Mutually Iso-recursive Subtyping (Expanded)

ANDREAS ROSSBERG, Munich, Germany

Iso-recursive types are often taken as a type-theoretic model for type recursion as present in many programming languages, e.g., classes in object-oriented languages or algebraic datatypes in functional languages. Their main advantage over an equi-recursive semantics is that they are simpler and algorithmically less expensive, which is an important consideration when the cost of type checking matters, such as for intermediate or low-level code representations, virtual machines, or runtime casts. However, a closer look reveals that iso-recursion cannot, in its standard form, efficiently express essential type system features like mutual recursion or non-uniform recursion. While it has been folklore that mutual recursion and non-uniform type parameterisation can nicely be handled by generalising to higher kinds, this encoding breaks down when combined with subtyping: the classic "Amber" rule for subtyping iso-recursive types is too weak to express mutual recursion without falling back to encodings of quadratic size.

We present a foundational core calculus of iso-recursive types with *declared* subtyping that can express both inter- and intra-recursion subtyping without such blowup, including subtyping between constructors of higher or mixed kind. In a second step, we identify a syntactic fragment of this general calculus that allows for more efficient type checking without "deep" substitutions, by observing that higher-kinded iso-recursive types can be inserted to "guard" against unwanted β-reductions. This fragment closely resembles the structure of typical nominal subtype systems, but without requiring nominal semantics. It has been used as the basis for a proposed extension of WebAssembly with recursive types.

CCS Concepts: • **Software and its engineering → Data types and structures**; **Semantics**; *Polymorphism.*

Additional Key Words and Phrases: type systems, recursive types, subtyping, higher-order subtyping

ACM Reference Format:
Andreas Rossberg. 2023. Mutually Iso-recursive Subtyping (Expanded) . *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 234 (October 2023), 58 pages. https://doi.org/10.1145/3622809

1 INTRODUCTION

Recursive types are a central element of typed programming languages. Object-oriented programming would be impossible without recursion between classes or their instances' object types. Similarly, typed functional programming heavily depends on the ability to form recursive algebraic data types. Even classical procedural languages usually have the means to define recursive record types. In all cases, multiple types can also be *mutually* recursive with each other.

Two different semantic interpretations of type recursion have been studied [Pierce 2002; Crary et al. 1999]: *equi-recursive* types, which are implicitly equivalent to their unrollings, and *iso-recursive* types, where folding and unfolding are explicit injection/projection operators on the term level.

Although equi-recursive types are generally more flexible and expressive, their coinductive nature results in a considerably more complicated meta-theory as well as higher algorithmic complexity. Types correspond to regular trees, i.e., cyclic graphs in this model, and relations like type equivalence or subtyping are as complex to verify as equivalence or inclusion between finite state automata [Kozen et al. 1993].

2475-1421/2023/10-ART234
https://doi.org/10.1145/3622809

Iso-recursive types on the other hand stay nicely inductive, with a simple interpretation as trees. That way, common meta operations are efficiently implementable with bottom-up algorithms:

- type equivalence check,
- subtyping check,
- type canonicalisation (i.e., unifying the representation of equivalent types).

Interestingly, despite their simpler nature, Abadi & Fiore [1996] have shown that every typing derivation in an equi-recursive setting can still be embedded into an iso-recursive type system, for the price of inserting suitable conversion terms.

These properties make iso-recursive types the far more attractive choice for type systems that require efficient decision procedures. Our particular motivation lies in type-safe low-languages like WebAssembly (Wasm) [Haas et al. 2017]. For Wasm, type checking has to be performed online, each time a new module is validated, instantiated, and linked, so that Wasm engines crucially dependent on all of the above operations. Consequently, an extension of Wasm with recursive types, as needed for proposed extensions like garbage collection [Rossberg 2022] or continuations [Phipps-Costin et al. 2023; Lindley et al. 2022], want these algorithms to be as cheap as possible.

However, it turns out that iso-recursive types, as usually defined, still have shortcomings:

- *Expressiveness.* As we will see, the standard bare-bones semantics for iso-recursive types is neither sufficient to maintain linear size in the presence of mutual type recursion, nor can it fully express recursion between generic types. And while it is folklore knowledge how to extend iso-recursion to *n*-ary and higher-kinded fixpoints, these encodings break down once subtyping is thrown into the mix. In other words, standard iso-recursive types cannot adequately express central features in contemporary programming languages.
- *Performance.* For common operations like equivalence and subtyping that are performed many, many times during type checking, even time linear in the size of the types is much too expensive. Equivalence can be brought down to constant time with the help of type canonicalisation. Subtyping can only be improved by caching checks already made [Gapeyev et al. 2002]. However, managing such caches complicates engines. For their purposes, an *eager* approach is more desirable, where the subtyping relation is known before-hand.

In this paper, we set out to answer two questions:

(1) What is missing from the standard theory of iso-recursive types to express mutual subtyping faithfully, and what is the minimal extension to bridge the gap?
(2) What is a suitable fragment of such a general calculus that minimises the algorithmic cost of checking subtyping while still covering a broad range of use cases?

Our contributions hence are as follows:

- We recap the folklore encoding of mutual as well as non-uniform recursive types in an extension of higher-kinded iso-recursive types and analyse its limitations. (Section 2)
- We demonstrate that these encodings are insufficient in the presence of subtyping and show how the subtyping mechanism for iso-recursive types can be modified to allow faithful encoding of subtyping between individual members of multiple recursion groups. (Section 3)
- We develop a foundational calculus of higher-order iso-recursive types with *pre-declared* subtyping bounds that can express these encodings. We prove relevant properties like type soundness and cut elimination, yielding a sound and complete subtype algorithm. (Section 4)
- We identify a practical fragment of the general calculus in which reduction and substitution remain *shallow*, and hence cheap enough for real-world implementations in a language runtime. We prove that this fragment is still sufficient to encode the subtyping relation of a suitable variant of Java with generics and briefly report on its use in Wasm. (Section 5)

We conclude with discussion and related work (Sections 6 & 7).

2 ISO-RECURSIVE TYPES

Under its textbook semantics, an iso-recursive type is formed as $\mu\alpha.\tau$, binding α as a type variable under which the type can refer to itself recursively within its actual definition τ. For example, consider the following record type, defining nodes for a singly-linked list of integers:

$$\begin{array}{l}\mathsf{struct}\ \mathit{List}\ \{\\ \quad \mathit{head} : \mathit{Int}\\ \quad \mathit{tail} : \mathit{List}\\ \}\end{array}$$

Such a type could be represented as follows:

$$\mathit{list} \quad = \quad \mu\alpha.\{\mathit{head} : \mathsf{int}, \mathit{tail} : \alpha\}$$

Values of iso-recursive type must be introduced and eliminated using term-level operators roll and unroll (often called fold and unfold), whose respective typing rules add and remove one level of recursion from their operand's type, respectively:

$$\frac{\tau = \mu\alpha.\tau' \qquad \Gamma \vdash e : \tau'[\tau/\alpha]}{\Gamma \vdash \mathsf{roll}_\tau\ e : \tau}\ (\textsc{T-Roll}) \qquad \frac{\tau = \mu\alpha.\tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{unroll}\ e : \tau'[\tau/\alpha]}\ (\textsc{T-Unroll})$$

The type annotation on the roll-operator is necessary to ensure that its type is uniquely determined.

With that, a value of type *list* — say, the list $[1, 2]$ — is representable with the following term:

$$l \quad = \quad \mathsf{roll}_{\mathit{list}}\ \{\mathit{head} = 1, \mathit{tail} = \mathsf{roll}_{\mathit{list}}\ \{\mathit{head} = 2, \mathit{tail} = \mathsf{null}\}\}$$

(For the sake of simple examples, we dare to assume that record types are inhabited by null.)

Accordingly, to inspect the list — e.g., access its first element — its type has to be unrolled first:

$$h \quad = \quad (\mathsf{unroll}\ l).\mathit{head}$$

In this paper we never need the ability to reorder the fields of record types — our records are effectively just tuples. In the remainder of the paper we will hence omit record labels to avoid clutter, and express the prior definitions more tersely:

$$\begin{array}{rcl}\mathit{list} & = & \mu\alpha.\{\mathsf{int}, \alpha\}\\ l & = & \mathsf{roll}_{\mathit{list}}\ \{1, \mathsf{roll}_{\mathit{list}}\ \{2, \mathsf{null}\}\}\\ h & = & (\mathsf{unroll}\ l).1\end{array}$$

2.1 Mutual recursion

Let us now consider two mutually recursive types: a node type for n-ary trees, whose children are represented as a forest, which in turn is a list of trees:

$$\begin{array}{l}\mathsf{struct}\ \mathit{Tree}\ \{\\ \quad \mathit{value} : \mathit{Int}\\ \quad \mathit{children} : \mathit{Forest}\\ \}\end{array} \qquad \begin{array}{l}\mathsf{struct}\ \mathit{Forest}\ \{\\ \quad \mathit{child} : \mathit{Tree}\\ \quad \mathit{rest} : \mathit{Forest}\\ \}\end{array}$$

Since this recursion spans two types, it can only be expressed by nesting the μ-operator:

$$\begin{array}{rcl}\mathit{tree} & = & \mu\alpha.\{\mathsf{int}, \mu\beta.\{\alpha, \beta\}\}\\ \mathit{forest} & = & \mu\beta.\{\mathit{tree}, \beta\}\end{array}$$

Each type needs to have a closed definition, so we are forced to inline a local view of *forest* into *tree*'s definition (of course, we could also do it the other way round). Note in particular that this "inlined" version needs to be syntactically different from *forest* itself, because it has to use the *internal* type variable α to refer to *tree* recursively. So both type expressions cannot be deduped.

Still, that may not look so bad with just two types, but to see where it leads, consider a schematic case of a recursion group of three types, with additional fields u, v, w to make them different:

$$\begin{array}{l}\text{struct } A\ \{u : U;\ \ a : A; b : B; c : C\}\\ \text{struct } B\ \{v : V;\ \ a : A; b : B; c : C\}\\ \text{struct } C\ \{w : W; a : A; b : B; c : C\}\end{array}$$

Here is how these types would have to be expressed with vanilla iso-recursive types:

$$\begin{array}{lcl}A & = & \mu a.\{U, a, (\mu b.\{V, a, b, (\mu c.\{W, a, b, c\})\}), (\mu c.\{W, a, (\mu b.\{V, a, b, c\}), c)\}\}\\ B & = & \mu b.\{V, A, b, (\mu c.\{W, A, b, c\})\}\\ C & = & \mu c.\{W, A, B, c\}\end{array}$$

Inlining once is not enough, we have to do it — including transitive inlinings — 5 times! And all inlinings of the same type are syntactically different, such that no sharing of its occurrences is possible. It is not difficult to see that the total size of the type definitions grows *quadratically* with the number of types in a mutual recursion group — or even worse when each type has more than one occurrence of the other type names.

With higher-order types at hand, we could try to factor out the commonalities by functorising each type definition:

$$\begin{array}{lcl}AF(BG, CG) & = & \mu a.\{U, a, BG(a), CG(a)\}\\ BF(AG, CG) & = & \mu b.\{V, AG(b), b, CG(b)\}\\ CF(AG, BG) & = & \mu c.\{W, AG(c), BG(c), c\}\end{array}$$

Here, AF, BF, CF are definitions of A, B, C, parameterised by the respective other types within their recursion group. The parameters themselves in turn need to be parameterised by the recursive type they are woven into, in order to vary the occurrences, making this a higher-order abstraction.

That looks fancy, but unfortunately, still leads to quadratic growth when tying the knots:

$$\begin{array}{lcl}A & = & AF(\lambda a.BF(\lambda b.a, \lambda b.CF(\lambda c.a, \lambda c.b)), \lambda a.CF(\lambda c.a, \lambda c.BF(\lambda b.a, \lambda b.c)))\\ B & = & BF(\lambda b.A, \lambda b.CF(\lambda c.A, \lambda c.b))\\ C & = & CF(\lambda c.A, \lambda c.B)\end{array}$$

And of course, the normal forms of these types will also be no different from our first attempt.

Overall, we can conclude that vanilla iso-recursive types scale poorly. On paper, they maintain linear type checking, but their encoding of a program with mutually recursive types is in fact polynomial in size.[1] That makes them rather unsuitable for use in certain systems, such as intermediate languages or low-level languages like Wasm.

2.2 Non-uniform recursion

Another limitation of vanilla iso-recursive types is that they can only express *regular* types — sometimes called *uniform* recursion — where each recursive use of a constructor must preserve the head shape, i.e., the original parameters. For example, a generic version of our prior list node type is fine, because *List* is only applied to the original X recursively:

$$\begin{array}{l}\text{struct } List\langle X\rangle\ \{\\ \quad head : X\\ \quad tail : List\langle X\rangle\\ \}\end{array}$$

[1] Note that this is worse than for equi-recursive types. While their *textual* representation grows in the same manner, the respective terms merely express a cyclic graph that is linear in size when minimised. That is not possible in the iso-recursive setting, because the inlined occurrences are actually different types, only equivalent after *explicit* unrolling.

A regular recursive type like this can be expressed as a type constructor wrapped around an ordinary iso-recursive type (we write Ω for the ground *kind*):

$$list \quad = \quad \lambda x{:}\Omega.\mu\alpha.\{x, \alpha\}$$

Notably, the recursive application has been replaced by a plain α, and the lambda has effectively been lifted out of the recursion.

But what if that is not possible? Many programming languages actually allow non-uniform recursion and irregular types. For example, consider the following Haskell data type that defines square matrices [Okasaki 1999]:

$$\begin{array}{l}\text{type } \mathit{Quad}\ a \ \ = (a, a, a, a)\\ \text{data } \mathit{Square}\ a = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Square}\ (\mathit{Quad}\ a))\end{array}$$

Note how the recursive application of *Square* is to *Quad a*, not just *a* like on the left-hand side. Lifting the abstraction out of the recursion becomes impossible.

Various patterns of object-oriented programming also use irregular types. For example, a container library for Java might want to provide an interface method like the following:

$$\begin{array}{l}\text{interface } \mathit{List}\langle X\rangle\ \{\\ \quad \mathit{List}\langle \mathit{List}\langle X\rangle\rangle\ \mathit{groupBy}(\mathit{Func}\langle X, X, \mathit{Bool}\rangle\ \mathit{eq});\\ \}\end{array}$$

2.3 Higher-kinded iso-recursion

The folklore solution to both problems described so far is to generalise iso-recursion to both *n*-ary fixpoints and higher kinds [Pierce 2002]. In fact, higher kinds are sufficient to express *n*-ary fixpoints as well, since they allow the introduction of *tuple kinds*. For example, a variation of this solution has been adopted in the type-theoretic interpretation of Standard ML [Harper and Stone 2000].

To see how this works, consider a type system with the following kinds:

$$\kappa ::= \Omega \mid \kappa \times \kappa \mid \kappa \longrightarrow \kappa$$

At the type level, we have corresponding introduction and elimination forms for these higher kinds, namely type pairs and type functions:

$$\tau ::= \ldots \mid \langle \tau, \tau \rangle \mid \tau.i \mid \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau$$

With this, we can allow iso-recursive types to be introduced at any kind κ, such that they take the form $\mu\alpha{:}\kappa.\tau$. But power comes with responsibility, so we must also do more work in the typing rules for roll/unroll:

$$\frac{\tau = \mu\alpha{:}\kappa.\tau' \qquad \Gamma \vdash e : T[\tau'][\tau/\alpha]}{\Gamma \vdash \mathsf{roll}_{T[\tau]}\ e : T[\tau]}\ \text{(T-Roll-HO)} \qquad \frac{\tau = \mu\alpha{:}\kappa.\tau' \qquad \Gamma \vdash e : T[\tau]}{\Gamma \vdash \mathsf{unroll}\ e : T[\tau'][\tau/\alpha]}\ \text{(T-Unroll-HO)}$$

To formulate these rules in a compact and uniform fashion, we define *type elimination contexts* T:

$$T ::= [\_] \mid T.i \mid T\ \tau$$

The first brackets in $T[\tau'][\tau/\alpha]$ fill the hole in that context, while the second express a substitution. Let's digest this with an example, a type constructor $\tau = \mu\alpha{:}(\Omega \to \Omega).\lambda\beta{:}\Omega.\{\beta, \alpha(\beta)\}$. Then the term

$$\mathsf{roll}_{\tau(nat)}\{1,\ \mathsf{roll}_{\tau(nat)}\{2,\ \mathsf{roll}_{\tau(nat)}\{3,\ \mathsf{roll}_{\tau(nat)}\ \mathsf{null}\}\}\} \quad : \quad \tau(nat)$$

creates what can be interpreted as the list $[1, 2, 3]$ of natural numbers: at each roll, T is $[\_](nat)$ and the operand is of type $T[\lambda\beta{:}\Omega.\{\beta, \alpha(\beta)\}][\tau/\alpha] \equiv (\lambda\beta{:}\Omega.\{\beta, \tau(\beta)\})(nat) \equiv \{nat, \tau(nat)\}$.

Since terms can only have types of ground kind Ω, the context T occurring in the typing rules necessarily has to eliminate *all* higher kinds from κ in order for roll/unroll terms to be well-typed.

Okay, but how does this solve our previous problems?

Tuple kinds can express a single fixpoint over a tuple of multiple mutually recursive types. For example, the forest from Section 2.1 can be encoded almost directly and without any duplication:

$$\begin{array}{rcl} \mathit{tree\_forest} & = & \mu\alpha : \Omega \times \Omega.\ \langle \{\mathsf{int}, \alpha.2\}, \{\alpha.1, \alpha.2\} \rangle \\ \mathit{tree} & = & \mathit{tree\_forest}.1 \\ \mathit{forest} & = & \mathit{tree\_forest}.2 \end{array}$$

At this stage we should point out that we are dealing with a calculus here, and the equations above are simply meta-level definitions. Technically, they mean that *tree* is in fact the type $(\mu\alpha : \Omega \times \Omega.\langle\{\mathsf{int}, \alpha.2\}, \{\alpha.1, \alpha.2\}\rangle).1$. And similarly, *forest* contains the entirety of the same μ-type. So, are we not buying into some textual "code duplication" with this as well?

Well, only virtually. Of course, a real programming language will offer language-level means to name types, avoiding textual duplication. And a real implementation can *share* the representation for occurrences of the same type. Hence, we liberally use meta-level definitions as if they were language-level definitions, and assume that multiple uses of the same type incur no extra cost (and come back to explicitly named type definitions in Section 5.2).

The fact that all occurrences of the *tree_forest* type are syntactically the same is what sets this representation apart from the quadratic encoding shown in Section 2.1, because that had to duplicate type terms with slight modifications for each of the occurrences, such that they cannot be named or shared in a single definition!

Let us return to the problem at hand, and turn to arrow kinds. Given those, it is possible to keep the lambda encoding of a type parameter *inside* the recursion and apply it heterogeneously, unlike before. Recall the square matrix example from Section 2.2, which can now be expressed as:

$$\begin{array}{rcl} \mathit{quad} & = & \lambda a : \Omega.\ a \times a \times a \times a \\ \mathit{square} & = & \mu\beta : \Omega \to \Omega.\ \lambda a : \Omega.\ a + \beta\ (\mathit{quad}\ a) \end{array}$$

Of course, both tuple and arrow kinds can also be combined freely, forming non-uniform mutually recursive generic types. We will see an example later.

It is important to note that despite the fancy extra kinds, we are still dealing with iso-recursive types, meaning that there is no implicit unfolding on the type level. Consequently, a type like *tree*, that is, $(\mu\alpha : \Omega\times\Omega.\ \langle\{\mathsf{int}, \alpha.2\}, \{\alpha.1, \alpha.2\}\rangle).1$, or an application like $(\mu\alpha : \Omega \to \Omega.\lambda\beta : \Omega.\{\beta, \alpha\})\ \mathit{int}$, are already in normal form and cannot be reduced further! A redex is only produced in a controlled manner by the unrollings applied in the typing rules for roll and unroll. That ensures that even this higher-order extension still is algorithmically well-behaved and normalisation terminates.

As mentioned, this technique has been successfully applied to model real-world languages like Standard ML. And at this point it may seem like we have a satisfying and scalable solution to the problem of modelling programming language with iso-recursive types.

But do we? Well, wait until subtyping complicates everything, like it usually does.

3 SUBTYPING

Imagine the tree and forest node definitions from Section 2.1 were actually defined in an object-oriented language, as classes:

```
class Tree {                    class Forest {
  value : Int                     child : Tree
  children : Forest               rest : Forest
}                               }
```

The types of instances of these classes, when expressed with higher-order iso-recursive types, remain exactly the same as given in Section 2.3 (ignoring method tables):

$$\begin{array}{rcl} \mathit{tree\_forest} & = & \mu\alpha : \Omega \times \Omega.\ \langle \{\mathsf{int}, \alpha.2\}, \{\alpha.1, \alpha.2\} \rangle \\ \mathit{tree} & = & \mathit{tree\_forest}.1 \\ \mathit{forest} & = & \mathit{tree\_forest}.2 \end{array}$$

But according to object-oriented philosophy, it ought to be possible to define a subclass of *Tree* that adds an additional field:

$$\begin{array}{l} \mathsf{class}\ \mathit{ExtTree}\ \mathsf{extends}\ \mathit{Tree}\ \{ \\ \quad \mathit{info} : \mathit{Int} \\ \} \end{array}$$

The natural encoding of the instance type of this class would be:

$$\mathit{exttree} \quad = \quad \mu\alpha : \Omega.\ \{\mathsf{int}, \mathit{forest}, \mathsf{int}\}$$

But in order to be an adequate encoding, we also need this to be a subtype of the type *tree*!

So let us recall how subtyping works. The standard rule for subtyping iso-recursive types is sometimes dubbed the "Amber" rule, due its initial appearance in Cardelli's experimental language Amber [Cardelli 1986]. This rule, generalised to higher-kinded μ, looks as follows:

$$\frac{\Gamma, \alpha_2{:}\kappa, \alpha_1 {\leq} \alpha_2 \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \mu\alpha_1{:}\kappa.\tau_1 \leq \mu\alpha_2{:}\kappa.\tau_2}\ (\textsc{S-Rec-Amber})$$

Turning to type tuples, the most liberal rule that we could define would support both width and depth subtyping:

$$\frac{\Gamma \vdash \tau_{11} \leq \tau_{21} \qquad \cdots \qquad \Gamma \vdash \tau_{1N} \leq \tau_{2N}}{\Gamma \vdash \langle \tau_{11}, \ldots, \tau_{1N}, \ldots, \tau_{1M} \rangle \leq \langle \tau_{21}, \ldots, \tau_{2N} \rangle}$$

But combining these rules, there is no way in which $\mathit{exttree} \leq \mathit{tree\_forest}.1$ could possibly be derived. These types do not even have the same shape, despite already being in normal form! With just these rules and the previous encoding of mutually recursive types, only types from larger recursion groups can be subtypes of smaller (or equally sized) recursion groups, and only if there was a 1-to-1 subtype correspondence between the members of the groups.

Clearly, that's not sufficient. In a programming language, any member of a recursive group of types can be in a subtype relation with any member of another (or its own!) recursion group.

3.1 Coinductive generalisations of iso-recursive subtyping

One attempt at a fix could be to build unrolling into the subtyping rule:

$$\frac{\Gamma \vdash \tau_1[\mu\alpha_1{:}\kappa.\tau_1/\alpha_1] \leq \tau_2[\mu\alpha_2{:}\kappa.\tau_2/\alpha_2]}{\Gamma \vdash \mu\alpha_1{:}\kappa.\tau_1 \leq \mu\alpha_2{:}\kappa.\tau_2}\ (\textsc{S-Rec-Unroll})$$

A version of such a rule has previously been proposed by Ligatti et al. [2017], for example.

However, one problem with this rule is that it is no longer inductive. A subtyping assertion like $\mu\alpha.\{\alpha, \mathit{int}\} \leq \mu\alpha.\{\alpha\}$ would be reduced to the goal $\{\mu\alpha.\{\alpha, \mathit{int}\}, \mathit{int}\} \leq \{\mu\alpha.\{\alpha\}\}$, and then to $\mu\alpha.\{\alpha, \mathit{int}\} \leq \mu\alpha.\{\alpha\}$ again. To make this work, we need to assume recursively that the original relation holds, but then we are essentially in a coinductive setting.[2] At that point, we could just as well go back to equi-recursive types, since rule S-Rec-Unroll suffers from many of the same problems. In particular, every type is a subtype of its unrolling, e.g., $\mu\alpha.\{\alpha\} \leq \mu\alpha.\{\mu\alpha.\{\alpha\}\}$ (and vice versa), which can be repeated indefinitely by transitivity. Consequently, the algorithmic complexity

[2]Technically, a coinductive relation can be formulated with inductive rules by introducing an environment of assumptions to the subtyping judgement, like done by Ligatti et al. and originally in Amadio & Cardelli [1993]. The close relation between such a formulation and coinduction was first pointed out by Brandt and Henglein [1998]. Unfortunately, this trick does not eliminate the meta-theoretical and algorithmic repercussions.

of checking subtyping is no better than for equi-recursive types (concretely, it is $O(m \cdot n)$ vs $O(n^2)$, but with m, the number of recursive binders, which itself is linear in n in the worst case).

The other issue is that this rule does not even solve our original problem. It could still not handle *exttree* $\leq$ *tree_forest*.1, because the right-hand side of that assertion is not a μ-type syntactically, nor can it be reduced to one. We would need a higher-order extension, analogous to the typing rules T-Roll-HO and T-Unroll-HO, which might look as follows:

$$\frac{\Gamma \vdash T_1[\tau_1][\mu\alpha_1{:}\kappa_1.\tau_1/\alpha_1] \leq T_2[\tau_2][\mu\alpha_2{:}\kappa_2.\tau_2/\alpha_2]}{\Gamma \vdash T_1[\mu\alpha_1{:}\kappa_1.\tau_1] \leq T_2[\mu\alpha_2{:}\kappa_2.\tau_2]}\ \text{(S-Rec-Unroll-HO)}$$

This indeed would allow deriving *exttree* $\leq$ *tree_forest*.1, because both sides would be unfolded to records $\{\mathsf{int}, \textit{tree\_forest}.2, \dots\}$ in the premise. But the meta-theory of such a complex rule is rather non-obvious, not least due to its asymmetric elimination contexts.

A cleaner approach might be to separate the unrolling. For example, by adding a type equivalence rule that is a version of "Shao's equation" [Crary et al. 1999]:

$$\frac{}{\mu\alpha.\tau \equiv \mu\alpha.\tau[\mu\alpha.\tau/\alpha]}\ \text{(E-Rec-Shao)}$$

In its bare form, this rule is not quite enough, we would again need a higher-order generalisation that mirrors the unrolling happening in the typing rules T-Roll and T-Unroll:

$$\frac{}{T[\mu\alpha{:}\kappa.\tau] \equiv \mu\alpha{:}\Omega.T[\tau][\mu\alpha{:}\kappa.\tau/\alpha]}\ \text{(E-Rec-Shao-HO)}$$

Combined with S-Rec-Amber and transitivity, this rule appears to subsume S-Rec-Unroll-HO. And it would allow deriving *exttree* $\leq$ *tree_forest*.1, because *tree_forest*.1 $\equiv \mu\alpha{:}\Omega.\{\mathsf{int}, \textit{tree\_forest}.2\}$ under this rule. However, such a rule obviously has the same problem as S-Rec-Unroll, namely that it puts us into coinductive territory, which is not where we want to be. Note in particular that Shao's equation is almost the same as the unrolling rule for equi-recursive types, except that the μ-binder is not eliminated on the outside. So, roll/unroll operators are still enforced on the term level, but we have not won anything for the type level — leaving us with the worst of both worlds.

3.2 Declared Subtyping

The previous attempts at solving the problem had in common that they tried to meddle with the "use site" of subtyping: by making the subtype relation more powerful, they would essentially allow to *infer* the desired subtype assertions where they are needed. Unfortunately, that has the inherent problem that more expressive subtyping will almost inevitably be more costly to check, and the respective computation has to be performed each time subsumption is invoked Or alternatively, extra overhead has to be invested into caching it.

In this paper, we hence explore the opposite direction and focus on the "definition site" of subtyping: that is, we require that all recursive subtype assertions used in a program are *declared* upfront with a recursive type's definition. That allows them to be verified once at definition time (ideally, in linear time), and after that, subsumption can be checked in approximately constant time.

While seemingly odd from the perspective of type theory, this of course is the dominant approach in practical programming languages. Although this design choice is typically related to *nominal* typing, where declaring supertype bounds upfront is inevitable, there is no principle reason why the approach cannot be applied in a *structurally* typed system, like an iso-recursive theory. It is particularly attractive for "internal" low-level languages like Wasm, for which the convenience of "inferred" subtyping is not relevant — it is fine to offload the work of collecting the necessary assertions to the source language compiler and have it insert the necessary declarations.

To this end, we propose a minimal modification of subtyping for iso-recursive types, where the μ-operator is extended with an upper bound[3]:

$$\mu\alpha{\leq}\tau_1.\tau_2$$

As usual in type systems with higher-order subtyping [Pierce and Steffen 1997; Compagnoni 1995], the bound τ_1 replaces the kind annotation, as it is now implied by τ_1. The bound declares the immediate supertype of the formed types — no other supertypes will be recognised than its immediate or indirect, declared supertypes. Consequently, the subtyping rule for μ-types can now be vastly simplified. Assuming that there are separate rules for reflexivity and transitivity, it's just:

$$\frac{}{\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 \leq \tau_1}\ \textsc{S-Rec-Sup}$$

But of course, in order for this rule to be sound, the validity of the declared subtype now needs to be verified as part of well-formedness of μ-types, hence the following kinding rule:

$$\frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma, \alpha \leq \tau_1 \vdash \tau_2 : \kappa \qquad \Gamma, \alpha \leq \tau_1 \vdash \tau_2 \leq \mathsf{unroll}_\kappa(\tau_1)}{\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 : \kappa}\ \textsc{K-Rec-Sup}$$

This rule requires that τ_2 is a suitable subtype of the *unrolling* of τ_1. The unroll meta-function is a kind-indexed generalisation of the unrolling substitution over a type elimination context that we saw earlier:

$$\begin{array}{lcll}
\mathrm{unroll}_\Omega(\tau) & = & T[\tau_2][\mu\alpha{\leq}\tau_1.\tau_2/\alpha] & \text{if } \tau \equiv T[\mu\alpha{\leq}\tau_1.\tau_2] \\
\mathrm{unroll}_\Omega(\tau) & = & \top & \text{if } \tau \equiv \top \\
\mathrm{unroll}_{\kappa_1\times\kappa_2}(\tau) & = & \langle \mathrm{unroll}_{\kappa_1}(\tau.1), \mathrm{unroll}_{\kappa_2}(\tau.2)\rangle & \\
\mathrm{unroll}_{\kappa_1\to\kappa_2}(\tau) & = & \lambda\alpha{:}\kappa_1.\mathrm{unroll}_{\kappa_2}(\tau\,\alpha) &
\end{array}$$

At ground kind, unrolling is only defined for μ-types and for the top type. For higher kinds the unrolling is pushed inwards by means of η-expansion.[4] Consequently, the bound on a μ-type either has to be top or a μ-type itself, or a higher-order type producing those.

One technical complication with these two rules is that kinding and subtyping relations now are mutually dependent. That could cause severe pain for the meta-theory, but it turns out that it is harmless in this instance, because the subtyping premise on K-Rec-Sup can be "ignored" for the purpose of type normalisation, thereby allowing to stratify the meta-theoretical development; see Appendix A.4 and A.5 for details.

The typing rules for the roll and unroll operators remain unchanged as given in Section 2.3.

We can now use the desired definition for encoding the mutually recursive instance types of the *Tree* and *Forest* classes, except that we have to give them bounds. Since there are no supertypes, we use the top type for both:

$$\begin{array}{rcl}
\mathit{tree_forest} & = & \mu\alpha \leq \langle\top,\top\rangle.\ \langle\{\mathsf{int}, \alpha.2\}, \{\alpha.1, \alpha.2\}\rangle \\
\mathit{tree} & = & \mathit{tree_forest}.1 \\
\mathit{forest} & = & \mathit{tree_forest}.2
\end{array}$$

The instance type of the subclass *ExtTree* gets a more interesting supertype:

$$\mathit{exttree} \quad = \quad \mu\alpha \leq \mathit{tree}.\ \{\mathsf{int}, \mathit{forest}, \mathsf{int}\}$$

It is easy to check that *tree_forest* is well-formed according to Rule K-Rec-Sup, since in the premise, $\mathrm{unroll}_{\Omega\times\Omega}(\langle\top,\top\rangle) \equiv \langle\top,\top\rangle$, so we only need to check each record against $\top$.

[3]We defer the discussion of multiple supertypes to Section 7.

[4]In a type system with η-convertibility already built into the type equivalence relation, the η-expansion and thus the indexing by kinds wouldn't be necessary, and $\mathrm{unroll}(T[\mu\alpha{\leq}\tau_1.\tau_2]) = T[\tau_2][\mu\alpha{\leq}\tau_1.\tau_2/\alpha]$ would suffice for all kinds.

For *exttree*, unrolling yields $\mathsf{unroll}_\Omega(\mathit{tree}) \equiv \{\mathsf{int}, \mathit{tree\_forest}.2\}$, which is the instance type of *tree* and a straightforward width supertype of *exttree*'s representation.

Note that the type term for *exttree* will virtually contain its entire superclass hierarchy in the bound, which may seem to induce serious "code duplication". But as pointed out in Section 2.3, this is merely an artefact of dealing with a substitution-based *calculus*. In a proper programming language, the usual naming mechanisms prevent it from manifesting physically.

3.3 Intra-Recursion Subtyping

But we are still not quite done. There is one more annoying complication: we must also allow subtyping between the types of a single recursion group! For example, the analogue of the following code is perfectly legal in most object-oriented languages:

$$\mathsf{class}\ A\ \{\ x : B\ \} \qquad\qquad \mathsf{class}\ B\ \mathsf{extends}\ A\ \{\ y : A\ \}$$

If we were to try translating that into iso-recursive types with our recipe above, we would end up with something like the following:

$$AB \quad = \quad \mu\alpha \leq \langle \top, \alpha.1 \rangle . \langle \{\alpha.2\}, \{\alpha.2, \alpha.1\} \rangle$$

But obviously, this is not well-formed, because α is not in scope in its own bound.

Unfortunately, this example is not expressible with the rules given so far, even with a more sophisticated encoding. To see why, observe that we ultimately need the following relation to hold:

$$AB.2 \quad \leq \quad AB.1$$

But the only direct supertypes of a μ-type like AB are either $\top$ or another μ-type. In the latter case, this other type (or a type equivalent to it) has to occur as the bound, i.e., as a subterm, in order for rule S-Rec-Sup to apply. The argument can be iterated transitively for any indirect supertypes. Consequently, for the above to hold, the whole μ-type AB would have to contain itself as a proper subterm, which is syntactically impossible, no matter how clever we are trying to be about decomposing and layering the recursion (e.g., with nested μ-types).

What now?

One solution would be to generalise μ-types to an F-bounded semantics, where α is allowed to occur freely within its own bound [Canning et al. 1989; Baldan et al. 1999]. Such an extension is interesting in its own right, because many object-oriented languages indeed support F-bounded inheritance. However, F-bounded quantification complicates the meta-theory significantly. It hence seems wiser to investigate such a generalisation separately in future work, while also demonstrating that it is not required to address the more earthly problem of intra-recursion subtyping.

In this light, we restrain ourselves to a more moderate extension. Namely, we make μ-types with tuple bounds primitive:

$$\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle . \tau$$

Here, the later tuple component in the bound is allowed to refer to earlier one, but not vice versa:

$$\frac{\begin{array}{cc} \Gamma \vdash \tau_1 : \kappa_1 & \Gamma, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \vdash \tau : \kappa_1 \times \kappa_2 \\ \Gamma, \alpha_1 \leq \tau_1 \vdash \tau_2 : \kappa_2 & \Gamma, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \vdash \tau \leq \mathsf{unroll}^{[\tau.1/\alpha_1]}_{\kappa_1 \times \kappa_2}(\langle \tau_1, \tau_2 \rangle) \end{array}}{\Gamma \vdash \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle . \tau : \kappa_1 \times \kappa_2}\ \text{K-Rec-Tup}$$

Since the bound τ_2 may refer to α_1, which would involve checking that the second component is a subtype of the first, its representation has to be substituted during the subtype check. For that purpose, the definition of unrolling is extended with a variable case and a substitution parameter

applied in that case. The base case of unrolling is also modified accordingly:

$$\begin{array}{lcll} \mathrm{unroll}^{\sigma}_{\Omega}(T[\tau]) & = & T[\tau'][\tau.1/\alpha_1, \tau.2/\alpha_2] & \text{if } \tau \equiv \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau' \\ \mathrm{unroll}^{\sigma}_{\Omega}(T[\tau]) & = & T[\sigma(\alpha)] & \text{if } \tau \equiv \alpha \end{array}$$

For example, consider:

$$\tau \quad = \quad \mu\langle \alpha_1 \leq \top, \alpha_2 \leq \alpha_1 \rangle.\langle \{\alpha_2\}, \{\alpha_2, \alpha_1\} \rangle$$

Because α_2 is declared to be a subtype of α_1, our definition should better verify $\{\alpha_2, \alpha_1\} \leq \{\alpha_2\}$ somehow. This internal subtyping is accounted, because the unroll function expands α_1 in root position and thus the last premise of rule K-Rec-Tup checks:[5]

$$\begin{array}{lcll} \langle \{\alpha_2\}, \{\alpha_2, \alpha_1\} \rangle & \leq & \mathrm{unroll}^{\sigma}_{\Omega\times\Omega}(\langle \top, \alpha_1 \rangle) & \text{where } \sigma = [\langle \{\alpha_2\}, \{\alpha_2, \alpha_1\} \rangle.1/\alpha_1] \\ & \equiv & \langle \top, \mathrm{unroll}^{\sigma}_{\Omega}(\alpha_1) \rangle & \\ & = & \langle \top, \langle \{\alpha_2\}, \{\alpha_2, \alpha_1\} \rangle.1 \rangle & \\ & \equiv & \langle \top, \{\alpha_2\} \rangle & \end{array}$$

The other crucial change to support dependent bounds occurs in the subtyping rule:

$$\frac{\tau = \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau'}{\Gamma \vdash \tau \leq \langle \tau_1, \tau_2[\tau.1/\alpha_1] \rangle} \text{ S-Rec-Tup}$$

By substituting the first projection of the whole type for α_1, the dependency is preserved as needed.

With this, we can define:

$$AB \quad = \quad \mu\langle \alpha_1 \leq \top, \alpha_2 \leq \alpha_1 \rangle.\langle \{\alpha_2\}, \{\alpha_2, \alpha_1\} \rangle$$

We leave it as an exercise to the reader to convince themselves that this type is well-formed according to rule K-Rec-Tup, and more interestingly, that the $AB.2 \leq AB.1$ holds with rule S-Rec-Tup.

There is but one wrinkle: while tuples of higher arity can generally be encoded in terms of nested binary tuples given subtyping and a top type (including both depth and width subtyping), such a decomposition is not possible for the μ-operator — pretty much for the same reason that the binary operator itself is not encodable. To support intra-subtyping within larger recursion groups, a generalised version of the operator for tuples of arbitrary arity hence has to be made primitive. But since the generalisation is obvious, we only spell out the binary version here to avoid clutter.

Bounds can only refer to earlier variables, and as a consequence, all subtyping in a recursion group has to be ordered linearly with respect to their declared supertype relation. That implicitly prevents *cyclic* subtyping. This restriction is present in most programming languages as well. Fortunately, if a subtype relation is indeed non-cyclic, a linear order can always be constructed by topologically sorting the types by their supertype dependencies.

4 THE λ_{misu} CALCULUS

In the remainder of this paper we formalise the ideas sketched in the previous sections. We proceed in two steps. First, in this section, we formalise them in an extension of System $\mathrm{F}^{\omega}_{\leq}$, the polymorphic λ-calculus with higher-order subtyping [Pierce and Steffen 1997; Compagnoni 1995; Pierce 2002]. We call it the *misu* calculus, for "mutually iso-recursive subtyping". This calculus realises the idea of declared subtyping in its most general form, in a minimal extension to a standard formulation of iso-recursive types. It therefore provides a type-theoretic justification for the more limited system that we derive in a second step, in the next section. The advantage of doing it this way instead of constructing the latter system directly is that it guarantees that the final system is coherent wrt. a general and canonical semantics, and has a clear extension path to cover more ground if needs be.

[5] Note that it would not be correct to simply substitute α_1 in the *result* of the unrolling, as per $\mathrm{unroll}_{\kappa}(\langle \tau_1, \tau_2 \rangle)[\tau.1/\alpha_1]$. Other occurrences of α_1 must not be affected, such that, e.g., $\mu\langle \alpha_1 \leq \top, \alpha_2 \leq \mu\_.\{\alpha_1 \to \alpha_1\} \rangle.\langle \{\}, \{\alpha_1 \to \alpha_1\} \rangle$ is well-formed. For the same reason, it would work neither to blindly substitute the type beforehand, like with $\mathrm{unroll}_{\kappa}(\langle \tau_1, \tau_2[\tau.1/\alpha_1] \rangle)$.

4.1 Syntax and Semantics

To layer the presentation a little bit, we start out smoothly by presenting the *basic* λ_{misu} calculus, which only features the simpler unary μ-type from Section 3.2. In the following subsection, we then show the more uncomely modifications to obtain the *full* calculus, which also supports intra-recursion subtyping, as developed in Section 3.3.

Figure 1 shows the complete definition of the basic λ_{misu} calculus. It consists of mostly standard ingredients [Pierce 2002], with the novel parts being highlighted in red. As usual for $F^{\omega}_{\leq}$, it features function and tuple types and a top type, as well as universal types with *kernel* subtyping rule [Cardelli and Wegner 1985]. In addition, it has both arrow and product *kinds*, with respective introduction and elimination forms on the type level. We limit the presentation to binary tuples on type and term level, although we take the liberty to assume other arities in some of our examples — the encoding of n-ary tuples $\langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n$ as nested pairs $\langle e_1, \langle \ldots, \langle e_n, \langle\rangle \rangle \rangle \rangle : \tau_1 \times (\cdots \times (\tau_n \times \top))$ is straightforward and provides both depth and width subtyping [Cardelli 1994].

The only interesting addition over the standard textbook calculus are recursive types along the lines developed in the previous Section, i.e., an iso-recursive μ-constructor with an upper bound, and respective roll and unroll operators for injection and projection into/out of such types. Judgement forms and rules are largely standard as well. We formalise call-by-value reduction using evaluation contexts. Typing, kinding, and subtyping operate relative to an environment Γ recording the type of term variables and the bounds of type variables. To handle type-level λ, whose parameter is only given a kind, we define higher-order top types $\top_\kappa$ as an abbreviation [Pierce 2002]. The only novelty is in the kinding and subtyping rules for μ, which resemble K-Rec-Sup and S-Rec-Sup from Section 3.2, and the typing rules for roll and unroll, which are equivalent to T-Roll-HO and T-Unroll-HO from Section 2.3, but reusing the unroll_κ meta-function for notational convenience.

4.2 The Full Calculus

With the basic calculus set up, Figure 2 shows the modifications for obtaining the *full* λ_{misu} calculus. Again, we highlight the differences in red, but this time relative to the formulation in Figure 1. The majority of rules is unmodified and hence omitted.

For the most part, the changes are fairly mechanical, adapting to the extended syntax of the μ-constructor, and folding in the tuple semantics. Other than that, the only substantial changes are the following, dealing with the possibility of the first type variable occurring in the second bound:

- In the kinding rule, the second bound is kinded with the first variable in scope.
- The unroll meta-function adds a case for the occurrence of this variable as a root, which just gets replaced by the first projection of the μ-type's body, accomplished by a substitution parameter σ (which we omit when it is empty).
- In the subtyping rule, any free occurrence of the first type variable in the second bound is substituted by the first projection from the complete type.

As mentioned, we only show the extended μ-operator for binary tuples. The generalisation to larger arities can be found in Appendix C.2 .

In the full calculus, the unary version of μ is encodable in a straightforward manner:

$$\mu\alpha{\leq}\tau_1.\tau_2 \quad = \quad (\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\top\rangle.\langle\tau_2, \top\rangle).1$$

It is easy to show that the original kinding, subtyping, and reduction rules are admissible.

Syntax

(kinds)	κ	::=	$\Omega \mid \kappa \times \kappa \mid \kappa \to \kappa$
(types)	τ	::=	$\alpha \mid \top \mid \tau \times \tau \mid \tau \to \tau \mid \forall \alpha{\leq}\tau.\tau \mid \mu\alpha{\leq}\tau.\tau \mid \langle \tau, \tau \rangle \mid \tau.i \mid \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau$
(values)	v	::=	$\langle v, v \rangle \mid \lambda x{:}\tau.e \mid \lambda\alpha{\leq}\tau.e \mid \mathsf{roll}_\tau\ v$
(terms)	e	::=	$x \mid v \mid \langle e, e \rangle \mid e.i \mid e\ e \mid e\ \tau \mid \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e$
(type contexts)	T	::=	$\_ \mid T.i \mid T\ \tau$
(eval contexts)	E	::=	$\_ \mid E.i \mid E\ e \mid v\ E \mid E\ \tau \mid \mathsf{roll}\ E \mid \mathsf{unroll}\ E$
(environments)	Γ	::=	$\cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{\leq}\tau$

Top ($\top_\kappa$)

$$\top_\Omega = \top \qquad \top_{\kappa_1 \times \kappa_2} = \langle \top_{\kappa_1}, \top_{\kappa_2} \rangle \qquad \top_{\kappa_1 \to \kappa_2} = \lambda\alpha{:}\kappa_1.\top_{\kappa_2}$$

Reduction ($e \hookrightarrow e'$)

$$\langle v_1, v_2 \rangle.i \hookrightarrow v_i \qquad (\lambda x{:}\tau.e_1)\ v_2 \hookrightarrow e_1[v_2/x] \qquad (\lambda\alpha{\leq}\tau_1.e)\ \tau_2 \hookrightarrow e[\tau_2/\alpha] \qquad \mathsf{unroll}\ (\mathsf{roll}_\tau\ v) \hookrightarrow v$$

$$\frac{e \hookrightarrow e'}{E[e] \hookrightarrow E[e']}$$

Unrolling ($\mathsf{unroll}_\kappa(\tau)$)

$$\mathsf{unroll}_\Omega(T[\tau]) = T[\tau_2][\tau/\alpha] \quad \text{if } \tau \equiv \mu\alpha{\leq}\tau_1.\tau_2$$
$$\mathsf{unroll}_\Omega(\tau) = \top \quad \text{if } \tau \equiv \top$$
$$\mathsf{unroll}_{\kappa_1 \times \kappa_2}(\tau) = \langle \mathsf{unroll}_{\kappa_1}(\tau.1), \mathsf{unroll}_{\kappa_2}(\tau.2) \rangle$$
$$\mathsf{unroll}_{\kappa_1 \to \kappa_2}(\tau) = \lambda\alpha{:}\kappa_1.\ \mathsf{unroll}_{\kappa_2}(\tau\ \alpha)$$

Context Formation ($\vdash \Gamma\ \mathsf{ok}$)

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \qquad \Gamma \vdash \tau : \Omega \qquad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x{:}\tau\ \mathsf{ok}} \qquad \frac{\vdash \Gamma\ \mathsf{ok} \qquad \Gamma \vdash \tau : \kappa \qquad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha{\leq}\tau\ \mathsf{ok}}$$

Type Formation ($\Gamma \vdash \tau : \kappa$)

$$\frac{\Gamma \vdash \Gamma(\alpha) : \kappa}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \tau_2 : \kappa \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \tau_2 \leq \mathsf{unroll}_\kappa(\tau_1)}{\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 : \kappa}$$

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\Gamma \vdash \top : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \times \tau_2 : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \tau_2 : \Omega}{\Gamma \vdash \forall\alpha{\leq}\tau_1.\tau_2 : \Omega}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \times \kappa_2} \qquad \frac{\Gamma \vdash \tau : \kappa_1 \times \kappa_2}{\Gamma \vdash \tau.i : \kappa_i} \qquad \frac{\Gamma, \alpha{\leq}\top_{\kappa_1} \vdash \tau_2 : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}$$

Term Formation ($\Gamma \vdash e : \tau$)

$$\frac{\vdash \Gamma\ \mathsf{ok}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.i : \tau_i}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma, \alpha{\leq}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\alpha{\leq}\tau_1.e : \forall\alpha{\leq}\tau_1.\tau_2} \qquad \frac{\Gamma \vdash e : \forall\alpha{\leq}\tau_1.\tau_2 \qquad \Gamma \vdash \tau \leq \tau_1 : \kappa}{\Gamma \vdash e\ \tau : \tau_2[\tau/\alpha]}$$

$$\frac{\tau = \mu\alpha{\leq}\tau_1.\tau_2 \qquad \Gamma \vdash e : \mathsf{unroll}_\Omega(T[\tau]) \qquad \Gamma \vdash T[\tau] : \Omega}{\Gamma \vdash \mathsf{roll}_{T[\tau]}\ e : T[\tau]} \qquad \frac{\tau = \mu\alpha{\leq}\tau_1.\tau_2 \qquad \Gamma \vdash e : T[\tau]}{\Gamma \vdash \mathsf{unroll}\ e : \mathsf{unroll}_\Omega(T[\tau])}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau' : \Omega \qquad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

Type Equivalence ($\tau \equiv \tau'$)

reflexive, transitive, symmetric, congruent closure of $\langle \tau_1, \tau_2 \rangle.i \equiv \tau_i \qquad (\lambda\alpha{:}\kappa.\tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]$

Subtyping ($\Gamma \vdash \tau \leq \tau'$) $\qquad (\Gamma \vdash \tau_1 \leq \tau_2 : \kappa \iff \Gamma \vdash \tau_1 : \kappa \wedge \Gamma \vdash \tau_2 : \kappa \wedge \Gamma \vdash \tau_1 \leq \tau_2)$

$$\frac{\tau \equiv \tau'}{\Gamma \vdash \tau \leq \tau'} \qquad \frac{\Gamma \vdash \tau \leq \tau' \qquad \Gamma \vdash \tau' : \kappa \qquad \Gamma \vdash \tau' \leq \tau''}{\Gamma \vdash \tau \leq \tau''} \qquad \frac{}{\Gamma \vdash \alpha \leq \Gamma(\alpha)} \qquad \frac{}{\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 \leq \tau_1}$$

$$\frac{\Gamma \vdash \tau : \Omega}{\Gamma \vdash \tau \leq \top} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_1' \qquad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \qquad \frac{\Gamma \vdash \tau_1' \leq \tau_1 \qquad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \qquad \frac{\Gamma, \alpha{\leq}\tau_1 \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1.\tau_2'}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_1' \qquad \Gamma \vdash \tau_2 \leq \tau_2'}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle \leq \langle \tau_1', \tau_2' \rangle} \qquad \frac{\Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \tau.i \leq \tau'.i} \qquad \frac{\Gamma, \alpha{\leq}\top_\kappa \vdash \tau \leq \tau'}{\Gamma \vdash \lambda\alpha{:}\kappa.\tau \leq \lambda\alpha{:}\kappa.\tau'} \qquad \frac{\Gamma \vdash \tau_1 \leq \tau_1'}{\Gamma \vdash \tau_1\ \tau_2 \leq \tau_1'\ \tau_2}$$

Fig. 1. The basic λ_{misu} calculus

Syntax

$$\tau \quad ::= \quad \ldots \mid \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau$$

Unrolling ($\text{unroll}_\kappa(\tau)$)

$$\begin{array}{lll}
\text{unroll}^\sigma_\Omega(T[\tau]) & = T[\tau'][\tau.1/\alpha_1, \tau.2/\alpha_2] & \text{if } \tau \equiv \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau' \\
\text{unroll}^\sigma_\Omega(T[\tau]) & = T[\sigma(\alpha)] & \text{if } \tau \equiv \alpha \\
\text{unroll}^\sigma_\Omega(\tau) & = \top & \text{if } \tau \equiv \top \\
\text{unroll}^\sigma_{\kappa_1 \times \kappa_2}(\tau) & = \langle \text{unroll}^\sigma_{\kappa_1}(\tau.1), \text{unroll}^\sigma_{\kappa_2}(\tau.2) \rangle & \\
\text{unroll}^\sigma_{\kappa_1 \to \kappa_2}(\tau) & = \lambda\alpha{:}\kappa_1.\, \text{unroll}^\sigma_{\kappa_2}(\tau\,\alpha) &
\end{array}$$

Type Formation ($\Gamma \vdash \tau : \kappa$)

$$\frac{\begin{array}{cc} \Gamma \vdash \tau_1 : \kappa_1 & \Gamma, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \vdash \tau : \kappa_1 \times \kappa_2 \\ \Gamma, \alpha \leq \tau_1 \vdash \tau_2 : \kappa_2 & \Gamma, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \vdash \tau \leq \text{unroll}^{[\tau.1/\alpha_1]}_{\kappa_1 \times \kappa_2}(\langle \tau_1, \tau_2 \rangle) \end{array}}{\Gamma \vdash \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau : \kappa_1 \times \kappa_2}$$

Term Formation ($\Gamma \vdash e : \tau$)

$$\frac{\tau = \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau' \qquad \Gamma \vdash e : \text{unroll}_\Omega(T[\tau]) \qquad \Gamma \vdash T[\tau] : \Omega}{\Gamma \vdash \text{roll}_{T[\tau]}\ e : T[\tau]} \qquad \frac{\tau = \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau' \qquad \Gamma \vdash e : T[\tau]}{\Gamma \vdash \text{unroll}\ e : \text{unroll}_\Omega(T[\tau])}$$

Subtyping ($\Gamma \vdash \tau \leq \tau'$)

$$\frac{\tau = \mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau'}{\Gamma \vdash \tau \leq \langle \tau_1, \tau_2[\tau.1/\alpha_1] \rangle}$$

Fig. 2. Modifications for the full λ_{misu} calculus

4.3 Example

To convince ourselves that this calculus scales to more complex examples, consider the following set of (admittedly artificial) classes, which mix all interesting features. To make it even more interesting, we allow the types of (immutable) fields to be overridden covariantly:

```
class A {                class B⟨X⟩ <: A {      class C⟨X⟩ <: B⟨X⟩ {     class D <: C⟨Float⟩ {
  x : Int                  z : X                  override v : C⟨X⟩        override v : D
  y : B⟨Int⟩ → A           v : A                  w : D                    override y : A → C⟨Int⟩
}                        }                      }                        }
```

In this synthetic example, A and B form a recursion group, as do C and D, both groups using *intra*-recursion subtyping, plus there is *inter*-recursion subtyping between C and B. The latter is kind-*homogeneous*, while subtyping is *heterogeneous* between B and A or D and C, introducing, resp. specialising a type parameter. The override of y in D is contra/co-variant in the function type.

The canonical, subtype-preserving λ_{misu}-encoding of the instance type hierarchy produced by these classes would be the following. We liberally assume the addition of higher-arity tuples and width subtyping on term-level tuples, which we write with record notation $\{\ldots\}$ instead of $\langle\ldots\rangle$ for better readability:

$$\begin{array}{l}
ab = \mu\langle \alpha_1 \leq \top, \alpha_2 \leq \lambda\chi{:}\Omega.\alpha_1 \rangle.\langle \\
\qquad \{int, \alpha_2(int) \to \alpha_1\}, \\
\qquad \lambda\chi{:}\Omega.\{int, (\alpha_2(int) \to \alpha_1), \chi, \alpha_1\} \rangle
\end{array}
\qquad
\begin{array}{l}
cd = \mu\langle \beta_1 \leq ab.2, \beta_2 \leq \beta_1(float) \rangle.\langle \\
\qquad \lambda\chi{:}\Omega.\{int, (ab.2(int) \to ab.1), \chi, \beta_1(\chi), \beta_2\}, \\
\qquad \{int, (ab.1 \to \beta_1(int)), float, \beta_2, \beta_2\} \rangle
\end{array}$$

With this, $ab.1$ and $ab.2$ represent instances of A and B, while $cd.1$ and $cd.2$ define C and D, respectively. Inside ab, the variables α_1 and α_2 name A and B recursively, similarly β_1 and β_2 in cd. In each case, the bound of the variable corresponds to the supertype declared for the original class, or $\top$ where none is given. The records in the recursive types consist of all instance fields for the respective class, including inherited ones. In the case of generic classes, their instance type has a respective parameter χ, both in its declared bound and its definition. As a minor point, we can η-reduce the bound of β_1 to just $ab.2$, but the expansion $\lambda\chi{:}\Omega.\, ab.2(\chi)$ would work as well.

Proving that these types are well-formed in the calculus is primarily a matter of proving that the unrolling of their bounds is a supertype of their bodies. For example, in the case of cd,

$$\mathrm{unroll}^{\sigma}_{(\Omega\to\Omega)\times\Omega}(\langle ab.2, \beta_1(float)\rangle) \quad = \quad \langle ab.2, \{int, (ab.2(int) \to ab.1), float, \beta_1(float), \beta_2\}\rangle$$

(with $\sigma = [\lambda\chi{:}\Omega.\{int, (ab.2(int) \to ab.1), \chi, \beta_1(\chi), \beta_2\}/\beta_1]$), is a supertype of cd's body, under the assumption $\beta_1 \le ab.2, \beta_2 \le \beta_1(float)$. We leave that as an exercise to the reader, as well as showing that all the subtyping relations declared in the source program are preserved by this encoding.

A simple constructor function for instances of class $C\langle X\rangle$ could be encoded as follows:

$$new\_c \quad = \quad \lambda\chi{\le}\top.\, \lambda x{:}int.\, \lambda y{:}(ab.2(int){\to}ab.1).\, \lambda z{:}\chi.\, \lambda v{:}cd.1(\chi).\, \lambda w{:}cd.2\,.\, \mathsf{roll}_{cd.1(\chi)}\{x, y, z, v, w\}$$

This function is polymorphic over type χ and takes parameters x, y, z, v, w corresponding to the fields in the class, including the inherited ones, with their respective types. It then creates a record of these fields and applies roll to form a value of the recursive type $cd.1$, also known as C.

Now imagine a generic function that performs a simple computation on an object of type $C\langle X\rangle$:

$$\mathsf{func}\ f\langle X\rangle(c : C\langle X\rangle) : Int\ \{\mathsf{return}\ c.x + c.v.x + c.w.y(c.w.v).z\}$$

This function is directly representable in our calculus (assuming the addition of a + operator):

$$f \quad = \quad \lambda\chi{\le}\top.\, \lambda c : cd.1(\chi)\,.\, c.1 + c.4.1 + c.5.2(c.5.4).3$$

Field accesses directly map to projections of the respective record/tuple component. And because subtyping is faithfully preserved in the encoding, no further magic is needed.

Finally, another function applying f to values that are subtypes of the declared parameters,

$$\mathsf{func}\ g(d : D) : Int\ \{\mathsf{return}\ f\langle Float\rangle(d.v, d)\}$$

can likewise be translated one-to-one, and since subtyping between D and $C\langle X\rangle$ is preserved in the encoding, the parameters can be used as is in the application to f:

$$g \quad = \quad \lambda d : cd.2\,.\, f(float)(d.4)(d)$$

So far, we have only considered instance fields of classes and conveniently skipped over methods. That is because our subject of interest is subtyping, whereas full-blown class encodings are quite a different challenge, for reasons that go beyond subtyping [Cook et al. 1989]. Bruce et al. [1999] give a nice overview of basic encodings. The λ_{misu}-calculus as defined here can handle their basic OR encoding, i.e., the classical objects-as-records-of-closures model, where methods are fields of function type like y in class A above. But that encoding has limitations, in particular, it cannot express inheritance with method overrides. Additional type-level constructs are required for that, specifically the ability to encode *self types* to type the implicit receiver parameter of a (pre-)method. This is a well-known problem, and various authors have investigated and addressed it. Abadi et al. [1996] first gave a solution using bounded existential quantification, which Bruce et al. also discuss, but which requires Full $F_{\le}$. We do not expect a problem with applying more complex encodings like Bruce et al.'s to a suitably enriched version of our calculus, since they already use iso-recursive types. A simpler, possibly more attractive alternative has been described by Glew [Glew 2000]. A comprehensive rehash of these complex techniques is outside the scope of our paper, though, since it should neither affect nor is affected by the specific problem we set out to solve.

4.4 Subtyping Algorithm

Our subtyping rules are declarative, and the possible occurrence of type-level redexes allows for non-trivial uses of transitivity. Hence, as usual with higher-order subtyping, it is not immediately apparent how to formulate an algorithm [Pierce and Steffen 1997; Compagnoni 1995], and moreover, whether the new rules for recursive types make that more difficult.

Basic System

$$
\begin{array}{lcl}
sub(\Gamma, \tau_1, \tau_2) & = & \tau_1 \equiv \tau_2 \vee sub'(\Gamma, norm(\tau_1), norm(\tau_2)) \\
sub'(\Gamma, \_, \top) & = & \mathsf{true} \\
sub'(\Gamma, \tau_{11} \times \tau_{12}, \tau_{21} \times \tau_{22}) & = & sub(\Gamma, \tau_{11}, \tau_{21}) \wedge sub(\Gamma, \tau_{12}, \tau_{22}) \\
sub'(\Gamma, \tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}) & = & sub(\Gamma, \tau_{21}, \tau_{11}) \wedge sub(\Gamma, \tau_{12}, \tau_{22}) \\
sub'(\Gamma, \forall\alpha{\leq}\tau_{11}.\tau_{12}, \forall\alpha{\leq}\tau_{21}.\tau_{22}) & = & \tau_{11} = \tau_{21} \wedge sub((\Gamma, \alpha{\leq}\tau_{11}), \tau_{12}, \tau_{22}) \\
sub'(\Gamma, \langle\tau_{11}, \tau_{12}\rangle, \langle\tau_{21}, \tau_{22}\rangle) & = & sub(\Gamma, \tau_{11}, \tau_{21}) \wedge sub(\Gamma, \tau_{12}, \tau_{22}) \\
sub'(\Gamma, \lambda\alpha{:}\kappa_1.\tau_1, \lambda\alpha{:}\kappa_2.\tau_2) & = & \kappa_1 = \kappa_2 \wedge sub((\Gamma, \alpha{\leq}\top_{\kappa_1}), \tau_1, \tau_2) \\
sub'(\Gamma, T[\alpha], \tau_2) & = & sub(\Gamma, T[\Gamma(\alpha)], \tau_2) \\
sub'(\Gamma, T[\mu\alpha{\leq}\tau_{11}.\tau_{12}], \tau_2) & = & sub(\Gamma, T[\tau_{11}], \tau_2) \\
sub'(\Gamma, \_, \_) & = & \mathsf{false}
\end{array}
$$

Full System (diff)

$$
\begin{array}{lcl}
sub'(\Gamma, T[\tau_1], \tau_2) & = & sub(\Gamma, T[\langle\tau_{11}, \tau_{12}[\tau_1.1/\alpha_1]\rangle], \tau_2) \\
 & & \text{iff } \tau_1 = \mu\langle\alpha_1{\leq}\tau_{11}, \alpha_2{\leq}\tau_{12}\rangle.\tau_1'
\end{array}
$$

Fig. 3. Subtyping Algorithm for λ_{misu}

Fortunately, it turns out that μ-types are an easy addition to the known algorithms. This is fairly intuitive after observing that subtyping on μ-types is no different from what can already be expressed with higher-kinded type variables. Concretely, with respect to subtyping and equivalence, a type $\mu\alpha{\leq}\tau_1.\tau_2$ of kind κ behaves like the higher-order type application

$$\mu_\kappa\ (\tau_1)\ (\lambda\alpha{:}\kappa.\tau_2)$$

provided μ_κ is an abstract type variable recorded in the context Γ with bound

$$\mu_\kappa \leq \lambda u{:}\kappa.\ \lambda f{:}(\kappa \to \kappa).\ u$$

In particular, the higher-order subtyping rules of $F^{\omega}_{\leq}$ are sufficient to derive

$$\mu_\kappa\ (\tau_1)\ (\lambda\alpha{:}\kappa.\tau_2) \quad \leq \quad (\Gamma(\mu_\kappa))\ (\tau_1)\ (\lambda\alpha{:}\kappa.\tau_2) \quad \equiv \quad \tau_1$$

as demanded by our rule for μ-types.

Consequently, we can simply implement the case for μ by following the case for type variables. That is, going from a recursive type to its bound is a case of *promotion* [Pierce and Steffen 1997].

With that in mind, Figure 3 shows an appropriate algorithm. Modulo minor case reordering, it is essentially the one given by Pierce & Steffen [1997], except for the addition of type tuples and μ-types. If the types are not equivalent already, the algorithm first reduces them to their normal forms.[6] It then proceeds by case analysis. Most cases follow directly from the declarative rules.

Type tuples and type-level projections are a fairly straightforward addition. Tuples must simply match pointwise. A projection that is not a redex can only be a subtype of another if both are either equivalent (already covered), or if they are part of an elimination context around a variable or μ-type that can be promoted to create a redex that allows eliminating it. Consequently, where Steffen & Pierce's algorithm only applies promotion under a sequence of applications, the extended algorithm generalises that to type elimination contexts T, which may mix applications with projections.

In the case of a variable on the left, it is promoted to its upper bound as recorded in the context. The variable may occur inside a type elimination context, in which case the promotion may enable

[6] As noted by Pierce & Steffen [1997], weak-head normalisation would suffice here, and is more efficient in practice.

further reduction. A variable on the right can only be matched by an equivalent type, so is already covered by the initial equivalence check.

As observed above, μ-types can be handled analogously, by promotion. But instead of looking up the promoted type from the context, it is simply read off the type itself.

This algorithm (for both variants of the calculus) terminates and is sound and complete:

Theorem 4.1 (Algorithmic Soundness). *If* $sub(\Gamma, \tau_1, \tau_2) = \mathsf{true}$*, then* $\Gamma \vdash \tau_1 \leq \tau_2$.

Theorem 4.2 (Algorithmic Completeness). *If* $\Gamma \vdash \tau_1 \leq \tau_2$*, then* $sub(\Gamma, \tau_1, \tau_2) = \mathsf{true}$.

Theorem 4.3 (Algorithmic Termination). *If* $\Gamma \vdash \tau_1 : \kappa$ *and* $\Gamma \vdash \tau_2 : \kappa$*, then* $sub(\Gamma, \tau_1, \tau_2) = b$.

Proofs are given in Appendix A.6 to A.9 and involve the usual tedium of proving transitivity elimination for the subtyping relation.

Note that the subtyping algorithm never needs to look at the body τ_{12} of a recursive type, i.e., it's actual definition. It merely inspects the bound τ_{11}. That is the central feature of our semantics. Only the equivalence relation needs to consider τ_{12} — unless that can also be avoided through the use of canonicalisation, as we will explore further in Section 5. Hence, a subtyping check $\tau_1 \leq \tau_2$ between two recursive types simply consists of climbing the supertype ladder of τ_1 until τ_2 is found. It fails when the top type is reached before that event. In other words, up to equivalence, subtyping works like with *nominal* subtyping (a.k.a. inheritance), and the cost of a subtyping check is at worst linear in the depth of the *declared* supertype hierarchy of the left-hand type (assuming equivalence is made constant-time via canonicalisation). That a recursive type is in fact a correct subtype of its supertype is only checked "once", when the type's wellformedness is verified.

4.5 Metatheory

The λ_{misu} calculus enjoys the standard syntactic type soundness properties:

Theorem 4.4 (Preservation). *If* $\cdot \vdash e : \tau$ *and* $e \hookrightarrow e'$*, then* $\cdot \vdash e' : \tau$.

Theorem 4.5 (Progress). *If* $\cdot \vdash e : \tau$ *and* e *is not a value* v*, then* $e \hookrightarrow e'$ *for some* e'.

The proofs of these properties are given in Appendix A.10 to A.12 . Due to the presence of higher-order types, they involve the usual diversion through an algorithmic subtype relation, in order to be able prove appropriate inversion lemmas. This essentially is the algorithm already given in Figure 3.

As a side effect, the correctness proof for this algorithm also establishes termination, and thereby decidability of the subtype relation, and in turn decidability of type checking in general:

Theorem 4.6 (Decidability). *All relations defined in Figures 1 and 2 are decidable.*

5 A FRAGMENT WITH SHALLOW SUBTYPING

Now that we have a general and well-behaved theory of pre-declared subtyping between iso-recursive types, we can investigate its algorithmic implications a bit more. As an immediate observation, the λ_{misu} calculus is completely liberal in its use of higher-order types. That comes at a price, in terms of both semantic and algorithmic complexity. Can we isolate a fragment that avoids this cost but is still sufficiently expressive for many practical purposes?

5.1 Analysis

Let us take a closer look at the subtyping algorithm from Figure 3 again. Assuming Γ, τ_1, and τ_2 are already normalised, most cases just traverse the structure of these types. The only critical cases are those for *neutral* types. In conventional systems of higher-order subtyping, neutral types are those

of the form $T[\alpha]$, which cannot be reduced. In λ_{misu}, neutral types include $T[\mu\ldots]$, i.e., the *root* inside the elimination context T can either be a type variable or a μ-type.

As discussed in Section 4.4, the root of a neutral type is promoted to its bound by the algorithm, i.e., its least supertype. If that supertype is either a type tuple or a type function, this will generally produce a new redex at the innermost level of T. For example, $T[\alpha][\langle\tau_1, \tau_2\rangle/\alpha]$ with $T = [\_].2$ produces $\langle\tau_1, \tau_2\rangle.2$.

Redexes for tuple projection are not a big deal, since they only create a level of indirection that could be handled directly by the algorithm without actually rewriting the type. However, if the new redex is an application, then β-reduction implies substitution. Consider these type functions:

$$\tau_1 \;=\; \lambda\alpha{:}\Omega.\ \mu\beta{\leq}\tau_0.\ \{\ldots\} \qquad \tau_2 \;=\; \lambda\alpha{:}\Omega.\ \mu\beta{\leq}\tau_1(\alpha\times\alpha).\ \{\ldots\} \qquad \tau_3 \;=\; \mu\beta{\leq}\tau_2.\ \ldots$$

Let's say type-checking a program involves checking $\tau_3(\mathit{nat}) \leq \tau_1(\mathit{nat}\times\mathit{nat})$. Assuming that both types are already normalised (e.g., because we normalise all types occurring in the source upfront), we enter sub' and reach the case for μ with $T = [\_]\ \mathit{nat}$. That produces the term $T[\tau_2] = \tau_2(\mathit{nat})$ on the right-hand side. With that we recurse. But $\tau_2(\mathit{nat})$ is a new redex that didn't exist in the program. So, the algorithm needs to normalise it *now*, which involves β-reduction and substituting *nat* for α in τ_2's entire, potentially large, body. Worse, it has to transitively substitute α in the bound $\tau_1(\alpha\times\alpha)$ — technically, it goes up the entire supertype chain, and in the worst case recomputes all their bodies. In other words, although we avoid *textual* "code duplication" (in the sense of Section 2.3), we still have a lot of *algorithmic* redundancy.

We cannot prevent substitution altogether — except by removing type functions. But it would be a jolly big improvement of we could keep substitutions *shallow*, i.e., avoid traversals deep into complex type definitions. What if we could syntactically restrict the use of types such that every complex type — for some definition of "complex" — that occurs during type checking of terms is *closed* by construction? Then substitution can just ignore and skip over them.

Wait, you'll think, wouldn't that be useless? The whole point of having type functions is to be able to parameterise complex types.

Well, it turns out that higher-order iso-recursive types give us a cute way out: we can parameterise them on the inside, without applications becoming redexes — because $(\mu\beta{\leq}\tau.\lambda\alpha{:}\kappa.\tau_1)(\tau_2)$ is already in normal form. That is, we can abuse recursive types to "guard" against β-reduction, without losing the ability to add parameters. As a simple example, consider this polymorphic function:

$$\mathit{fst} \quad = \quad \lambda\alpha \leq \top.\, \lambda x : \{\alpha, \alpha, \alpha\}.\ x.1$$

Let's say we consider records, a.k.a. structs, "complex" types — like many languages do by requiring a definition for them — and want to avoid having to substitute into $\{\ldots\}$, ever. If we defined

$$\mathit{triple} \quad = \quad \mu\_ \leq \top_{\Omega\to\Omega}.\ \lambda\alpha : \Omega.\ \{\alpha, \alpha, \alpha\}$$

and then, instead of the above, wrote the following term of type $\forall\alpha \leq \top.\ \mathit{triple}(\alpha) \to \alpha$:

$$\mathit{fst}' \quad = \quad \lambda\alpha \leq \top.\, \lambda x : \mathit{triple}(\alpha).\ (\mathsf{unroll}\ x).1$$

then instantiating that generic function, say, via $\mathit{fst}'\ \mathit{int}$, would produce the perfectly accurate function type $\mathit{triple}(\mathit{int}) \to \mathit{int}$. Yet the substitution doesn't need to touch *triple* at all, because its definition is closed and moreover, the application $\mathit{triple}(\mathit{int})$ is irreducible.[7]

[7] As usual, abstracting into a type constructor prevents covariant subtyping on α here, but that can be recovered by adding variance annotations to the language, which are an orthogonal extension [Steffen 1997], see Section 7.

5.2 A Syntactic Restriction

We can take the scheme just described to its logical conclusion where all complex types are "locked away" inside a μ-type, and moreover all μ-types are essentially *closed*. Then substitutions can ignore complex types entirely.

Let's make this more systematic. Consider the following stratified type grammar for λ_{misu}:

$$\begin{array}{lrcl}
\text{(value types)} & \tau_v & ::= & b \mid \top \mid \alpha(\overline{\tau_v}) \mid \alpha.i(\overline{\tau_v}) \\
\text{(composite types)} & \tau_c & ::= & \tau_v \mid \tau_c \times \tau_c \mid \tau_c \to \tau_c \mid \forall \alpha{\leq}\tau_v.\tau_c \\
\text{(defined types)} & \tau_d & ::= & \lambda(\overline{\alpha{:}\kappa}).\tau_v \mid \mu\overline{\langle \alpha{\leq}\lambda\overline{\alpha{:}\kappa}.\tau_v \rangle}.\langle \overline{\lambda\overline{\alpha{:}\kappa}.\tau_c} \rangle \\
\\
\text{(expressions)} & e & ::= & \lambda x{:}\tau_v.e \mid \lambda\alpha{\leq}\tau_v.e \mid \mathsf{roll}_{\tau_v}\ e \mid \ldots \\
\text{(programs)} & p & ::= & \mathsf{type}\ \alpha{=}\tau_d\ \mathsf{in}\ p \mid e
\end{array}$$

This grammar implements several crucial restrictions:

(1) Types occurring in terms e can only take the simple form of *value types* τ_v, which are either primitive, like $\top$ or a base type b (thrown in here for illustration), or neutral ones of the form $T[\alpha]$ with a T also consisting of only value types.
(2) In contrast, *composite types* τ_c can only occur inside recursive types. Likewise type tuples.
(3) Recursive types, in turn, have to be *named* by a *type definition*, which can only occur in *global* scope. Type definitions may also introduce (parameterised) aliases for value types.

Here, we are making the naming mechanism for types that we alluded to in Section 2.3 explicit. Semantically, named types are still just equivalent to their expansion. However, we impose that each defined type τ_d has to be well-formed *before* it is substituted, i.e., it has to be closed, up to preceding definitions that have already been checked and substituted into it. In other words, we enforce the following typing rule for programs p (note the empty context):

$$\frac{\cdot \vdash \tau_d : \kappa \qquad \cdot \vdash p'[\tau_d/\alpha] : \tau}{\cdot \vdash \mathsf{type}\ \alpha{=}\tau_d\ \mathsf{in}\ p' : \tau}$$

By restricting the surface syntax like this, all type applications initially form neutral types, all type arguments are primitive or neutral types, and so are all type bounds. More importantly, composite types are always guarded by a μ-binder, with no outer λ-bound type variables in scope.

For the most part, these restrictions are preserved under substitution, except that substitution may make lambdas and (closed!) μ-types appear in value types. Consequently, after expansion of type definitions, the structure of value types is slightly richer:

$$\text{(extended value types)} \quad \tau_{v'} \quad ::= \quad b \mid \top \mid \alpha(\overline{\tau_{v'}}) \mid \tau_d.i(\overline{\tau_{v'}}) \mid \lambda(\overline{\alpha{:}\kappa}).\tau_{v'}$$

Yet, altogether, this still ensures that no β-reduction ever needs to perform substitution inside a composite or recursive type, because they are all closed and remain so. The only types that substitutions have to traverse are type applications, abstractions, and projections. We call such substitutions *shallow*, write them $\tau_{v'}\lceil\tau'_{v'}/\alpha\rceil$, and we can define them as follows:

$$\begin{array}{rcll}
\alpha\lceil\tau/\alpha\rceil & = & \tau & \\
(\tau'.i)\lceil\tau/\alpha\rceil & = & (\tau'\lceil\tau/\alpha\rceil).i & \\
(\tau_1\ \tau_2)\lceil\tau/\alpha\rceil & = & (\tau_1\lceil\tau/\alpha\rceil\ \tau_2\lceil\tau/\alpha\rceil)^{\Downarrow} & \\
(\lambda\alpha'{:}\kappa.\tau')\lceil\tau/\alpha\rceil & = & \lambda\alpha'{:}\kappa.\ \tau'\lceil\tau/\alpha\rceil & \\
\tau'\lceil\tau/\alpha\rceil & = & \tau' & \text{otherwise}
\end{array}$$

The case for applications, where such a substitution may still introduce a redex, is defined in the style of a *hereditary substitution* that performs reduction on the fly and thereby preserves normal

forms [Watkins et al. 2004; Abel 2009]. This *shallow reduction* $\tau^{\Downarrow}$ is defined as follows:[8]

$$\begin{array}{rcll} ((\lambda\alpha{:}\kappa.\tau_1)\ \tau_2)^{\Downarrow} & = & \tau_1\lceil\tau_2/\alpha\rceil & \\ \tau^{\Downarrow} & = & \tau & \text{otherwise} \end{array}$$

That leads to the following, efficient subtype algorithm on extended value types:

$$\begin{array}{lcll} sub_v(\Gamma, \tau_1, \tau_2) & = & \tau_1 \equiv \tau_2 \vee sub'_v(\Gamma, \tau_1, \tau_2) & \\ sub'_v(\Gamma, \_, \top) & = & \text{true} & \\ sub'_v(\Gamma, \alpha\ \overline{\tau}, \tau_2) & = & sub_v(\Gamma, (\tau_1\ \overline{\tau})^{\Downarrow}, \tau_2) & \text{where } \tau_1 = \Gamma(\alpha) \\ sub'_v(\Gamma, \tau'.i\ \overline{\tau}, \tau_2) & = & sub_v(\Gamma, (\tau_1\ \overline{\tau})^{\Downarrow}, \tau_2) & \text{where } \tau_1 = bound(\tau.i) \\ sub'_v(\Gamma, \lambda\alpha{:}\kappa.\tau_1, \lambda\alpha{:}\kappa.\tau_2) & = & sub_v((\Gamma, \alpha{\le}\top_\kappa), \tau_1, \tau_2) & \\ sub'_v(\Gamma, \_, \_) & = & \text{false} & \text{otherwise} \\ bound(\tau.1) & = & \tau_1 & \text{where } \tau = \mu\langle\alpha_1{\le}\tau_1, \alpha_2{\le}\tau_2\rangle.\tau' \\ bound(\tau.2) & = & \tau_2\lceil\tau.1/\alpha_1\rceil & \end{array}$$

The auxiliary function *bound* produces the direct supertype of a recursive type according to rule S-Rec-Tup. This function can be pre-computed, such that the substitution it performs for $\tau.2$ (and others in the case of the n-ary generalisation) can be dealt with once at definition time of the type.

We still have to check *type equivalence* first in sub_v. But by the same argument, it suffices to perform a shallow traversal for this, provided that all recursive types are canonicalised initially, such that comparing μ's is a constant-time check. The upshot then is that neither of these type comparison algorithms ever needs to touch composite types or the bodies of recursive types.

As for checking type definitions themselves, that *will* of course have to perform a subtype check on the composite types involved. But every composite type τ_c occurring in a program is only ever involved in a subtype check on two occasions: (1) once to verify its originating definition (with τ_c on the left), (2) each time a subtype of it is declared (with τ_c on the right). Furthermore, since this check again merely involves sub_v at the leaves, it is typically linear in the size of τ_c.

All this seems very convenient! But what do we lose? Is it a reasonable restriction to work with?

Yes. Of course, this is exactly what languages with nominal subtyping (and generics) have always been happy with. In such type systems, all types are either primitive, or nominal types, possibly applied to a list of type arguments of similar shape, like our value types. In checking the subtype relation, the languages' type checkers have to expand (read: substitute) type parameters, but they never need to substitute into type *definitions*, such as class bodies.[9]

In other words, what we have arrived at can be viewed as a type-theoretic reconstruction of the subtyping semantics in nominal type systems — except that all of our types are still structural in the sense that equivalent recursive types are identified. Which disentangles subtyping from nominal semantics and lets us dub this semantics *"nominal typing up to canonicalisation"*.

Finally, note that there is some freedom regarding the role of different composite type constructors in the definition of the grammar above. For example, even nominal languages often treat (non-recursive) function types structural. That can be reflected easily by moving arrow types from τ_c to τ_v, for the price of somewhat less shallow substitutions.

[8] We don't need to worry about reducing projection $\langle\tau_1, \tau_2\rangle.i$, because the stratified syntax ruled out bare tuples.

[9] However, unlike most of those languages, our system also allows for higher-order "generics", which is why we also have "shallow lambdas" arise. Of course, a restriction to first-order type constructors would get rid of those.

$$\begin{array}{llll} \text{(types)} & T ::= \mathtt{X} \mid N & \text{(classes)} & C ::= \text{class } \mathtt{C}\langle \overline{\mathtt{X}<:N}\rangle <: N\{\overline{T\ \mathtt{x}};\ \ldots\} \\ \text{(nominal types)} & N ::= \mathtt{C}\langle \overline{T}\rangle & \text{(programs)} & P ::= \overline{C} \end{array}$$

Types ($[\![T]\!] \in \tau_v$)	$[\![\mathtt{X}]\!]$	$= \alpha_{\mathtt{X}}$	
	$[\![\mathtt{Object}]\!]$	$= \top$	
	$[\![\mathtt{C}\langle \overline{T}\rangle]\!]$	$= \alpha_{\mathtt{C}}\ \overline{[\![T]\!]}$	
Classes ($[\![C]\!] \in \lambda\overline{\alpha{:}\kappa}.\tau_v$)	$[\![\text{class } \mathtt{C}\langle \overline{\mathtt{X}<:N}\rangle <: N'\{\overline{T\ \mathtt{x}}\}]\!]$	$= \lambda\overline{\alpha_{\mathtt{X}}{:}\Omega}.[\![N']\!]$	
Type Representation ($[\![N]\!]^*_P \in \tau_c$)	$[\![\mathtt{Object}]\!]^*_P$	$= \{\}$	
	$[\![\mathtt{C}\langle \overline{T}\rangle]\!]^*_P$	$= \tau[\overline{[\![T]\!]/\alpha}]$	iff $[\![P(\mathtt{C})]\!]^*_P = \lambda\overline{\alpha{:}\kappa}.\tau$
Class Representation ($[\![C]\!]^*_P \in \lambda\overline{\alpha{:}\kappa}.\tau_c$)	$[\![\text{class } \mathtt{C}\langle \overline{\mathtt{X}<:N}\rangle <: N'\{\overline{T\ \mathtt{x}}\}]\!]^*_P$	$= \lambda\overline{\alpha_{\mathtt{X}}{:}\Omega}.\{\overline{\tau}, \overline{[\![T]\!]}\}$	iff $[\![N']\!]^*_P = \{\overline{\tau}\}$
Programs ($[\![P]\!] \in \tau_d$)	$[\![P]\!]$	$= \mu\langle \overline{\alpha_{\mathtt{C}} \le [\![C]\!]}\rangle.\langle \overline{[\![C]\!]^*_P}\rangle$	iff $P = \overline{C}$

Fig. 4. FGJ syntax and translation

5.3 Encoding Featherweight Generic Java

In order to have evidence that the λ_{misu}-calculus and the fragment we just defined really is expressive enough to encode the subtyping exhibited by typical languages, we give an encoding of the instance types of Featherweight Generic Java (FGJ) [Igarashi et al. 2001] as a representative candidate.

Figure 4 shows the syntax of FGJ class definitions and rules for translating their instance types into corresponding types from the shallow λ_{misu} fragment. We consider a variant of FGJ without F-bounded quantification, since we omitted F-bounded quantification from the presentation in this paper. Furthermore, we make the simplifying assumption that the classes in the program have been ordered with respect to their subtyping hierarchy, i.e., classes only inherit from classes earlier in the program — inheritance must be non-cyclic in Java, so programs can always be rearranged that way. The details of the FGJ type system can be found in Appendix C.1 .

For the reasons discussed at the end of Section 4.3, our translation only is for the type hierarchy of a FGJ program, and does not consider method bodies. While it is possible to express regular methods in plain λ_{misu}, overriding cannot be translated without further additions to the calculus.

FGJ types T are translated to *value* types $[\![T]\!]$. The translation of classes C is split into translating their bound $[\![C]\!]$ into (functions over) value types, and translating their representation $[\![C]\!]^*$ into (functions over) *composite* types, with the auxiliary translation of type representations $[\![N]\!]^*$ to handle superclasses. We write n-ary tuples using record syntax and rely on depth/width subtyping of their encoding (cf. Section 4.1). Because all classes can be mutually recursive, a program is translated into a single *defined* μ-type (assuming the n-ary generalisation of μ-types here). This matches the fact that Java has a single global namespace for classes.

A key lemma is that this translation preserves subtyping, as it should:

Lemma 5.1 (FGJ Subtype Translation). *Let* $\Delta \vdash_{\text{FGJ}} \text{ok}$.

(1) If $\Delta \vdash_{\text{FGJ}} T_1 \le T_2$ *and* $\Delta \vdash_{\text{FGJ}} T_1\ \text{ok}$, *then* $[\![\Delta]\!] \vdash_{misu} [\![T_1]\!] \le [\![T_2]\!]$.
(2) If $\Delta \vdash_{\text{FGJ}} N_1 \le N_2$ *and* $\Delta \vdash_{\text{FGJ}} N_1\ \text{ok}$, *then* $[\![\Delta]\!] \vdash_{misu} [\![N_1]\!]^*_P \le [\![N_2]\!]^*_P$.
(3) If $\Delta \vdash_{\text{FGJ}} C\ \text{ok}$, *then* $[\![\Delta]\!] \vdash_{misu} [\![C]\!]^*_P \le [\![C]\!]$.

From that and other auxiliary lemmas we can prove correctness of the translation:

Theorem 5.2 (FGJ Translation Correctness). *If* $\vdash_{\text{FGJ}} P\ \text{ok}$, *then* $\cdot \vdash_{misu} [\![P]\!] : \kappa$.

5.4 Wasm

An iso-recursive semantics with a type structure like shown in Section 5.2 has been adopted for a proposed extension of Wasm [Haas et al. 2017] with recursive types [Rossberg 2022].[10] Wasm's

[10]Though in the first iteration of the proposal, type functions are not yet included, which simplifies the semantics further.

module system provides strong modularity with type-safe separate compilation and linking and no ambient name space, which requires a structural notion of types that can be duplicated, matched up, and checked for equivalence at link time. Iso-recursive types fit the bill.

Currently, "composite" types in that proposal encompasses functions, structures, and arrays. All these types are always interpreted as iso-recursive. However, Wasm does not require explicit roll/unroll operations; rather, as is standard practice, these are implicit in the typing rules of the respective introduction and elimination forms (i.e., instructions).

Wasm's semantics depends on subtype checks at various stages:

(1) during *compilation*, to validate the input module;
(2) during *linking*, to ensure that provided import values match declared import types;
(3) during *execution*, for checking the types of values against cast instructions.

Since even compilation and linking of a Wasm module can be dynamic — e.g., an app may load additional modules in reaction to user activity — it is vital that subtype checks are efficient in all phases, because many such checks are often involved. Production Wasm engines are expected to employ runtime optimisations like cross-module type canonicalisation. Dynamic casts are implemented using standard techniques for casting along linear supertype hierarchies in constant time [Ducournau 2011], by assigning type identities to distinct canonical types.

The presented type system has been implemented in Wasm engines like V8 and SpiderMonkey. Multiple compilers have already been developed that target it, including ones for Java and Dart, as well as ongoing work compiling OCaml, Scheme, Kotlin, and others.

As a low-level VM, compilation to Wasm requires laying out runtime data structures in the same manner as a native code compilation would. For example, consider the following classes:

$$
\begin{array}{ll}
\mathsf{class}\ C\ \{ & \mathsf{class}\ D <: C\ \{ \\
\quad a : \mathit{Int} & \quad b : \mathit{Int} \\
\quad f(x : \mathit{Int}) : \mathit{Int}\ \{\mathsf{return}\ x + a\} & \quad \mathsf{override}\ f(x : \mathit{Int}) : \mathit{Int}\ \{\mathsf{return}\ x + a + b\} \\
\} & \quad g(x : \mathit{Int}) : \mathit{Bool}\ \{\mathsf{return}\ x < f(a)\} \\
 & \}
\end{array}
$$

A compiler targeting Wasm could produce something akin to the following types and functions:

$$
\begin{array}{rcl}
c & = & \mu\langle\alpha{\leq}\top, \beta{\leq}\top\rangle.\langle\{\mathsf{vtable} : \beta, \mathsf{a} : \mathit{int}\}, \{\mathsf{f} : \alpha \to \mathit{int} \to \mathit{int}\}\rangle \\
d & = & \mu\langle\alpha{\leq}c.1, \beta{\leq}c.2\rangle.\langle\{\mathsf{vtable} : \beta, \mathsf{a} : \mathit{int}, \mathsf{b} : \mathit{int}\}, \{\mathsf{f} : \alpha \to \mathit{int} \to \mathit{int}, \mathsf{g} : \alpha \to \mathit{int} \to \mathit{bool}\}\rangle \\
 & & \\
c\_f & = & \lambda \mathit{this} : c.1.\, \lambda x : \mathit{int}.\ \mathsf{let}\ \mathit{this}' = \mathsf{unroll}\ \mathit{this}\ \mathsf{in}\ x + \mathit{this}'.\mathsf{a} \\
d\_f & = & \lambda \mathit{this} : c.1.\, \lambda x : \mathit{int}.\ \mathsf{let}\ \mathit{this}' = \mathsf{unroll}\ (\mathsf{cast}\ \mathit{this} <: d.1)\ \mathsf{in}\ x + \mathit{this}'.\mathsf{a} + \mathit{this}'.\mathsf{b} \\
d\_g & = & \lambda \mathit{this} : d.1.\, \lambda x : \mathit{int}.\ \mathsf{let}\ \mathit{this}' = \mathsf{unroll}\ \mathit{this}\ \mathsf{in}\ x < \mathit{this}'.\mathsf{vtable.f}(\mathit{this}'.\mathsf{a})
\end{array}
$$

This is a standard implementation of objects with "vtables" of pre-method pointers. Instance type and vtable type of each class are mutually recursive. However, Wasm does not currently support self types (cf. Section 4.3), so that the compilation of an overriding function like d_f has to insert a (checked) downcast as an escape hatch for covariantly specialising the receiver argument. Notably, this is the only place where such a cast is needed outside the compilation of generics or interfaces.

6 RELATED WORK

The λ_{misu} calculus integrates iso-recursive types with higher-order subtyping. We are not aware of previous work that investigates a similar combination, but both have been investigated separately.

Iso-recursive types. Iso-recursive types are routinely viewed as a type-theoretic model of type recursion as it commonly appears in programming languages. However, there is surprisingly little literature on iso-recursive types, their expressiveness, and their practical implementation.

The most important result probably is by Abadi & Fiore [1996], who proved that iso-recursive types are as expressive as equi-recursive ones, for the price of a program transformation that synthesises coercion terms where the equi-recursive system would merely invoke the type conversion rule. But these coercions typically have to copy the entire value, so are very costly. They do not consider subtyping, but we conjecture that their argument would extend to a system with subtyping, because subsumption can as well be replaced by a narrowing coercion operator.

Zhou et al. [2022] propose an alternative formulation of Amber-style subtyping that allows a more efficient check, but it remains unclear how well it scales to richer type systems. The case they optimise does not arise in our system with declared subtyping, and their optimisation does not help to avoid the problematic size blow-up in encodings of mutually recursive types.

Most work on the semantics of subtyping for recursive types focusses on the equi-recursive model [Amadio and Cardelli 1993; Brandt and Henglein 1998; Gapeyev et al. 2002]. The first introduction of iso-recursive subtyping is probably due to Cardelli [1986] and his experimental language *Amber*; which prompted the "Amber rule". Ligatti et al. [2017] suggest a more expressive subtyping relation; however, in doing so, they essentially fall back to coinduction, thus giving up on the primary advantage of iso-recursion. A similar relaxation of iso-recursion towards coinduction has been brought up earlier by Shao, as reported by Crary et al. [1999] (who also coined the term "iso-recursive"), though they were not interested in subtyping but rather motivated by type abstraction and modularity concerns. Our work goes in the opposite direction: give up some expressiveness (and convenience) for the sake of enabling a practical implementation in a runtime.

The idea of employing higher-kinded iso-recursive types to express irregular data types and mutual recursion seems to be folklore knowledge. The first (and only) published use that we know of was in the type-theoretic formalisation of Standard ML by Harper & Stone [2000]. Interestingly, although not entirely trivial, this generalisation apparently seemed obvious enough to Harper & Stone that they neither discussed it nor provided a reference.

Higher-order Subtyping. Pierce & Steffen [1997], and in parallel, Compagnoni [1995], were the first to develop a meta-theory for higher-order subtyping. Our proofs mostly follow the structure of Compagnoni's development (minus intersection types, which her work includes), with some elements taken from Pierce & Steffen. Steffen [1997] afterwards extended this line of work with constructor polarities (variance annotations). Abel & Rodriguez [2008] gave a somewhat simpler account of the meta-theory that proves strong normalisation with the help of hereditary substitutions, which is more amendable to mechanisation. None of these works considers recursive types.

F-bounded Quantification. Our tuple-bounded μ-types as in full λ_{misu} have an internal dependency that has some resemblance to F-bounded quantification. Originally suggested by Canning et al. [1989], F-bounded quantification is available in many object-oriented languages. A meta-theory for F-bounds has later been developed by Ghelli and others [Ghelli 1997; Baldan et al. 1999], but their system is very general and undecidable. Glew [2012] gives an efficient subtype algorithm for a decidable version of that system, more similar to what is typically employed in practical languages.

F-bounded quantification for μ-types would be a natural extension to our system (Section 7). But we are unaware of any investigation of combining F-bounded quantification with higher-order subtyping or with iso-recursive types.

Nominal Subtyping. The shallow fragment of λ_{misu} introduced in Section 5 has close similarities with nominal subtyping. In the literature, nominal subtyping is typically modelled in a more monolithic fashion. The most dominant example of this probably is the formalisation of Featherweight Java [Igarashi et al. 2001]. In that system, the class hierarchy, and therefore the subtype relation, exists globally in an ambient program context that is invoked by the typing rules and that encompasses the entire program. Our μ-types can be viewed as a minimal, type-theoretic reconstruction

of the subtyping semantics in such a system, but without actually depending on nominality itself (which could hence be introduced orthogonally).

Declared Subtyping. In an unpublished manuscript, Hosoya et al. [1998] have a different take on user-declared subtyping. In their system, the subtype declaration is not part of the type definition, but an independent, term-level assertion. That provides some extra flexibility, although it is unclear to us what that would be used for in practice. The downside is a more complex operational semantics, because subtype declarations need to be maintained and scope-extruded during reduction, in order to maintain Type Preservation. Likewise, all typing rules need to be indexed with an extra set of subtype assumptions, in addition to the subtype bounds appearing in the regular context. In contrast, the approach taken with λ_{misu} is unintrusive and structurally minimalist, both static and dynamic semantics keep their standard form. Another difference is that their system only allows subtyping to be declared between type *constructors* of the same kind. That makes it incapable of handling common examples of *heterogeneous* subtyping: neither class B nor D from the example in Section 4.3 is expressible in their system. It might be possible to extend it via more complex subtype assumptions, but the details of how to maintain decidability for such an extension are not obvious, because their definition of "consistent assumptions" L to check subtype declarations essentially implements a coinductive semantics (L is checked with itself in the context).

DOT Calculi. Another system able to readily express mutually recursive subtyping is the DOT calculus [Rompf and Amin 2016]. Its notion of path-dependent types allows types to refer to themselves recursively by detour through the term level and by relying on the dependently-typed fixpoint operator that is built into the semantics. In his recent thesis, Martres [2022] shows that DOT can encode FGJ, including methods. However, DOT and its descendent calculi come with a rather heavyweight meta-theory and type-checking procedure, whereas our goals are minimising algorithmic cost as well as staying close to more standard semantics.

7 DISCUSSION AND FUTURE WORK

The λ_{misu} calculus provides a fairly complete semantics for recursive types as found in mainstream programming languages, including both regular and irregular inter- and intra-recursion subtyping at higher kinds — with two primary omissions subject to future work, which we discuss below: (1) subtyping is *singular*, i.e., every type can have at most one supertype; (2) bounds cannot recursively refer to themselves, i.e., F-bounded recursion [Canning et al. 1989; Ghelli 1997] is not supported.

Other than that, the calculus' main limitation (as well as feature) compared to more puristic type theories is that recursive types can only be in a subtype relation if declared so upfront. The slogan of "*nominal typing up to canonicalisation*" describes the intuition behind this. We conjecture that under this general approach, (a) non-recursive structural types, (b) recursive structural types without subtyping, and (c) recursive nominal types with subtyping, can all be expressed, covering a fairly wide range of practical programming languages (though some languages would obviously benefit from also adding the — mostly orthogonal — features discussed below). There are few practical languages that inhabit the missing point in the spectrum, that of fully structural recursive subtyping. In Wasm, compiling languages outside this range will require additional runtime casts.

Multiple Supertypes. Extending λ_{misu} with support for multiple supertypes amounts to adding intersection types to the system. For example, they would allow forming a type like

$$\tau \quad = \quad \mu\alpha{\leq}(\tau_1 \wedge \tau_2 \wedge \tau_3).\tau'$$

such that $\tau \leq \tau_i$ for all $i \in \{1, 2, 3\}$. Compagnoni [1995] already developed the meta-theory of higher-order subtyping with intersection types. Their impact is almost entirely orthogonal to our extension with iso-recursive types, and in principle, we foresee no problem in combining both.

However, subtyping and type checking become rather expensive in the presence of intersection types. Subtyping for μ-types would essentially need to perform a search in a directed acyclic graph, versus just a linear search in a linear subtype hierarchy in λ_{misu} as is. Type checking would have to compute least upper bounds when elimination constructs like application or projection encounter intersection types. While it is probably not possible to bring down the cost of the subtyping check itself — programming languages with multiple inheritance have the same problem — it may still be possible to reduce the overall cost of type checking and its use of these expensive subtype checks through suitable restrictions on the formation of intersections. Interestingly, the problem can be vastly reduced if we limit the use of intersections to μ-bounds: intersections will only enter the type environment as the bounds of μ-type variables, and hence arise only during the well-formedness check for μ, which only involves subtyping. Intersections would not show up during type checking of terms. But it is not clear whether such a system would be sufficient: intersections are e.g. desirable on the bounds of quantified types. Perhaps it is sufficient to use a μ-type itself as a bound in such a case. That is, in place of abstracting $\forall\alpha \leq \tau_1 \wedge \tau_2.\tau$, it may conceivable to use $\forall\alpha \leq (\mu\alpha' \leq \tau_1 \wedge \tau_2.\alpha').\tau$. We leave further investigation to future work.

F-bounded μ-types. Another feature that shows up in many contemporary object-oriented languages is F-bounded quantification [Canning et al. 1989; Ghelli 1997], where a type variable may appear in its own bound. In Section 3.3, we consciously skirted the introduction of F-bounded quantification for expressing intra-recursion subtyping. Of course, if we were to buy into F-bounds anyway, then the extension with tuple-kinded μ-types as in the full λ_{misu} calculus would become unnecessary, since they could be encoded easily:

$$\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau \quad := \quad \mu\alpha{\leq}\langle\tau_1, \tau_2[\alpha.1/\alpha_1]\rangle.\tau[\alpha.1/\alpha_1, \alpha.2/\alpha_2]$$

Unfortunately, a naive extension with unrestricted F-bounds would immediately render the type system undecidable, due to its higher-order nature. For example, the type $\mu\alpha{\leq}\alpha(\alpha(\tau)).\tau'$ would cause promotion in the subtype algorithm from Figure 3 to loop with an ever-expanding type elimination context. There are solutions to this, e.g., involving suitable syntactic restrictions on F-bounds, but working out the details such that they support our encodings and other common use cases, while maintaining decidability, requires some investigation.

Variance on Type Parameters. Another extension often desired for the combination of type constructors and subtyping is the ability to annotate type parameters with *variances*, such that applications of a constructor can still subtype co- or contravariantly. For example, this would allow the type $\mathit{triple}(\mathit{nat})$ from Section 5.1 to be recognised as a subtype of $\mathit{triple}(\mathit{int})$.

Fortunately, Steffen already worked out all the necessary machinery for adding such annotations to higher-order subtyping in his thesis [Steffen 1997]. This extension is completely orthogonal to the mechanism considered in this paper and could readily be added to λ_{misu}.

Mechanisation. We have proved all relevant properties of λ_{misu} in the Appendix. Fortunately, we were able to closely follow the ground works of Comapgnoni [1995] and Pierce and Steffen [1997]. But we would not mind to verify these proofs the "contemporary" way, in a theorem prover like Coq. However, the meta-theory of higher-order subtyping in plain $F^{\omega}_{\leq}$ already is quite involved, and especially the proof for transitivity elimination via rewriting of derivations difficult to render in Coq. We are not aware of prior work tackling the mechanisation of $F^{\omega}_{\leq}$. We hence have to leave mechanised proofs for λ_{misu} to future work.

ACKNOWLEDGEMENTS

We would like to thank Sam Lindley, Luke Wagner, Steven Keuchel, and Conrad Watt for insightful discussions and pointers to kick off this work, the WebAssembly CG, especially Thomas Lively, for a wealth of implementation feedback, and the anonymous referees for their valuable comments.

REFERENCES

Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. 1996. An Interpretation of Objects and Object Types. In *Principles of Programming Languages (POPL)*. ACM Press, 396–409.

Martín Abadi and Marcelo Fiore. 1996. Syntactic considerations on recursive types. In *Logic in Computer Science*. IEEE, 242–252.

Andreas Abel. 2009. Implementing a normalizer using sized heterogeneous types. *Journal of Functional Programming* 19, 3–4 (2009), 287–310.

Andreas Abel and Dulma Rodriguez. 2008. Syntactic Metatheory of Higher-Order Subtyping. In *Computer Science Logic*, Michael Kaminski and Simone Martini (Eds.). Springer-Verlag, 446–460.

Roberto Amadio and Luca Cardelli. 1993. Subtyping recursive types. *Transactions on Programming Languages and Systems* 15, 4 (1993), 575–631.

Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaetà. 1999. Basic theory of F-bounded quantification. *Information and Computation* 153, 2 (1999), 173–237.

Henk Barendregt. 1984. *The Lambda Calculus* (2nd ed.). North-Holland.

Henk Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Samson Abramsky, Dov Gabbay, and T.S.E. Maibaum (Eds.). Vol. 2. Oxford University Press, Oxford, Chapter 2, 117–309.

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33 (1998), 309–338.

Kim Bruce, Luca Cardelli, and Benjamin Pierce. 1999. Comparing Object Encodings. *Information and Computation* 155, 1/2 (1999), 108–133.

Peter Canning, William Cook, Walter Hill, John Mitchell, and William Olthoff. 1989. F-bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 273–280.

Luca Cardelli. 1986. Amber. In *Combinators and Functional Programming Languages (LNCS, 242)*, Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet (Eds.). Springer-Verlag, 21–47.

Luca Cardelli. 1994. Extensible Records in a Pure Calculus of Subtyping. In *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 373–425.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17, 4 (1985), 471–522.

Adriana Compagnoni. 1995. Decidability of higher-order subtyping with intersection types. In *Annual Conference of the European Association for Computer Science Logic (CSL '94) (LNCS, 933)*. Springer-Verlag, 46–60.

Adriana Compagnoni. 1997. *Subject Reduction and Minimal Types for Higher Order Subtyping*. Technical Report ECS-LFCS-97-363. University of Edinburgh.

William Cook, Walter Hill, and Peter Canning. 1989. Inheritance Is Not Subtyping. In *Principles of Programming Languages (POPL)*. ACM Press, 125–135.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *Programming Language Design and Implementation (PLDI)*. ACM Press, Boston, USA, 50–63.

Roland Ducournau. 2011. Implementing Statically Typed Object-Oriented Programming Languages. *Comput. Surveys* 43, 4 (2011), 80–131.

Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. 2002. Recursive subtyping revealed. *Journal of Functional Programming* 12, 6 (2002), 511–548.

Giorgio Ghelli. 1997. Termination of system F-bounded: A complete proof. *Information and Computation* 139, 1 (1997), 39–56.

Neal Glew. 2000. An efficient class and object encoding. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 311–324.

Neal Glew. 2012. *Subtyping for F-Bounded Quantifiers and Equirecursive Types*. Springer-Verlag, 66–82.

Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Notices* 52, 6 (2017), 185–200.

Robert Harper and Chris Stone. 2000. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press.

Haruo Hosoya, Benjamin Pierce, and David Turner. 1998. Datatypes and Subtyping.

Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450.

Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. 1993. Efficient recursive subtyping. In *Principles of Programming Languages (POPL)*. ACM Press, Charleston, USA, 419–428.

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. 2017. On Subtyping-Relation Completeness, with an Application to Iso-Recursive Types. *Transactions on Programming Languages and Systems* 39, 1 (2017), 1–36.

Sam Lindley, Daniel Hillerström, Luna Phipps-Costin, Andreas Rossberg, Matija Pretnar, KC Sivaramakrishnan, Arjun Guha, and Daan Leijen. 2022. Stack Switching Proposal for WebAssembly. https://github.com/WebAssembly/stack-switching/.

Guillaume Martres. 2022. *Type-Preserving Compilation of Class-Based Languages.* Ph. D. Dissertation. École Polytechnique Fédérale de Lausanne.

Chris Okasaki. 1999. From fast exponentiation to square matrices: an adventure in types. In *International Conference on Functional Programming (ICFP).* ACM Press, 28–35.

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press, to appear.

Benjamin Pierce. 2002. *Types and Programming Languages.* MIT Press, Boston, USA.

Benjamin Pierce and Martin Steffen. 1997. Higher-Order Subtyping. *Theoretical Computer Science* 176, 1–2 (1997), 235–282.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press, 624–641.

Andreas Rossberg. 2022. GC Proposal for WebAssembly. https://github.com/WebAssembly/gc/.

Martin Steffen. 1997. *Polarized Higher-Order Subtyping.* Ph. D. Dissertation. Universität Erlangen-Nürnberg.

William Tait. 1975. A realizability interpretation of the theory of species. In *Logic Colloquium*, R. Parikh| (Ed.). Number 453 in LNCS. Springer-Verlag, 240–251.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2004. A Concurrent Logical Framework: The Propositional Fragment. In *Types for Proofs and Programs.* LNCS, Vol. 3085. Springer-Verlag, 355–377.

Yaoda Zhou, Junixu Zhao, and Bruno C. D. S. Oliveira. 2022. Revisiting Iso-Recursive Subtyping. *Transactions on Programming Languages and Systems* 44, 4 (2022), 1–54.

A META-THEORY OF BASIC λ_{misu}

In this section, we prove the meta-theory of basic λ_{misu} with recursive types of the form $\mu\alpha{\leq}\tau.\tau'$. Appendix B will treat the extension to the full calculus with the more powerful form $\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau'$.

For most part, our development closely follows either that of Pierce & Steffen [1997] or Compagnoni's [**?**], minus intersection types (other than the empty one, which corresponds to the top type), but with the addition of type tuples and higher-order iso-recursive types.

A.1 Regularity

First, a few simple facts about higher-order top types.

Lemma A.1 (Kinding of Higher-Order Top). *If* $\vdash \Gamma$ ok, *then* $\Gamma \vdash \top_\kappa : \kappa$.

Proof. By induction on κ. □

Lemma A.2 (Subtyping of Higher-Order Top). *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \vdash \tau \leq \top_\kappa$.

Proof. By induction on the derivation. The interesting cases are:

- Case $\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 : \kappa$
  - by inversion, $\Gamma \vdash \tau_1 : \kappa$
  - by induction, $\Gamma \vdash \tau_1 \leq \top_\kappa$
  - by subtyping rule for recursive types, $\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 \leq \tau_1$
  - by subtyping transitivity rule, $\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 \leq \top_\kappa$
- Case $\Gamma \vdash \tau.i : \kappa_i$
  - by inversion, $\Gamma \vdash \tau : \kappa_1 \times \kappa_2$
  - by induction, $\Gamma \vdash \tau \leq \top_{\kappa_1\times\kappa_2}$
  - by subtyping rule for projection, $\Gamma \vdash \tau.i \leq \top_{\kappa_1\times\kappa_2}.i$
  - by kinding rule for projection, $\Gamma \vdash \top_{\kappa_1\times\kappa_2}.i : \kappa_i$
  - by Lemma A.1, $\Gamma \vdash \top_{\kappa_i} : \kappa_i$
  - by β-reduction of projection, $\top_{\kappa_1\times\kappa_2}.i \equiv \top_{\kappa_i}$
  - by subtyping reflexivity rule, $\Gamma \vdash \top_{\kappa_1\times\kappa_2}.i \leq \top_{\kappa_i}$
  - by subtyping transitivity rule, $\Gamma \vdash \tau.i \leq \top_{\kappa_i}$
- Case $\Gamma \vdash \tau_1\ \tau_2 : \kappa$
  - by inversion, $\Gamma \vdash \tau_1 : \kappa_2 \longrightarrow \kappa$ and $\Gamma \vdash \tau_2 : \kappa_2$
  - by induction, $\Gamma \vdash \tau_1 \leq \top_{\kappa_2\to\kappa}$
  - by subtyping rule for application, $\Gamma \vdash \tau_1\ \tau_2 \leq \top_{\kappa_2\to\kappa}\ \tau_2$
  - by kinding rule for application, $\Gamma \vdash \top_{\kappa_2\to\kappa}\ \tau_2 : \kappa$
  - by Lemma A.1, $\Gamma \vdash \top_\kappa : \kappa$
  - by β-reduction of application, $\top_{\kappa_2\to\kappa}\ \tau_2 \equiv \top_\kappa$
  - by subtyping reflexivity rule, $\Gamma \vdash \top_{\kappa_2\to\kappa}\ \tau_2 \leq \top_\kappa$
  - by subtyping transitivity rule, $\Gamma \vdash \tau_1\ \tau_2 \leq \top_\kappa$ □

This establishes $\top_\kappa$ as a maximum element of kind κ.

Lemma A.3 (Regularity).

(1) If $\Gamma \vdash \tau : \kappa$, *then* $\vdash \Gamma$ ok.
(2) If $\Gamma \vdash e : \tau$, *then* $\vdash \Gamma$ ok *and* $\Gamma \vdash \tau : \Omega$.

Proof. Both by induction on the derivation. □

A.2 Type Reduction

Definition A.1 (Type Reduction). *One-step β-*reduction *on types, written $\tau \hookrightarrow \tau'$, is defined by the congruent closure of the following rules:*

$$\begin{array}{lcl} \langle \tau_1, \tau_2 \rangle.i & \hookrightarrow & \tau_i \\ (\lambda \alpha{:}\kappa.\tau_1)\ \tau_2 & \hookrightarrow & \tau_1[\tau_2/\alpha] \end{array}$$

The relation $\hookrightarrow$ on types is extended pointwise to contexts Γ. We write the reflexive, transitive closure of $\hookrightarrow$ as $\hookrightarrow^*$. The reflexive, transitive, symmetric closure is written $\equiv$.

Definition A.2 (Neutral Type). *A* neutral *type is one of the form $T[\alpha]$ or $T[\mu\alpha{\leq}\tau_1.\tau_2]$.*

The rest of this section proves the Church-Rosser property and closely follows the development of Pierce & Steffen [1997], Section 3. First, we define *parallel reduction* $\hookrightarrow\!\!\!\!\rightarrow$.

Definition A.3 (Parallel Reduction).

$$\frac{}{\tau \hookrightarrow\!\!\!\!\rightarrow \tau} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\tau_1 \times \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\tau_1 \to \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \to \tau_2'}$$

$$\frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\forall \alpha{\leq}\tau_1.\tau_2 \hookrightarrow\!\!\!\!\rightarrow \forall \alpha{\leq}\tau_1'.\tau_2'} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\mu \alpha{\leq}\tau_1.\tau_2 \hookrightarrow\!\!\!\!\rightarrow \mu \alpha{\leq}\tau_1'.\tau_2'}$$

$$\frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\langle \tau_1, \tau_2 \rangle.i \hookrightarrow\!\!\!\!\rightarrow \langle \tau_1', \tau_2' \rangle} \qquad \frac{\tau \hookrightarrow\!\!\!\!\rightarrow \tau'}{\tau.i \hookrightarrow\!\!\!\!\rightarrow \tau'.i} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\langle \tau_1, \tau_2 \rangle.i \hookrightarrow\!\!\!\!\rightarrow \tau_i'}$$

$$\frac{\tau \hookrightarrow\!\!\!\!\rightarrow \tau'}{\lambda \alpha{:}\kappa.\tau \hookrightarrow\!\!\!\!\rightarrow \lambda \alpha{:}\kappa.\tau'} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{\tau_1\ \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_1'\ \tau_2'} \qquad \frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'}{(\lambda \alpha{:}\kappa.\tau_1)\ \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_1'[\tau_2'/\alpha]}$$

Parallel reduction is related to single- and multi-step reduction as follows:

Lemma A.4 (Relating Reductions).

(1) $\hookrightarrow \subset \hookrightarrow\!\!\!\!\rightarrow$
(2) $\hookrightarrow^* \supset \hookrightarrow\!\!\!\!\rightarrow$
(3) $\hookrightarrow^* = \hookrightarrow\!\!\!\!\rightarrow^*$

The following auxiliary lemma on substitutions is needed for the proof of Lemma A.6:

Lemma A.5 (Commuting Substitutions).
If $\alpha_1 \neq \alpha_2$ and $\alpha_1 \notin \mathrm{ftv}(\tau_2)$, then $\tau[\tau_1/\alpha_1][\tau_2/\alpha_2] = \tau[\tau_2/\alpha_2][\tau_1[\tau_2/\alpha_2]/\alpha_1]$.

Substitution commutes with reductions:

Lemma A.6 (Substitution for Reduction).

(1) If $\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1'$ and $\tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2'$, then $\tau_1[\tau_2/\alpha] \hookrightarrow\!\!\!\!\rightarrow \tau_1'[\tau_2'/\alpha]$.
(2) If $\tau_1 \hookrightarrow\!\!\!\!\rightarrow^* \tau_1'$ and $\tau_2 \hookrightarrow\!\!\!\!\rightarrow^* \tau_2'$, then $\tau_1[\tau_2/\alpha] \hookrightarrow\!\!\!\!\rightarrow^* \tau_1'[\tau_2'/\alpha]$.
(3) If $\tau_1 \hookrightarrow^* \tau_1'$ and $\tau_2 \hookrightarrow^* \tau_2'$, then $\tau_1[\tau_2/\alpha] \hookrightarrow^* \tau_1'[\tau_2'/\alpha]$.

As a consequence, an outermost redex can always be reduced directly:

Lemma A.7 (Outermost Reduction).

(1) If $\langle \tau_1, \tau_2 \rangle.i \hookrightarrow^* \tau'$ with $\tau' \neq \langle \tau_1', \tau_2' \rangle.i$ for any $\tau_1 \hookrightarrow^* \tau_1'$ and $\tau_2 \hookrightarrow^* \tau_2'$, then $\tau_i \hookrightarrow^* \tau'$.
(2) If $(\lambda \alpha{:}\kappa.\tau_1)\ \tau_2 \hookrightarrow^* \tau'$ with $\tau' \neq (\lambda \alpha{:}\kappa.\tau_1')\ \tau_2'$ for any $\tau_1 \hookrightarrow^* \tau_1'$ and $\tau_2 \hookrightarrow^* \tau_2'$, then $\tau_1[\tau_2/\alpha] \hookrightarrow^* \tau'$.

Lemma A.8 (Diamond Property for Parallel Reduction).
If $\tau \hookrightarrow\!\!\!\!\rightarrow \tau_1$ and $\tau \hookrightarrow\!\!\!\!\rightarrow \tau_2$, then $\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau'$ and $\tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau'$ for some τ'.

Proof. See Barendregt [?]. □

Corollary A.9 (Confluence ("Church-Rosser")).
If $\tau \hookrightarrow^* \tau_1$ and $\tau \hookrightarrow^* \tau_2$, then $\tau_1 \hookrightarrow^* \tau'$ and $\tau_2 \hookrightarrow^* \tau'$ for some τ'.

A.3 Contexts

Context formation supports the obvious inversion lemmas:

Lemma A.10 (Inversion of Context Formation).

(1) If $\vdash \Gamma_1, x{:}\tau, \Gamma_2$ ok, *then* $\Gamma_1 \vdash \tau : \Omega$ *by a proper subderivation.*
(2) If $\vdash \Gamma_1, \alpha{\leq}\tau, \Gamma_2$ ok, *then* $\Gamma_1 \vdash \tau : \kappa$ *by a proper subderivation.*

Furthermore, the following properties hold type variables occurring in kinding derivations:

Lemma A.11 (Free Variables).

(1) If $\Gamma \vdash \tau : \kappa$, *then* $\mathrm{ftv}(\tau) \subseteq \mathrm{dom}(\Gamma)$.
(2) If $\vdash \Gamma$ ok, *then each* $\alpha \in \mathrm{dom}(\Gamma)$ *is bound exactly once.*

A.4 Pre-Kinding

One structural difference between λ_{misu} and $\mathrm{F}^{\omega}_{\leq}$ is that its kinding and subtyping judgements are mutually dependent, due to the subtyping premise in rule K-Rec-Sup. In order to stratify the formal development, we hence define a variation of the kinding relation that drops that premise. That is okay for the purpose of proving strong normalisation of type reduction, because this premise is only needed to ensure soundness of subtyping and term-level reduction, it does not affect type-level reductions.

Definition A.4 (Pre-Kinding). *The relation* $\Gamma \vdash' \tau : \kappa$ *is the same as* $\Gamma \vdash \tau : \kappa$, *but with the following replacement rule for* μ*-types:*

$$\frac{\Gamma \vdash' \tau_1 : \kappa \qquad \Gamma \vdash' \tau_2 : \kappa}{\Gamma \vdash' \mu\alpha{\leq}\tau_1.\tau_2 : \kappa}$$

Context pre-kinding $\vdash' \Gamma$ ok *is the same as* $\vdash \Gamma$ ok, *but using pre-kinding.*

This relation is implied by ordinary kinding:

Lemma A.12 (Kinding Implies Pre-Kinding).

(1) If $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \vdash' \tau : \kappa$.
(2) If $\vdash \Gamma$ ok, *then* $\vdash' \Gamma$ ok.

With this, we can largely follow Pierce & Steffen [1997], Section 4.

Definition A.5 (Algorithmic Pre-Kinding). *The relation* $\Gamma \vdash'' \tau : \kappa$ *is the same as* $\Gamma \vdash' \tau : \kappa$, *but with the following replacement rule for type variables:*

$$\frac{\Gamma_1 \vdash'' \tau : \kappa \qquad \vdash'' \Gamma_1, \alpha{\leq}\tau, \Gamma_2 \text{ ok}}{\Gamma_1, \alpha{\leq}\tau, \Gamma_2 \vdash'' \alpha : \kappa}$$

Algorithmic context pre-kinding $\vdash'' \Gamma$ ok *is the same as* $\vdash' \Gamma$ ok, *but using algorithmic pre-kinding.*

Lemma A.13 (Strengthening for Algorithmic Pre-Kinding).

(1) If $\Gamma_1, x{:}\tau', \Gamma_2 \vdash'' \tau : \kappa$, *then* $\Gamma_1, \Gamma_2 \vdash'' \tau : \kappa$.
(2) If $\Gamma_1, \alpha{\leq}\tau', \Gamma_2 \vdash'' \tau : \kappa$ *and* $\alpha \notin \mathrm{ftv}(\Gamma_2) \cup \mathrm{ftv}(\tau)$, *then* $\Gamma_1, \Gamma_2 \vdash'' \tau : \kappa$.
(3) If $\vdash'' \Gamma_1, x{:}\tau', \Gamma_2$ ok, *then* $\vdash'' \Gamma_1, \Gamma_2$ ok.
(4) If $\vdash'' \Gamma_1, \alpha{\leq}\tau', \Gamma_2$ ok *and* $\alpha \notin \mathrm{ftv}(\Gamma_2)$, *then* $\vdash'' \Gamma_1, \Gamma_2$ ok.

Both formulations of pre-kinding are equivalent:

Lemma A.14 (Equivalence of Algorithmic Pre-Kinding).

(1) $\Gamma \vdash' \tau : \kappa$ *if and only if* $\Gamma \vdash'' \tau : \kappa$.
(2) $\vdash' \Gamma$ ok *if and only if* $\vdash'' \Gamma$ ok.

Proof. Each direction by induction on derivations. The only interesting case is the variable rule, since all others are unchanged. □

Lemma A.15 (Decidability of Algorithmic Pre-Kinding).

(1) The relation $\Gamma \vdash'' \tau : \kappa$ *is decidable.*
(2) The relation $\vdash'' \Gamma$ ok *is decidable.*

Proof. The rules of algorithmic pre-kinding can be read as an algorithm bottom-up. This algorithm terminates, because for every rule, all premises are on smaller syntactic structures than the conclusion. □

Corollary A.16 (Decidability of Pre-Kinding).

(1) The relation $\Gamma \vdash' \tau : \kappa$ *is decidable.*
(2) The relation $\vdash' \Gamma$ ok *is decidable.*

From the algorithmic formulation it also follows that all kinds are uniquely determined.

Corollary A.17 (Uniqueness of Kinds).

(1) If $\Gamma \vdash'' \tau : \kappa_1$ *and* $\Gamma \vdash'' \tau : \kappa_2$, *then* $\kappa_1 = \kappa_2$.
(2) If $\Gamma \vdash' \tau : \kappa_1$ *and* $\Gamma \vdash' \tau : \kappa_2$, *then* $\kappa_1 = \kappa_2$.
(3) If $\Gamma \vdash \tau : \kappa_1$ *and* $\Gamma \vdash \tau : \kappa_2$, *then* $\kappa_1 = \kappa_2$.

Proof.

(1) By induction on τ, using that algorithmic kinding is syntax-directed.
(2) Follows immediately from (1) and Lemma A.14.
(3) Follows immediately from (2) and Lemma A.12. □

Finally, we have a number of useful structural properties related to pre-kinding.

Lemma A.18 (Transposition of Pre-Kinding).

If $\Gamma_1, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2, \Gamma_2 \vdash' \tau : \kappa$ *and* $\vdash' \Gamma_1, \alpha_2 \leq \tau_2, \alpha_1 \leq \tau_1, \Gamma_2$ ok, *then* $\Gamma_1, \alpha_2 \leq \tau_2, \alpha_1 \leq \tau_1, \Gamma_2 \vdash' \tau : \kappa$.

Lemma A.19 (Weakening of Pre-Kinding).

If $\Gamma \vdash' \tau : \kappa$ *and* $\vdash' \Gamma'$ ok *with* $\Gamma \subseteq \Gamma'$, *then* $\Gamma' \vdash' \tau : \kappa$.

Lemma A.20 (Context Update with Pre-Kinding).

If $\Gamma_1, \alpha \leq \tau_1, \Gamma_2 \vdash' \tau : \kappa$ *and* $\Gamma_1 \vdash' \tau_1 : \kappa_1$ *and* $\Gamma_1 \vdash' \tau_2 : \kappa_1$, *then* $\Gamma_1, \alpha \leq \tau_2, \Gamma_2 \vdash' \tau : \kappa$.

Lemma A.21 (Substitution for Pre-Kinding). *Suppose* $\Gamma_1 \vdash' \tau : \kappa$.

(1) If $\Gamma_1, \alpha \leq \top_\kappa, \Gamma_2 \vdash' \tau' : \kappa'$, *then* $\Gamma_1, \Gamma_2[\tau/\alpha] \vdash' \tau'[\tau/\alpha] : \kappa'$.
(2) If $\Gamma_1, \alpha \leq \top_\kappa, \Gamma_2$ ok, *then* $\vdash' \Gamma_1, \Gamma_2[\tau/\alpha]$ ok.

Lemma A.22 (Preservation of Pre-Kinding).

(1) If $\Gamma \vdash' \tau : \kappa$ *and* $\tau \hookrightarrow^* \tau'$, *then* $\Gamma \vdash' \tau' : \kappa$.
(2) If $\Gamma \vdash' \tau : \kappa$ *and* $\Gamma \hookrightarrow^* \Gamma'$, *then* $\Gamma' \vdash' \tau : \kappa$.

Corollary A.23 (Preservation of Kinding).

(1) If $\Gamma \vdash \tau : \kappa$ *and* $\tau \hookrightarrow^* \tau'$, *then* $\Gamma \vdash \tau' : \kappa$.
(2) If $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \hookrightarrow^* \Gamma'$, *then* $\Gamma' \vdash \tau : \kappa$.

Lemma A.24 (Pre-Kind Invariance).

If $\Gamma \vdash' \tau_1 : \kappa_1$ *and* $\Gamma \vdash' \tau_2 : \kappa_2$ *and* $\tau_1 \equiv \tau_2$, *then* $\kappa_1 = \kappa_2$.

A.5 Strong Normalisation of Types

We prove strong normalisation – or termination – of type reduction by using Tait's method [**??**], based on pre-kinding.

Definition A.6 (Terminating Types).

(1) τ *normal* $\Leftrightarrow \neg\exists\tau'.\ \tau \hookrightarrow \tau'$
(2) τ *terminates* $\Leftrightarrow \forall\tau'.\ \tau \hookrightarrow \tau' \Rightarrow \tau'$ *terminates*
(3) T *terminates* $\Leftrightarrow T[\alpha]$ *terminates*

With this, we can define the standard logical predicate:

Definition A.7 (Type Interpretation).

$$\begin{array}{rcl} [\![\Omega]\!] & = & \{\tau \mid \tau \textit{ terminates}\} \\ [\![\kappa_1 \times \kappa_2]\!] & = & \{\tau \mid \tau.1 \in [\![\kappa_1]\!] \wedge \tau.2 \in [\![\kappa_2]\!]\} \\ [\![\kappa_1 \to \kappa_2]\!] & = & \{\tau \mid \forall \tau_1 \in [\![\kappa_1]\!].\ \tau\ \tau_2 \in [\![\kappa_2]\!]\} \\ [\![\Gamma]\!] & = & \{\gamma \mid \forall (\alpha \leq \tau) \in \Gamma.\, \Gamma \vdash \tau : \kappa \Rightarrow \gamma(\alpha) \in [\![\kappa]\!]\} \end{array}$$

Lemma A.25 (Soundness wrt. Termination ("Main Lemma")).
If $\tau \in [\![\kappa]\!]$, then τ terminates.

Proof. By induction on κ. □

We need the following lemmas to prove the Fundamental Property of the logical relation.

Lemma A.26 (Neutral Types).

(1) If $T[\alpha]$ terminates, then $T[\alpha] \in [\![\kappa]\!]$.
(2) If $T[\mu\alpha{\leq}\tau_1.\tau_2]$ terminates, then $T[\mu\alpha{\leq}\tau_1.\tau_2] \in [\![\kappa]\!]$.

Proof.

(1) By induction on κ.
- Case Ω
  - immediate by definition of $[\![\Omega]\!]$
- Case $\kappa_1 \times \kappa_2$
  - let $T_1 = T.1$ and $T_2 = T.2$
  - then $T_1[\alpha]$ and $T_2[\alpha]$ terminates
  - by induction, $T_1[\alpha] \in [\![\kappa_1]\!]$ and $T_2[\alpha] \in [\![\kappa_2]\!]$
  - by definition of $[\![\kappa_1 \times \kappa_2]\!]$, $T[\alpha] \in [\![\kappa_1 \times \kappa_2]\!]$
- Case $\kappa_1 \to \kappa_2$
  - assume $\tau_1 \in [\![\kappa_1]\!]$
  - by Main Lemma, τ_1 terminates
  - let $T' = T\ \tau_1$
  - then $T'[\alpha]$ terminates
  - by induction, $T'[\alpha] \in [\![\kappa_2]\!]$
  - by definition of $[\![\kappa_1 \to \kappa_2]\!]$, $T[\alpha] \in [\![\kappa_1 \to \kappa_2]\!]$

(2) Analogous. □

Lemma A.27 (Termination under Substitution). *If $\tau[\tau'/\alpha]$ terminates, then τ terminates.*

Lemma A.28 (Termination in Context). *If both T and $T[\tau]$ terminate, then τ terminates.*

Lemma A.29 (Closure under Expansion).

(1) If T_1 and T_2 terminate and $T_1[\tau_1] \in [\![\kappa_1]\!]$ and $T_2[\tau_2] \in [\![\kappa_2]\!]$, then $T_i[\langle\tau_1, \tau_2\rangle.i] \in [\![\kappa_i]\!]$.
(2) If T terminates and $T[\tau_2[\tau_1/\alpha]] \in [\![\kappa]\!]$ and $\tau_1 \in [\![\kappa_1]\!]$, then $T[(\lambda\alpha{:}\kappa_1.\tau_2)\ \tau_1] \in [\![\kappa]\!]$.

Proof.

(1) By induction on κ_i.
- Case Ω
  - by definition of $[\![\Omega]\!]$, $T_1[\tau_1]$ and $T_2[\tau_2]$ terminate
  - hence, τ_1 and τ_2 terminate
  - hence, there exists normal forms T_1', T_2', τ_1', τ_2'
  - hence, $T_i[\langle\tau_1, \tau_2\rangle.i] \hookrightarrow^* T_i'[\langle\tau_1', \tau_2'\rangle.i]$
  - at that point, the only possible reduction is $T_i'[\langle\tau_1', \tau_2'\rangle.i] \hookrightarrow T_i'[\tau_i']$

  - but also, $T_i[\tau_i] \hookrightarrow^* T_i'[\tau_i']$
  - because the former terminates, so does the latter
  - hence, $T_i[\langle \tau_1, \tau_2 \rangle.i]$ also terminates
  - by definition of $[\![\Omega]\!]$, $T_i[\langle \tau_1, \tau_2 \rangle.i] \in [\![\Omega]\!]$
- Case $\kappa_{i1} \times \kappa_{i2}$
  - by definition of $[\![\kappa_{i1} \times \kappa_{i2}]\!]$, $T_i[\tau_i].1 \in [\![\kappa_{i1}]\!]$ and $T_i[\tau_i].2 \in [\![\kappa_{i2}]\!]$
  - by assumption, $T_{1-i}[\tau_{1-i}] \in [\![\kappa_{1-i}]\!]$
  - let $T_{i1} = T_i.1$ and $T_{i2} = T_i.2$
  - by induction (1), $T_i[\langle \tau_1, \tau_2 \rangle.i].1 = T_{i1}[\langle \tau_1, \tau_2 \rangle.i] \in [\![\kappa_{i1}]\!]$ and $T_i[\langle \tau_1, \tau_2 \rangle.i].2 = T_{i2}[\langle \tau_1, \tau_2 \rangle.i] \in [\![\kappa_{i2}]\!]$
  - by definition of $[\![\kappa_{i1} \times \kappa_{i2}]\!]$, $T_i[\langle \tau_1, \tau_2 \rangle.i] \in [\![\kappa_{i1} \times \kappa_{i2}]\!]$
- Case $\kappa_{i1} \to \kappa_{i2}$
  - assume $\tau_{i1} \in [\![\kappa_{i1}]\!]$
  - by definition of $[\![\kappa_{i1} \to \kappa_{i2}]\!]$, $T_i[\tau_i]\ \tau_{i1} \in [\![\kappa_{i1}]\!]$
  - by assumption, $T_{1-i}[\tau_{1-i}] \in [\![\kappa_{1-i}]\!]$
  - let $T_i' = T_i\ \tau_{i1}$
  - by induction (1), $T_i[\langle \tau_1, \tau_2 \rangle.i]\ \tau_{i1} = T_i'[\langle \tau_1, \tau_2 \rangle.i] \in [\![\kappa_{i2}]\!]$
  - by definition of $[\![\kappa_{i1} \to \kappa_{i2}]\!]$, $T_i[\langle \tau_1, \tau_2 \rangle.i] \in [\![\kappa_{i1} \to \kappa_{i2}]\!]$

(2) By induction on κ.

- Case Ω
  - by definition of $[\![\Omega]\!]$, $T[\tau_2[\tau_1/\alpha]]$ and τ_1 terminate
  - hence, $\tau_2[\tau_1/\alpha]$ terminates
  - hence, τ_2 terminates
  - hence, there exists normal forms T', τ_1', τ_2'
  - hence, $T[(\lambda\alpha{:}\kappa_1.\tau_2)\ \tau_1] \hookrightarrow^* T'[(\lambda\alpha{:}\kappa_1.\tau_2')\ \tau_1']$
  - at that point, the only possible reduction is $T'[(\lambda\alpha{:}\kappa_1.\tau_2')\ \tau_1'] \hookrightarrow T'[\tau_2'[\tau_1'/\alpha]]$
  - but also, $T[\tau_2[\tau_1/\alpha]] \hookrightarrow^* T'[\tau_2'[\tau_1'/\alpha]]$
  - because the former terminates, so does the latter
  - hence, $T[(\lambda\alpha{:}\kappa_1.\tau_2)\ \tau_1]$ also terminates
  - by definition of $[\![\Omega]\!]$, $T[(\lambda\alpha{:}\kappa_1.\tau_2)\ \tau_1] \in [\![\Omega]\!]$
- Case $\kappa_{i1} \times \kappa_{i2}$
  - analogous to (1)
- Case $\kappa_{i1} \to \kappa_{i2}$
  - analogous to (1) □

Lemma A.30 (Completeness wrt. Kinding ("Fundamental Property")).
If $\Gamma \vdash' \tau : \kappa$ *and* $\gamma \in [\![\Gamma]\!]$, *then* $\gamma(\tau) \in [\![\kappa]\!]$.

Proof. By induction on the derivation.

- Case $\Gamma \vdash' \alpha : \kappa$
  - by inversion, $\Gamma \vdash' \Gamma(\alpha) : \kappa$
  - by definition of $[\![\Gamma]\!]$, $\gamma(\alpha) \in [\![\kappa]\!]$
- Case $\Gamma \vdash' \top : \Omega$
  - immediate by definition of $[\![\Omega]\!]$
- Case $\Gamma \vdash' \tau_1 \times \tau_2 : \Omega$
  - by inversion, $\Gamma \vdash' \tau_1 : \Omega$ and $\Gamma \vdash' \tau_2 : \Omega$
  - by induction, $\gamma(\tau_1) \in [\![\Omega]\!]$ and $\gamma(\tau_2) \in [\![\Omega]\!]$
  - by definition of $[\![\Omega]\!]$, $\gamma(\tau_1)$ and $\gamma(\tau_2)$ terminate

  - hence $\gamma(\tau_1) \times \gamma(\tau_2)$ terminates
  - by definition of $[\![\Omega]\!]$, $\gamma(\tau_1 \times \tau_2) = \gamma(\tau_1) \times \gamma(\tau_2) \in [\![\Omega]\!]$
- Case $\Gamma \vdash' \tau_1 \to \tau_2 : \Omega$
  - analogous
- Case $\Gamma \vdash' \forall\alpha{\leq}\tau_1.\tau_2 : \Omega$ (assuming α is fresh)
  - by inversion, $\Gamma \vdash' \tau_1 : \kappa_1$ and $\Gamma, \alpha{\leq}\tau_1 \vdash' \tau_2 : \Omega$
  - by induction, $\gamma(\tau_1) \in [\![\kappa_1]\!]$
  - let $\Gamma' = \Gamma, \alpha{\leq}\tau_1$
  - let $\gamma' = \gamma \circ [\alpha/\alpha]$
  - by Neutral Types, $\alpha \in [\![\kappa_1]\!]$
  - by definition of $[\![\Gamma']\!]$, $\gamma' \in [\![\Gamma']\!]$
  - by induction, $\gamma(\tau_2) = \gamma'(\tau_2) \in [\![\Omega]\!]$
  - by Main Lemma, $\gamma(\tau_1)$ and $\gamma(\tau_2)$ terminate
  - hence $\forall\alpha{\leq}\gamma(\tau_1).\gamma(\tau_2)$ terminates
  - by definition of $[\![\Omega]\!]$, $\gamma(\forall\alpha{\leq}\tau_1.\tau_2) = \forall\alpha{\leq}\gamma(\tau_1).\gamma(\tau_2) \in [\![\Omega]\!]$
- Case $\Gamma \vdash' \mu\alpha{\leq}\tau_1.\tau_2 : \kappa$ (assuming α is fresh)
  - by inversion, $\Gamma \vdash' \tau_1 : \kappa$ and $\Gamma, \alpha{\leq}\tau_1 \vdash' \tau_2 : \kappa$
  - by induction, $\gamma(\tau_1) \in [\![\kappa]\!]$
  - let $\Gamma' = \Gamma, \alpha{\leq}\tau_1$
  - let $\gamma' = \gamma \circ [\alpha/\alpha]$
  - by Neutral Types, $\alpha \in [\![\kappa]\!]$
  - by definition of $[\![\Gamma']\!]$, $\gamma' \in [\![\Gamma']\!]$
  - by induction, $\gamma(\tau_2) = \gamma'(\tau_2) \in [\![\kappa]\!]$
  - by Main Lemma, $\gamma(\tau_1)$ and $\gamma(\tau_2)$ terminate
  - hence, $\mu\alpha{\leq}\gamma(\tau_1).\gamma(\tau_2)$ terminates
  - by Neutral Types, $\gamma(\mu\alpha{\leq}\tau_1.\tau_2) = \mu\alpha{\leq}\gamma(\tau_1).\gamma(\tau_2) \in [\![\kappa]\!]$
- Case $\Gamma \vdash' \langle\tau_1, \tau_2\rangle : \kappa_1 \times \kappa_2$
  - by inversion, $\Gamma \vdash' \tau_1 : \kappa_1$ and $\Gamma \vdash' \tau_2 : \kappa_2$
  - by induction, $\gamma(\tau_1) \in [\![\kappa_1]\!]$ and $\gamma(\tau_2) \in [\![\kappa_2]\!]$
  - by reduction, $\langle\gamma(\tau_1), \gamma(\tau_2)\rangle.1 \hookrightarrow \gamma(\tau_1)$ and $\langle\gamma(\tau_1), \gamma(\tau_2)\rangle.2 \hookrightarrow \gamma(\tau_2)$
  - by Closure under Expansion, $\langle\gamma(\tau_1), \gamma(\tau_2)\rangle.i \in [\![\kappa_i]\!]$
  - by definition of $[\![\kappa_1 \times \kappa_2]\!]$, $\gamma(\langle\tau_1, \tau_2\rangle) = \langle\gamma(\tau_1), \gamma(\tau_2)\rangle \in [\![\kappa_1 \times \kappa_2]\!]$
- Case $\Gamma \vdash' \tau.i : \kappa_i$
  - by inversion, $\Gamma \vdash' \tau : \kappa_1 \times \kappa_2$
  - by induction, $\gamma(\tau) \in [\![\kappa_1 \times \kappa_2]\!]$
  - by definition of $[\![\kappa_1 \times \kappa_2]\!]$, $\gamma(\tau.i) = \gamma(\tau).i \in [\![\kappa_i]\!]$
- Case $\Gamma \vdash' \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \to \kappa_2$ (assuming α is fresh)
  - by inversion, $\Gamma, \alpha{\leq}\top_{\kappa_1} \vdash' \tau_2 : \kappa_2$
  - assume $\tau_1 \in [\![\kappa_1]\!]$
  - let $\Gamma' = \Gamma, \alpha{\leq}\top_{\kappa_1}$
  - let $\gamma' = \gamma \circ [\tau_1/\alpha]$
  - by Kinding of Higher-order Top, $\Gamma' \vdash' \top_{\kappa_1} : \kappa_1$
  - by definition of $[\![\Gamma']\!]$, $\gamma' \in [\![\Gamma']\!]$
  - by induction, $\gamma(\tau_2)[\tau_1/\alpha] = \gamma'(\tau_2) \in [\![\kappa_2]\!]$
  - by reduction, $(\lambda\alpha{:}\kappa_1.\gamma(\tau_2))\ \tau_1 \hookrightarrow \gamma(\tau_2)[\tau_1/\alpha]$
  - by Closure under Expansion, $\gamma(\lambda\alpha{:}\kappa_1.\tau_2)\ \tau_1 = (\lambda\alpha{:}\kappa_1.\gamma(\tau_2))\ \tau_1 \in [\![\kappa_2]\!]$
  - by definition of $[\![\kappa_1 \to \kappa_2]\!]$, $\gamma(\lambda\alpha{:}\kappa_1.\tau_2) \in [\![\kappa_1 \to \kappa_2]\!]$
- Case $\Gamma \vdash' \tau_1\ \tau_2 : \kappa$

– by inversion, $\Gamma \vdash' \tau_1 : \kappa_2 \to \kappa$ and $\Gamma \vdash' \tau_2 : \kappa_2$
– by induction, $\gamma(\tau_1) \in [\![\kappa_2 \to \kappa]\!]$ and $\gamma(\tau_2) \in [\![\kappa_2]\!]$
– by definition of $[\![\kappa_2 \to \kappa]\!]$, $\gamma(\tau_1\ \tau_2) = \gamma(\tau_1)\ \gamma(\tau_2) \in [\![\kappa]\!]$ □

Theorem A.31 (Strong Normalisation of Types).

(1) If $\Gamma \vdash' \tau : \kappa$, *then* $\tau \hookrightarrow^* \tau'$ *such that* τ' *normal.*
(2) If $\Gamma \vdash \tau : \kappa$, *then* $\tau \hookrightarrow^* \tau'$ *such that* τ' *normal.*

Proof.

(1) By the Main Lemma and the Fundamental Property with the identity substitution as γ, which is in $[\![\Gamma]\!]$ due to Lemma A.26.
(2) Follows directly from (1) and Lemma A.12. □

This result justifies a syntactic shorthand for denoting the unique normal form of types and environments.

Definition A.8 (Normal Form).

$$\begin{array}{lcl} \tau^{\downarrow} & = & \tau' \textit{ for which } \tau \hookrightarrow^* \tau' \wedge \tau' \text{ normal} \\ \Gamma^{\downarrow} & = & \Gamma' \textit{ for which } \mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \wedge \forall \alpha/x \in \mathrm{dom}(\Gamma).\, \Gamma'(\alpha/x) = \Gamma(\alpha/x)^{\downarrow} \end{array}$$

A.6 Normal Subtyping

Having proved decidability of type equivalence via string normalisation in the previous section, we now turn to subtyping. We follow Compagnoni [?], Sections 6–9, by first defining an equivalent *normal* subtype relation that only considers types in normal form, for which it is easier to prove cut elimination. That is, derivations can be simplified by eliminating the use of transitivity and restricting reflexivity to syntactic equality. From this, an algorithmic formulation of subtyping can then be derived.

First, we define the notion of *promotion* [Pierce and Steffen 1997], which generalises a neutral type to its least supertype. [11]

Definition A.9 (Promotion).

$$\begin{array}{lcl} \mathrm{promote}_\Gamma(\alpha) & = & \Gamma(\alpha) \\ \mathrm{promote}_\Gamma(\mu\alpha{\leq}\tau_1.\tau_2) & = & \tau_1 \end{array}$$

Promotion is a partial function, only defined on variables and μ-types, i.e., the roots of neutral types. When applied to neutral types, it preserves kinding and produces a supertype (Lemma A.32). With that, we can define a variant of the subtyping relation that operates on normal types only.

Definition A.10 (Normal Subtyping). *The relation* $\Gamma \vdash_n \tau \leq \tau'$ *is the same as* $\Gamma \vdash \tau \leq \tau'$, *but where (1) all types in the derivation are restricted to normal and (2) with the following replacement rules for reflexivity and universal types, and a single rule for applying promotion to neutral types replacing the rules for variables, μ-types, application, and projection:*

$$\frac{}{\Gamma \vdash_n \tau \leq \tau}\,\textit{NS-Refl} \qquad \frac{\Gamma, \alpha{\leq}\tau_1 \vdash_n \tau_2 \leq \tau_2'}{\Gamma \vdash_n \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1.\tau_2'}\,\textit{NS-All}$$

$$\frac{\Gamma \vdash_n T[\mathrm{promote}_\Gamma(\tau)]^{\downarrow} \leq \tau'}{\Gamma \vdash_n T[\tau] \leq \tau'}\,\textit{NS-Neutr}$$

[11] Compagnoni called it *lub* instead.

Unlike in Compagnoni's definition, we integrate the rule for variables (and projection) into a single rule for neutral types, which reduces some repetition in the proofs, especially with the addition of projections.

Promotion maintains the following propoerties.

Lemma A.32 (Properties of Promotion). *If* $\text{promote}_\Gamma(\tau)$ *is defined, then:*
(1) If $\Gamma \vdash T[\tau] : \kappa$, *then* $\Gamma \vdash T[\text{promote}_\Gamma(\tau)] : \kappa$.
(2) $\Gamma \vdash T[\tau] \leq T[\text{promote}_\Gamma(\tau)]$.
(3) $\Gamma \vdash_n T[\tau] \leq T[\text{promote}_\Gamma(\tau)]^\downarrow$.

Proof. Each by induction on the structure of $T[\tau]$. □

A.7 Cut Elimination for Normal Subtyping

Normal subtyping has a general reflexivity rule NS-Refl, but it is possible to eliminate most of its uses with the following rewrite steps on derivations.

Definition A.11 (Reflexivity Elimination).

$$\frac{}{\Gamma \vdash_n \tau_1 \times \tau_2 \leq \tau_1 \times \tau_2}\,\textit{NS-Refl} \Longrightarrow \frac{\dfrac{}{\Gamma \vdash_n \tau_1 \leq \tau_1}\,\textit{NS-Refl} \qquad \dfrac{}{\Gamma \vdash_n \tau_2 \leq \tau_2}\,\textit{NS-Refl}}{\Gamma \vdash_n \tau_1 \times \tau_2 \leq \tau_1 \times \tau_2}\,\textit{NS-Tim}$$

$$\frac{}{\Gamma \vdash_n \tau_1 \to \tau_2 \leq \tau_1 \to \tau_2}\,\textit{NS-Refl} \Longrightarrow \frac{\dfrac{}{\Gamma \vdash_n \tau_1 \leq \tau_1}\,\textit{NS-Refl} \qquad \dfrac{}{\Gamma \vdash_n \tau_2 \leq \tau_2}\,\textit{NS-Refl}}{\Gamma \vdash_n \tau_1 \to \tau_2 \leq \tau_1 \to \tau_2}\,\textit{NS-Arr}$$

$$\frac{}{\Gamma \vdash_n \forall \alpha{\leq}\tau_1.\tau_2 \leq \forall \alpha{\leq}\tau_1.\tau_2}\,\textit{NS-Refl} \Longrightarrow \frac{\dfrac{}{\Gamma \vdash_n \tau_1 \leq \tau_1}\,\textit{NS-Refl} \qquad \dfrac{}{\Gamma, \alpha{\leq}\tau_1 \vdash_n \tau_2 \leq \tau_2}\,\textit{NS-Refl}}{\Gamma \vdash_n \forall \alpha{\leq}\tau_1.\tau_2 \leq \forall \alpha{\leq}\tau_1.\tau_2}\,\textit{NS-All}$$

$$\frac{}{\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \leq \langle \tau_1, \tau_2 \rangle}\,\textit{NS-Refl} \Longrightarrow \frac{\dfrac{}{\Gamma \vdash_n \tau_1 \leq \tau_1}\,\textit{NS-Refl} \qquad \dfrac{}{\Gamma \vdash_n \tau_2 \leq \tau_2}\,\textit{NS-Refl}}{\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \leq \langle \tau_1, \tau_2 \rangle}\,\textit{NS-Tup}$$

$$\frac{}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1.\tau_2 \leq \lambda \alpha{:}\kappa_1.\tau_2}\,\textit{NS-Refl} \Longrightarrow \frac{\dfrac{}{\Gamma, \alpha{\leq}\top_\kappa \vdash_n \tau_2 \leq \tau_2}\,\textit{NS-Refl}}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1.\tau_2 \leq \lambda \alpha{:}\kappa_1.\tau_2}\,\textit{NS-Lam}$$

Similarly, uses of transitivity can be eliminated by rewriting derivations according to the following set of rules.

Definition A.12 (Transitivity Elimination).

$$\frac{\dfrac{}{\Gamma \vdash_n \tau \leq \tau}\,\textit{NS-Refl} \qquad \dfrac{\vdots}{\Gamma \vdash_n \tau \leq \tau'}}{\Gamma \vdash_n \tau \leq \tau'}\,\textit{NS-Trans} \Longrightarrow \frac{\vdots}{\Gamma \vdash_n \tau \leq \tau'}$$

$$\frac{\dfrac{\vdots}{\Gamma \vdash_n \tau \leq \tau'} \qquad \dfrac{}{\Gamma \vdash_n \tau' \leq \tau}\,\textit{NS-Refl}}{\Gamma \vdash_n \tau \leq \tau'}\,\textit{NS-Trans} \Longrightarrow \frac{\vdots}{\Gamma \vdash_n \tau \leq \tau'}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash_n \tau \le \tau'} \qquad \dfrac{}{\Gamma \vdash_n \tau' \le \top}\,\textit{NS-Top}}{\Gamma \vdash_n \tau \le \top}\,\textit{NS-Trans} \Longrightarrow \dfrac{}{\Gamma \vdash_n \tau \le \top}\,\textit{NS-Top}$$

$$\dfrac{\dfrac{\Gamma \vdash_n \Gamma(\alpha) \le \tau'}{\Gamma \vdash_n \alpha \le \tau'}\,\textit{NS-Var} \qquad \dfrac{\vdots}{\Gamma \vdash_n \tau' \le \tau''}}{\Gamma \vdash_n \alpha \le \tau''}\,\textit{NS-Trans} \Longrightarrow \dfrac{\dfrac{\Gamma \vdash_n \Gamma(\alpha) \le \tau' \quad \Gamma \vdash_n \tau' \le \tau''}{\Gamma \vdash_n \Gamma(\alpha) \le \tau''}\,\textit{NS-Trans}}{\Gamma \vdash_n \alpha \le \tau''}\,\textit{NS-Var}$$

$$\dfrac{\dfrac{\Gamma \vdash_n \tau_1 \le \tau_1' \qquad \Gamma \vdash_n \tau_2 \le \tau_2'}{\Gamma \vdash_n \tau_1 \times \tau_2 \le \tau_1' \times \tau_2'}\,\textit{NS-Tim} \qquad \dfrac{\Gamma \vdash_n \tau_1' \le \tau_1'' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \tau_1' \times \tau_2' \le \tau_1'' \times \tau_2''}\,\textit{NS-Tim}}{\Gamma \vdash_n \tau_1 \times \tau_2 \le \tau_1'' \times \tau_2''}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\dfrac{\Gamma \vdash_n \tau_1 \le \tau_1' \qquad \Gamma \vdash_n \tau_1' \le \tau_1''}{\Gamma \vdash_n \tau_1 \le \tau_1''}\,\textit{NS-Trans} \qquad \dfrac{\Gamma \vdash_n \tau_2 \le \tau_2' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \tau_2 \le \tau_2''}\,\textit{NS-Trans}}{\Gamma \vdash_n \tau_1 \times \tau_2 \le \tau_1'' \times \tau_2''}\,\textit{NS-Tim}$$

$$\dfrac{\dfrac{\Gamma \vdash_n \tau_1' \le \tau_1 \qquad \Gamma \vdash_n \tau_2 \le \tau_2'}{\Gamma \vdash_n \tau_1 \to \tau_2 \le \tau_1' \to \tau_2'}\,\textit{NS-Arr} \qquad \dfrac{\Gamma \vdash_n \tau_1'' \le \tau_1' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \tau_1' \to \tau_2' \le \tau_1'' \to \tau_2''}\,\textit{NS-Arr}}{\Gamma \vdash_n \tau_1 \to \tau_2 \le \tau_1'' \to \tau_2''}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\dfrac{\Gamma \vdash_n \tau_1'' \le \tau_1' \qquad \Gamma \vdash_n \tau_1' \le \tau_1}{\Gamma \vdash_n \tau_1'' \le \tau_1}\,\textit{NS-Trans} \qquad \dfrac{\Gamma \vdash_n \tau_2 \le \tau_2' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \tau_2 \le \tau_2''}\,\textit{NS-Trans}}{\Gamma \vdash_n \tau_1 \to \tau_2 \le \tau_1'' \to \tau_2''}\,\textit{NS-Arr}$$

$$\dfrac{\dfrac{\Gamma, \alpha \le \tau_1 \vdash_n \tau_2 \le \tau_2'}{\Gamma \vdash_n \forall \alpha \le \tau_1 . \tau_2 \le \forall \alpha \le \tau_1 . \tau_2'}\,\textit{NS-All} \qquad \dfrac{\Gamma, \alpha \le \tau_1 \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \forall \alpha \le \tau_1 . \tau_2' \le \forall \alpha \le \tau_1 . \tau_2''}\,\textit{NS-All}}{\Gamma \vdash_n \forall \alpha \le \tau_1 . \tau_2 \le \forall \alpha \le \tau_1 . \tau_2''}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\Gamma, \alpha \le \tau_1 \vdash_n \tau_2 \le \tau_2' \qquad \Gamma, \alpha \le \tau_1 \vdash_n \tau_2' \le \tau_2''}{\Gamma, \alpha \le \tau_1 \vdash_n \tau_2' \le \tau_2''}\,\textit{NS-Trans}}{\Gamma \vdash_n \forall \alpha \le \tau_1 . \tau_2 \le \forall \alpha \le \tau_1 . \tau_2''}\,\textit{NS-All}$$

$$\dfrac{\dfrac{\Gamma \vdash_n \tau_1 \le \tau_1' \qquad \Gamma \vdash_n \tau_2 \le \tau_2'}{\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \le \langle \tau_1', \tau_2' \rangle}\,\textit{NS-Tup} \qquad \dfrac{\Gamma \vdash_n \tau_1' \le \tau_1'' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \langle \tau_1', \tau_2' \rangle \le \langle \tau_1'', \tau_2'' \rangle}\,\textit{NS-Tup}}{\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \le \langle \tau_1'', \tau_2'' \rangle}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\dfrac{\Gamma \vdash_n \tau_1 \le \tau_1' \qquad \Gamma \vdash_n \tau_1' \le \tau_1''}{\Gamma \vdash_n \tau_1 \le \tau_1''}\,\textit{NS-Trans} \qquad \dfrac{\Gamma \vdash_n \tau_2 \le \tau_2' \qquad \Gamma \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \tau_2 \le \tau_2''}\,\textit{NS-Trans}}{\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \le \langle \tau_1'', \tau_2'' \rangle}\,\textit{NS-Tup}$$

$$\dfrac{\dfrac{\Gamma, \alpha \le \top_{\kappa_1} \vdash_n \tau_2 \le \tau_2'}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1 . \tau_2 \le \lambda \alpha{:}\kappa_1 . \tau_2'}\,\textit{NS-Lam} \qquad \dfrac{\Gamma, \alpha \le \top_{\kappa_1} \vdash_n \tau_2' \le \tau_2''}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1 . \tau_2' \le \lambda \alpha{:}\kappa_1 . \tau_2''}\,\textit{NS-Lam}}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1 . \tau_2 \le \lambda \alpha{:}\kappa_1 . \tau_2''}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\Gamma, \alpha \le \top_{\kappa_1} \vdash_n \tau_2 \le \tau_2' \qquad \Gamma, \alpha \le \top_{\kappa_1} \vdash_n \tau_2' \le \tau_2''}{\Gamma, \alpha \le \top_{\kappa_1} \vdash_n \tau_2' \le \tau_2''}\,\textit{NS-Trans}}{\Gamma \vdash_n \lambda \alpha{:}\kappa_1 . \tau_2 \le \lambda \alpha{:}\kappa_1 . \tau_2''}\,\textit{NS-Lam}$$

$$\dfrac{\dfrac{\Gamma \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow \le \tau'}{\Gamma \vdash_n T[\tau] \le \tau'}\,\textit{NS-Neutr} \qquad \dfrac{\vdots}{\Gamma \vdash_n \tau' \le \tau''}}{\Gamma \vdash_n T[\tau] \le \tau''}\,\textit{NS-Trans}$$

$$\Longrightarrow \dfrac{\dfrac{\Gamma \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow \le \tau' \qquad \Gamma \vdash_n \tau' \le \tau''}{\Gamma \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow \le \tau''}\,\textit{NS-Trans}}{\Gamma \vdash_n T[\tau] \le \tau''}\,\textit{NS-Neutr}$$

Having defined these rules, we can further define the notion of derivations that are *normalised* according to the rewrite rules.

Definition A.13 (Normalised Derivation).

(1) A derivation is reflexivity-normalised *if no Reflexivity Elimination rule is applicable.*
(2) A derivation is transitivity-normalised *if no Transitivity Elimination rule is applicable.*
(3) A derivation is normalised *if it is both reflexivity-normalised and transitivity-normalised.*

Iterative application of the rewrite rules indeed terminates.

Lemma A.33 (Termination of Reflexivity and Transitivity Elimination).

(1) A normal subtyping derivation with only one application of rule NS-Refl has a reflexivity-normalised form.
(2) A normal subtyping derivation with only one application of rule NS-Trans has a transitivity-normalised form.

Proof.

(1) By induction on the size of the types. Each rewrite rule produces uses of NS-Refl on strictly smaller terms.
(2) By induction on the size of the derivation. Each rewrite rule either eliminates the use of NS-Trans or produces uses with strictly smaller derivations as premises. □

Corollary A.34 (Existence of Normalised Derivations). *If there exists a derivation for* $\Gamma \vdash_n \tau \le \tau'$*, then there also exists a normalised derivation for it.*

Finally, by iteratively eliminating reflexivity and transitivity from subderivations, we can construct cut-free derivations.

Lemma A.35 (Last Rule of Normalised Derivations).

(1) A normalised derivation whose last rule is NS-Refl is a proof for a neutral type.
(2) A normalised derivation does not have NS-Trans as its last rule.

Proof.

(1) By inspection of the rules for Reflexivity Elimination, any other case constitutes a redex.
(2) By induction on the size of the derivation. Each rewrite rule either eliminates the use of NS-Trans or produces uses with strictly smaller derivations as premises. □

Corollary A.36 (Cut-Free Derivations). *If there exists a derivation for* $\Gamma \vdash_n \tau \le \tau'$*, then there also exists a derivation for it with no application of NS-Trans and in which NS-Refl is only applied to neutral types.*

A.8 Equivalence of Normal Subtyping

We are now in shape to prove that normal subtyping is in fact equivalent to ordinary subtyping for normal types. We had to defer that proof, because proving the completeness direction depends on the existence of cut-free derivations.

We start with soundness.

Proposition A.37 (Soundness of Normal Subtyping). *If* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa'$ *and* $\Gamma \vdash_n \tau \leq \tau'$, *then* $\Gamma \vdash \tau \leq \tau'$.

Proof. By induction on the derivation of $\vdash_n$. The only interesting cases are the new rules:

- Case $\Gamma \vdash_n \tau \leq \tau$
  - by reflexivity of $\equiv$, $\tau \equiv \tau$
  - by the reflexivity rule, $\Gamma \vdash \tau \leq \tau$
- Case $\Gamma \vdash_n \forall \alpha{\leq}\tau_1.\tau_2 \leq \forall \alpha{\leq}\tau_1.\tau_2'$
  - by inversion, $\Gamma, \alpha{\leq}\tau_1 \vdash_n \tau_2 \leq \tau_2'$
  - by reflexivity of $\equiv$, $\tau_1 \equiv \tau_1$
  - by the original rule for $\forall$, $\Gamma \vdash \forall \alpha{\leq}\tau_1.\tau_2 \leq \forall \alpha{\leq}\tau_1.\tau_2'$
- Case $\Gamma \vdash_n T[\tau] \leq \tau'$
  - by inversion, $\Gamma \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow \leq \tau'$
  - by Lemma A.32 (1), $\Gamma \vdash T[\text{promote}_\Gamma(\tau)] : \kappa$
  - by Preservation of Kinding, $\Gamma \vdash T[\text{promote}_\Gamma(\tau)]^\downarrow : \kappa$
  - by Lemma A.32 (2), $\Gamma \vdash T[\tau] \leq T[\text{promote}_\Gamma(\tau)]$
  - by the definition of $\equiv$, $T[\text{promote}_\Gamma(\tau)] \equiv T[\text{promote}_\Gamma(\tau)]^\downarrow$
  - by the reflexivity rule, $\Gamma \vdash T[\text{promote}_\Gamma(\tau)] \leq T[\text{promote}_\Gamma(\tau)]^\downarrow$
  - by the transitivity rule, $\Gamma \vdash T[\tau] \leq T[\text{promote}_\Gamma(\tau)]^\downarrow$
  - by induction, $\Gamma \vdash T[\text{promote}_\Gamma(\tau)]^\downarrow \leq \tau'$
  - by the transitivity rule, $\Gamma \vdash T[\tau] \leq \tau'$ □

To prove completeness of normal subtyping, we first need to show that ordinary subtyping can be limited to normal types as well.

Lemma A.38 (Completeness of Normal Environments).

(1) If $\vdash \Gamma$ ok, *then* $\vdash \Gamma^\downarrow$ ok.

(2) If $\Gamma \vdash \tau : \kappa$, *then* $\Gamma^\downarrow \vdash \tau : \kappa$.

(3) If $\Gamma \vdash \tau \leq \tau'$, *then* $\Gamma^\downarrow \vdash \tau \leq \tau'$.

Proof. By simultaneous induction on derivations. □

Lemma A.39 (Soundness of Normal Environments). *If* $\vdash \Gamma_1, \Gamma_2$ ok, *then:*

(1) If $\Gamma_1^\downarrow, \Gamma_2 \vdash \tau : \kappa$, *then* $\Gamma_1, \Gamma_2 \vdash \tau : \kappa$.

(2) If $\Gamma_1^\downarrow, \Gamma_2 \vdash \tau \leq \tau'$, *then* $\Gamma_1, \Gamma_2 \vdash \tau \leq \tau'$.

Proof. By simultaneous induction on derivations. □

Lemma A.40 (Equivalence of Normal Environments). *Let* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$. *Then* $\Gamma \vdash \tau \leq \tau'$ *if and only if* $\Gamma^\downarrow \vdash \tau^\downarrow \leq \tau'^\downarrow$.

Proof. Follows from Soundness and Completeness of Normal Environments and Preservation of Kinding. □

Lemma A.41 (Normal Subtyping of Higher-Order Top). *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \vdash \tau \leq \top_\kappa$.

Proof. By induction on the derivation. □

Lemma A.42 (Kinding under Elimination Contexts). *Let* $\Gamma \vdash T[\tau] : \kappa$. *Then:*

(1) If $\tau = \langle \tau_1, \tau_2 \rangle$, *then* $T = T'[\_.i]$.

(2) If $\tau = \lambda \alpha{:}\kappa_1.\tau_2$, *then* $T = T'[\_\ \tau_1]$.

Proof. Each by induction on the derivation. □

Lemma A.43 (Kind Preservation for Normal Subtyping). *If* $\Gamma \vdash_n \tau \leq \tau'$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa'$, *then* $\kappa = \kappa'$.

Proof. By induction on the derivation of $\leq$. □

Lemma A.44 (Normal Subtyping Under Type Elimination). *Let* $\Gamma \vdash_n \tau \leq \tau'$.

(1) If $\Gamma \vdash \tau : \kappa_1 \times \kappa_2$ *and* $\Gamma \vdash \tau' : \kappa_1 \times \kappa_2$, *then* $\Gamma \vdash_n (\tau.i)^\downarrow \leq (\tau'.i)^\downarrow$.
(2) If $\Gamma \vdash \tau : \kappa_1 \to \kappa_2$ *and* $\Gamma \vdash \tau' : \kappa_1 \to \kappa_2$ *and* $\Gamma \vdash \tau_1 : \kappa_1$, *then* $\Gamma \vdash_n (\tau\ \tau_1)^\downarrow \leq (\tau'\ \tau_1)^\downarrow$.
(3) If $\Gamma \vdash T[\tau] : \kappa$ *and* $\Gamma \vdash T[\tau'] : \kappa$, *then* $\Gamma \vdash_n T[\tau]^\downarrow \leq T[\tau']^\downarrow$.
Proof.

(1) By induction on the derivation of $\leq$.
- Case NS-Refl
  - immediate
- Case NS-Trans
  - by inversion, $\Gamma \vdash_n \tau \leq \tau''$ and $\Gamma \vdash \tau'' \leq \tau'$ and $\Gamma \vdash \tau'' : \kappa''$
  - by Kind Preservation for Normal Subtyping, $\kappa'' = \kappa_1 \times \kappa_2$
  - by induction, $\Gamma \vdash_n (\tau.i)^\downarrow \leq (\tau''.i)^\downarrow$ and $\Gamma \vdash (\tau''.i)^\downarrow \leq (\tau'.i)^\downarrow$
  - by kinding rule, $\Gamma \vdash \tau''.i : \kappa_i$
  - by rule NS-Trans, $\Gamma \vdash_n (\tau.i)^\downarrow \leq (\tau'.i)^\downarrow$
- Case $\tau = \langle \tau_1, \tau_2 \rangle$ and $\tau' = \langle \tau_1', \tau_2' \rangle$ (NS-Tup)
  - by inversion, $\Gamma \vdash_n \tau_1 \leq \tau_1'$ and $\Gamma \vdash_n \leq \tau_2 \leq \tau_2'$
  - by reduction rules, $(\tau.i)^\downarrow = \tau_i$ and $(\tau'.i)^\downarrow = \tau_i'$
- Case $\tau = T[\tau_1]$ (NS-Neutr)
  - by inversion, $\Gamma \vdash_n T[\text{promote}_\Gamma(\tau_1)]^\downarrow \leq \tau'$
  - by Properties of Promotion, $\Gamma \vdash_n T[\text{promote}_\Gamma(\tau_1)] : \kappa_1 \times \kappa_2$
  - by induction, $\Gamma \vdash_n (T[\text{promote}_\Gamma(\tau_1)].i)^\downarrow \leq (\tau'.i)^\downarrow$
  - by assumption, $T[\tau_1]$ normal and neutral
  - hence, $T[\tau_1].i$ normal and $T[\tau_1].i = (T[\tau_1].i)^\downarrow$
  - by rule NS-Neutr, $\Gamma \vdash_n (T[\tau_1].i)^\downarrow \leq (\tau'.i)^\downarrow$

(2) Analogously.

(3) By induction on the structure of T, using parts (1) and (2). □

Lemma A.45 (Substitution for Normal Subtyping). *Let* $\Gamma = \Gamma_1, \alpha \leq \top_\kappa, \Gamma_2$. *If* $\Gamma \vdash_n \tau \leq \tau'$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\Gamma_1 \vdash \tau_\alpha : \kappa_\alpha$, *then* $\Gamma_1, \Gamma_2[\tau_\alpha/\alpha]^\downarrow \vdash_n \tau[\tau_\alpha/\alpha]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$.

Proof. By induction on the derivation of $\leq$. The interesting case is rule NS-Neutr:

- Case $\Gamma \vdash_n T[\tau] \leq \tau'$ (NS-Neutr)
  - let $\Gamma = \Gamma_1, \alpha \leq \top_{\kappa_\alpha}, \Gamma_2$ and $\Gamma' = \Gamma_1, \Gamma_2[\tau_\alpha/\alpha]^\downarrow$
  - by inversion, $\Gamma \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow \leq \tau'$
  - by induction, $\Gamma' \vdash_n T[\text{promote}_\Gamma(\tau)]^\downarrow[\tau_\alpha/\alpha]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
  - let $T' = T[\tau_\alpha/\alpha]^\downarrow$
  - by definition of normal form,
    $T[\text{promote}_\Gamma(\tau)]^\downarrow[\tau_\alpha/\alpha]^\downarrow = T[\text{promote}_\Gamma(\tau)][\tau_\alpha/\alpha]^\downarrow = T'[\text{promote}_\Gamma(\tau)[\tau_\alpha/\alpha]]^\downarrow$
  - following from the partial definition of promote_Γ, there are 4 subcases to consider:
    - subcase $\tau = \alpha$
      - then $\text{promote}_\Gamma(\alpha) = \top_{\kappa_\alpha}$
      - by definition, $\alpha \notin \text{fv}(\top_{\kappa_\alpha})$

· hence, $\Gamma' \vdash_n T'[\top_{\kappa_\alpha}]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by Preservation of Kinding, $\Gamma \vdash \tau_\alpha^\downarrow : \kappa_\alpha$
· by Normal Subtyping of Higher-Order Top, $\Gamma_1 \vdash_n \tau_\alpha^\downarrow \leq \top_{\kappa_\alpha}$
· by Weakening, $\Gamma' \vdash \tau_\alpha : \kappa_\alpha$ and $\Gamma' \vdash_n \tau_\alpha^\downarrow \leq \top_{\kappa_\alpha}$
· by Properties of Higher-order Top, $\Gamma' \vdash \top_{\kappa_\alpha} : \kappa_\alpha$
· by Normal Subtyping Under Type Elimination, $\Gamma' \vdash_n T'[\tau_\alpha^\downarrow]^\downarrow \leq T'[\top_{\kappa_\alpha}]^\downarrow$
· by Confluence, $T'[\tau_\alpha^\downarrow]^\downarrow = T'[\tau_\alpha]^\downarrow = T[\alpha][\tau_\alpha/\alpha]^\downarrow$
· by transitivity rule, $\Gamma' \vdash_n T[\alpha][\tau_\alpha/\alpha]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$

∗ subcase $\tau = \alpha_1 \in \mathrm{dom}(\Gamma_1)$ (with $\alpha_1 \neq \alpha$)
· by Free Variables, $\alpha_1 \notin \mathrm{fv}(\Gamma_1(\alpha_1))$
· hence, $\Gamma' \vdash_n T'[\Gamma_1(\alpha_1)]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by Properties of Promotion, $\Gamma' \vdash_n T'[\alpha_1] \leq T'[\Gamma_1(\alpha_1)]^\downarrow$
· by transitivity rule, $\Gamma' \vdash_n T'[\alpha_1] \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by Confluence, $T'[\alpha_1] = T[\alpha_1][\tau_\alpha/\alpha]^\downarrow$

∗ subcase $\tau = \alpha_2 \in \mathrm{dom}(\Gamma_2)$ (with $\alpha_2 \neq \alpha$)
· let $\Gamma_2' = \Gamma_2[\tau_\alpha/\alpha]$
· by Properties of Promotion, $\Gamma' \vdash_n T'[\alpha_2] \leq T'[\Gamma_2'(\alpha_2)]^\downarrow$
· by definition, $\Gamma_2'(\alpha_2) = \Gamma_2(\alpha_2)[\tau_\alpha/\alpha]$
· by transitivity rule, $\Gamma' \vdash_n T'[\alpha_2] \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by Confluence, $T'[\alpha_2] = T[\alpha_2][\tau_\alpha/\alpha]^\downarrow$

∗ subcase $\tau = \mu\alpha_1 \leq \tau_1.\tau_2$
· then $\mathrm{promote}_\Gamma(\tau) = \tau_1$
· by definition, $\mathrm{promote}_{\Gamma'}(\tau) = \tau_1 = \mathrm{promote}_\Gamma(\tau)$
· also by definition, $\mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha]) = \tau_1[\tau_\alpha/\alpha]$
· hence, $\Gamma' \vdash_n T'[\mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha])]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by Confluence, $T'[\mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha])^\downarrow]^\downarrow$
· because τ is a μ-type, $\mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha])^\downarrow = \mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha]^\downarrow)$
· hence, $\Gamma' \vdash_n T'[\mathrm{promote}_{\Gamma'}(\tau[\tau_\alpha/\alpha]^\downarrow)]^\downarrow \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· because τ is a μ-type, $T'[\tau[\tau_\alpha/\alpha]^\downarrow]$ is normal
· by rule NS-Neutr, $\Gamma' \vdash_n T'[\tau[\tau_\alpha/\alpha]^\downarrow] \leq \tau'[\tau_\alpha/\alpha]^\downarrow$
· by definition, $T'[\tau[\tau_\alpha/\alpha]^\downarrow] = T[\tau][\tau_\alpha/\alpha]^\downarrow$ □

Lemma A.46 (Normal Subtyping under Elimination). *If* $\Gamma \vdash_n \tau \leq \tau'$ *and* $\Gamma \vdash T[\tau] : \kappa$ *and* $\Gamma \vdash T[\tau'] : \kappa'$, *then* $\Gamma \vdash_n T[\tau]^\downarrow \leq T[\tau']^\downarrow$.

Proof. By induction on the normalised derivation of $\leq$. Interesting cases are:

- Case NS-Refl, $\Gamma \vdash_n \tau \leq \tau$ with $\tau' = \tau$
  - then $T[\tau] = T[\tau']$
  - hence, $T[\tau]^\downarrow = T[\tau']^\downarrow$
  - by NS-Refl, $\Gamma \vdash_n T[\tau]^\downarrow \leq T[\tau']^\downarrow$
- Case NS-Neutr, $\Gamma \vdash_n T'[\tau] \leq \tau'$
  - by inversion, $\Gamma \vdash_n T'[\mathrm{promote}_\Gamma(\tau)]^\downarrow \leq \tau'$
  - by Properties of Promotion, $\Gamma T'[\mathrm{promote}_\Gamma(\tau)] : \kappa$
  - by Preservation of Kinding, $\Gamma T'[\mathrm{promote}_\Gamma(\tau)]^\downarrow : \kappa$
  - by induction, $\Gamma \vdash_n T[T'[\mathrm{promote}_\Gamma(\tau)]^\downarrow]^\downarrow \leq T[\tau']^\downarrow$
  - by Uniqueness of Normal Forms, $T[T'[\mathrm{promote}_\Gamma(\tau)]^\downarrow]^\downarrow = T[T'[\mathrm{promote}_\Gamma(\tau)]]^\downarrow$
  - let $T'' = T[T']$
  - by NS-Neutr, $\Gamma \vdash_n T[T'[\tau]]^\downarrow \leq T[\tau']^\downarrow$

- Case $\Gamma \vdash_n \langle \tau_1, \tau_2 \rangle \leq \langle \tau_1', \tau_2' \rangle$
  - by inversion, $\Gamma \vdash_n \tau_1 \leq \tau_1'$ and $\Gamma \vdash_n \tau_2 \leq \tau_2'$
  - by Kinding under Elimination Contexts, $T = T'[\_.i]$
  - then $T[\langle \tau_1, \tau_2 \rangle]^\downarrow = T'[\tau_i]^\downarrow$ and $T[\langle \tau_1', \tau_2' \rangle]^\downarrow = T'[\tau_i']^\downarrow$
  - by inversion of kinding, $\Gamma \vdash \tau_i : \kappa_i$ and $\Gamma \vdash \tau_i' : \kappa_i'$
  - by induction, $\Gamma \vdash_n T'[\tau_i]^\downarrow \leq T'[\tau_i']^\downarrow$
- Case $\Gamma \vdash_n \lambda\alpha{:}\kappa_1.\tau_2 \leq \lambda\alpha{:}\kappa_1.\tau_2'$
  - by inversion, $\Gamma, \alpha{\leq}\top_{\kappa_1} \vdash_n \tau_2 \leq \tau_2'$
  - by Kinding under Elimination Contexts, $T = T'[\_\ \tau_1]$
  - then $T[\lambda\alpha{:}\kappa_1.\tau_2]^\downarrow = T'[\tau_2[\tau_1/\alpha]]^\downarrow$ and $T[\lambda\alpha{:}\kappa_1.\tau_2]^\downarrow = T'[\tau_2'[\tau_1/\alpha]]^\downarrow$
  - by inversion of kinding, $\Gamma, \alpha{\leq}\top_{\kappa_1} \vdash \tau_2 : \kappa_2$ and $\Gamma, \alpha{\leq}\top_{\kappa_1} \vdash \tau_2' : \kappa_2'$
  - by induction, $\Gamma \vdash_n T'[\tau_i]^\downarrow \leq T'[\tau_i']^\downarrow$ □

Proposition A.47 (Completeness of Normal Subtyping). *If* $\Gamma \vdash \tau \leq \tau'$, *then* $\Gamma^\downarrow \vdash_n \tau^\downarrow \leq \tau'^\downarrow$.

Proof. By induction on the derivation. The only interesting cases are the eliminated rules:

- Case $\Gamma \vdash \tau \leq \tau'$ where $\tau \equiv \tau'$
  - by Strong Normalisation, $\tau^\downarrow = \tau'^\downarrow$
  - by rule NS-Refl, $\Gamma^\downarrow \vdash_n \tau^\downarrow \leq \tau'^\downarrow$
- Case $\Gamma \vdash \forall\alpha{\leq}\tau_1.\tau_2 \leq \forall\alpha{\leq}\tau_1'.\tau_2'$
  - by inversion, $\tau_1 \equiv \tau_1'$ and $\Gamma, \alpha{\leq}\tau_1 \vdash \tau_2 \leq \tau_2'$
  - by Strong Normalisation, $\tau_1^\downarrow = \tau_1'^\downarrow$
  - by induction, $\Gamma^\downarrow, \alpha{\leq}\tau_1^\downarrow \vdash_n \tau_2^\downarrow \leq \tau_2'^\downarrow$
  - by rule NS-All, $\Gamma \vdash_n (\forall\alpha{\leq}\tau_1.\tau_2)^\downarrow \leq (\forall\alpha{\leq}\tau_1.\tau_2')^\downarrow$
- Case $\Gamma \vdash \alpha \leq \Gamma(\alpha)$
  - by definition, $\text{promote}_{\Gamma^\downarrow}(\alpha) = \Gamma^\downarrow(\alpha)$
  - by definition of normal form, $\alpha^\downarrow = \alpha$
  - by definition, $\Gamma^\downarrow(\alpha) = \Gamma(\alpha)^\downarrow$
  - by rule NS-Refl, $\Gamma^\downarrow \vdash_n \Gamma(\alpha)^\downarrow \leq \Gamma(\alpha)^\downarrow$
  - let $T = \_$
  - by definition of normal form, $\Gamma^\downarrow(\alpha)^\downarrow = (\Gamma(\alpha)^\downarrow)^\downarrow = \Gamma(\alpha)^\downarrow$
  - by rule NS-Neutr, $\Gamma^\downarrow \vdash_n \alpha^\downarrow \leq \Gamma(\alpha)^\downarrow$
- Case $\Gamma \vdash \mu\alpha{\leq}\tau_1.\tau_2 \leq \tau_1$
  - by definition, $\text{promote}_{\Gamma^\downarrow}((\mu\alpha{\leq}\tau_1.\tau_2)^\downarrow) = \tau_1^\downarrow$
  - by rule NS-Refl, $\Gamma^\downarrow \vdash_n \tau_1^\downarrow \leq \tau_1^\downarrow$
  - let $T = \_$
  - by rule NS-Neutr, $\Gamma^\downarrow \vdash_n (\mu\alpha{\leq}\tau_1.\tau_2)^\downarrow \leq \tau_1^\downarrow$
- Case $\Gamma \vdash \tau.i \leq \tau'.i$
  - by inversion, $\Gamma \vdash \tau \leq \tau'$
  - by induction, $\Gamma \vdash_n \tau^\downarrow \leq \tau'^\downarrow$
  - by Normal Subtyping under Elimination, $\Gamma \vdash_n (\tau^\downarrow.i)^\downarrow \leq (\tau'^\downarrow.i)^\downarrow$
  - by Confluence, $(\tau^\downarrow.i)^\downarrow = (\tau.i)^\downarrow$ and $(\tau'^\downarrow.i)^\downarrow = (\tau'.i)^\downarrow$
- Case $\Gamma \vdash \tau\ \tau_1 \leq \tau'\ \tau_1$
  - by inversion, $\Gamma \vdash \tau \leq \tau'$
  - by induction, $\Gamma \vdash_n \tau^\downarrow \leq \tau'^\downarrow$
  - by Normal Subtyping under Elimination, $\Gamma \vdash_n (\tau^\downarrow\ \tau_1)^\downarrow \leq (\tau'^\downarrow\ \tau_1)^\downarrow$
  - by Confluence, $(\tau^\downarrow\ \tau_1)^\downarrow = (\tau\ \tau_1)^\downarrow$ and $(\tau'^\downarrow\ \tau_1)^\downarrow = (\tau'\ \tau_1)^\downarrow$

□

Theorem A.48 (Equivalence of Normal Subtyping). *Let* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$. *Then* $\Gamma \vdash \tau \leq \tau'$ *if and only if* $\Gamma^{\downarrow} \vdash_n \tau^{\downarrow} \leq \tau'^{\downarrow}$.

Proof. Follows from Soundness and Completeness of Normal Subtyping and Lemma A.40. □

Corollary A.49 (Reflexivity and Transitivity of Normal Subtyping).

(1) If $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$ *and* $\tau \equiv \tau'$, *then* $\Gamma \vdash_n \tau \leq \tau'$.
(2) If $\Gamma \vdash_n \tau \leq \tau' : \kappa$ *and* $\Gamma \vdash_n \tau' \leq \tau'' : \kappa$, *then* $\Gamma \vdash_n \tau \leq \tau''$.

Proof. By converting to the respective ordinary subtyping derivation, applying reflexivity or transitivity rules, and converting back. □

A.9 Algorithmic Subtyping

If we only consider cut-free derivations of normal subtyping, then all rules are syntax-directed. We can hence interpret the normal subtyping relation under such a restricted rule set as an algorithm.[12]

Definition A.14 (Algorithmic Subtyping). *The relation* $\Gamma \vdash_a \tau \leq \tau'$ *is the same as* $\Gamma \vdash_n \tau \leq \tau'$, *but with the transitivity rule removed.*

Theorem A.50 (Equivalence of Algorithmic Subtyping). *Let* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$. *Then:*

(1) $\Gamma \vdash_a \tau \leq \tau'$ *if and only if* $\Gamma \vdash_n \tau \leq \tau'$.
(2) $\Gamma^{\downarrow} \vdash_a \tau^{\downarrow} \leq \tau'^{\downarrow}$ *if and only if* $\Gamma \vdash \tau \leq \tau'$.

Proof.

(1) One direction follows from Cut-Free Derivations, the other is immediate from the definition.
(2) By part (1) and Equivalence of Normal Subtyping. □

It is not difficult to see that the resulting rule set corresponds directly to the functional presentation given in Figure 3.

Corollary A.51 (Reflexivity and Transitivity of Algorithmic Subtyping).

(1) If $\Gamma^{\downarrow} \vdash \tau^{\downarrow} : \kappa$ *and* $\Gamma^{\downarrow} \vdash \tau'^{\downarrow} : \kappa$ *and* $\tau^{\downarrow} \equiv \tau'^{\downarrow}$, *then* $\Gamma^{\downarrow} \vdash_a \tau^{\downarrow} \leq \tau'^{\downarrow}$.
(2) If $\Gamma^{\downarrow} \vdash_a \tau^{\downarrow} \leq \tau'^{\downarrow} : \kappa$ *and* $\Gamma^{\downarrow} \vdash_a \tau'^{\downarrow} \leq \tau''^{\downarrow} : \kappa$, *then* $\Gamma^{\downarrow} \vdash_a \tau^{\downarrow} \leq \tau''^{\downarrow}$.

Proof. Follows from the definition and Reflexivity and Transitivity of Normal Subtyping. □

A.10 Unrolling

To prove type soundness, we need a few properties of higher-order unrolling. In particular, the key characteristic to show is that unrolling preserves subtyping. But for that, we first need a couple of simple lemmas about type elimination contexts.

Lemma A.52 (Properties of Type Elimination).

(1) If $\Gamma \vdash T[\tau] : \kappa'$, *then* $\Gamma \vdash \tau : \kappa$.
(2) If $\Gamma \vdash T[\tau] : \kappa'$ *and* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \tau' : \kappa$, *then* $\Gamma \vdash T[\tau'] : \kappa'$.
(3) If $\Gamma \vdash \tau \leq \tau'$, *then* $\Gamma \vdash T[\tau] \leq T[\tau']$.

Proof. Each by induction on the (first) derivation. □

Lemma A.53 (Possible Type Eliminations). *Let* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash T[\tau] : \kappa'$.

(1) If $\kappa = \Omega$, *then* $T = \_$ *and* $\kappa' = \kappa$.
(2) If $\kappa = \kappa_1 \times \kappa_2$, *then either* $T = \_$ *and* $\kappa' = \kappa$, *or* $T = T'[\_.i]$ *for some* i.
(3) If $\kappa = \kappa_1 \to \kappa_2$, *then either* $T = \_$ *and* $\kappa' = \kappa$, *or* $T = T'[\_\ \tau_1]$ *with* $\Gamma \vdash \tau_1 : \kappa_1$.

[12] Unlike Compagnoni, we do not restrict reflexivity here, since that is actually irrelevant to the algorithm — in fact, an implementation that already canonicalises types will benefit from a more general reflexivity rule.

Proof. By induction on the second derivation. □

With that, we can prove that unrolling has the needed properties. One technical complication is that equivalences and subtyping of higher-order unrollings (the last two parts of the following lemma) generally only hold inside a suitable type elimination context (T' below) that observes them at ground kind Ω. These relations would only hold directly at higher kinds if we included η-equivalence rules for types into the calculus (in which case the definition of unrolling could also be simplified and skip the η-expansion).

Lemma A.54 (Properties of Unrolling).

(1) If $\Gamma \vdash \tau : \kappa$ *and* $\mathrm{unroll}_\kappa(\tau)$ *is defined, then* $\mathrm{unroll}_\kappa(\tau[\tau_1/\alpha]) \equiv \mathrm{unroll}_\kappa(\tau)[\tau_1/\alpha]$.
(2) If $\Gamma \vdash \tau : \kappa$ *and* $\mathrm{unroll}_\kappa(\tau)$ *is defined, then* $\Gamma \vdash \mathrm{unroll}_\kappa(\tau) : \kappa$.
(3) If $\Gamma \vdash T[\tau] : \kappa$ *and* $\mathrm{unroll}_\kappa(T[\tau])$ *is defined, then* $\mathrm{unroll}_\kappa(T[\tau]) \equiv T[\mathrm{unroll}_{\kappa'}(\tau)]$.
(4) If $\Gamma \vdash T[\mu\alpha{\le}\tau_1.\tau_2] : \kappa$ *and* $\Gamma \vdash T'[T[\mu\alpha{\le}\tau_1.\tau_2]] : \Omega$,
then $T'[\mathrm{unroll}_\kappa(T[\mu\alpha{\le}\tau_1.\tau_2])] \equiv T'[T[\tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha]]]$.
(5) If $\Gamma \vdash \tau \le \tau' : \kappa$ *and* $\mathrm{unroll}_\kappa(\tau)$ *and* $\mathrm{unroll}_\kappa(\tau')$ *are defined and* $\Gamma \vdash T'[\tau] : \Omega$,
then $\Gamma \vdash T'[\mathrm{unroll}_\Omega(\tau)] \le T'[\mathrm{unroll}_\kappa(\tau')]$.

Proof.

(1) By induction on κ.
(2) By induction on κ.
- Case $\kappa = \Omega$ and $\tau = \top$
  - immediate
- Case $\kappa = \Omega$ and $\tau = T[\mu\alpha{\le}\tau_1.\tau_2]$
  - by Properties of Type Elimination (1), $\Gamma \vdash \mu\alpha{\le}\tau_1.\tau_2 : \kappa'$
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa'$ and $\Gamma, \alpha{\le}\tau_1 \vdash \tau_2 : \kappa'$
  - by rule S-Rec-Sup, $\Gamma \vdash \mu\alpha{\le}\tau_1.\tau_2 \le \tau_1$
  - by Substitution, $\Gamma \vdash \tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha] : \kappa'$
  - by Properties of Type Elimination (2), $\Gamma \vdash T[\tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha]] : \kappa$
- Case $\kappa = \kappa_1 \times \kappa_2$
  - by kinding rule for projection, $\Gamma \vdash \tau.1 : \kappa_1$ and $\Gamma \vdash \tau.2 : \kappa_2$
  - by induction, $\Gamma \vdash \mathrm{unroll}_{\kappa_1}(\tau.1) : \kappa_1$ and $\Gamma \vdash \mathrm{unroll}_{\kappa_2}(\tau.2) : \kappa_2$
  - by kinding rule for tuples, $\Gamma \vdash \langle \mathrm{unroll}_{\kappa_1}(\tau.1), \mathrm{unroll}_{\kappa_2}(\tau.2)\rangle : \kappa_1 \times \kappa_2$
- Case $\kappa = \kappa_1 \to \kappa_2$
  - by Regularity, $\vdash \Gamma$ ok
  - by Properties of Higher-order Top, $\Gamma \vdash \top_{\kappa_1} : \kappa_1$
  - by environment rule, $\vdash \Gamma, \alpha{\le}\top_{\kappa_1}$ ok
  - by kinding rule for variables, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \alpha : \kappa_1$
  - by Weakening, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \tau : \kappa_1 \to \kappa_2$
  - by kinding rule for application, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \tau\,\alpha : \kappa_2$
  - by induction, $\Gamma, \alpha{\le}\top_{\kappa_2} \vdash \mathrm{unroll}_{\kappa_2}(\tau\,\alpha) : \kappa_2$
  - by kinding rule for abstraction, $\Gamma \vdash \lambda\alpha{:}\kappa_1.\,\mathrm{unroll}_{\kappa_2}(\tau\,\alpha) : \kappa_1 \to \kappa_2$
(3) By induction on the structure of T.
- Case $T = \_$
  - then $\mathrm{unroll}_\kappa(T[\tau]) = \mathrm{unroll}_\kappa(\tau) = T[\mathrm{unroll}_\kappa(\tau)]$
  - by reflexivity, $\mathrm{unroll}_\kappa(T[\tau]) \equiv T[\mathrm{unroll}_\kappa(\tau)]$
- Case $T = T'.i$
  - by inversion of kinding, $\Gamma \vdash T'[\tau] : \kappa_1 \times \kappa_2$ and $\kappa = \kappa_i$
  - by induction, $\mathrm{unroll}_{\kappa_1\times\kappa_2}(T'[\tau]) \equiv T'[\mathrm{unroll}_{\kappa_1\times\kappa_2}(\tau)]$
  - by definition, $\mathrm{unroll}_{\kappa_1\times\kappa_2}(T'[\tau]) = \langle \mathrm{unroll}_{\kappa_1}(T'[\tau].1), \mathrm{unroll}_{\kappa_2}(T'[\tau].2)\rangle$

  - by reduction, $\langle \mathrm{unroll}_{\kappa_1}(T'[\tau].1), \mathrm{unroll}_{\kappa_2}(T'[\tau].2)\rangle.i \equiv \mathrm{unroll}_{\kappa_i}(T'[\tau].i)$
  - by symmetry and transitivity, $\mathrm{unroll}_{\kappa_i}(T'[\tau].i) \equiv T'[\mathrm{unroll}_{\kappa_1\times\kappa_2}(\tau)].i$
- Case $T = T'\ \tau_1$
  - by inversion of kinding, $\Gamma \vdash T'[\tau] : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_1 : \kappa_1$ and $\kappa = \kappa_2$
  - by induction, $\mathrm{unroll}_{\kappa_1\to\kappa_2}(T'[\tau]) \equiv T'[\mathrm{unroll}_{\kappa_1\to\kappa_2}(\tau)]$
  - by definition, $\mathrm{unroll}_{\kappa_1\to\kappa_2}(T'[\tau]) = \lambda\alpha{:}\kappa_1.\, \mathrm{unroll}_{\kappa_2}(T'[\tau]\ \alpha)$
  - by reduction, $(\lambda\alpha{:}\kappa_1.\, \mathrm{unroll}_{\kappa_2}(T'[\tau]\ \alpha))\ \tau_1 \equiv \mathrm{unroll}_{\kappa_2}(T'[\tau]\ \tau_1)$
  - by symmetry and transitivity, $\mathrm{unroll}_{\kappa_2}(T'[\tau]\ \tau_1) \equiv T'[\mathrm{unroll}_{\kappa_1\to\kappa_2}(\tau)]\ \tau_1$

(4) By induction on κ. Let $\tau = \mu\alpha{\le}\tau_1.\tau_2$.
- Case $\kappa = \Omega$
  - by definition, $\mathrm{unroll}_\Omega(T[\tau]) = T[\tau_2[\tau/\alpha]]$
  - by reflexivity, $T'[\mathrm{unroll}_\Omega(T[\tau])] \equiv T'[T[\tau_2[\tau/\alpha]]]$
- Case $\kappa = \kappa_1 \times \kappa_2$
  - by Possible Type Eliminations, $T' = T''[\_.i]$
  - by kinding rule for projection, $\Gamma \vdash T[\tau].1 : \kappa_1$ and $\Gamma \vdash T[\tau].2 : \kappa_2$
  - by induction, $T''[\mathrm{unroll}_{\kappa_i}(T[\tau].i)] \equiv T''[T[\tau_2[\tau/\alpha]].i]$
  - by definition, $\mathrm{unroll}_{\kappa_1\times\kappa_2}(T[\tau]) = \langle \mathrm{unroll}_{\kappa_1}(T[\tau].1), \mathrm{unroll}_{\kappa_2}(T[\tau].2)\rangle$
  - hence by reduction, $T'[\mathrm{unroll}_{\kappa_1\times\kappa_2}(T[\tau])] \equiv T''[\mathrm{unroll}_{\kappa_i}(T[\tau].i)]$
  - by transitivity, $T'[\mathrm{unroll}_{\kappa_1\times\kappa_2}(T[\tau])] \equiv T''[T[\tau_2[\tau/\alpha]].i] = T'[T[\tau_2[\tau/\alpha]]]$
- Case $\kappa = \kappa_1 \to \kappa_2$
  - by Possible Type Eliminations, $T' = T''[\_\ \tau_a]$ and $\Gamma \vdash \tau_a : \kappa_1$
  - by kinding rule for application, $\Gamma \vdash T[\tau]\ \tau_a : \kappa_2$
  - by induction, $T''[\mathrm{unroll}_\kappa(T[\tau]\ \tau_a)] \equiv T''[T[\tau_2[\tau/\alpha]]\ \tau_a]$
  - by definition, $\mathrm{unroll}_{\kappa_1\to\kappa_2}(T[\tau]) = \lambda\alpha'{:}\kappa_1.\, \mathrm{unroll}_{\kappa_2}(T[\tau]\ \alpha')$
  - hence by reduction, $T'[\mathrm{unroll}_{\kappa_1\to\kappa_2}(T[\tau])] \equiv T''[\mathrm{unroll}_{\kappa_2}(T[\tau]\ \tau_a)]$
  - by transitivity, $T'[\mathrm{unroll}_{\kappa_1\to\kappa_2}(T[\tau])] \equiv T''[T[\tau_2[\tau/\alpha]]\ \tau_a] = T'[T[\tau_2[\tau/\alpha]]]$

(5) By induction on the subtyping derivation. The only possible cases are the following.
- Case reflexivity rule
  - by inversion, $\tau \equiv \tau'$
  - by definition, $\mathrm{unroll}_\kappa(\tau) = \mathrm{unroll}_\kappa(\tau')$
  - by reflexivity, $\Gamma \vdash \mathrm{unroll}_\kappa(\tau) \le \mathrm{unroll}_\kappa(\tau')$
- Case transitivity rule
  - by inversion, $\Gamma \vdash \tau \le \tau''$ and $\Gamma \vdash \tau'' \le \tau'$ and $\Gamma \vdash \tau'' : \kappa$
  - by induction, $\Gamma \vdash \mathrm{unroll}_\kappa(\tau) \le \mathrm{unroll}_\kappa(\tau'')$ and $\Gamma \vdash \mathrm{unroll}_\kappa(\tau'') \le \mathrm{unroll}_\kappa(\tau')$
  - by transitivity, $\Gamma \vdash \mathrm{unroll}_\kappa(\tau) \le \mathrm{unroll}_\kappa(\tau')$
- Case $\Gamma \vdash \tau \le \top$
  - by inversion of kinding, $\kappa = \Omega$
  - by Possible Type Eliminations, $T' = \_$
  - by definition, $\mathrm{unroll}_\Omega(\top) = \top$
  - by subtyping rule for top, $\Gamma \vdash \mathrm{unroll}_\Omega(\tau) \le \top$
- Case $\Gamma \vdash \mu\alpha{\le}\tau_1.\tau_2 \le \tau_1$ (S-Rec-Sup)
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa$ and $\Gamma, \alpha{\le}\tau_1 \vdash \tau_2 : \kappa$ and $\Gamma, \alpha{\le}\tau_1 \vdash \tau_2 \le \mathrm{unroll}_\kappa(\tau_1)$
  - by Substitution, $\Gamma \vdash \tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha] \le \mathrm{unroll}_\kappa(\tau_1)$
  - by Properties of Type Elimination (3), $\Gamma \vdash T'[\tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha]] \le T'[\mathrm{unroll}_\kappa(\tau_1)]$
  - by part (4), $T'[\mathrm{unroll}_\kappa(\mu\alpha{\le}\tau_1.\tau_2)] \equiv T'[\tau_2[\mu\alpha{\le}\tau_1.\tau_2/\alpha]]$
  - by reflexivity and transitivity rules, $\Gamma \vdash T'[\mathrm{unroll}_\kappa(\mu\alpha{\le}\tau_1.\tau_2)] \le T'[\mathrm{unroll}_\kappa(\tau_1)]$
- Case $\Gamma \vdash \langle\tau_1, \tau_2\rangle \le \langle\tau_1', \tau_2'\rangle$
  - let $\tau = \langle\tau_1, \tau_2\rangle$ and $\tau' = \langle\tau_1', \tau_2'\rangle$

  - by inversion, $\Gamma \vdash \tau_1 \le \tau_1'$ and $\Gamma \vdash \tau_2 \le \tau_2'$
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2$ and $\Gamma \vdash \tau_1' : \kappa_1$ and $\Gamma \vdash \tau_2' : \kappa_2$ and $\kappa = \kappa_1 \times \kappa_2$
  - by Possible Type Eliminations, $T' = T''[\_.i]$
  - let $T_1 = T''[\_.1]$ and $T_2 = T''[\_.2]$
  - by reduction, $T'[\tau] \equiv T_1[\tau_1]$ and $T'[\tau'] \equiv T_1[\tau_1']$ and $T'[\tau] \equiv T_2[\tau_2]$ and $T'[\tau'] \equiv T_2[\tau_2']$
  - by induction, $\Gamma \vdash T''[\mathrm{unroll}_{\kappa_1}(\tau_1)] \le T''[\mathrm{unroll}_{\kappa_1}(\tau_1')]$ and $\Gamma \vdash T''[\mathrm{unroll}_{\kappa_2}(\tau_2)] \le T''[\mathrm{unroll}_{\kappa_2}(\tau_2')]$
  - by definition, $\mathrm{unroll}_\kappa(\tau) = \langle \mathrm{unroll}_{\kappa_1}(\tau.1), \mathrm{unroll}_{\kappa_2}(\tau.2)\rangle$ and $\mathrm{unroll}_\kappa(\tau') = \langle \mathrm{unroll}_{\kappa_1}(\tau'.1), \mathrm{unroll}_{\kappa_2}(\tau'.2)\rangle$
  - by reduction, $T'[\mathrm{unroll}_\kappa(\tau)] \equiv T''[\mathrm{unroll}_{\kappa_i}(\tau.i)] \equiv T''[\mathrm{unroll}_{\kappa_i}(\tau_i)]$ and $T'[\mathrm{unroll}_\kappa(\tau')] \equiv T''[\mathrm{unroll}_{\kappa_i}(\tau'.i)] \equiv T''[\mathrm{unroll}_{\kappa_i}(\tau_i')]$
  - by transitivity rule, $\Gamma \vdash T'[\mathrm{unroll}_\kappa(\tau)] \le T'[\mathrm{unroll}_\kappa(\tau')]$
- Case $\Gamma \vdash \tau.i \le \tau'.i$
  - by inversion, $\Gamma \vdash \tau \le \tau'$
  - by inversion of kinding, $\Gamma \vdash \tau : \kappa_1 \times \kappa_2$ and $\Gamma \vdash \tau' : \kappa_1' \times \kappa_2'$ and $\kappa = \kappa_i = \kappa_i'$
  - let $T'' = T'[\_.i]$
  - by induction, $\Gamma \vdash T''[\mathrm{unroll}_{\kappa_1\times\kappa_2}(\tau)] \le T''[\mathrm{unroll}_{\kappa_1\times\kappa_2}(\tau')]$
- Case $\Gamma \vdash \lambda\alpha{:}\kappa_1.\tau_2 \le \lambda\alpha{:}\kappa_1.\tau_2'$
  - let $\tau = \lambda\alpha{:}\kappa_1.\tau_2$ and $\tau' = \lambda\alpha{:}\kappa_1.\tau_2'$
  - by inversion, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \tau_2 \le \tau_2'$
  - by inversion of kinding, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \tau_2 : \kappa_2$ and $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash \tau_2' : \kappa_2$ and $\kappa = \kappa_1 \to \kappa_2$
  - by Possible Type Eliminations, $T' = T''[\_\ \tau_1]$ and $\Gamma \vdash \tau_1 : \kappa_1$
  - by reduction, $T'[\tau] \equiv T''[\tau_2[\tau_1/\alpha]]$ and $T'[\tau'] \equiv T''[\tau_2'[\tau_1/\alpha]]$
  - by Weakening, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash T''[\tau] : \kappa_2$ and $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash T''[\tau'] : \kappa_2$
  - by Properties of Type Elimination, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash T''[\tau_2] : \kappa_2$ and $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash T''[\tau_2'] : \kappa_2$
  - by induction, $\Gamma, \alpha{\le}\top_{\kappa_1} \vdash T''[\mathrm{unroll}_{\kappa_2}(\tau_2)] \le T''[\mathrm{unroll}_{\kappa_2}(\tau_2')]$
  - by Substitution, $\Gamma \vdash T''[\mathrm{unroll}_{\kappa_2}(\tau_2[\tau_1/\alpha])] \le T''[\mathrm{unroll}_{\kappa_2}(\tau_2'[\tau_1/\alpha])]$
  - by definition, $\mathrm{unroll}_\kappa(\tau) = \lambda\alpha{:}\kappa_1.\,\mathrm{unroll}_{\kappa_2}(\tau\ \alpha)$ and $\mathrm{unroll}_\kappa(\tau') = \lambda\alpha{:}\kappa_1.\,\mathrm{unroll}_{\kappa_2}(\tau'\ \alpha)$
  - by reduction, $T'[\mathrm{unroll}_\kappa(\tau)] = T''[\mathrm{unroll}_\kappa(\tau)\ \tau_1] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau\ \alpha)[\tau_1/\alpha]]$ and $T'[\mathrm{unroll}_\kappa(\tau')] = T''[\mathrm{unroll}_\kappa(\tau')\ \tau_1] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau'\ \alpha)[\tau_1/\alpha]]$
  - by part (1), $T''[\mathrm{unroll}_{\kappa_2}(\tau\ \alpha)[\tau_1/\alpha]] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau\ \tau_1)]$ and $T''[\mathrm{unroll}_{\kappa_2}(\tau'\ \alpha)[\tau_1/\alpha]] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau'\ \tau_1)]$
  - by reduction, $T''[\mathrm{unroll}_{\kappa_2}(\tau\ \tau_1)] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau_2[\tau_1/\alpha])]$ and $T''[\mathrm{unroll}_{\kappa_2}(\tau'\ \tau_1)] \equiv T''[\mathrm{unroll}_{\kappa_2}(\tau_2'[\tau_1/\alpha])]$
  - by transitivity rule, $\Gamma \vdash T'[\mathrm{unroll}_\kappa(\tau)] \le T'[\mathrm{unroll}_\kappa(\tau')]$
- Case $\Gamma \vdash \tau\ \tau_1 \le \tau'\ \tau_1$
  - by inversion, $\Gamma \vdash \tau \le \tau'$
  - by inversion of kinding, $\Gamma \vdash \tau : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau' : \kappa_1 \times \kappa_2$ and $\Gamma \vdash \tau_1 : \kappa_1$ and $\kappa = \kappa_2$
  - let $T'' = T'[\_\ \tau_1]$
  - by induction, $\Gamma \vdash T''[\mathrm{unroll}_{\kappa_1\to\kappa_2}(\tau)] \le T''[\mathrm{unroll}_{\kappa_1\to\kappa_2}(\tau')]$ □

A.11 Preservation

We are now equipped to prove type soundness the usual way, starting with preservation.

Lemma A.55 (Inversion of Typing). *Let* $\Gamma \vdash e : \tau$.

(1) If $e = x$, *then* $\Gamma \vdash \Gamma(x) \le \tau$.

(2) If $e = \langle v_1, v_2 \rangle$, *then* $\Gamma \vdash v_1 : \tau_1$ *and* $\Gamma \vdash v_2 : \tau_2$ *and* $\Gamma \vdash \tau_1 \times \tau_2 \leq \tau$.
(3) If $e = e'.i$, *then* $\Gamma \vdash e' : \tau_1 \times \tau_2$ *and* $\Gamma \vdash \tau_i \leq \tau$.
(4) If $e = \lambda x{:}\tau_1.e_2$, *then* $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \tau_1 \to \tau_2 \leq \tau$.
(5) If $e = e_1\, e_2$, *then* $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \tau_1 \leq \tau$.
(6) If $e = \lambda\alpha{\leq}\tau_1.e_2$, *then* $\Gamma, \alpha{\leq}\tau_1 \vdash e_2 : \tau_2$ *and* $\Gamma \vdash \forall\alpha{\leq}\tau_1.\tau_2 \leq \tau$.
(7) If $e = e_1\, \tau_2$, *then* $\Gamma \vdash e_1 : \forall\alpha{\leq}\tau_2'.\tau_1$ *and* $\Gamma \vdash \tau_2 \leq \tau_2' : \kappa$ *and* $\Gamma \vdash \tau_1[\tau_2/\alpha] \leq \tau$.
(8) If $e = \mathsf{roll}_{\tau'}\, v$, *then* $\tau' = T[\mu\alpha{\leq}\tau_1.\tau_2]$ *and* $\Gamma \vdash \tau' : \Omega$ *and* $\Gamma \vdash v : \mathrm{unroll}_\Omega(\tau')$ *and* $\Gamma \vdash \tau' \leq \tau$.
(9) If $e = \mathsf{unroll}\ e'$, *then* $\Gamma \vdash e' : T[\mu\alpha{\leq}\tau_1.\tau_2]$ *and* $\Gamma \vdash \mathrm{unroll}_\Omega(T[\mu\alpha{\leq}\tau_1.\tau_2]) \leq \tau$.

Proof. By induction on the derivation. □

Theorem A.56 (Preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \hookrightarrow e'$, *then* $\Gamma \vdash e' : \tau$.

Proof. By induction on the derivation of $\hookrightarrow$. Most cases are standard using Inversion of Typing, the only interesting new one is elimination of recursive types:

- Case $\mathsf{unroll}\ (\mathsf{roll}_{\tau'}\, v) \hookrightarrow v$
  - by Inversion of Typing, $\Gamma \vdash \mathsf{roll}_{\tau'}\, v : T[\mu\alpha{\leq}\tau_1.\tau_2]$ and $\Gamma \vdash \mathrm{unroll}_\Omega(T[\mu\alpha{\leq}\tau_1.\tau_2]) \leq \tau$
  - by Regularity, $\Gamma \vdash T[\mu\alpha{\leq}\tau_1.\tau_2] : \Omega$
  - by Inversion of Typing, $\tau' = T'[\mu\alpha{\leq}\tau_1'.\tau_2']$ and $\Gamma \vdash v : \mathrm{unroll}_\Omega(\tau')$ and $\Gamma \vdash \tau' : \Omega$ and $\Gamma \vdash \tau' \leq T[\mu\alpha{\leq}\tau_1.\tau_2]$
  - by Properties of Unrolling, $\Gamma \vdash \mathrm{unroll}_\Omega(\tau') \leq \mathrm{unroll}_\Omega(T[\mu\alpha{\leq}\tau_1.\tau_2])$
  - by transitivity rule for subtyping, $\Gamma \vdash \mathrm{unroll}_\Omega(\tau') \leq \tau$
  - by subsumption rule, $\Gamma \vdash v : \tau$ □

A.12 Progress

Finally, we prove progress. This is almost entirely standard as well.

Lemma A.57 (Canonical Subtypes). *Let* $\cdot \vdash \tau \leq \tau' : \kappa$.

(1) If $\tau' \equiv \tau_1' \times \tau_2'$, *then* $\tau \equiv \tau_1 \times \tau_2$ *such that* $\cdot \vdash \tau_1 \leq \tau_1'$ *and* $\cdot \vdash \tau_2 \leq \tau_2'$.
(2) If $\tau' \equiv \tau_1' \to \tau_2'$, *then* $\tau \equiv \tau_1 \to \tau_2$ *such that* $\cdot \vdash \tau_1' \leq \tau_1$ *and* $\cdot \vdash \tau_2 \leq \tau_2'$.
(3) If $\tau' \equiv \forall\alpha{\leq}\tau_1'.\tau_2'$, *then* $\tau \equiv \forall\alpha{\leq}\tau_1.\tau_2$ *such that* $\tau_1 \equiv \tau_1'$ *and* $\cdot \vdash \tau_2 \leq \tau_2'$.
(4) If $\tau' \equiv T'[\mu\alpha{\leq}\tau_1'.\tau_2']$, *then* $\tau \equiv T[\mu\alpha{\leq}\tau_1.\tau_2]$ *such that either* $\tau \equiv \tau'$ *or* $\vdash T[\tau_1] \leq T'[\mu\alpha{\leq}\tau_1'.\tau_2']$.
(5) If $\tau' \equiv \langle \tau_1', \tau_2' \rangle$, *then* $\tau \equiv \langle \tau_1, \tau_2 \rangle$ *such that* $\cdot \vdash \tau_1 \leq \tau_1'$ *and* $\cdot \vdash \tau_2 \leq \tau_2'$.
(6) If $\tau' \equiv \lambda\alpha{:}\kappa.\tau_2'$, *then* $\tau \equiv \lambda\alpha{:}\kappa.\tau_2$ *such that* $\cdot, \alpha{\leq}\top_\kappa \vdash \tau_2 \leq \tau_2'$.

Proof. By Completeness of Algorithmic Subtyping we have $\cdot \vdash_a \tau^{\downarrow} \leq \tau'^{\downarrow} : \kappa$. Each result then follows from inversion and Soundness of Algorithmic Subtyping. □

Lemma A.58 (Canonical Values). *Let* $\cdot \vdash v : \tau$.

(1) If $\cdot \vdash \tau \leq \tau_1 \times \tau_2$, *then* $v = \langle v_1, v_2 \rangle$.
(2) If $\cdot \vdash \tau \leq \tau_1 \to \tau_2$, *then* $v = \lambda x{:}\tau_1'.e$.
(3) If $\cdot \vdash \tau \leq \forall\alpha{\leq}\tau_1.\tau_2$, *then* $v = \lambda\alpha{\leq}\tau_1'.e$.
(4) If $\cdot \vdash \tau \leq T[\mu\alpha{\leq}\tau_1.\tau_2]$, *then* $v = \mathsf{roll}_{\tau'}\, v'$.

Proof. By induction on the derivation, using Canonical Subtypes in the case of the subsumption rule and to exclude incompatible rules. □

Theorem A.59 (Progress). *If* $\cdot \vdash e : \tau$ *and* $e \neq v$ *for any* v, *then* $e \hookrightarrow e'$ *for some* e'.

Proof. By induction on the derivation. The only non-standard case is unrolling:

- Case $e = \mathsf{unroll}\ e_1$

- by Inversion of Typing, $\cdot \vdash e_1 : T[\mu\alpha{\leq}\tau_1.\tau_2]$
- subcase $e_1 = v$
  - by reflexivity rule, $\cdot \vdash T[\mu\alpha{\leq}\tau_1.\tau_2] \leq T[\mu\alpha{\leq}\tau_1.\tau_2]$
  - by Canonical Values, $v = \mathsf{roll}_{\tau'}\ v'$
  - by reduction rule, $\mathsf{unroll}\ (\mathsf{roll}_{\tau'}\ v') \hookrightarrow v'$
- subcase $e_1 \neq v$
  - by induction, $e_1 \hookrightarrow e_1'$
  - by reduction rule for contexts, $\mathsf{unroll}\ e_1 \hookrightarrow \mathsf{unroll}\ e_1'$ □

B META-THEORY OF FULL λ_{misu}

In this section, we show the modifications necessary to the statements and proofs in order to cover full λ_{misu}. We only show definitions, lemmas and proof cases that change. The only deeply affected is the lemma on Properties of Unrolling.

B.1 Regularity

Lemma B.1 (Subtyping of Higher-Order Top). *If* $\Gamma \vdash \tau : \kappa$, *then* $\Gamma \vdash \tau \leq \top_\kappa$.

Proof. By induction on the derivation. The modified case is:

- Case $\Gamma \vdash \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau' : \kappa$ with $\kappa = \kappa_1 \times \kappa_2$
  - let $\tau = \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau'$
  - by inversion, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma, \alpha_1{\leq}\tau_1 \vdash \tau_2 : \kappa_2$
  - by induction, $\Gamma \vdash \tau_1 \leq \top_{\kappa_1}$ and $\Gamma, \alpha_1{\leq}\tau_1 \vdash \tau_2 \leq \top_{\kappa_2}$
  - by kinding rule for projection, $\Gamma \vdash \tau.1 : \kappa_1$
  - by Substitution, $\Gamma \vdash \tau_2[\tau.1/\alpha_1] \leq \top_{\kappa_2}$
  - by kinding rule for tuples, $\Gamma \vdash \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle : \kappa_1 \times \kappa_2$
  - by definition, $\top_\kappa = \langle\top_{\kappa_1}, \top_{\kappa_2}\rangle$
  - by subtyping rule for tuples, $\Gamma \vdash \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle \leq \langle\top_{\kappa_1}, \top_{\kappa_2}\rangle$
  - by subtyping rule for recursive types, $\Gamma \vdash \tau \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle$
  - by subtyping transitivity rule, $\Gamma \vdash \tau \leq \top_\kappa$ □

B.2 Type Reduction

Definition B.1 (Neutral Type). *A* neutral *type is one of the form* $T[\alpha]$ *or* $T[\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau']$.

Definition B.2 (Parallel Reduction).

$$\frac{\tau_1 \hookrightarrow\!\!\!\!\rightarrow \tau_1' \qquad \tau_2 \hookrightarrow\!\!\!\!\rightarrow \tau_2' \qquad \tau_3 \hookrightarrow\!\!\!\!\rightarrow \tau_3'}{\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau_3 \hookrightarrow\!\!\!\!\rightarrow \mu\langle\alpha_1{\leq}\tau_1', \alpha_2{\leq}\tau_2'\rangle.\tau_3'}$$

B.3 Pre-Kinding

Definition B.3 (Pre-Kinding). *The relation* $\Gamma \vdash' \tau : \kappa$ *is the same as* $\Gamma \vdash \tau : \kappa$, *but with the following replacement rule for* μ*-types:*

$$\frac{\Gamma \vdash' \tau_1 : \kappa_1 \qquad \Gamma, \alpha_1{\leq}\tau_2 \vdash' \tau_2 : \kappa_2 \qquad \Gamma, \alpha_1{\leq}\tau_2, \alpha_2{\leq}\tau_2 \vdash' \tau : \kappa_1 \times \kappa_2}{\Gamma \vdash' \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau : \kappa_1 \times \kappa_2}$$

B.4 Strong Normalisation

Lemma B.2 (Neutral Types).

(1) If $T[\alpha]$ *terminates, then* $T[\alpha] \in [\![\kappa]\!]$.
(2) If $T[\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau]$ *terminates, then* $T[\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau] \in [\![\kappa]\!]$.

Lemma B.3 (Completeness wrt. Kinding ("Fundamental Property")).
If $\Gamma \vdash' \tau : \kappa$ *and* $\gamma \in [\![\Gamma]\!]$, *then* $\gamma(\tau) \in [\![\kappa]\!]$.

Proof. By induction on the derivation.

- Case $\Gamma \vdash' \mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau : \kappa_1 \times \kappa_2$ (assuming α_1, α_2 are fresh)
  - by inversion, $\Gamma \vdash' \tau_1 : \kappa_1$ and $\Gamma, \alpha_1 \leq \tau_1 \vdash' \tau_2 : \kappa_2$ and $\Gamma, \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \vdash' \tau : \kappa_1 \times \kappa_2$
  - by induction, $\gamma(\tau_1) \in [\![\kappa]\!]$
  - let $\Gamma_1 = \Gamma, \alpha_1 \leq \tau_1$ and $\Gamma_2 = \Gamma_1, \alpha_2 \leq \tau_2$
  - let $\gamma_1 = \gamma \circ [\alpha_1/\alpha_1]$ and $\gamma_2 = \gamma_1 \circ [\alpha_2/\alpha_2]$
  - by Neutral Types, $\alpha_1 \in [\![\kappa_1]\!]$ and $\alpha_2 \in [\![\kappa_2]\!]$
  - by definition of $[\![\Gamma]\!]$, $\gamma_1 \in [\![\Gamma_1]\!]$ and $\gamma_2 \in [\![\Gamma_2]\!]$
  - by induction, $\gamma(\tau_2) = \gamma_1(\tau_2) \in [\![\kappa_1]\!]$ and $\gamma(\tau) = \gamma_1(\tau) = \gamma_2(\tau) \in [\![\kappa_1 \times \kappa_2]\!]$
  - by Main Lemma, $\gamma(\tau_1)$ and $\gamma(\tau_2)$ and $\gamma(\tau)$ terminate
  - hence, $\mu\langle\alpha_1 \leq \gamma(\tau_1), \alpha_2 \leq \gamma(\tau_2)\rangle.\gamma(\tau)$ terminates
  - by Neutral Types, $\gamma(\mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau) = \mu\langle\alpha_1 \leq \gamma(\tau_1), \alpha_2 \leq \gamma(\tau_2)\rangle.\gamma(\tau) \in [\![\kappa_1 \times \kappa_2]\!]$

□

B.5 Normal Subtyping

Definition B.4 (Promotion).

$$\begin{array}{lcl} \mathrm{promote}_\Gamma(\alpha) & = & \Gamma(\alpha) \\ \mathrm{promote}_\Gamma(\mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau) & = & \langle\tau_1, \tau_2[(\mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau).1/\alpha_1]\rangle \end{array}$$

B.6 Equivalence of Normal Subtyping

Lemma B.4 (Substitution for Normal Subtyping). *If* $\Gamma_1, \alpha \leq \top_\kappa, \Gamma_2 \vdash_n \tau \leq \tau'$ *and* $\Gamma \vdash \tau_a : \kappa$, *then* $\Gamma_1, \Gamma_2[\tau_a/\alpha]^\downarrow \vdash_n \tau[\tau_a/\alpha]^\downarrow \leq \tau'[\tau_a/\alpha]^\downarrow$.

Proof. As before. □

Proposition B.5 (Completeness of Normal Subtyping). *If* $\Gamma \vdash \tau \leq \tau'$, *then* $\Gamma^\downarrow \vdash_n \tau^\downarrow \leq \tau'^\downarrow$.

Proof. By induction on the derivation.

- Case $\Gamma \vdash \mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau_3 \leq \langle\tau_1, \tau_2[(\mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau_3).1/\alpha_1]\rangle$
  - let $\tau = \mu\langle\alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2\rangle.\tau_3$
  - by definition, $\mathrm{promote}_{\Gamma\downarrow}(\tau^\downarrow) = \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle^\downarrow$
  - by rule NS-Refl, $\Gamma^\downarrow \vdash_n \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle^\downarrow \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle^\downarrow$
  - let $T = \_$
  - by rule NS-Neutr, $\Gamma^\downarrow \vdash_n \tau^\downarrow \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle^\downarrow$

□

B.7 Unrolling

Note that the first and last part only hold for unrolling under an empty substitution parameter (as used in the typing rules for roll/unroll). Nevertheless, the auxiliary substitution parameter on the unroll function complicates the proof of the last part of the Unrolling lemma, in the case concerning μ-types.

Lemma B.6 (Properties of Unrolling). *Assume that for all* $\alpha \in \mathrm{dom}(\sigma)$, *(1) if* $\Gamma \vdash \Gamma(\alpha) : \kappa$, *then* $\Gamma \vdash \sigma(\alpha) : \kappa$; *and (2)* $\Gamma \vdash \sigma(\alpha) \leq \Gamma(\alpha)$. *Then:*

(1) If $\Gamma \vdash \tau : \kappa$ *and* $\mathrm{unroll}_\kappa(\tau)$ *is defined, then* $\mathrm{unroll}_\kappa(\tau[\tau_1/\alpha]) \equiv \mathrm{unroll}_\kappa(\tau)[\tau_1/\alpha]$.
(2) If $\Gamma \vdash \tau : \kappa$ *and* $\mathrm{unroll}^\sigma_\kappa(\tau)$ *is defined, then* $\Gamma \vdash \mathrm{unroll}^\sigma_\kappa(\tau) : \kappa$.
(3) If $\Gamma \vdash T[\tau] : \kappa$ *and* $\mathrm{unroll}^\sigma_\kappa(T[\tau])$ *is defined, then* $\mathrm{unroll}^\sigma_\kappa(T[\tau]) \equiv T[\mathrm{unroll}^\sigma_{\kappa'}(\tau)]$.

(4) If $\tau = \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau'$ *and* $\Gamma \vdash T[\tau] : \kappa$ *and* $\Gamma \vdash T'[T[\tau]] : \Omega$, *then* $T'[\mathrm{unroll}^{\sigma}_{\kappa}(T[\tau])] \equiv T'[T[\tau'[\tau.1/\alpha_1, \tau.2/\alpha_2]]]$.

(5) If $\Gamma \vdash \tau \leq \tau' : \kappa$ *and* $\mathrm{unroll}_{\kappa}(\tau)$ *and* $\mathrm{unroll}_{\kappa}(\tau')$ *are defined and* $\Gamma \vdash T'[\tau] : \Omega$, *then* $\Gamma \vdash T'[\mathrm{unroll}_{\Omega}(\tau)] \leq T'[\mathrm{unroll}_{\kappa}(\tau')]$.

Proof.

(1) By induction on κ.

(2) By induction on κ.
- Case $\kappa = \Omega$ and $\tau = T[\mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau_3]$
  - let $\tau' = \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau_3$.
  - by Properties of Type Elimination (1), $\Gamma \vdash \tau' : \kappa'$
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma, \alpha_1{\leq}\tau_1 \vdash \tau_2 : \kappa_2$ and $\Gamma, \alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2 \vdash \tau_3 : \kappa'$
  - by rule S-Rec-Sup, $\Gamma \vdash \tau' \leq \langle\tau_1, \tau_2[\tau'.1/\alpha_1]\rangle$
  - by subtyping rule for projection, $\Gamma \vdash \tau'.1 \leq \langle\tau_1, \tau_2[\tau'.1/\alpha_1]\rangle.1$ and $\Gamma \vdash \tau'.2 \leq \langle\tau_1, \tau_2[\tau'.1/\alpha_1]\rangle.2$
  - by type equivalence, $\langle\tau_1, \tau_2[\tau'.1/\alpha_1]\rangle.1 \equiv \tau_1$ and $\langle\tau_1, \tau_2[\tau'.1/\alpha_1]\rangle.2 \equiv \tau_2[\tau'.1/\alpha_1]$
  - by transitivity, $\Gamma \vdash \tau'.1 \leq \tau_1$ and $\Gamma \vdash \tau'.2 \leq \tau_2[\tau'.1/\alpha_1]$
  - by Substitution, $\Gamma, \alpha_2{\leq}\tau_2[\tau'.1/\alpha_1] \vdash \tau_3[\tau'.1/\alpha_1] : \kappa'$
  - by Substitution, $\Gamma \vdash \tau_3[\tau'.1/\alpha_1, \tau'.2/\alpha_2] : \kappa'$
  - by Properties of Type Elimination (2), $\Gamma \vdash T[\tau_3[\tau'.1/\alpha_1, \tau'.2/\alpha_2]] : \kappa$
- Case $\kappa = \Omega$ and $\tau = T[\alpha]$
  - by Properties of Type Elimination (1), $\Gamma \vdash \alpha : \kappa'$
  - by inversion of kinding, $\Gamma \vdash \Gamma(\alpha) : \kappa'$
  - by assumption, $\Gamma \vdash \sigma(\alpha) : \kappa'$
  - by Properties of Type Elimination (2), $\Gamma \vdash T[\sigma(\alpha)] : \kappa$

(3) By induction on the structure of T.

(4) By induction on κ. Let $\tau = \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau_3$.
- Case $\kappa = \Omega$
  - by definition, $\mathrm{unroll}^{\sigma}_{\Omega}(T[\tau]) = T[\tau_3[\tau.1/\alpha_1, \tau.2/\alpha_2]]$
  - by reflexivity, $T'[\mathrm{unroll}^{\sigma}_{\Omega}(T[\tau])] \equiv T'[T[\tau_3[\tau.1/\alpha_1, \tau.2/\alpha_2]]]$

(5) By induction on the subtyping derivation.
- Case $\Gamma \vdash \tau \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle$ with $\tau = \mu\langle\alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2\rangle.\tau_3$ (S-Rec-Tup)
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma, \alpha_1{\leq}\tau_1 \vdash \tau_2 : \kappa_2$ and $\Gamma, \alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2 \vdash \tau_3 : \kappa$ and $\Gamma, \alpha_1{\leq}\tau_1, \alpha_2{\leq}\tau_2 \vdash \tau_3 \leq \mathrm{unroll}^{\sigma}_{\kappa}(\langle\tau_1, \tau_2\rangle)$ with $\kappa = \kappa_1 \times \kappa_2$ and $\sigma = [\tau_3.1/\alpha_1]$
  - by subtyping rule for projection, $\Gamma \vdash \tau.1 \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle.1$ and $\Gamma \vdash \tau.2 \leq \langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle.2$
  - by kinding rule for projection, $\Gamma \vdash \tau.1 : \kappa_1$ and $\Gamma \vdash \tau.2 : \kappa_2$
  - by reduction, $\langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle.1 \equiv \tau_1$ and $\langle\tau_1, \tau_2[\tau.1/\alpha_1]\rangle.2 \equiv \tau_2[\tau.1/\alpha_1]$
  - by subtyping transitivity, $\Gamma \vdash \tau.1 \leq \tau_1$ and $\Gamma \vdash \tau.2 \leq \tau_2[\tau.1/\alpha_1]$
  - by inversion of kinding, $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2[\tau.1/\alpha_1] : \kappa_2$
  - by Substitution, $\Gamma \vdash \tau_3[\tau.1/\alpha_1, \tau.2/\alpha_2] \leq \mathrm{unroll}^{\sigma}_{\kappa}(\langle\tau_1, \tau_2\rangle)[\tau.1/\alpha_1, \tau.2/\alpha_2]$
  - Assertion: If $\Gamma, \alpha{\leq}\tau_1 \vdash \tau' : \kappa'$ and $\Gamma \vdash \tau'_3 \leq \mathrm{unroll}^{\sigma}_{\kappa'}(\tau')[\tau.1/\alpha_1, \tau.2/\alpha_2]$, then $\Gamma \vdash \tau'_3 \leq \mathrm{unroll}_{\kappa'}(\tau'[\tau.1/\alpha_1, \tau.2/\alpha_2])$.
    Proof by local induction on the normal form of τ'. The interesting case is the one for variables, which must be α_1 for the unrolling to be defined:
    - Case $\tau'^{\downarrow} = T[\alpha_1]$
      - by definition, $\mathrm{unroll}^{\sigma}_{\kappa'}(T(\alpha_1))[\tau.1/\alpha_1, \tau.2/\alpha_2] = T[\tau_3.1][\tau.1/\alpha_1, \tau.2/\alpha_2]$
      - by definition, $\mathrm{unroll}_{\kappa'}(T[\alpha_1][\tau.1/\alpha_1, \tau.2/\alpha_2]) = \mathrm{unroll}_{\kappa'}(T'[\tau.1])$ where $T'[\_] = T[\_][\tau.1/\alpha_1, \tau.2/\alpha_2]$
      - by definition, $\mathrm{unroll}_{\kappa'}(T'[\tau.1]) = T'[\tau_3.1][\tau.1/\alpha_1, \tau.2/\alpha_2]$

  - · by Free Variables, $\alpha_1, \alpha_2 \notin \mathrm{fv}(\tau)$
  - · hence, $T'[\_][\tau.1/\alpha_1, \tau.2/\alpha_2] = T'[\_] = T[\_][\tau.1/\alpha_1, \tau.2/\alpha_2]$
  - · hence, $\mathrm{unroll}_{\kappa'}(T[\alpha_1][\tau.1/\alpha_1, \tau.2/\alpha_2]) = T[\tau_3.1][\tau.1/\alpha_1, \tau.2/\alpha_2] = \mathrm{unroll}_{\kappa'}(T[\tau.1])$
  - by the assertion, $\Gamma \vdash \tau_3[\tau.1/\alpha_1, \tau.2/\alpha_2] \leq \mathrm{unroll}_{\kappa}(\langle \tau_1, \tau_2 \rangle[\tau.1/\alpha_1, \tau.2/\alpha_2])$
  - by Free Variables, $\alpha_1 \notin \mathrm{fv}(\tau_1)$ and $\alpha_2 \notin \mathrm{fv}(\tau_1)$ and $\alpha_2 \notin \mathrm{fv}(\tau_2)$
  - hence, $\langle \tau_1, \tau_2 \rangle[\tau.1/\alpha_1, \tau.2/\alpha_2] = \langle \tau_1, \tau_2[\tau.1/\alpha_1] \rangle$
  - by Properties of Type Elimination (3), $\Gamma \vdash T'[\tau_3[\tau.1/\alpha_1, \tau.2/\alpha_2]] \leq T'[\mathrm{unroll}_{\kappa}(\langle \tau_1, \tau_2[\tau.1/\alpha_1] \rangle)]$
- Case $\Gamma \vdash \alpha \leq \Gamma(\alpha)$
  - impossible, since unroll is undefined under an empty substitution □

B.8 Preservation

Lemma B.7 (Inversion of Typing). *Let* $\Gamma \vdash e : \tau$.

(1) If $e = \mathsf{roll}_{\tau'}\ v$, *then* $\tau' = T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]$ *and* $\Gamma \vdash \tau' : \Omega$ *and* $\Gamma \vdash v : \mathrm{unroll}_{\Omega}(\tau')$ *and* $\Gamma \vdash \tau' \leq \tau$.

(2) If $e = \mathsf{unroll}\ e'$, *then* $\Gamma \vdash e' : T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]$ *and* $\Gamma \vdash \mathrm{unroll}_{\Omega}(T[\mu\alpha \leq \tau_1.\tau_2]) \leq \tau$.

Proof. By induction on the derivation. □

Theorem B.8 (Preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \hookrightarrow e'$, *then* $\Gamma \vdash e' : \tau$.

Proof. By induction on the derivation of $\hookrightarrow$.

- Case $\mathsf{unroll}\ (\mathsf{roll}_{\tau'}\ v) \hookrightarrow v$
  - by Inversion of Typing, $\Gamma \vdash \mathsf{roll}_{\tau'}\ v : T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]$ and $\Gamma \vdash \mathrm{unroll}_{\Omega}(T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]) \leq \tau$
  - by Regularity, $\Gamma \vdash T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3] : \Omega$
  - by Inversion of Typing, $\tau' = T'[\mu\langle \alpha_1 \leq \tau_1', \alpha_2 \leq \tau_2' \rangle.\tau_3']$ and $\Gamma \vdash v : \mathrm{unroll}_{\Omega}(\tau')$ and $\Gamma \vdash \tau' : \Omega$ and $\Gamma \vdash \tau' \leq T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]$
  - by Properties of Unrolling, $\Gamma \vdash \mathrm{unroll}_{\Omega}(\tau') \leq \mathrm{unroll}_{\Omega}(T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3])$
  - by transitivity rule for subtyping, $\Gamma \vdash \mathrm{unroll}_{\Omega}(\tau') \leq \tau$
  - by subsumption rule, $\Gamma \vdash v : \tau$ □

B.9 Progress

Lemma B.9 (Canonical Subtypes). *Let* $\cdot \vdash \tau \leq \tau' : \kappa$.

(1)–(3) As before.

(4) If $\tau' \equiv T'[\mu\langle \alpha_1 \leq \tau_1', \alpha_2 \leq \tau_2' \rangle.\tau_3']$, *then* $\tau \equiv T[\mu\langle \alpha_1 \leq \tau_1', \alpha_2 \leq \tau_2' \rangle.\tau_3']$ *such that either* $\tau \equiv \tau'$ *or* $\vdash T[\tau_1] \leq T'[\mu\langle \alpha_1 \leq \tau_1', \alpha_2 \leq \tau_2' \rangle.\tau_3']$.

(5)–(6) As before.

Proof. As before. □

Lemma B.10 (Canonical Values). *Let* $\cdot \vdash v : \tau$.

(1)–(3) As before.

(4) If $\cdot \vdash \tau \leq T[\mu\langle \alpha_1 \leq \tau_1, \alpha_2 \leq \tau_2 \rangle.\tau_3]$, *then* $v = \mathsf{roll}_{\tau'}\ v'$.

Proof. As before. □

Theorem B.11 (Progress). *If* $\cdot \vdash e : \tau$ *and* $e \neq v$ *for any* v, *then* $e \hookrightarrow e'$ *for some* e'.

Proof. By induction on the derivation. The difference to the proof for the basic calculus is purely syntactic, given the updated Canonical Values lemma. □

Syntax

$$\begin{array}{lrcl} \text{(types)} & T & ::= & \mathtt{X} \mid N \\ \text{(nominal types)} & N & ::= & \mathtt{C}\langle\overline{T}\rangle \\ \text{(classes)} & C & ::= & \text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N\{\overline{T\ \mathtt{x}};\ \ldots\} \\ \text{(programs)} & P & ::= & \overline{C} \\ \text{(contexts)} & \Delta & ::= & P; \overline{\mathtt{X}\leq N} \end{array}$$

Type Well-formedness ($\Delta \vdash_{\text{FGJ}} T$ ok)

$$\frac{\mathtt{X}<:N \in \Delta}{\Delta \vdash_{\text{FGJ}} \mathtt{X}\ \text{ok}} \qquad \frac{}{\Delta \vdash_{\text{FGJ}} \mathtt{Object}\ \text{ok}} \qquad \frac{\text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\ldots\} \in \Delta \qquad \overline{\Delta \vdash_{\text{FGJ}} T\ \text{ok}} \qquad \overline{\Delta \vdash_{\text{FGJ}} T \leq N'[\overline{T/\mathtt{X}}]}}{\Delta \vdash_{\text{FGJ}} \mathtt{C}\langle\overline{T}\rangle\ \text{ok}}$$

Subtyping ($\Delta \vdash_{\text{FGJ}} T \leq T$)

$$\frac{}{\Delta \vdash_{\text{FGJ}} T \leq T} \qquad \frac{\Delta \vdash_{\text{FGJ}} T \leq T' \qquad \Delta \vdash_{\text{FGJ}} T' \leq T''}{\Delta \vdash_{\text{FGJ}} T \leq T''} \qquad \frac{\mathtt{X}<:N \in \Delta}{\Delta \vdash_{\text{FGJ}} \mathtt{X} \leq N} \qquad \frac{\text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\ldots\} \in \Delta}{\Delta \vdash_{\text{FGJ}} \mathtt{C}\langle\overline{T}\rangle \leq N'[\overline{T/\mathtt{X}}]}$$

Context Well-formedness ($\Delta \vdash_{\text{FGJ}}$ ok)

$$\frac{}{P; \epsilon \vdash_{\text{FGJ}} \text{ok}} \qquad \frac{\Delta \vdash_{\text{FGJ}} \text{ok} \qquad \Delta \vdash_{\text{FGJ}} N\ \text{ok}}{\Delta, \mathtt{X}<:N \vdash_{\text{FGJ}} \text{ok}}$$

Program Well-formedness ($P \vdash_{\text{FGJ}} P'$ ok)

$$\frac{}{P \vdash_{\text{FGJ}} \epsilon\ \text{ok}} \qquad \frac{P; C, P' \vdash_{\text{FGJ}} C\ \text{ok} \qquad P, C \vdash_{\text{FGJ}} P'\ \text{ok}}{P \vdash_{\text{FGJ}} C, P'\ \text{ok}}$$

Class Well-formedness ($P; P' \vdash_{\text{FGJ}} C$ ok)

$$\frac{\Delta = P; \overline{\mathtt{X}<:N} \qquad \Delta \vdash_{\text{FGJ}} \text{ok} \qquad \Delta \vdash_{\text{FGJ}} N'\ \text{ok} \qquad \Delta' = P, P'; \overline{\mathtt{X}<:N} \qquad \overline{\Delta' \vdash_{\text{FGJ}} T\ \text{ok}} \qquad \ldots}{P; P' \vdash_{\text{FGJ}} \text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\overline{T\ \mathtt{x}};\ \ldots\}\ \text{ok}}$$

Fig. 5. FGJ syntax, type well-formedness, and subtyping

C ENCODING FGJ

C.1 Defining FGJ

We consider a fragment of FGJ [Igarashi et al. 2001] without F-bounded quantification, since we omitted F-bounded quantification from the presentation in this paper. Figure 5 gives the definition and semantics of types and classes for this fragment, omitting method definitions. The main change relative to Igarashi et al. is that we do not allow a class or generic variable to appear in their own bounds. Furthermore, our rules make the program explicit as part of the context. And they formulate the constraint that inheritance must not be cyclic explicitly as well, by linearising the well-formedness for the classes appearing in it. To that end, the rules assume that all classes appear ordered topologically with respect to their subtyping hierarchy (which could be achieved by straightforward preprocessing). They then distinguish the "left" part P of the program, containing the classes *before* the current and therefor allowed in bounds or superclasses, and the "right" part P', which contains the classes *after* (and the class itself). The latter may still be referenced from the current class definition, but merely in its body, not its head.

We define $\Delta' \supseteq \Delta$ to mean that $\Delta' = \Delta, \overline{\mathtt{X}<:N}$, i.e., they record the same program, but Δ' may bind additional type variables.

We will need the following lemmas asserting properties of the FGJ type system.

Lemma C.1 (FGJ Substitution). *Let* $\overline{\Delta \vdash_{\text{FGJ}} T\ \text{ok}}$.

(1) If $\Delta \vdash_{\text{FGJ}} T'$ ok, *then* $\Delta \vdash_{\text{FGJ}} T'[\overline{T/\mathtt{X}}]$ ok

(2) If $\Delta \vdash_{\text{FGJ}} T_1 \leq T_2$, *then* $\Delta \vdash_{\text{FGJ}} T_1[\overline{T/\mathtt{X}}] \leq T_2[\overline{T/\mathtt{X}}]$.

Proof. By simultaneous induction on the derivation. □

Lemma C.2 (FGJ Supertype Validity). *If* $\Delta \vdash_{\text{FGJ}} T_1 \leq T_2$ *and* $\Delta \vdash_{\text{FGJ}} \text{ok}$ *and* $\Delta \vdash_{\text{FGJ}} T_1$ ok, *then* $\Delta \vdash_{\text{FGJ}} T_2$ ok.

Proof. By induction on the subtyping derivation and the well-formedness of Δ. □

Lemma C.3 (FGJ Program Well-formedness Inversion). *If* $P \vdash_{\text{FGJ}} P'$ ok *and* $C \in P'$, *then there exists a subderivation* $P, P'_1; P'_2 \vdash_{\text{FGJ}} C$ ok *with* $P'_1, P'_2 = P'$.

Proof. By induction on the derivation. □

C.2 Higher-arity Tuples

To encode FGJ programs with more than two classes, we need the n-ary version of the μ-operator at this point. Spelling out the its well-formedness rule it looks like this:

$$\frac{\Gamma' = \Gamma, \overline{\alpha \leq \tau} \qquad \Gamma' \vdash \langle \overline{\alpha} \rangle : \times(\overline{\kappa}) \qquad \Gamma' \vdash \tau' : \times(\overline{\kappa}) \qquad \Gamma' \vdash \tau' \leq \text{unroll}^{[\overline{\tau'.i/\alpha}^{\,i}]}_{\times(\overline{\kappa})}(\langle \overline{\tau} \rangle)}{\Gamma \vdash \mu \langle \overline{\alpha \leq \tau} \rangle . \tau' : \times(\overline{\kappa})}$$

This is assuming an encoding of n-ary type tuples $\langle \overline{\tau} \rangle$ into nested type pairs as follows. We likewise use an encoding for n-ary tuple types $\{\overline{\tau}\}$:

$$\begin{array}{llllllllllll} \times() & := & \Omega & & \langle\rangle & := & \top & & \tau.1 & := & \tau.1 & & \{\} & := & \top \\ \times(\kappa, \overline{\kappa}) & := & \kappa \times (\times(\overline{\kappa})) & & \langle \tau, \overline{\tau} \rangle & := & \langle \tau, \langle \overline{\tau} \rangle \rangle & & \tau.(n{+}1) & := & \tau.2.n & & \{\tau, \overline{\tau}\} & := & \tau \times \{\overline{\tau}\} \end{array}$$

(We are omitting the term-level encodings, since we are only interested in the type level.) The following typing rules are then derivable:

Lemma C.4 (Tuple Typing).

(1) If $\overline{\Gamma \vdash \tau : \kappa}$, *then* $\Gamma \vdash \langle \overline{\tau} \rangle : \times(\overline{\kappa})$.
(2) If $\Gamma \vdash \tau : \times(\overline{\kappa})$ *and* $i \leq |\overline{\kappa}|$, *then* $\Gamma \vdash \tau.i : \kappa_i$.
(3) If $\overline{\Gamma \vdash \tau : \Omega}$, *then* $\Gamma \vdash \{\overline{\tau}\} : \Omega$.
(4) If $\Gamma \vdash \{\overline{\tau}_1, \overline{\tau}_2\} : \Omega$, *then* $\Gamma \vdash \{\overline{\tau}_1, \overline{\tau}_2\} \leq \{\overline{\tau}_1\}$.

Proof. Each by induction on the length of $\overline{\tau}$. □

Likewise, corresponding inversion principles:

Lemma C.5 (Tuple Typing Inversoin).

(1) If $\Gamma \vdash \langle \overline{\tau} \rangle : \times(\overline{\kappa})$, *then* $\overline{\Gamma \vdash \tau : \kappa}$.
(2) If $\Gamma \vdash \{\overline{\tau}\} : \Omega$, *then* $\overline{\Gamma \vdash \tau : \Omega}$.

Proof. Each by induction on the length of $\overline{\tau}$. □

C.3 Translating FGJ

Figure 6 defines the translation of FGJ class definitions into λ_{misu} types that represent their instances.

First off, this translation commutes with substitution:

Lemma C.6 (FGJ Translation Substitution).

(1) $[\![T'[\overline{T/\mathsf{X}}]]\!] = [\![T']\!][\overline{[\![T]\!]/\alpha_{\mathsf{X}}}]$
(2) $[\![C[\overline{T/\mathsf{X}}]]\!] = [\![C]\!][\overline{[\![T]\!]/\alpha_{\mathsf{X}}}]$
(3) $[\![N[\overline{T/\mathsf{X}}]]\!]^*_P = [\![N]\!]^*_P[\overline{[\![T]\!]/\alpha_{\mathsf{X}}}]$
(4) $[\![C[\overline{T/\mathsf{X}}]]\!]^*_P = [\![C]\!]^*_P[\overline{[\![T]\!]/\alpha_{\mathsf{X}}}]$

Proof. By simultaneous induction on the structure of the type. Part (4) is a trivial consequence of the fact that classes cannot contain free type variables. □

Contexts ($[\![\Delta]\!]$)	$[\![\overline{C}; \overline{\mathtt{X}<:N}]\!]$	=	$\overline{\alpha_{\mathtt{C}} \leq [\![C]\!]}, \overline{\alpha_{\mathtt{X}} \leq \top}$	
Types ($[\![T]\!]$)	$[\![\mathtt{X}]\!]$	=	$\alpha_{\mathtt{X}}$	
	$[\![\mathtt{Object}]\!]$	=	$\{\}$	
	$[\![\mathtt{C}\langle\overline{T}\rangle]\!]$	=	$\alpha_{\mathtt{C}}\ \overline{[\![T]\!]}$	
Classes ($[\![C]\!]$)	$[\![\mathtt{class\ C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\overline{T\ \mathtt{x}}\}]\!]$	=	$\lambda\overline{\alpha_{\mathtt{X}}:\Omega}.[\![N']\!]$	
Type Representation ($[\![N]\!]^*_P$)	$[\![\mathtt{Object}]\!]^*_P$	=	$\{\}$	
	$[\![\mathtt{C}\langle\overline{T}\rangle]\!]^*_P$	=	$[\![P(\mathtt{C})]\!]^*_P\ \overline{[\![T]\!]}$	
Class Representation ($[\![C]\!]^*_P$)	$[\![\mathtt{class\ C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\overline{T\ \mathtt{x}}\}]\!]^*_P$	=	$\lambda\overline{\alpha_{\mathtt{X}}:\Omega}.\{\overline{\tau}, \overline{[\![T]\!]}\}$	iff $[\![N']\!]^*_P \equiv \{\overline{\tau}\}$
Programs ($[\![P]\!]$)	$[\![P]\!]$	=	$\mu\langle\overline{\alpha_{\mathtt{C}} \leq [\![C]\!]}\rangle.\langle\overline{[\![C]\!]^*_P}\rangle$	iff $P = \overline{C}$

Fig. 6. FGJ type translation

Furthermore, a simple weakening principle applies to the translation of representations:
With that, we can first show that the translation preserves well-formedness of types and contexts:

Theorem C.7 (FGJ Type Translation). *Let* $\Delta \supseteq P, P'; \epsilon$ *and* $\epsilon \vdash_{\text{FGJ}} P, P'$ ok *and* $\Delta \vdash_{\text{FGJ}}$ ok.
(1) If $P \vdash_{\text{FGJ}} P'$ ok *and* $\vdash [\![P; \epsilon]\!]$ ok, *then* $\vdash [\![P, P'; \epsilon]\!]$ ok.
(2) If $\Delta \vdash_{\text{FGJ}}$ ok, *then* $\vdash [\![\Delta]\!]$ ok.
(3) If $\Delta \vdash_{\text{FGJ}} T$ ok, *then* $[\![\Delta]\!] \vdash [\![T]\!] : \Omega$.
(4) If $\Delta \vdash_{\text{FGJ}} N$ ok, *then* $[\![\Delta]\!] \vdash [\![N]\!]^*_P : \Omega$ *with* $[\![N']\!]^*_P \equiv \{\overline{\tau}\}$.
(5) If $P; P' \vdash_{\text{FGJ}} C$ ok, *then* $[\![P; \epsilon]\!] \vdash [\![C]\!] : \overline{\Omega} \to \Omega$ *with* $|\overline{\Omega}| = \text{arity}(C)$.
(6) If $P; P' \vdash_{\text{FGJ}} C$ ok, *then* $[\![P, P'; \epsilon]\!] \vdash [\![C]\!]^*_P : \overline{\Omega} \to \Omega$ *with* $|\overline{\Omega}| = \text{arity}(C)$.

Proof. By simultaneous induction on the combined length of the derivations.

(1) By induction on the derivation of $P \vdash_{\text{FGJ}} P'$ ok
- subcase $P' = \epsilon$
  - by definition, $[\![P, P'; \epsilon]\!] = [\![P; \epsilon]\!]$
  - by assumption, $\vdash [\![P; \epsilon]\!]$ ok
- subcase $P' = C, P''$
  - by definition, $[\![P, P'; \epsilon]\!] = [\![P; \epsilon]\!], \alpha_{\mathtt{C}} \leq [\![C]\!], [\![P'; \epsilon]\!]$
  - by inversion, $P; C, P'' \vdash_{\text{FGJ}} C$ ok and $P, C \vdash_{\text{FGJ}} P''$ ok
  - by induction (5), $[\![P; \epsilon]\!] \vdash [\![C]\!] : \overline{\Omega} \to \Omega$
  - by context formation, $\vdash [\![P; \epsilon]\!], \alpha_{\mathtt{C}} \leq [\![C]\!]$ ok
  - by definition, $[\![P; \epsilon]\!], \alpha_{\mathtt{C}} \leq [\![C]\!] = [\![P, C; \epsilon]\!]$
  - by induction, $\vdash [\![P, C, P''; \epsilon]\!]$ ok

(2) By induction on the derivation of $\Delta \vdash_{\text{FGJ}}$ ok
- Case $\Delta = P, P'; \epsilon$
  - by definition, $[\![\epsilon; \epsilon]\!] = \cdot$
  - by context formation, $\vdash \cdot$ ok
  - by induction (1), $\vdash [\![P, P'; \epsilon]\!]$ ok
- Case $\Delta = \Delta', \mathtt{X}<:N$
  - by definition, $[\![\Delta', \mathtt{X}<:N]\!] = [\![\Delta']\!], \alpha_{\mathtt{X}} \leq \top$
  - by inversion, $\Delta' \vdash_{\text{FGJ}}$ ok
  - by induction (2), $\vdash [\![\Delta']\!]$ ok
  - by kinding rule for top, $[\![\Delta']\!] \vdash \top$ ok
  - by context wf rule, $\vdash [\![\Delta']\!], \alpha_{\mathtt{X}} \leq \top$ ok

(3) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} T$ ok
- Case $T = \mathtt{X}$
  - by inversion, $\mathtt{X} <: N \in \Delta$
  - by definition, $[\![\mathtt{X}]\!] = \alpha_{\mathtt{X}}$
  - by definition, $[\![\Delta]\!](\alpha_{\mathtt{X}}) = \top$
  - by kinding rule for top, $[\![\Delta]\!] \vdash \top : \Omega$
  - by kinding rule for type variables, $[\![\Delta]\!] \vdash \alpha_{\mathtt{X}} : \Omega$
- Case $T = \mathtt{Object}$
  - by definition, $[\![\mathtt{Object}]\!] = \{\}$
  - by Tuple Typing, $[\![\Delta]\!] \vdash \{\} : \Omega$
- Case $T = \mathtt{C}\langle\overline{T}\rangle$
  - by inversion, $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X} <: N}\rangle <: N' \{\dots\} \in \Delta$ and $\overline{\Delta \vdash_{\text{FGJ}} T \text{ ok}}$ and $|\overline{T}| = |\overline{\mathtt{X}}|$
  - by definition, $[\![\mathtt{C}\langle\overline{T}\rangle]\!] = \alpha_{\mathtt{C}}\ \overline{[\![T]\!]}$
  - by induction (3), $\overline{[\![\Delta]\!] \vdash [\![T]\!] : \Omega}$
  - by induction (5), $[\![P; \epsilon]\!] \vdash [\![C]\!] : \overline{\Omega} \to \Omega$ with $|\overline{\Omega}| = |\overline{\mathtt{X}}|$
  - by Weakening, $[\![\Delta]\!] \vdash [\![C]\!] : \overline{\Omega} \to \Omega$
  - by definition, $[\![\Delta]\!](\alpha_{\mathtt{C}}) = [\![C]\!]$
  - by kinding rule for type variables, $[\![\Delta]\!] \vdash \alpha_{\mathtt{C}} : \overline{\Omega} \to \Omega$
  - by iterating kinding rule for application, $[\![\Delta]\!] \vdash \alpha_{\mathtt{C}}\ \overline{[\![T]\!]} : \Omega$

(4) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} N$ ok
- Case $N = \mathtt{Object}$
  - by definition, $[\![\mathtt{Object}]\!]^*_P = \{\}$
  - by Tuple Typing, $[\![\Delta]\!] \vdash \{\} : \Omega$
- Case $N = \mathtt{C}\langle\overline{T}\rangle$
  - by inversion, $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X} <: N'}\rangle <: N'' \{\dots\} \in \Delta$ and $\overline{\Delta \vdash_{\text{FGJ}} T \text{ ok}}$ and $|\overline{T} = |\overline{\mathtt{X}}|$
  - by definition, $[\![\mathtt{C}\langle\overline{T}\rangle]\!]^*_P = [\![C]\!]^*_P\ \overline{[\![T]\!]}$
  - by induction (3), $\overline{[\![\Delta]\!] \vdash [\![T]\!] : \Omega}$
  - by FGJ Program WF Inversion, $P_1; P_2 \vdash_{\text{FGJ}} C$ ok for some $P_1, P_2 = P, P'$
  - by induction (6), $[\![P, P'; \epsilon]\!] \vdash [\![C]\!]^*_P : \overline{\Omega} \to \Omega$ with $|\overline{\Omega}| = |\overline{\mathtt{X}}|$ and $[\![C]\!]^*_P \equiv \lambda\overline{\alpha{:}\Omega}.\{\overline{\tau}\}$
  - by Weakening, $[\![\Delta]\!] \vdash [\![C]\!]^*_P : \overline{\Omega} \to \Omega$
  - by iterating kinding rule for application, $[\![\Delta]\!] \vdash [\![C]\!]^*_P\ \overline{[\![T]\!]} : \Omega$
  - by reduction, $[\![C]\!]^*_P\ \overline{[\![T]\!]} \equiv \{\overline{\tau}\}[\overline{[\![T]\!]/\alpha}]$

(5) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} C$ ok
- by inversion $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X} <: N}\rangle <: N' \{\dots\}$
- let $\Delta = P; \overline{\mathtt{X} <: N}$
- by inversion, $\Delta \vdash_{\text{FGJ}}$ ok and $\Delta \vdash_{\text{FGJ}} N'$ ok
- by definition, $[\![C]\!] = \lambda\overline{\alpha_{\mathtt{X}}{:}\Omega}.[\![N']\!]$
- by induction (2), $\vdash [\![\Delta]\!]$ ok
- by induction (3), $[\![\Delta]\!] \vdash [\![N']\!] : \Omega$
- by definition, $[\![\Delta]\!] = [\![P; \epsilon]\!], \overline{\alpha_{\mathtt{X}} \leq \top}$
- by iterating kinding rule for lambdas, $[\![\Delta]\!] \vdash \lambda\overline{\alpha_{\mathtt{X}}{:}\Omega}.[\![N']\!] : \overline{\Omega} \to \Omega$

(6) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} C$ ok
- by inversion, $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X} <: N}\rangle <: N' \{\dots\}$
- let $\Delta = P; \overline{\mathtt{X} <: N}$ and $\Delta' = P, P'; \overline{\mathtt{X} <: N}$
- by inversion, $\Delta \vdash_{\text{FGJ}}$ ok and $\Delta \vdash_{\text{FGJ}} N'$ ok and $\overline{\Delta' \vdash_{\text{FGJ}} T \text{ ok}}$

- by induction (2), $\vdash [\![\Delta]\!]$ ok and $\vdash [\![\Delta']\!]$ ok
- by induction (3), $\overline{[\![\Delta']\!] \vdash [\![T]\!] : \Omega}$
- by induction (4), $[\![\Delta]\!] \vdash [\![N']\!]^*_P : \Omega$ with $[\![N']\!]^*_P \equiv \{\overline{\tau}\}$
- by definition, $[\![C]\!]^*_P = \lambda \overline{\alpha_X{:}\Omega}.\{\overline{\tau}, \overline{[\![T]\!]}\}$
- by Type Preservation, $[\![\Delta]\!] \vdash \{\overline{\tau}\} : \Omega$
- by Weakening, $[\![\Delta']\!] \vdash \{\overline{\tau}\} : \Omega$
- by Tuple Typing Inversion, $\overline{[\![\Delta']\!] \vdash \tau : \Omega}$
- by Tuple Typing, $[\![\Delta']\!] \vdash \{\overline{\tau}, \overline{[\![T]\!]}\} : \Omega$
- by definition, $[\![\Delta']\!] = [\![P, P'; \epsilon]\!], \overline{\alpha_X \leq \top}$
- by iterating kinding rule for lambda, $[\![P, P'; \epsilon]\!] \vdash \lambda \overline{\alpha_X{:}\Omega}.\{\overline{\tau}, \overline{[\![T]\!]}\} : \overline{\Omega} \to \Omega$ □

Second, we can prove that the translation of classes preserves subtyping:

Theorem C.8 (FGJ Subtype Translation). *Let* $\Delta \supseteq P, P'; \epsilon$ *and* $\epsilon \vdash_{\text{FGJ}} P, P'$ ok *and* $\Delta \vdash_{\text{FGJ}}$ ok.
(1) If $\Delta \vdash_{\text{FGJ}} T_1 \leq T_2$ *and* $\Delta \vdash_{\text{FGJ}} T_1$ ok, *then* $[\![\Delta]\!] \vdash [\![T_1]\!] \leq [\![T_2]\!]$.
(2) If $\Delta \vdash_{\text{FGJ}} N_1 \leq N_2$ *and* $\Delta \vdash_{\text{FGJ}} N_1$ ok, *then* $[\![\Delta]\!] \vdash [\![N_1]\!]^*_P \leq [\![N_2]\!]^*_P$.
(3) If $P; P' \vdash_{\text{FGJ}} C$ ok, *then* $[\![P, P'; \epsilon]\!] \vdash [\![C]\!]^*_P \leq [\![C]\!]$.
Proof.

(1) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} T_1 \leq T_2$
- Case reflexivity
  - by inversion, $T_1 = T_2$
  - by definition, $[\![T_1]\!] = [\![T_2]\!]$
  - by reflexivity, $[\![T_1]\!] \equiv [\![T_2]\!]$
  - by subtyping rule, $[\![\Delta]\!] \vdash [\![T_1]\!] \leq [\![T_2]\!]$
- Case transitivity
  - by inversion, $\Delta \vdash_{\text{FGJ}} T_1 \leq T'$ and $\Delta \vdash_{\text{FGJ}} T' \leq T_2$
  - by induction (1), $[\![\Delta]\!] \vdash [\![T_1]\!] \leq [\![T']\!]$ and $[\![\Delta]\!] \vdash [\![T']\!] \leq [\![T_2]\!]$
  - by FGJ Supertype Validity, $\Delta \vdash_{\text{FGJ}} T'$ ok
  - by FGJ Type Translation (3), $[\![\Delta]\!] \vdash [\![T']\!] : \Omega$
  - by subtyping rule, $[\![\Delta]\!] \vdash [\![T_1]\!] \leq [\![T_2]\!]$
- Case $\mathtt{X} \leq \Delta(\mathtt{X})$
  - by inversion, $T_1 = \mathtt{X}$ and $T_2 = N$ and $\mathtt{X}{<:}N \in \Delta$
  - by definition, $[\![T_1]\!] = \alpha_X$ and $[\![T_2]\!] = [\![N]\!]$
  - by definition, $[\![\Delta]\!](\alpha_X) = [\![N]\!]$
  - by subtyping rule for variables, $[\![\Delta]\!] \vdash \alpha_X \leq [\![N]\!]$
- Case $\mathtt{C}\langle\overline{T}\rangle \leq N'[\overline{T/\mathtt{X}}]$
  - by inversion, $T_1 = \mathtt{C}\langle\overline{T}\rangle$ and $T_2 = N'[\overline{T/\mathtt{X}}]$ and $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X}{<:}N}\rangle{<:}N'\{\dots\} \in \Delta$
  - by definition, $[\![T_1]\!] = \alpha_C\ \overline{[\![T]\!]}$ and $[\![T_2]\!] = [\![N'[\overline{T/\mathtt{X}}]]\!]$
  - by FGJ Translation Substitution (1), $[\![T_2]\!] = [\![N'[\overline{T/\mathtt{X}}]]\!] = [\![N']\!]\,[\overline{[\![T]\!]/\alpha_X}]$
  - by FGJ Program WF Inversion, $P_1; P_2 \vdash_{\text{FGJ}} C$ ok for some $P_1, P_2 = P, P'$
  - by inversion, $P_1; \overline{\mathtt{X}{<:}N} \vdash N'$ ok
  - by FGJ Type Translation (2), $[\![P_1; \epsilon]\!], \overline{\alpha_X \leq \top} \vdash [\![N']\!] : \Omega$
  - by Weakening, $[\![\Delta]\!], \overline{\alpha_X \leq \top} \vdash [\![N']\!] : \Omega$
  - by inversion of $\Delta \vdash_{\text{FGJ}} T_1$ ok assumption, $\overline{\Delta \vdash T \text{ ok}}$
  - by FGJ Type Translation (2), $\overline{[\![\Delta]\!] \vdash [\![T]\!] : \Omega}$
  - by Substitution, $[\![\Delta]\!] \vdash [\![N']\!]\,[\overline{[\![T]\!]/\alpha_X}] : \Omega$
  - by definition, $[\![C]\!] = \lambda \overline{\alpha_X{:}\Omega}.[\![N']\!]$

- by beta reduction rule, $[\![C]\!]\ \overline{[\![T]\!]} \equiv [\![N'[\overline{T}/\overline{\mathtt{X}}]]\!]$
- by definition, $[\![\Delta]\!](\alpha_{\mathtt{C}}) = [\![C]\!]$
- by subtyping rule for variables, $[\![\Delta]\!] \vdash \alpha_{\mathtt{C}} \leq [\![C]\!]$
- by reflexivity, $\overline{[\![T]\!] \equiv [\![T]\!]}$
- by subtyping rule, $\overline{[\![\Delta]\!] \vdash [\![T]\!] \leq [\![T]\!]}$
- by subtyping rule for application, $[\![\Delta]\!] \vdash \alpha_{\mathtt{C}}\ \overline{[\![T]\!]} \leq [\![C]\!]\ \overline{[\![T]\!]}$
- by subtyping transitivity, $[\![\Delta]\!] \vdash \alpha_{\mathtt{C}}\ \overline{[\![T]\!]} \leq [\![N'[\overline{T}/\overline{\mathtt{X}}]]\!]$

(2) By induction on the derivation of $\Delta \vdash_{\text{FGJ}} N_1 \leq N_2$

- Case reflexivity
  - by inversion, $N_1 = N_2$
  - by definition, $[\![N_1]\!]^*_P = [\![N_2]\!]^*_P$
  - by reflexivity, $[\![N_1]\!]^*_P \equiv [\![N_2]\!]^*_P$
  - by subtyping rule, $[\![\Delta]\!] \vdash [\![N_1]\!]^*_P \leq [\![N_2]\!]^*_P$
- Case transitivity
  - by inversion, $\Delta \vdash_{\text{FGJ}} N_1 \leq N'$ and $\Delta \vdash_{\text{FGJ}} N' \leq N_2$
  - by induction (2), $[\![\Delta]\!] \vdash [\![N_1]\!]^*_P \equiv [\![N']\!]^*_P$ and $[\![\Delta]\!] \vdash [\![N']\!]^*_P \equiv [\![N_2]\!]^*_P$
  - by FGJ Supertype Validity, $\Delta \vdash_{\text{FGJ}} N'$ ok
  - by Type Translation (4), $[\![\Delta]\!] \vdash [\![N']\!]^*_P : \Omega$
  - by subtyping rule, $[\![\Delta]\!] \vdash [\![N_1]\!]^*_P \equiv [\![N_2]\!]^*_P$
- Case $\mathtt{X} \leq \Delta(\mathtt{X})$
  - impossible
- Case $\mathtt{C}\langle\overline{T}\rangle \leq N'[\overline{T}/\overline{\mathtt{X}}]$
  - by inversion, $N_1 = \mathtt{C}\langle\overline{T}\rangle$ and $N_2 = N'[\overline{T}/\overline{\mathtt{X}}]$ and $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\overline{T'\ \mathtt{x}}, \dots\} \in \Delta$
  - by definition, $[\![N_1]\!]^*_P = [\![C]\!]^*_P\ \overline{[\![T]\!]}$ and $[\![N_2]\!]^*_P = [\![N'[\overline{T}/\overline{\mathtt{X}}]]\!]^*_P$
  - by inversion of $\Delta \vdash_{\text{FGJ}} N_1$ ok assumption, $\overline{\Delta \vdash T \text{ ok}}$
  - by FGJ Type Translation (3), $\overline{[\![\Delta]\!] \vdash [\![T]\!] : \Omega}$
  - by FGJ Program WF Inversion, $P_1; P_2 \vdash_{\text{FGJ}} C$ ok for some $P_1, P_2 = P, P'$
  - by inversion, $P_1; \overline{\mathtt{X}<:N} \vdash N'$ ok
  - by FGJ Type Translation (4), $[\![P_1; \epsilon]\!], \overline{\alpha_{\mathtt{X}} \leq \top} \vdash [\![N']\!]^*_P : \Omega$ with $[\![N']\!]^*_P \equiv \{\overline{\tau}\}$
  - by Weakening, $[\![\Delta]\!], \overline{\alpha_{\mathtt{X}} \leq \top} \vdash [\![N']\!]^*_P : \Omega$
  - by definition, $[\![C]\!]^*_P = \lambda\overline{\alpha_{\mathtt{X}}:\Omega}.\{\overline{\tau}, \overline{[\![T']\!]}\}$
  - by FGJ Translation Substitution (3), $[\![N_2]\!]^*_P = [\![N'[\overline{T}/\overline{\mathtt{X}}]]\!]^*_P = [\![N']\!]^*_P[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}] = \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}]\}$
  - by FGJ Type Translation (4), $[\![\Delta]\!] \vdash [\![N_1]\!]^*_P : \Omega$
  - by Substitution, $[\![\Delta]\!] \vdash [\![N']\!]^*_P[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}] : \Omega$
  - by definition, $[\![N']\!]^*_P[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}] = \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}]\}$
  - by beta reduction rule, $[\![C]\!]^*_P\ \overline{[\![T]\!]} \equiv \{\overline{\tau}, \overline{[\![T']\!]}\}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}] = \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}], \overline{[\![T']\!][\![T]\!]/\alpha_{\mathtt{X}}}]\}$
  - by Type Preservation, $[\![\Delta]\!] \vdash \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}], \overline{[\![T']\!][\![T]\!]/\alpha_{\mathtt{X}}}]\} : \Omega$
  - by Tuple Typing (3), $[\![\Delta]\!] \vdash \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}], \overline{[\![T']\!][\![T]\!]/\alpha_{\mathtt{X}}}]\} \leq \{\overline{\tau}[\overline{[\![T]\!]/\alpha_{\mathtt{X}}}]\}$
  - by subtyping transitivity, $[\![\Delta]\!] \vdash [\![C]\!]^*_P\ \overline{[\![T]\!]} \leq [\![N']\!]^*_P[\overline{T}/\overline{\mathtt{X}}]$

(3) By induction on the derivation of $P; P' \vdash_{\text{FGJ}} C$ ok

- by inversion, $C = \text{class } \mathtt{C}\langle\overline{\mathtt{X}<:N}\rangle<:N'\{\overline{T\ \mathtt{x}}\}$
- let $\Delta = P; \overline{X<:N}$ and $\Delta' = P, P'; \overline{X<:N}$
- by inversion, $\Delta \vdash_{\text{FGJ}}$ ok and $\Delta \vdash_{\text{FGJ}} N'$ ok and $\overline{\Delta' \vdash_{\text{FGJ}} T \text{ ok}}$
- by definition, $[\![C]\!] = \lambda\overline{\alpha_{\mathtt{X}}:\Omega}.[\![N']\!]$ and $[\![C]\!]^*_P = \lambda\overline{\alpha_{\mathtt{X}}:\Omega}.\{\overline{\tau}, \overline{[\![T]\!]}\}$
- by FGJ Type Translation (4), $[\![N']\!]^*_P \equiv \{\overline{\tau}\}$

- by symmetry, $\{\overline{\tau}\} \equiv [\![N']\!]^*_P$
- by subtyping reflexivity, $[\![\Delta]\!] \vdash \{\overline{\tau}\} \leq [\![N']\!]^*_P$
- by Tuple Typing (3), $[\![\Delta]\!] \vdash \{\overline{\tau}, \overline{[\![T]\!]}\} \leq \{\overline{\tau}\}$
- by subtyping transitivity, $[\![\Delta]\!] \vdash \{\overline{\tau}, \overline{[\![T]\!]}\} \leq [\![N']\!]^*_P$
- by iterating subtyping rule for lambda, $[\![\Delta]\!] \vdash \overline{\lambda\alpha_X{:}\Omega.}\{\overline{\tau}, \overline{[\![T]\!]}\} \leq \overline{\lambda\alpha_X{:}\Omega.}[\![N']\!]$ □

As a last interesting step, we need to relate the translation to unrolling:

Lemma C.9 (FGJ Unrolling Translation).

Let $P = \overline{C} = \overline{\text{class } \mathsf{C}\langle \overline{\mathsf{X}{<:}N}\rangle {<:} N'\{\dots\}}^{\,i}$ *and* $\overline{\kappa = \Omega^{|\mathsf{X}|} \to \Omega}^{\,i}$ *and* $\sigma = [\overline{\langle\overline{[\![C]\!]}^*_P\rangle.i/\alpha_\mathsf{C}}^{\,i}]$.

(1) $\text{unroll}^\sigma_{\Omega^{|\overline{\mathsf{X}_i}|-|\overline{\alpha}|}\to\Omega}([\![C_i]\!]\ \overline{\alpha}) \equiv (\overline{\lambda\alpha_{\mathsf{X}_i}{:}\Omega.}[\![N'_i]\!]^*_P)\ \overline{\alpha}$.

(2) $\text{unroll}^\sigma_{\kappa_i}([\![C_i]\!]) \equiv \overline{\lambda\alpha_{\mathsf{X}_i}{:}\Omega.}[\![N'_i]\!]^*_P$.

Proof.

(1) By induction on the length of $|\mathsf{X}_i|$, observing in the base case that $\langle\overline{[\![C]\!]}^*_P\rangle.i \equiv [\![C_i]\!]^*_P$.
(2) Follows directly from (1). □

Finally, from all that it follows that the translation of programs, and hence mutually recursive classes, is well-formed:

Theorem C.10 (FGJ Program Translation). *Let* $\epsilon \vdash_{\text{FGJ}} P, \overline{C}:$ ok *and* $\overline{\kappa = \Omega^{\text{arity}(C)} \to \Omega}$.

(1) $[\![P, \overline{C}]\!] \vdash \langle\overline{\alpha_\mathsf{C}}\rangle : \times(\overline{\kappa})$.

(2) $[\![P, \overline{C}]\!] \vdash \langle\overline{[\![C]\!]^*_P}\rangle : \times(\overline{\kappa})$.

(3) $[\![P, \overline{C}]\!] \vdash \langle\overline{[\![C]\!]^*_P}\rangle \leq \text{unroll}^\sigma_{\times(\overline{\kappa})}(\langle\overline{[\![C]\!]}\rangle)$ *where* $\sigma = [\overline{\langle\overline{[\![C]\!]}^*_P\rangle.i/\alpha_\mathsf{C}}^{\,i}]$

Proof. Each by induction on the length of $\overline{C}$.

(1) Follows from the definition of $[\![P, \overline{C}]\!]$ and the kinding rule for type variables.
(2) Follows from FGJ Type Translation (6).
(3) Follows from FGJ Subtype Translation (4) and FGJ Unrolling Translation and subtyping transitivity. □

Corollary C.11 (FGJ Translation Correctness). *If* $\epsilon \vdash_{\text{FGJ}} P$ ok, *then* $\cdot \vdash [\![P]\!] : \kappa$.

Proof. By FGJ Program Translation with empty P and the typing rule for n-ary μ-types. □

Received 2023-04-14; accepted 2023-08-27