

[Wei02] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, 2002.

[WF94] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon – The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX Association, 1996.