Alice Through the Looking Glass (Extended Mix)*

Andreas Rossberg Didier Le Botlan Guido Tack
Thorsten Brunklaus Gert Smolka

Universität des Saarlandes, Saarbrücken, Germany
{rossberg,botlan,tack,brunklaus,smolka}@ps.uni-sb.de

April 15, 2005

Abstract: We present Alice ML, a functional programming language that has been designed with strong support for *typed open programming*. It incorporates concurrency with data flow synchronisation, higher-order modularity, dynamic modules, and type-safe pickling as a minimal and generic set of simple, orthogonal features providing that support. Based on these mechanisms Alice offers a flexible notion of component, and high-level facilities for distributed programming.

1 Introduction

Software is decreasingly 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 omni-present, 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 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:

*This paper is an extended version of [RLT+05].

- *Modularity*, to flexibly combine software blocks that were created separately.
- *Dynamicity*, to import *and* export software blocks in running programs.
- *Security*, to safely deal with unknown or untrusted software blocks.
- *Distribution*, to communicate data and software blocks over networks.
- *Concurrency*, to deal with asynchronous events and non-sequential tasks.

Software blocks intended for dynamic combination are usually called *components*.

Most practical programming languages today have not been designed with open programming in mind. Even the few that have been – primarily Java [GJS96] – do not adequately address all of the above points. For example, Java is not statically type-safe, has only weak support for import/export, and rather clunky distribution and concurrency mechanisms.

Our claim is that only a few simple, orthogonal concepts are necessary to provide a flexible, *strongly typed* framework supporting all aspects of open programming. Components are a central notion in this framework, but instead of being primitive they can be derived from the following simpler, orthogonal concepts:

- *Futures*, which allow for light-weight concurrency and *laziness*.
- *Higher-order modules*, to provide genericity and encapsulation.
- *Packages*, to wrap modules into self-describing, dynamically typed entities.
- *Pickling*, to enable generic persistence and distribution.
- *Proxy functions*, to enable remote calls into other processes.

To substantiate our claim, we developed the programming language *Alice ML*, a conservative extension of Standard ML [MTHM97]. It is largely inspired by Oz [Smo95, Moz04, VH04], a relational language with advanced support for open programming, but lacking any notion of static typing. The aim of the Alice project [Ali04] is to systematically reconstruct the essential functionality of Oz on top of a simple and well-understood, typed functional core language. Alice is strongly typed and supports typeful programming [Car91] with abstraction safety (encapsulation) preserved at all times.

The Alice ML language has been implemented in the Alice Programming System [Ali04], a full-featured programming environment based on a VM with just-in-time compilation, support for platform-independent persistence and platform-independent mobile code, and a rich library for constraint programming.

Organisation of the paper

This paper describes the concepts introduced in Alice to enable open programming. In order to support concurrency and laziness, Alice provides the concept of futures, described in Section 2. A higher-order extension to the SML module system enhances modularity and is briefly sketched in Section 3. Type-safe import and export is realized by introducing dynamically typed modules called *packages* that can be *pickled*, as described in Section 4. In Section 5 we introduce the component model and show in Section 6 how it can be decomposed. Communication through a network based on components is considered in Section 7. We illustrate most of the features of Alice in Section 8 considering the case study of a distributed solver for constraint programming. We discuss related work in Section 10 and conclude in Section 11.

2 Futures

Programs communicating with the outside world usually have to deal with non-deterministic, asynchronous events. Purely sequential programming cannot adequately handle such scenarios. Support for concurrency hence is vital for open programming.

Concurrency in Alice is based uniformly on the concept of *futures*, which has been mostly adapted from Multilisp [Hal85], where they were introduced as an approach to make automatic parallelisation of functional programs effective. A future is a transparent place-holder for a yet undetermined value that allows for implicit synchronisation based on data flow. There are different kinds of futures, which we will describe in the following sections:

- *concurrent futures* hold place for the result of an evaluation in a thread,
- *lazy futures* hold place for the result of a delayed computation,
- *promised futures* are created as holes to be filled later by an explicit operation,
- *failed futures* mark futures for which no value could be determined.

A formal semantics of the future mechanism present in Alice can be found in [NSS02], more examples are presented in [Smo99].

Futures are a generic mechanism for communication and synchronisation. As such, they are comparatively simple, but expressive enough to enable formulation of a broad range of concurrency abstractions.

2.1 Concurrency

Future-based concurrency is very light-weight: any expression can be evaluated in its own thread, a simple keyword allows forking off a concurrent computation for evaluating an expression:

spawn *exp*

This phrase immediately evaluates to a fresh *concurrent future*, standing for the yet unknown result of *exp*. Simultaneously, evaluation of *exp* is initiated in a new thread. As soon as the thread terminates, the result 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. 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. For that reason, we call them *functional threads.*

Functional threads allow turning a synchronous function call to a function 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 7), 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.

Futures already provide for complex 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. The computation for the table entries in the second declaration depends on that value, but since the entries are computed concurrently, construction of the table can proceed without delay. However, the individual threads computing its content 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 offers primitives for explicit synchronisation:[1]

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

The function await is a strict variant of the identity function: if applied to a future, it blocks until the future has been replaced by a proper value. A straightforward abstraction using this function is the barrier function that implements a *join point* by applying a list of functions concurrently and waiting 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
```

2.2 Laziness

SML is an eager language. While eager evaluation has advantages (e.g. making algorithmic complexity more predictable), certain algorithms are expressed more elegantly or more efficiently with *lazy evaluation*. It has become a common desire to marry eager and lazy evaluation, and the future mechanism provides an elegant way to do so. While keeping eager semantics the default behaviour, full support for laziness is available through a lazy variant of futures. A *lazy future* is introduced analogously to a concurrent one: the phrase

```
lazy exp
```

will not evaluate *exp*, but instead returns a fresh lazy future, standing for the yet unknown result of *exp*. Evaluation of *exp* is triggered by a thread first touching 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.

In other words, lazy evaluation can be selected for individual expressions. For example, the expression (fn x ⇒ 5) (lazy raise E) will *not* raise E. A fully lazy evaluation regime can be emulated by prefixing *every* subexpression with the lazy keyword, but usually only few strategic annotations are necessary.

In order to support the definition of lazy functions conveniently, Alice extends the definition of SML's sugared function declaration syntax with support for the lazy keyword. For example, a lazy mapping function on lists can be defined as

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

```
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
```

In fact, spawn is supported likewise, for the definition of asynchronous functions.

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

2.3 Failed Futures

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?

Failed futures allow structured propagation of exceptions between threads. A failed future carries an exception. When a thread terminates with an exception, the associated future becomes failed with that exception. Any attempt to touch a failed future re-raises its exception in the respective thread. For example, after evaluating

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

the expression x+1 will raise Empty.

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 in certain ways. There is no way to eliminate the future in that case, 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.

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* important thread expressions, so that exceptions are ensured not to escape. No commitment is made by the language semantics, futures represent the primitive that enables programming different strategies as higher-level abstractions.

2.4 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 × α → 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 a usual references in that they are created uninitialised and may only be assigned once. Dereferencing them prior to assignment delivers a future standing for the later (immutable) content.

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 1. However, they are particularly important for 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.

2.5 Thread Safety

Basic *thread safety* is provided by defining all primitive operations that access or modify state as atomic. Of course, that alone is not sufficient for thread-safe programming: when

```
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 1: Tail-recursive append with promises

multiple threads share mutable state it is imperative to synchronise access, usually by forms of locking on critical sections. Alice requires no primitive locking mechanisms, they can be fully bootstrapped from promises, plus references with an atomic exchange operation, a variant of the fundamental test-and-set operation [Hal85]:

```
val exchange : α ref × α → α
```

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

As a simple example demonstrating this, Figure 2 presents a higher-order function implementing mutex locks for synchronising an arbitrary number of functions.[2] 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"))
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. As soon as that content is determined to () 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.

It should be noted that promises are derivable in a language with concurrent futures and atomic exchange. Figure 3 shows a non-primitive implementation of 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.

[2] 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.

```
(* mutex : unit → (α → β) → (α → β) *)
fun mutex () =
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 2: Mutexes for synchronised functions

```
type α promise = {free : bool ref, value : α option ref}

exception Promise

fun promise () = {free = ref true, value = ref NONE}

fun poll r = case !r of SOME x ⇒ x
                     | NONE   ⇒ poll r

fun future {free, value} = spawn poll value

fun fulfill ({free, value}, x) =
    if exchange (free, false)
    then value := SOME x
    else raise Promise

fun fail (p, e) = fulfill (p, spawn raise e)
```

Figure 3: A non-primitive implementation of promises

2.6 Modules and Types

Futures are not restricted to the core language, entire modules can be futures, too. In particular, module expressions can be evaluated lazily or concurrently, by explicitly prefixing them with the corresponding keywords lazy or spawn. More importantly, we will see in Section 5 that lazy module futures are ubiquitous as a consequence of the lazy linking mechanism for Alice components.

Evaluating a structure lazily implies that its body is evaluated not before one of its fields is touched. That is, a projection M.a, where the structure M is lazy, does not by itself touch M. Only when the projected field is used in a future-strict position evaluation of M will be triggered to determine M.a. In other words, projection from structures via long identifiers is implicitly lazy. This semantics 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 identification changes strictness – otherwise modular programming would become an obstacle for laziness (and particularly lazy linking, see Section 5).

Module futures can also be failed. Exceptions from failed module futures can be caught because Alice allows modules to be declared in local scope (Section 3), which can be surrounded by a handler.

The combination of module futures and dynamic types (Section 4) also implies the existence of *type futures*. They are touched only by the unpack operation (Section 4.1) and by pickling (Section 4.2). Touching a type generally can trigger arbitrary computations, e.g. by loading a component (Section 5).

3 Higher-order Modules

For open programming, good language support for modularity is essential. The SML module system is quite advanced, but still limited by its restriction to first-order functors (parameterised modules) and its stratified design (modules cannot be declared locally). Adopting a long line of work [DCH03, Ler95, Lil97, Rus98], Alice extends the SML module system in three ways:

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

Local modules are important for dealing with packages (Section 4), while the other two extensions allow more general forms of abstraction. In particular, they turn structures into a general container for all language entities, which is crucial for the design of the Alice component system (Section 5).

Nested and abstract signatures are less standard than the other two features. But they are interesting because they enable the definition of *polymorphic functors*, exemplified by a general application functor:

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

Polymorphic functors are used in the Alice library to provide certain functionality at the module level. We will see an example of this in Section 8.

More importantly, nested signatures turn structures into a general container for all language entities, which is crucial for the design of the component system (Section 5).

Space consideration preclude a detailed presentation of the Alice module language, but the knowledgeable reader will realise that the above extensions turn it into a higher-order functional language that closely mirrors the module language of Objective Caml [Ler03, Ler94], except that functors are not applicative [Ler95]. Like in Objective Caml, the presence of abstract signatures renders module type checking undecidable [Lil97], but this has not turned out to be a problem in practice.

4 Packages

When a program is able to import and export data and functionality dynamically, from statically unknown sources, then a certain amount of runtime checking is inevitable to ensure the integrity of the program and the runtime system. In a language with a strong type system, like ML, it particularly must be ensured that dynamic imports cannot undermine the invariants established by the type system. How can the tension be resolved?

Dynamics [Myc83, ACPR95, LM93] complement static typing with dynamic type checking. They provide a universal type dyn of 'dynamic values' that carry runtime type information. Values of every type can be injected into this type. Projection usually is a complex type-case operation that dispatches on the runtime type found in the dynamic value.

Dynamics adequately solve the problem of typed open programming by demanding external values to uniformly have type dyn. We see several hurdles that nevertheless prevented the wide-spread adoption of dynamics in practice: (1) their too fine level of granularity, (2) the complexity of the type-case construct, (3) the lack of flexibility with matching types.

Dynamics maintain most properties of the static type system by isolating dynamic typing with the projection operation, and they solve the problem of open programming by demanding external values to uniformly have type dyn. We see several hurdles that nevertheless prevented the wide-spread adoption of dynamics in practice: (1) the improper level of granularity they provide for wrapping objects, (2) the complexity of the type-case construct, particularly with respect to polymorphic types, (3) the lack of flexibility

with matching types, which makes them fragile against interface changes. Moreover, dynamics destroy parametricity, a desirable property that we will discuss in Section 4.4.

We modified the concept of dynamics slightly: instead of plain values dynamics in Alice, called *packages*, contain *modules*. Projection simply matches the runtime *package signature* against a statically specified one – with full respect for subtyping. Reusing module subtyping instead of type matching and dispatch has several advantages: (1) it keeps the language simple, (2) it is flexible and sufficiently robust against interface evolution, and (3) it allows the programmer to naturally adopt idioms already known from modular programming, particularly with respect to type sharing. Moreover, packages allow modules to be passed as first-class values, a capability that is sometimes being missed from ML, and becomes increasingly important with open programming.

A formal semantics for packages can be found in [Ros05].

4.1 Basics

A package is a value of the abstract type package. Intuitively, it contains a module, along with a dynamic description of its signature. We call that signature the *package signature*.

There are only two basic operations on packages. A package is created by injecting a module, expressed by a structure expressions *strexp* in SML [3] into the type package:

pack *strexp* : *sigexp*

The signature expression *sigexp* defines the package signature. Of course, the module expression *strexp* 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* and extracts the contained module, provided that the package signature matches the *target signature* denoted by *sigexp*. Statically, the expression has the signature *sigexp*. If the dynamic check fails, the pre-defined exception Unpack is raised.

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

```
val p = pack List : LIST
```

and unpack it successfully using the same signature:

```
structure List1 = unpack p : LIST
```

[3] Note that Alice supports higher-order modules, such that *strexp* includes functor expressions.

Any attempt to unpack p with an incompatible signature will fail. Note that all subsequent accesses to List1 or members of it are statically type-safe, no further checks are required.

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 is that first-class modules are statically typed (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.

4.2 Pickling

The primary purpose of packages is to type dynamic import and export of high-level language objects. At the core of this functionality lies a service called *pickling*. Pickling takes a value and produces a transitively closed, platform-independent representation of it that is transferable to other processes, where an equivalent copy of the original value can be constructed. Since ML is a language with first-class functions, a pickle can naturally include higher-order data, i.e. closures and code. Thanks to packages, even entire modules can be pickled.

One obvious application of pickling is *persistence*, available through two primitives 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. Any future occurring in the package (including lazy ones) will be touched (Section 2.1). If the package contains a local *resource*, i.e. a value that is private to a process, then the exception Sited is raised (we return to the issue of resources in Section 5.3). The inverse operation load retrieves a package from a file.

For example, we can write the library structure Array to disk, using the following idiomatic code:

```
Pickle.save ("/tmp/array.alc", pack Array : ARRAY)
```

It can be retrieved again with the inverse sequence of operations:

```
structure Array1 = unpack Pickle.load "/tmp/array.alc" : ARRAY
```

Any attempt to unpack it with an incompatible signature will fail with an Unpack exception. All subsequent accesses to Array1 or members of it are statically type-safe, no further checks are required. The only possible point of type failure is the unpack operation.

4.3 Dynamic Type Sharing

Note that the type Array1.array from the example in the previous section will be statically incompatible with the original type Array.array, since there is no way to know statically what type identities are found in a package, and all types in the target signature must hence be considered abstract. If compatibility is required, it can be enforced in the usual ML way, namely by putting *sharing constraints* on the target signature:

```
structure Array1 = unpack Pickle.load "/tmp/array.alc"
                          : ARRAY where type array = Array.array
```

The constraint effectively expresses *dynamic type sharing*. By restricting the target signature we ensure static compatibility, but of course we also preclude successful loading of non-standard implementations of arrays. Much like for programming with functors, it depends on the application how much sharing is required.

Note that 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. Campaigns can be created with different configurations (e.g. different map sizes), and for each campaign, an arbitrary number of snapshots (saved games) can be stored. 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. A snapshot only is valid in conjunction with the campaign that it belongs to. Thus, a snapshot can be modelled as an abstract type that is created along with each individual campaign and becomes part of a campaign pickle. Loading a snapshot then requires dynamic sharing between the type of snapshots of the current campaign and the snapshot's signature.

4.4 Parametricity

By utilising dynamic type sharing it is possible to dynamically test for type equivalences. In other words, evaluation is no longer *parametric* [Rey83]. For example,

```
functor F (type t) = unpack load file : (val it : t)
```

is a functor that behaves differently depending on what type t it is passed.[4]

Parametricity has important advantages: it induces strong static invariants about polymorphic types [Wad89], which allow deriving a variety of useful laws and particularly guarantee *abstraction* [Rey83, MP88] and enable efficient type erasing compilation, such that compiled programs need not maintain costly type information at runtime.

On the other hand, packages enforce the presence of non-parametric behaviour. Alice thus has been designed such that the *core* language, where polymorphism is ubiquitous, maintains parametricity. Only the *module* level employs dynamic type information –

[4] Alice allows abbreviating signatures sig ... end with (...), likewise for structures.

the evaluation of module expressions can be type-dependent. This design reduces the costs for dynamic types and provides a clear model for the programmer: the only type information relevant to the dynamic semantics (and its costs) are *explicitly declared* in the program. Implicit type information on the core level is not relevant.[5]

The restriction of type-driven evaluation to the module language is not without drawback: a parametric core language significantly restricts the expressive power of dynamic typing. In our experience however, there is no strong need for a more liberal regime. Thanks to higher-order and local modules (Section 3) it is often possible to lift the procedure in question to a functor. Moreover, packages provide a systematic way to work around the restriction, because they can be abused to pass dynamic type information to core functions if really required.

4.5 Abstraction Safety and Generativity

The absence of parametricity on the module level still raises the question of how dynamic typing interferes with type abstraction. Can we sneak through an abstraction barrier by dynamically discovering an abstract type's representation? For instance:

```
signature S = sig type t; val x : t end
structure M = struct type t = int; val x = 37 end :> S
structure N = unpack (pack M : S) : (S where type t = int)
val y = N.x + 1
```

Fortunately, the unpack operation will fail at runtime. This behaviour is achieved by a *dynamically generative* interpretation of type abstraction: with every abstraction operator :> evaluated, fresh type names are generated dynamically [Ros03]. Abstraction safety is maintained even across process boundaries, because type names are globally unique. There is no way within the language to break abstraction, even when whole modules and *implementations* of abstract types are exported.

The generative semantics of abstract types implies that execution of the same implementation of an abstraction will create incompatible instances between runs of the same program, or between different processes loading it. However, this is not a severe restriction, because the availability of module-level pickling enables us to execute it only once and then share the same *evaluated* instance of the abstraction between processes. As we will see in Section 5.4, pickling a module actually creates a proper component that is interchangeable with components generated by the compiler.

[5] Note that SML does not allow type or signature declarations to refer to free type variables, so that no interference is possible.

5 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 and configured dynamically. Such building blocks are called *components*.

We distinguish components from modules: while modules provide 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.

Alice incorporates a powerful notion of component that is a refinement and extension of the component system found in the Oz language (where components are called *functors* by slight abuse of terminology) [DKSS98], which in turn was partially inspired by Java [GJS96]. It provides all of the following:

- *Separate compilation.* Components are physically separate program units.
- *Lazy dynamic linking.* Loading is performed automatically when needed.
- *Static linking.* Components can be bundled into larger components off-line.
- *Dynamic creation.* Components can be computed and exported at runtime.
- *Type safety.* Components carry type information and linking fully checks it.
- *Flexibility.* Type checking is tolerant against interface changes via subtyping.
- *Sandboxing.* Custom *component managers* enable selective import policies.

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

- *Higher-order modules*, for encapsulating all possible language entities,
- *Packages*, for dynamic type checking of imports,
- *Futures*, for performing linking lazily,
- *Pickling*, for representing components externally.

We will first give a general overview of components before we see how precisely they relate to the mechanisms listed above.

5.1 Introduction

Components are the unit of compilation as well as the unit of deployment in Alice. A program consists of a – potentially open – set of components that are created separately and loaded dynamically. Static linking allows both to be performed on a different level of granularity if desired, by bundling given components to form larger ones. Every component defines a module – its *export* – and accesses an arbitrary number of modules from other components – its *imports*. Both, import and export interfaces, are fully typed by ML signatures.

Each Alice source file defines, and is compiled into, a component. Syntactically, a component definition is a sequence of SML declarations that 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. Because of Alice's higher-order modules (Section 3), these entities can include functors and even signatures. The string contains the URL under which the component is to be acquired at runtime. The exact interpretation of the URL is up to the component manager and its resolver (Section 5.3), but usually it is either a local file, an HTTP web address, or a virtual URL denoting local library components. 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://domain.org/server"
```

For convenience, Alice allows the type annotations in import specifications to be dropped. In that case, the imported component must be accessible (in compiled form) during compilation, so that the compiler can insert the respective types from its signature. For example, the previous import declarations could be written as

```
import structure Pickle from "x-alice:/lib/system/Pickle"
import structure Server from "http://domain.org/server"
```

This feature is particularly needed for large modules, where repeating the signature would be tedious. 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.

5.2 Program Execution and Dynamic Linking

A designated *root* is the main component of a program. To execute a program, its root component is evaluated. 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 2.2).

The process of loading a component requested as import by another one is referred to as *dynamic linking*. It involves several steps:

1. *Resolution.* The import URL is normalised to a canonical form.
2. *Acquisition.* If the component is being requested for the first time, it is loaded.
3. *Evaluation.* If the component has been loaded afresh, its body is evaluated.
4. *Type Checking.* The component's export signature is matched against the respective import signature.

Each of the steps can fail: 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, the respective future is failed with the standard exception Component.Failure that carries a description of the precise cause of the failure.

5.3 Component Managers and Sandboxing

Linking is performed with the help of a *component manager*, which is a module of the runtime library, similar to a class loader in Java [GJS96]. It is responsible for locating and loading components, and keeping a table of components already loaded by a process.

In an open setting it is important to be able to deal with untrusted components. For example, they should not be given write access to the local file system. To deal with this, we adopt the approach taken by Java: components can be executed in a *sandbox*. Sandboxing relies on two factors: (1) all resources and capabilities a component needs for execution are *sited* and have to be acquired via import through a local component manager (in particular, they cannot be stored in a pickle); (2) it is possible to create custom managers and explicitly link components through them. The implementation of a custom manager can only use capabilities provided by its 'parent' manager, so it can never grant more access than it has itself. Moreover, a component manager is inherited, i.e. all imports directly or indirectly requested by a component will be linked using the same manager. A custom manager hence represents a proper sandbox.

5.4 Dynamic Creation of Components

The external representation of a component is a pickle. It is hence possible to create a component not only statically by compilation, but also dynamically, by a running Alice program. In fact, a pickle created with the Pickle.save function (Section 4.2) *is* a component and can be imported as if it had been created through compilation.

The ability to create components dynamically is particularly important for distribution (Section 8). Basically, it enables components to capture dynamically obtained information, e.g. configuration data or connections to other processes, or dynamic type names (Section 4.5).

6 Decomposing Components

What *are* components? The close relation to concepts presented in previous chapters, like modules, packages and futures is obvious, so one might hope that there exists a simple reduction from components to simpler concepts. And indeed, components are merely syntactic sugar. Basically, a component defined by a sequence of declarations *dec* is interpreted as a higher-order procedure:

```
fn link ⇒ pack struct dec end : sigexp
```

where *link* is a reserved identifier and *sigexp* is the component signature derived by the compiler (the principal signature of the structure). In *dec*, every import declaration

```
import spec from s
```

is rewritten as[6]

```
structure strid = lazy unpack link s : sig spec end
open strid
```

where *strid* is a fresh identifier. The expansion makes laziness and dynamic type checking of imports immediately obvious. Component acquisition is encapsulated in the component manager represented by the *link* procedure. Every component receives that procedure for acquiring its imports and evaluates to a package that contains its own export. The *link* procedure has type string → package, taking a URL and returning a package representing the export of the component identified by the URL. Imports then are simply structure declarations that lazily unpack that package.

The *link* procedure represents the core of a component manager. 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. Figure 4 contains a simple model implementation of such a procedure, that assumes existence of two auxiliary procedures *resolve*, for normalising URLs relative to the URL of the parent component,

[6] An open declaration merely affects scoping, it does not touch its argument.

```
val table = ref () : (url × package) list ref

fun link parent url =
let
   val url' = resolve (parent, url)                    (* get absolute URL *)
   val p = promise ()
   val table' = exchange (table, future p)             (* lock table *)
in
   case List.find (fn (x,y) ⇒ x = url') table' of
      SOME package ⇒                                   (* already loaded *)
         (fulfill (p, table'); package)                (* unlock, return *)
    | NONE ⇒                                           (* not loaded yet *)
      let
         val component = acquire url'                  (* load component *)
         val package = lazy component(link url')       (* evaluate *)
      in
          fulfill (p, (url',package) :: table');       (*  unlock  *)
          package
      end
end
```

Figure 4: The essence of a component manager

and *acquire* for loading a component from a normalised URL. The procedure takes the parent URL as an additional parameter in order to allow the respective resolution. To achieve proper re-entrancy, the manager uses promises to implement locking on the component table (Section 2.5), and unlocks it *before* evaluating the component (hence the lazy application).

Giving this reduction of components, execution of an Alice program can be thought of as evaluation of the simple application

```
link "." root
```

where link is the initial component manager and *root* is the URL of the program's root component, resolved relative to the "current" location, which we indicate by a dot here.

7 Distribution

Distributed programming can be based on only a few high-level primitives that suffice to hide all the embarrassing details of low-level communication.

We first explain the basic mechanism for inter-process communication in Alice, and then the mechanisms conveniently allowing to set up a connection[7] for such communication.

[7]Where "connection" means a logical connection, and is definitely not a permanent connection at the network level.

7.1 Proxies

The central concept for distribution in Alice are *proxies*. A proxy is a mobile wrapper for a stationary function: it can be pickled and transferred to other processes without transferring the wrapped function itself. When a proxy function is applied, the call is automatically forwarded to the original site as a remote invocation, where argument and result are automatically transferred by means of pickling.

Proxies are created using the primitive

```
val proxy : (α → β) → (α → β)
```

For instance, the declaration

```
val f = proxy (fn x ⇒ x+1)
```

defines a simple proxy function. As a higher-level abstraction, a polymorphic functor (Section 3) Remote.Proxy is available to transform all functions of a given structure into proxies:

```
functor Proxy (signature S; structure X : S) : S
```

Communication between processes simply consists in calling proxy functions that have been acquired from another process. All invocations of a proxy are synchronous: the call

```
val n = f 4
```

will result in a synchronous remote function call and does not return before the remote process has computed and returned the result. In order to make an asynchronous call, it suffices to wrap the application using spawn (Section 2.1):

```
val n = spawn f 4
```

This immediately returns a future that will be replaced by the result of the remote call automatically once it terminates.

Note that all calls through proxies are statically typed, because proxies can be received through pickles or other proxy calls only. In both cases, type consistency is ensured.

7.2 Client/Server

To initiate a proxy connection, one pickle must first be transferred between processes by other means than proxy communication. The Alice library supports two main scenarios. In the *client/server* model a client establishes a connection to a known server. A service offered by a server takes the form of a local component, which we refer to as the *mobile* component. Mobile components can be made available in a network through a simple transfer mechanism adapted from Mozart [Moz04]. To employ it, a component is first packaged (Section 4), and then made available for download using the following library primitive:

```
val offer : package → url
```

Offering a package uses pickling (Section 4.2) in way very similar to saving it to a file. Given a package (Section 4), the offer function returns a globally unique URL, called a *ticket*. This ticket is the public identification of the server in the network. A ticket can be communicated to the outside world by conventional means such as web pages, email, phone, or pigeons.

A client can use a ticket to retrieve the package from the server:

```
val take : url → package
```

This primitive expects a valid ticket and retrieves the corresponding package from the server. The package 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. Noticeably, this is the only point where a dynamic type check is necessary. Indeed, from now on, static type checking suffices to ensure that all communication is well-typed.

In order to establish a permanent connection, the mobile component must contain proxies. Once the connection is thus established, these proxies serve as the communication channel(s).

So far, only the client can call the server side. To get a symmetrical connection, it suffices to send client-side proxies back to the server as arguments to higher-order proxies 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.

7.3 Master/Slave

In the client-server setting, clients choose independently to connect to a known server. In contrast, in an alternative scenario, a *master* initiates shifting computational tasks to a number of *slave* computers.

The library functor Remote.Run performs most of the respective procedure: it connects to a remote machine by using a low-level service (such as ssh), and starts a slave process that immediately connects to the master. It evaluates a component and sends back the resulting module.

The signature of Run is the following:

```
functor Run (val host : string
             signature RESULT
             functor F : COMPONENT'MANAGER → RESULT) : RESULT
```

Run is a polymorphic functor (Section 3) with two concrete arguments: the name of the remote host, and a functor that basically defines a dynamic component (Section 5.4) to be be evaluated on the remote machine. It takes a structure representing a component manager (Section 5.3), which can be used to access local libraries and resources on the remote host.

By using proxies defined *outside* of the functor F, and by creating proxies *inside* the functor and exporting them in the result structure a two-way communication is immediately established.

We illustrate the use of Remote.Run with a practical example in the next section.

8 Example: Distributed Solver

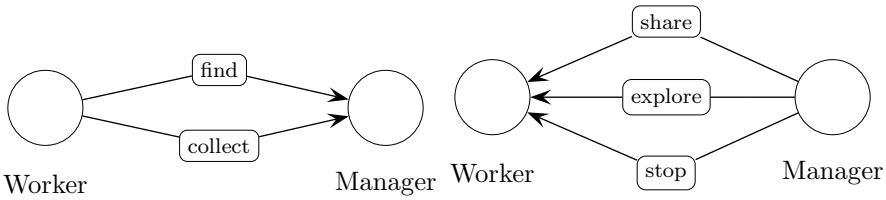
In this section, we illustrate the main features of Alice by showing the implementation of a distributed application, namely a distributed solver for constraint programming.

8.1 Description

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 failures (previous choices are inconsistent) or solutions. From a logical point of view, searching amounts only to traversing a tree and asking the status of each node.

In a distributed setting, each remote computer (*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. More information 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 follows (see Chapter 9 of [Sch02] for a more detailed presentation of this interface):



The find message is sent by a worker that is idle. It requests a job, that is, the path of a subtree that remains to be explored. When a worker encounters a solution, it sends

the collect message, along with the solution, to the manager. The share message 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 to the corresponding unexplored subtree (hence the type unit → path option below). The manager commands a worker to explore a subtree by sending the explore message along with the corresponding path. The stop message is used to stop the worker when the search is finished (for example, when the first solution was found in a one-solution search).

8.2 Implementation

The implementation of the distributed search engine consists basically of two components: the manager and the worker. The manager keeps a list of the workers it is connected to. Each worker has the same interface:

```
signature WORKER =
sig
    val share   : unit → path option
    val explore : path → unit
    val stop    : unit → unit
end
```

The manager creates workers by using the Remote.Run functor. Before that, the operations of the manager interface have to be defined. As explained above, the manager interface provides two functions: find and collect. They are implemented by proxies:

```
val find    = proxy (fn () ⇒ body)
val collect = proxy (fn sol ⇒ body)
```

The definition of a worker also takes place in the manager. Basically, we create a dynamic component (Section 5.4) in the form of a functor StartWorker. The worker interface provides three function operations: share, explore, and stop. They are also proxies. Additionally, the library used for constraint solving, named *Gecode*, is a native library that cannot be pickled, it is sited (Section 4.2). Thus, each worker needs to acquire this local library by using the component manager; this is the purpose of the call to the Link functor provided by the manager.

```
functor StartWorker (CM : COMPONENT_MANAGER) =
struct
    structure Gecode = CM.Link (val url = "x-alice:/lib/Gecode"
                                signature S = GECODE)
    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
```

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 functor Remote.Run repeatedly:

```
structure Worker = Remote.Run (val host      = hostname
                               signature RESULT = WORKER
                               functor Start    = StartWorker)
```

Each newly created worker is stored in the worker list. Then, the search starts by sending the root path of the search tree to the first worker of the list, then ask it for some work to give to other workers.

8.3 Concurrency

Note that the manager must be able to handle concurrent requests from workers. For example, the collect message stores the given solution in a list, which is protected using a locking mechanism (Section 2.5). Noticeably, the list of collected solutions is returned immediately when the search engine starts, in the form of a future. The list is built concurrently while solutions are sent to the manager.

9 Implementation

An implementation of Alice must meet two key requirements: dealing efficiently with the future-based concurrency model, and supporting open programming by providing a truly platform-independent and generic pickling mechanism.

The appropriate technology to achieve platform independence is to use a virtual machine (VM) together with just-in-time (JIT) compilation to native machine code. Futures and light-weight threads are implemented at the core of the system, making concurrency and data flow synchronisation efficient. More detailed implementation notes can be found in a technical report [BK02].

9.1 Pickling

Pickling and unpickling are core VM services available for all data in the store. The pickler takes a store object and transforms the subgraph of objects reachable from there into a platform-independent external representation (a pickle) that is suitable for transportation over a network or storage in a file. The unpickler recreates a copy of the original subgraph from a pickle. Pickling and unpickling preserve the structure of the original

graph, including cycles and sharing. The low-level pickling service thus implements the transitive closure semantics sketched in Section 4.2.

The system also features a minimisation mechanism that maximises sharing by merging all equivalent subgraphs of a data graph, using variants of algorithms from automata theory [Tac03]. This produces highly compact and efficient pickles.

In order to support platform-independent external, as well as an efficient internal data representations, the system offers a generic transformation mechanism: in a pickle, certain data is marked to be transformed on unpickling. It will be converted to an internal, possibly platform-dependent format. The internal data is also marked and preserves enough information to recreate the external version during pickling. Floating point numbers for instance, have different efficient internal representations on different platforms, but their external, pickled representation must be uniform (using standard IEEE format).

9.2 Codes and Interpreters

Code is managed as heap-allocated data, it is subject to garbage collection and can be pickled and unpickled. While there is exactly one external representation (called *Alice Abstract Code*), the VM allows for different internal types of code – e.g. JIT-compiled native code and interpreted byte code – to coexist and cooperate. Different codes and interpreters can be selected on a per-procedure basis. Pickler and unpickler automatically convert between these internal codes and the external format. More details can be found in a technical report [BK02].

The advantage of this generic model is that for different purposes, different internal codes can be employed: usually, JIT-compiled native code delivers the best execution speed. For code only executed once, direct interpretation of the abstract code is superior. A debugging interpreter can make use of additional annotations and check more invariants – the user can decide to run just a certain function in debug mode, with the rest of the system being JIT compiled.

JIT compilation elegantly builds on only two existing mechanisms: the transformation mechanism of the unpickler (Section 9.1), and lazy futures (Section 2.2). Unpickling transforms abstract code into a lazy future, which triggers JIT compilation of the code on request and results in the respective native machine code. Thus, native code is only created when actually needed.

JIT compilation is the adequate way to deal with a future-based and open programming language: at runtime, a lot of optimisations can be applied that are not possible for a static compiler. The JIT compiler can be reflective, taking advantage of the ability to dynamically inspect referenced values. This is particularly important for the optimization of futures and cross-component references represented as futures (Section 5).

9.3 Safety

The Alice implementation does not yet give any safety guarantees – in particular, the integrity and signature information of components is not verified currently. While type-annotated code is rather standard, the ability to dynamically create components via pickling also requires a certain amount of type information about data in the heap. It hence was a conscious decision for the Alice project to focus on language design first and consider implementation-level safety issues in a second phase.

9.4 Distribution

Thanks to the pickling-based approach, distribution requires almost no support from the VM. In particular, the copying-based semantics implied by pickling avoid complex inter-process references that require involved schemes for distributed garbage collection. The only possible inter-process references are proxies. Distributed collection for dead proxies is not yet implemented in the current version of Alice, but will be added in the future.

10 Related Work

There is numerous work and languages concerned with some of the issues addressed by Alice. We discuss those that are closest to our work, with focus on functional languages.

10.1 Java

Java [GJS96] was the first major language designed for open programming and has been successfully used in practice. Unlike Alice, it is object-oriented with a rather inexpressive and weak type system.

Open programming is based on *reflection*, which allows other components to exploit type information constructively. We feel that general reflection is expensive, invites abuse, and in practice demands a rather limited type system. Packages, as found in Alice, may be considered as providing a specialised, more limited form of reflection that avoids these issues.

Concurrency and serialisation are simplistic in Java and require considerable support from the programmer. Code cannot be serialised, only class names can be used to represent code, which is an inflexible and fragile abstraction.

Unlike in Alice, no structural type checks are performed when a class is loaded, subsequent method calls may cause a NoSuchMethodError exception any time, undermining the static type system to the point that Java has to be considered a dynamically typed

language as soon as it is used for open programming. In contrast, the dynamic type check performed by unpack in Alice ensures that all invariants of the static type system are maintained – execution cannot fail later due to a type error.[8]

10.2 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 multiple inheritance and mixins.

Abstraction mechanisms are very expressive thanks to *bounded views* that specify a lower bound as well as an upper bound for the type of the abstracted value. For comparison, a signature in Alice is a lower bound of the abstracted component.

Still, concurrency and distribution in Scala suffer from the same shortcomings as 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.

10.3 Oz/Mozart

Many of the concepts in Alice have been inherited from Oz [Smo95, VH04] and its implementation, the Mozart programming system [Moz04]. Similar to Alice, it has high-level support for concurrency, pickling, components and distribution.

Components are defined in terms of simpler concepts [DKSS98], albeit slightly differently than in Alice, due to the absence of a type system (see below). Components are represented as pickles, but are not identified with them, as in Alice.

The distribution subsystem is even more expressive than in Alice, supporting distributed state and futures. However, this is traded off with a considerable increase in complexity of semantics and implementation (roughly ten-fold for the latter).

Unlike Alice, Oz is based on a relational core language with logic variables being the primary means for data flow synchronisation. Futures have been added as an additional concept in later versions, to allow for safer concurrency abstractions.

The most significant shortcoming of Oz is its lack of a static type system, and the language's structure probably precludes adding one.

[8] More precisely, the use of laziness may delay type errors, but the package semantics prevents performing a call into a component successfully, and still encounter type failure for subsequent calls afterwards, due to inconsistent interface assumptions – *all* assumptions are checked before the very first use along a given import edge.

10.4 Acute

Acute [SLW+04] is an experimental, very expressive ML-based language for strongly typed open programming. It is probably closest in spirit to Alice.

Similar to Alice, it features pickling (marshalling), abstraction safety, and a component concept [Sew01, BHS+03]. Pickling incorporates a mechanism for rebinding resources in the target process in a controlled way, using explicit marks in the program. In contrast, Alice requires resources to be reimported from the component manager on the target site. This limitation is necessary for achieving sandboxing. No comparable safety mechanism is built-in in Acute, all details of distribution are left to the programmer.

Components are roughly comparable in power to Alice, but have additional support for versioning. Unlike Alice, Acute's component concept is not definable in terms of orthogonal, simpler concepts. We hence feel that it is more ad-hoc than Alice's.

Like in Alice, abstraction safety in Acute can be achieved by dynamic type generativity, but different levels of generativity are available [LPSW03], providing greater flexibility. In Alice, this has to be simulated by creating evaluated components via pickling (Section 4.5). As a fundamental philosophical difference, Acute allows abstractions to be broken by explicit means, for the sake of graceful dealing with versions.

10.5 JoCaml

JoCaml [FMS01] is an innovative distributed extension of Objective Caml [Ler03] based on the Join calculus [FGL+96].

Concurrency and distribution are more high-level than in Alice: channels are used to achieve both, concurrency and distribution, with expressive means of synchronisation and thread migration. Synchronisation, based on Join Patterns, can be programmed in a declarative manner, enabling elegant formulations of various concurrency abstractions. The future concept of Alice is more primitive, although sufficient to express join patterns as an abstraction with moderate size (ca. 200 lines for an experimental implementation).

On the downside, JoCaml is not open: dynamic typing and pickling is restricted to monomorphic values that can only be stored on a global name server, and there is no explicit component concept.

10.6 CML

CML [Rep99] is a mature concurrent extension of SML. It is based on first-class channels and synchronisation events, where synchronization has to be fully explicit.

The set of synchronisation events is primitive and cannot be extended easily by the programmer. In contrast, futures allow for straightforward programming of user-defined synchonisation abstractions.

CML does not address distribution or other aspects of open programming.

10.7 Erlang

Erlang [AVWW96, Arm03] is a language designed for embedded distributed applications, that has been applied successfully in the context of large industrial telecommunication projects.

It is purely functional with an additional, impure process layer. Processes communicate over implicit, process-global message channels. Erlang is untyped, but has a rich repertoire for dealing with failure.

Erlang is not designed for open programming, and does not directly support code mobility nor safety policies. A distinctive feature of the run-time environment is the ability to update code in active processes, however.

10.8 Clean

Clean is a concurrent, purely functional language with a type-safe import/export facility based on simple dynamics [Pv00, Pil96]. It does not have advanced support for components or distribution.

11 Outlook

We presented the design of Alice, a functional language for open programming. Alice provides a novel combination of coherent features to provide concurrency, modularity, a flexible component model and high-level support for distribution. Alice is strongly typed, and incorporates a module-based variant of dynamics to embrace openness, supporting abstraction safety and typeful programming dynamically.

Alice is fully implemented with a rich programming environment [Ali04], and some small to medium-size demo applications have already been implemented with it, including a distributed constraint solver, a multi-player network game, and a simple framework for generating web pages with active content scripted in Alice.

There is not yet a formal specification of the full language. Moreover, the implementation does not yet provide extra-lingual safety and security on the level of pickles. To that end, byte code *and* heap need to carry sufficient type information to allow creation of verifiable pickles. Research on these issues has been left for future work.

Other open questions we plan to address in the future are – among others – applicative type generativity, a semantics for unloading or even updating components in a manager, and higher-level library abstractions for concurrency and distribution.

Acknowledgements We thank our former colleague Leif Kornstaedt, who co-designed the Alice System and invested invaluable amounts of work into making it fly.

References

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1), 1995.

[Ali04] The Alice Project. http://www.ps.uni-sb.de/alice, 2004. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany.

[Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors.* Doctoral dissertation, Stockholm, Sweden, 2003.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice-Hall, 1996.

[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.

[BK02] Thorsten Brunklaus and Leif Kornstaedt. A virtual machine for multi-language execution. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[Car91] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts.* Springer-Verlag, Berlin, Germany, 1991.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Principles of Programming Languages*, New Orleans, USA, 2003.

[DKSS98] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 1998.

[FF95] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *Principled of Programming Languages*, San Francisco, USA, 1995.

[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*. Springer-Verlag, 1996.

[FMS01] Cédric Fournet, Luc Maranget, and Alan Schmitt. *The JoCaml Language beta release*. INRIA, http://pauillac.inria.fr/jocaml/htmlman/, 2001.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification*. Addison–Wesley, 1996.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 7(4), 1985.

[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 *Principles of Programming Languages*, San Francisco, USA, 1995. ACM.

[Ler03] Xavier Leroy. *The Objective Caml System*. INRIA, 2003.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA, 1997.

[LM93] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4), 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.

[Moz04] Mozart Consortium. The Mozart programming system, 2004. www.mozart-oz.org.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *TOPLAS*, 1988.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Myc83] Alan Mycroft. Dynamic types in ML, 1983. Draft article.

[NSS02] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. Concurrent computation in a lambda calculus with futures. Technical report, Universität des Saarlandes, 2002.

[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 Féd. de Lausanne, 2005.

[Pil96] Marco Pil. First class file I/O. In *IFL'96*, LNCS, vol.1268. Springer-Verlag, 1996.

[Pv00] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report*, 2000.

[Rep99] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Rey83] John Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, Amsterdam, 1983. North Holland.

[RLT+05] 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, 2005. Intellect.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Principles and Practice of Declarative Programming*, Uppsala, Sweden, 2003.

[Ros05] Andreas Rossberg. The definition of Standard ML with packages. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2005. http://www.ps.uni-sb.de/Papers/.

[Rus98] Claudio Russo. *Types for Modules.* Dissertation, University of Edinburgh, 1998.

[Rus00] Claudio Russo. First-class structures for Standard ML. In *ESOP*, Berlin, Germany, 2000.

[Sch02] Christian Schulte. *Programming Constraint Services*, volume 2302 of *LNAI*. 2002.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *28th Symposium on Principles of Programming Languages*, London, UK, January 2001.

[SLW+04] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. Technical Report RR-5329, INRIA, 2004.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*. 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.

[Tac03] Guido Tack. Linearisation, minimisation and transformation of data graphs with transients. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, 2003.

[TL04] Guido Tack and Didier Le Botlan. Compositional abstractions for search factories. In *MOZ 2004, Charleroi, Belgium*, LNCS. Springer-Verlag, 2004.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[Wad89] Philip Wadler. Theorems for free! In *FPCA*. ACM Press, 1989.