1ML – Core and Modules United (*F-ing First-class Modules*)

Andreas Rossberg
Google, Germany
rossberg@mpi-sws.org

Abstract

ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors. Modules form a separate, higher-order functional language on top of the core. There are both practical and technical reasons for this stratification; yet, it creates substantial duplication in syntax and semantics, and it reduces expressiveness. For example, selecting a module cannot be made a dynamic decision. Language extensions allowing modules to be packaged up as first-class values have been proposed and implemented in different variations. However, they remedy expressiveness only to some extent, are syntactically cumbersome, and do not alleviate redundancy.

We propose a redesign of ML in which modules are truly first-class values, and core and module layer are unified into one language. In this "1ML", functions, functors, and even type constructors are one and the same construct; likewise, no distinction is made between structures, records, or tuples. Or viewed the other way round, everything is just ("a mode of use of") modules. Yet, 1ML does not require dependent types, and its type structure is expressible in terms of plain System F_ω, in a minor variation of our F-ing modules approach. We introduce both an explicitly typed version of 1ML, and an extension with Damas/Milner-style implicit quantification. Type inference for this language is not complete, but, we argue, not substantially worse than for Standard ML.

An alternative view is that 1ML is a user-friendly surface syntax for System F_ω that allows combining term and type abstraction in a more compositional manner than the bare calculus.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Design, Theory

Keywords ML modules, first-class modules, type systems, abstract data types, existential types, System F, elaboration

1. Introduction

The ML family of languages is defined by two splendid innovations: parametric polymorphism with Damas/Milner-style type inference [18, 3], and an advanced module system based on concepts from dependent type theory [17]. Although both have contributed to the success of ML, they exist in almost entirely distinct parts of the language. In particular, the convenience of type inference is available only in ML's so-called *core language*, whereas the *module language* has more expressive types, but for the price of being painfully verbose. Modules form a separate language layered on top of the core. Effectively, ML is two languages in one.

This stratification makes sense from a historical perspective. Modules were introduced for programming-in-the-large, when the core language already existed. The dependent type machinery that was the central innovation of the original module design was alien to the core language, and could not have been integrated easily.

However, we have since discovered that dependent types are not actually necessary to explain modules. In particular, Russo [26, 28] demonstrated that module types can be readily expressed using only System-F-style quantification. The *F-ing modules* approach later showed that the entire ML module system can in fact be understood as a form of syntactic sugar over System F_ω [25].

Meanwhile, the second-class nature of modules has increasingly been perceived as a practical limitation. The standard example being that it is not possible to select modules at runtime:

```
module Table = if size > threshold then HashMap else TreeMap
```

A definition like this, where the choice of an implementation is dependent on dynamics, is entirely natural in object-oriented languages. Yet, it is not expressible with ordinary ML modules. What a shame!

1.1 Packaged Modules

It comes to no surprise, then, that various proposals have been made (and implemented) that enrich ML modules with the ability to package them up as first-class values [27, 22, 6, 25, 7]. Such *packaged modules* address the most imminent needs, but they are not to be confused with truly first-class modules. They require explicit injection into and projection from first-class core values, accompanied by heavy annotations. For example, in OCaml 4 the above example would have to be written as follows:

```
module Table = (val (if size > threshold
                     then (module HashMap : MAP)
                     else (module TreeMap : MAP))) : MAP)
```

which, arguably, is neither natural nor pretty. Packaged modules have limited expressiveness as well. In particular, type sharing with a packaged module is only possible via a detour through core-level polymorphism, such as in:

```
f : (module S with type t = 'a) → (module S with type t = 'a) → 'a
```

(where t is an abstract type in S). In contrast, with proper modules, the same sharing could be expressed as

```
f : (X : S) → (S with type t = X.t) → X.t
```

Because core-level polymorphism is first-order, this approach cannot express type sharing between type *constructors* – a complaint that has come up several times on the OCaml mailing list; for example, if one were to abstract over a monad:

```
map : (module MONAD with type 'a t = ?) → ('a → 'b) → ? → ?
```

There is nothing that can be put in place of the ?'s to complete this function signature. The programmer is forced to either use weaker types (if possible at all), or drop the use of packaged modules and lift the function (and potentially a lot of downstream code) to the functor level – which not only is very inconvenient, it also severely restricts the possible computational behaviour of such code. One could imagine addressing this particular limitation by introducing higher-kinded polymorphism into the ML core. But with such an extension type inference would require higher-order unification and hence become undecidable – unless accompanied by significant restrictions that are likely to defeat this example (or others).

1.2 First-Class Modules

Can we overcome this situation and make modules more equal citizens of the language? The answer from the literature has been: no, because first-class modules make type-checking undecidable and type inference infeasible.

The most relevant work is Harper & Lillibridge's calculus of *translucent sums* [9] (a precursor of later work on *singleton types* [31]). It can be viewed as an idealised functional language that allows types as components of (dependent) records, so that they can express modules. In the type of such a record, individual type members can occur as either transparent or opaque (hence, *translucent*), which is the defining feature of ML module typing.

Harper & Lillibrige prove that type-checking this language is undecidable. Their result applies to any language that has (a) contravariant functions, (b) both transparent and opaque types, and (c) allows opaque types to be subtyped with arbitrary transparent types. The latter feature usually manifests in a subtyping rule like

$$\frac{\{D_1[\tau/\mathsf{t}]\} \leq \{D_2[\tau/\mathsf{t}]\}}{\{\textbf{type}\ \mathsf{t}{=}\tau;D_1\} \leq \{\textbf{type}\ \mathsf{t};D_2\}}\textsc{forget}$$

which is, in some variation, at the heart of every definition of signature matching. In the premise the concrete type τ is substituted for the abstract t. Obviously, this rule is not inductive. The substitution can arbitrarily grow the types, and thus potentially require infinite derivations. A concrete example triggering non-termination is the following, adapted from Harper & Lillibridge's paper [9]:

```
type T = {type A; f : A → ()}
type U = {type A; f : (T where type A = A) → ()}
type V = T where type A = U
g (X : V) = X : U  (* V ≤ U ? *)
```

Checking $\mathsf{V} \leq \mathsf{U}$ would match **type** A with **type** A=U, substituting U for A accordingly, and then requires checking that the types of f are in a subtyping relation – which contravariantly requires checking that $(\mathsf{T}\ \textbf{where type}\ \mathsf{A} = \mathsf{A})[\mathsf{U}/\mathsf{A}] \leq \mathsf{A}[\mathsf{U}/\mathsf{A}]$, but that is the same as the $\mathsf{V} \leq \mathsf{U}$ we wanted to check in the first place.

In fewer words, signature matching is no longer decidable when module types can be abstracted over, which is the case if module types are simply collapsed into ordinary types. It also arises if "abstract signatures" are added to the language, as in OCaml, where the same divergent example can be constructed on the module type level alone.

Some may consider decidability a rather theoretical concern. However, there also is the – quite practical – issue that the introduction of signature matching into the core language makes ML-style type inference impossible. Obviously, Milner's algorithm $\mathcal{W}$ [18] is far too weak to handle dependent types. Moreover, modules introduce subtyping, which breaks unification as the basic algorithmic tool for solving type constraints. And while inference algorithms for subtyping exist, they have much less satisfactory properties than our beloved Hindley/Milner sweet spot.

Worse, module types do not even form a lattice under subtyping:

```
f1 : {type t a; x : t int} → int
f2 : {type t a; x : int} → int
g = if condition then f1 else f2
```

There are at least two possible types for g:

```
g : {type t a = int; x : int} → int
g : {type t a = a; x : int} → int
```

Neither is more specific than the other, so no least upper bound exists. Consequently, annotations are necessary to regain principal types for constructs like conditionals, in order to restore any hope for compositional type *checking*, let alone inference.

1.3 F-ing Modules

In our work on *F-ing modules* with Russo & Dreyer [25] we have demonstrated that ML modules can be expressed and encoded entirely in vanilla System F (or F_ω, depending on the concrete core language and the desired semantics for functors). Effectively, the F-ing semantics defines a type-directed desugaring of module syntax into System F types and terms, and inversely, interprets a stylised subset of System F types as module signatures.

The core language that we assume in that paper is System F (respectively, F_ω) itself, leading to the seemingly paradoxical situation that the core language appears to have *more* expressive types than the module language. That makes sense when considering that the module translation rules manipulate the sublanguage of module types in ways that would not generalise to arbitrary System F types. In particular, the rules *implicitly* introduce and eliminate universal and existential quantifiers, which is key to making modules a usable means of abstraction. But the process is guided by, and only meaningful for, module syntax; likewise, the built-in subtyping relation is only "complete" for the specific occurrences of quantifiers in module types.

Nevertheless, the observation that modules are just sugar for certain kinds of constructs that the core language can already express (even if less concisely), raises the question: what necessitates modules to be second-class in that system?

1.4 1ML

The answer to that question is: very little! And the present paper is motivated by exploring that answer.

In essence, the F-ing modules semantics reveals that the syntactic stratification between ML core and module language is merely a rather coarse means to enforce *predicativity* for module types: it prevents abstract types themselves from being instantiated with binders for abstract types. But this heavy *syntactic* restriction can be replaced by a more surgical *semantic* restriction! It is enough to employ a simple universe distinction between *small* and *large* types (reminiscent of Harper & Mitchell's XML [10]), and limit the equivalent of the FORGET rule shown earlier to only allow small types for subsitutition, which serves to exclude problematic quantifiers.

That would settle decidability, but what about type inference? Well, we can use the same distinction! A quick inspection of the subtyping rules in the F-ing modules semantics reveals that they, almost, degenerate to type equivalence when applied to *small* types — the only exception being width subtyping on structures. If we are willing to accept that inference is not going to be complete for records (which it already isn't in Standard ML), then a simple restriction to inferring only small types is sufficient to make type inference work almost as usual.

In this spirit, this paper presents *1ML*, an ML-dialect in which modules are truly first-class values. The name is both short for "1st-class module language" and a pun on the fact that it unifies core and modules of ML into one language. Our contributions are as follows:

- We present a decidable type system for a language of first-class modules that subsumes conventional second-class ML modules.
- We give an elaboration of this language into plain System F_ω.
- We show how Damas/Milner-style type inference can be integrated into such a language; it is incomplete, but only in ways that are already present in existing ML implementations.
- We develop the basis for a practical design of an ML-like language in which the distinction between core and modules has been eliminated.

We see several benefits with this redesign: it produces a language that is more *expressive* and *concise*, and at the same time, more *minimal* and *uniform*. "Modules" become a natural means to express all forms of (first-class) polymorphism, and can be freely intermixed with "computational" code and data. Type inference integrates in a rather seamless manner, reducing the need for explicit annotations to large types, module or not. Every programming concept is derived from a small set of orthogonal constructs, over which general and uniform syntactic sugar can be defined.

2. 1ML with Explicit Types

To separate concerns a little, we will start out by introducing 1ML_{ex}, a sublanguage of 1ML proper that is explicitly typed and does not support any type inference. Its kernel syntax is given in Figure 1. Let us take a little tour of 1ML_{ex} by way of examples.

Functional Core A major part of 1ML_{ex} consists of fairly conventional functional language constructs. On the expression level, as a representative for a base type, we have Booleans; in examples that follow, we will often assume the presence of an integer type and respective constructs as well. Then there are records, which consist of a sequence of bindings. And of course, it wouldn't be a functional language without functions.

In a first approximation, these forms are reflected on the type level as one would expect, except that for functions we allow two forms of arrows, distinguishing pure function types (⇒) from impure ones (→) (discussed later).

Like in the F-ing modules paper [25], most elimination forms in the kernel syntax only allow variables as subexpressions. However, the general expression forms are all definable as straightforward syntactic sugar, as shown in the lower half of Figure 1. For example,

```
(fun (n : int) ⇒ n + n) 3
```

desugars into

```
let f = fun (n : int) ⇒ n + n; x = 3 in f x
```

and further into

```
{f = fun (n : int) ⇒ n + n; x = 3; body = f x}.body
```

This works because records actually behave like ML structures, such that every bound identifier is in scope for later bindings – which enables encoding let-expressions.

Also, notably, if-expressions require a type annotation in 1ML_{ex}. As we will see, the type language subsumes module types, and as discussed in Section 1.2 there wouldn't generally be a unique least upper bound otherwise. However, in Section 4 we show that this annotation can usually be omitted in full 1ML.

Reified Types The core feature that makes 1ML_{ex} able to express modules is the ability to embed types in a first-class manner: the expression **type** T reifies the type T *as a value*.[1] Such an expression has type **type**, and thereby can be abstracted over. For example,

```
id = fun (a : type) ⇒ fun (x : a) ⇒ x
```

defines a polymorphic identity function, similar to how it would be written in dependent type theories. Note in particular that a is a *term* variable, but it is used as a *type* in the annotation for x. This is enabled by the "path" form E in the syntax of types, which expresses the (implicit) projection of a type from a term, provided this term has type **type**. Consequently, all variables are term variables in 1ML, there is no separate notion of type variable.

More interestingly, a function can *return* types, too. Consider

```
pair = fun (a : type) ⇒ fun (b : type) ⇒ type {fst : a; snd : b}
```

which takes a type and returns a type, and effectively defines a type *constructor*. Applied to a reified type it yields a reified type. Again, the implicit projection from "paths" enables using this as a type:

```
second = fun (a : type) ⇒ fun (b : type) ⇒ fun (p : pair a b) ⇒ p.snd
```

In this example, the whole of "pair a b" is a term of type **type**.

Figure 1 also defines a bit of syntactic sugar to make function and type definitions look more like in traditional ML. For example, the previous functions could equivalently be written as

```
id a (x : a) = x
type pair a b = {fst : a; snd : b}
second a b (p : pair a b) = p.snd
```

It may seem surprising that we can just reify types as first-class values. But reified types (or "atomic type modules") have been common in module calculi for a long time [16, 6, 24, 25]. We are merely making them available in the source language directly. For the most part, this is just a notational simplification over what first-class modules already offer: instead of having to define a spurious module T = {**type** t = int} : {**type** t} and then refer to T.t, we allow injecting types into modules (i.e., values) *anonymously*, without wrapping them into a structure; thus t = (**type** int) : **type**, which can be referred to as just t.

Translucency The type **type** allows classifying types abstractly: given a value of type **type**, nothing is known about *what* type it is. But for modular programming it is essential that types can selectively be specified *transparently*, which enables expressing the vital concept of *type sharing* [12].

As a simple example, consider these type aliases:

```
type size = int
type pair a b = {fst : a; snd : b}
```

According to the idea of translucency, the variables defined by these definitions can be classified in one of two ways. Either opaquely:

```
size : type
pair : (a : type) ⇒ (b : type) ⇒ type
```

Or transparently:

```
size : (= type int)
pair : (a : type) ⇒ (b : type) ⇒ (= type {fst : a; snd : b})
```

The latter use a variant of *singleton types* [31, 6] to reveal the definitions: a type of the form "$=E$" is inhabited only by values that are "structurally equivalent" to E, in particular, with respect to parts of type **type**. It allows the type system to infer, for example, that the application pair size size is equivalent to the (reified) type

[1] Ideally, "**type** T" should be written just "T", like in dependently typed systems. However, that would create various syntactic ambiguities, e.g. for phrases like "{}", which could only be avoided by moving to a more artificial syntax for types themselves. Nevertheless, we at least allow writing "$E\ T$" for the application "E (**type** T)" if T unambiguously is a type.

$$
\begin{array}{lll}
\text{(identifiers)} & X & \\
\text{(types)} & T & ::= \; E \mid \mathbf{bool} \mid \{D\} \mid (X{:}T) \stackrel{\Rightarrow}{\rightarrow} T \mid \mathbf{type} \mid {=}E \mid T \;\mathbf{where}\; (\overline{.X}{:}T) \\
\text{(declarations)} & D & ::= \; X : T \mid \mathbf{include}\; T \mid D;D \mid \epsilon \\
\text{(expressions)} & E & ::= \; X \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\; X \;\mathbf{then}\; E \;\mathbf{else}\; E{:}T \mid \{B\} \mid E.X \mid \mathbf{fun}\; (X{:}T) {\Rightarrow} E \mid X\;X \mid \mathbf{type}\; T \mid X{:>}T \\
\text{(bindings)} & B & ::= \; X{=}E \mid \mathbf{include}\; E \mid B;B \mid \epsilon
\end{array}
$$

$$
\begin{array}{ll}
\text{(types)} & \\
\mathbf{let}\; B \;\mathbf{in}\; T & := \{B;X{=}\,\mathbf{type}\; T\}.X \\
T_1 \stackrel{\Rightarrow}{\rightarrow} T_2 & := (X{:}T_1) \stackrel{\Rightarrow}{\rightarrow} T_2 \\
T \;\mathbf{where}\; (\overline{.X}\;\overline{P}{=}E) & := T \;\mathbf{where}\; (\overline{.X}{:}\;\overline{P \Rightarrow} ({=}E)) \\
T \;\mathbf{where}\; (\mathbf{type}\;\overline{.X}\;\overline{P}{=}T') & := T \;\mathbf{where}\; (\overline{.X}{:}\;\overline{P \Rightarrow} ({=}\,\mathbf{type}\; T')) \\
\text{(declarations)} & \\
\mathbf{local}\; B \;\mathbf{in}\; D & := \mathbf{include}\; (\mathbf{let}\; B \;\mathbf{in}\; \{D\}) \\
X\;\overline{P}{:}T & := X : \overline{P \Rightarrow} T \\
X\;\overline{P}{=}E & := X : \overline{P \Rightarrow} ({=}\,E) \\
\mathbf{type}\; X\;\overline{P} & := X : \overline{P \Rightarrow} \mathbf{type} \\
\mathbf{type}\; X\;\overline{P}{=}T & := X : \overline{P \Rightarrow} ({=}\,\mathbf{type}\; T)
\end{array}
$$

where: (parameter) $P ::= (X{:}T)$ with abbreviation $X := (X{:}\,\mathbf{type})$

$$
\begin{array}{ll}
\text{(expressions)} & \\
\mathbf{let}\; B \;\mathbf{in}\; E & := \{B;X{=}E\}.X \\
\mathbf{if}\; E_1 \;\mathbf{then}\; E_2 \;\mathbf{else}\; E_3{:}T & := \mathbf{let}\; X{=}E_1 \;\mathbf{in}\;\mathbf{if}\; X \;\mathbf{then}\; E_2 \;\mathbf{else}\; E_3{:}T \\
E_1\;E_2 & := \mathbf{let}\; X_1{=}E_1;\; X_2{=}E_2 \;\mathbf{in}\; X_1\;X_2 \\
E\;T & := E\;(\mathbf{type}\; T) \qquad \text{(if } T \text{ unambiguous)} \\
E : T & := (\mathbf{fun}\; (X{:}T) \Rightarrow X)\; E \\
E :> T & := \mathbf{let}\; X{=}E \;\mathbf{in}\; X :> T \\
\mathbf{fun}\; \overline{P} \Rightarrow E & := \overline{\mathbf{fun}\; P \Rightarrow} E \\
\text{(bindings)} & \\
\mathbf{local}\; B \;\mathbf{in}\; B' & := \mathbf{include}\; (\mathbf{let}\; B \;\mathbf{in}\; \{B'\}) \\
X\;\overline{P}\;\overline{:T'}\;\overline{:>T''}{=}E & := X = \mathbf{fun}\; \overline{P} \Rightarrow E\;\overline{:T'}\;\overline{:>T''} \\
\mathbf{type}\; X\;\overline{P}{=}T & := X = \mathbf{fun}\; \overline{P} \Rightarrow \mathbf{type}\; T
\end{array}
$$

(Identifiers X only occurring on the right-hand side are considered fresh)

Figure 1. $1\mathrm{ML}_{\mathrm{ex}}$ syntax and syntactic abbreviations

{fst : int; snd : int}. A type $=E$ is a subtype of the type of E itself, and consequently, transparent classifications define subtypes of opaque ones, which is the crux of ML signature matching.

Translucent types usually occur as part of module type declarations, where 1ML can abbreviate the above to the more familiar

```
type size                                  type size = int
type pair a b      or, respectively,       type pair a b = {fst : a; snd : b}
```

i.e., as in ML, transparent declarations look just like definitions.

Singletons can be formed over arbitrary values. This gives the ability to express *module sharing* and *aliases*. In the basic semantics described in this paper, this is effectively a shorthand for sharing all types contained in the module (including those defined inside transparent functors, see below). We leave the extension to full *value* equivalence (including primitive types like Booleans), as in our F-ing semantics for applicative functors [25], to future work.

Functors Returning to the 1ML grammar, the remaining constructs of the language are typical for ML modules, although they are perhaps a bit more general than what is usually seen. Let us explain them using an example that demonstrates that our language can readily express "real" modules as well. Here is the (unavoidable, it seems) functor that defines a simple map ADT:

```
type EQ =
{
  type t;
  eq : t → t → bool
};
type MAP =
{
  type key;
  type map a;
  empty a : map a;
  add a : key → a → map a → map a;
  lookup a : key → map a → opt a
};
Map (Key : EQ) :> MAP where (type .key = Key.t) =
{
  type key = Key.t;
  type map a = key → opt a;
  empty a = fun (k : key) ⇒ none a;
  lookup a (k : key) (m : map a) = m k;
  add a (k : key) (v : a) (m : map a) =
    fun (x : key) ⇒ if Key.eq x k then some a v else m x : opt a
}
```

The record type EQ amounts to a module signature, since it contains an abstract type component t. It is referred to in the type of eq, which shows that record types are seemingly "dependent": like for terms, earlier components are in scope for later components – the key insight of the F-ing approach is that this dependency is benign, however, and can be translated away, as we will see in Section 3.

Similarly, MAP defines a signature with abstract key and map types. Note how type parameters on the left-hand side conveniently and uniformly generalise to value declarations, avoiding the need for brittle implicit scoping rules like in conventional ML: as shown in Figure 1, "empty a : map a" abbreviates "empty : (a : **type**) ⇒ map a", in a generalisation of the syntax for type specifications introduced earlier, where "**type** t a" desugars into "t a : **type**" and then "t : (a : **type**) ⇒ **type**".

The Map function is a functor: it takes a value of type EQ, i.e., a module. From that it constructs a naive implementation of maps. "$X{:>}T$" is the usual *sealing* operator that opaquely ascribes a type (i.e., signature) to a value (a.k.a. module). The *type refinement* syntax "T **where** (**type** $.X{=}T$)" should be familiar from ML, but here it actually is derived from a more general construct: "T **where** $(\overline{.X}{:}U)$" refines T's subcomponent at path $\overline{.X}$ to type U, which can be any subtype of what's declared by T. That form subsumes module sharing as well as other forms of refinement.

Applicative vs. Generative In this paper, we stick to a relatively simple semantics for functor-like functions, in which Map is *generative* [28, 4, 25]. That is, like in Standard ML, each application will yield a fresh map ADT, because sealing occurs inside the functor:

```
M₁ = Map IntEq;
M₂ = Map IntEq;
m = M₁.add int 7 M₂.empty  (* ill-typed: M₁.map ≠ M₂.map *)
```

But as we saw earlier, type constructors like pair or map are essentially functors, too! Sealing the body of the Map functor hence implies higher-order sealing of the nested map "functor", as if performing map :> **type** ⇒ **type**. It is vital that the resulting functor has *applicative* semantics [15, 25], so that

```
type map a = M_1.map a;
type t = map int;
type u = map int
```

yields t = u, as one would expect from a proper type constructor.

We hence need applicative functors as well. To keep things simple, we restrict ourselves to the simplest possible semantics in this paper, in which we distinguish between pure (⇒, i.e. applicative) and impure (→, i.e. generative) function types, but sealing is always impure (or *strong* [6]). That is, sealing *inside* a functor always makes it generative. The only way to produce an applicative functor is by sealing a (fully transparent) functor *as a whole*, with applicative functor type, as for the map type constructor above. Given:

```
F = (fun (a : type) ⇒ type {x : a}) :> type ⇒ type
G = (fun (a : type) ⇒ type {x : a}) :> type → type
H = fun (a : type) ⇒ (type {x : a} :> type)
J  = G :> type ⇒ type                              (* ill-typed! *)
```

F is an applicative functor, such that F int = F int. G and H on the other hand are generative functors; the former because it is sealed with impure function type, the latter because sealing occurs inside its body. Consequently, G int or H int are impure expressions and invalid as type paths (though it is fine to bind their result to a name, e.g., "**type** w = G int", and use the constant w as a type). Lastly, J is ill-typed, because applicative functor types are subtypes of generative ones, but not the other way round.

This semantics for applicative functors (which is very similar to the applicative functors of Shao [30]) is somewhat limited, but just enough to encode sealing over type constructors and hence recover the ability to express type definitions as in conventional ML. An extension of 1ML to applicative functors with *pure* sealing à la F-ing modules [25] is given in the Technical Appendix [23].

The purity distinction would naturally extend to other relevant effects, such as state. For example, the assignment operator := would need to be typed as impure (because there is no sound elaboration for it otherwise), while other operators, such as +, could be pure. However, we do not explore that space further here, and conservatively treat all "core-like" functions as impure for now.

Higher Polymorphism So far, we have only shown how 1ML recovers constructs well-known from ML. As a first example of something that cannot directly be expressed in conventional ML, consider first-class polymorphic arguments:

```
f (id : (a : type) ⇒ a → a) = {x = id int 5; y = id bool true}
```

Similarly, existential types are directly expressible:

```
type SHAPE = {type t; area : t → float; v : t}
volume (height : int) (x : SHAPE) = height * x.area (x.v)
```

SHAPE can either be read as a module signature or an existential type, both are indistinguishable. The function volume is agnostic about the actual type of the shape it is given.

It turns out that the previous examples can still be expressed with packaged modules (Section 1.1). But now consider:

```
type COLL c =
{
  type key;
  type val;
  empty : c;
  add : c → key → val → c;
  lookup : c → key → opt val;
  keys : c → list key
};
entries c (C : COLL c) (xs : c) : list (C.key × C.val) = ...
```

COLL amounts to a *parameterised signature*, and is akin to a Haskell-style type class [34]. It contains two abstract type specifications, which are known as *associated types* in the type class literature (or in C++ land). The function entries is parameterised over a corresponding module C – an (explicit) type class instance if you want. Its result type depends directly on C's definition of the associated types. Such a dependency can be expressed in ML on the module level, but not at the core level.[2]

Moving to higher kinds, things become even more interesting:

```
type MONAD (m : type ⇒ type) =
{
  return a : a → m a;
  bind a b : m a → (a → m b) → m b
};
map a b (m : type ⇒ type) (M : MONAD m) (f : a → b) (mx : m a) =
  M.bind a b mx (fun (x : a) ⇒ M.return b (f x))  (* : m b *)
```

Here, MONAD is again akin to a type class, but over a type constructor. As explained in Section 1.1, this kind of polymorphism cannot be expressed even in MLs with packaged modules.

Computed Modules Just for completeness, we should mention that the motivating example from Section 1 can of course be written (almost) as is in $1\mathrm{ML}_{\mathrm{ex}}$:

```
Table = if size > threshold then HashMap else TreeMap : MAP
```

The only minor nuisance is the need to annotate the type of the conditional. As explained earlier, the annotation is necessary in general to achieve unique types, but can usually be inferred once we add inference to the mix (Section 4).

Predicativity What is the restriction we employ to maintain decidability? It is simple: during subtyping (a.k.a. signature matching) the type **type** can only be matched by *small* types, which are those that do not themselves contain the type **type**; or in other words, monomorphic types. Small types thus exclude first-class abstract types, actual functors (functions taking type parameters), and type constructors (which are just functors). For example, all of the following define *large* types:

```
type T_1 = type;                type T_4 = (x : {}) → type;
type T_2 = {type u};            type T_5 = (a : type) ⇒ {};
type T_3 = {type u = T_2};      type T_6 = {type u a = bool};
```

None of these are expressible as type expressions in conventional ML, and vice versa, all ML type expressions materialise as small types in 1ML, so nothing is lost in comparison.

The restriction on subtyping affects annotations, parameterisation over types, and the formation of abstract types. For example, for all of the above T_i, all of the following definitions are ill-typed:

```
type U = pair T_i T_i;  (* error *)
A = (type T_i) : type;  (* error *)
B = {type u = T_i} :> {type u};  (* error *)
C = if b then T_i else int : type  (* error *)
```

Notably, the case A with T_1 literally implies **type type** $\not:$ **type** (although **type type** itself *is* a well-formed expression!). The main challenge with first-class modules is preventing such a type:type situation, and the separation into a small universe (denoted by **type**) and a large one (for which no syntax exists) achieves that.

A *transparent* type is small as long as it reveals a small type:

```
type T'_1 = (= type int);
type T'_2 = {type u = int}
```

would *not* cause an error when inserted into the above definitions.

[2] In OCaml 4, this example can be approximated with heavy fibration:

```
module type COLL = sig type coll type key type val ... end
let entries (type c) (type k) (type v)
        (module C : COLL with
            type coll = c and type key = k and type value = v)
        (xs : c) : (k * v) list = ...
```

Recursion The 1ML$_{\text{ex}}$ syntax we give in Figure 1 omits a couple of constructs that one can rightfully expect from any serious ML contender: in particular, there is no form of recursion, neither for terms nor for types. It turns out that those are largely orthogonal to the overall design of 1ML, so we only sketch them here.

ML-style recursive functions can be added simply by throwing in a primitive polymorphic fixpoint operator

$$\mathsf{fix\ a\ b} : (\mathsf{a} \to \mathsf{b}) \to (\mathsf{a} \to \mathsf{b})$$

plus perhaps some suitable syntactic sugar:

$$\begin{array}{l}\textbf{rec}\, X\, \overline{Y}\, (Z{:}T) : U{=}E \quad := \\ X = \textbf{fun}\, \overline{Y} \Rightarrow \mathsf{fix}\, T\, U\, (\textbf{fun}(X{:}(Z{:}T) \to T') \Rightarrow \textbf{fun}(Z{:}T) \Rightarrow E)\end{array}$$

Given an appropriate fixpoint operator, this generalises to mutually recursive functions in the usual ways. Note how the need to specify the result type b (respectively, U) prevents using the operator to construct transparent recursive types, because U has no way of referring to the result of the fixpoint. Moreover, fix yields an impure function, so even an attempt to define an abstract type recursively,

$$\textbf{rec}\ \mathsf{stream}\ (\mathsf{a} : \textbf{type}) : \textbf{type} = \textbf{type}\ \{\mathsf{head} : \mathsf{a};\ \mathsf{tail} : \mathsf{stream\ a}\}$$

won't type-check, because stream wouldn't be an applicative functor, and so the term stream a on the right-hand side is not a valid type — fortunately, because there would be no way to translate such a definition into System F$_\omega$ with a conventional fixpoint operator.

Recursive (data)types have to be added separately. One approach, that has been used by Harper & Stone's type-theoretic account of Standard ML [13], is to interpret a recursive datatype like

$$\textbf{datatype}\ \mathsf{t} = \mathsf{A} \mid \mathsf{B}\ \textbf{of}\ T$$

as a module defining a primitive ADT with the signature

$$\{\textbf{type}\ \mathsf{t};\ \mathsf{A} : \mathsf{t};\ \mathsf{B} : T \Rightarrow \mathsf{t};\ \mathsf{expose\ a} : (\{\} \to \mathsf{a}) \Rightarrow (T \to \mathsf{a}) \Rightarrow \mathsf{t} \to \mathsf{a}\}$$

where expose is a case-operator accessed by pattern matching compilation. We refer to [13] for more details on this approach. There is one caveat, though: datatypes expressed as ADTs require sealing. With the simple system presented in this paper, they hence could not be defined inside applicative functors. However, this limitation is removed by the aforementioned generalisation to pure sealing described in the Technical Appendix [23].

Impredicativity Reloaded Predicativity is a severe restriction. Can we enable impredicative type abstraction without breaking decidability? Yes we can. One possibility is the usual trick of piggy-backing datatypes: we can allow their data constructors to have large parameters. Because datatypes are *nominal* in ML, impredicativity is "hidden away" and does not interfere with subtyping.

Structural impredicative types are also possible, as long as large types are injected into the small universe *explicitly*, by way of a special type, say, "**wrap** T". The gist of this approach is that subtyping does not extend to such wrapped types. It is an easy extension, the Technical Appendix [23] gives the details.

3. Type System and Elaboration

So much for leisure, now for work. The general recipe for 1ML$_{\text{ex}}$ is simple: take the semantics from F-ing modules [25], collapse the levels of modules and core, and impose the predicativity restriction needed to maintain decidability. This requires surprisingly few changes to the whole system. Unfortunately, space does not permit explaining all of the F-ing semantics in detail, so we encourage the reader to refer to [25] (mostly Section 4) for background, and will focus primarily on the differences and novelties in what follows.

3.1 Internal Language

System F$_\omega$ The semantics is defined by elaborating 1ML$_{\text{ex}}$ types and terms into types and terms of (call-by-value, impredicative) System F$_\omega$, the higher-order polymorphic λ-calculus [1], extended with simple record types (Figure 2). The semantics is completely standard; we omit it here and reuse the formulation from [25]. The only point of note is that it allows term (but not type) variables in the environment Γ to be shadowed without α-renaming, which is convenient for translating bindings.

$$\begin{array}{llcl}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa \to \kappa \\
\text{(types)} & \tau & ::= & \alpha \mid \tau \to \tau \mid \{\overline{l{:}\tau}\} \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \\
 & & & \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau \\
\text{(terms)} & e, f & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid \{\overline{l{=}e}\} \mid e.l \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \\
 & & & \mathsf{pack}\ \langle\tau, e\rangle_\tau \mid \mathsf{unpack}\ \langle\alpha, x\rangle{=}e\ \mathsf{in}\ e \\
\text{(environ's)} & \Gamma & ::= & \cdot \mid \Gamma, \alpha{:}\kappa \mid \Gamma, x{:}\tau
\end{array}$$

Figure 2. Syntax of F$_\omega$

$$\begin{array}{llcl}
\text{(abstracted)} & \Xi & ::= & \exists\overline{\alpha}.\Sigma \\
\text{(large)} & \Sigma & ::= & \pi \mid \mathsf{bool} \mid [= \Xi] \mid \{\overline{l{:}\Sigma}\} \mid \forall\overline{\alpha}.\Sigma \to_\iota \Xi \\
\text{(small)} & \sigma & ::= & \pi \mid \mathsf{bool} \mid [= \sigma] \mid \{\overline{l{:}\sigma}\} \mid \sigma \to_{\mathtt{I}} \sigma \\
\text{(paths)} & \pi & ::= & \alpha \mid \pi\,\overline{\sigma} \\
\text{(purity)} & \iota & ::= & \mathtt{P} \mid \mathtt{I}
\end{array}$$

Desugarings into F$_\omega$:

$$\begin{array}{lcl@{\qquad}lcl}
\text{(types)} & & & \text{(terms)} & & \\
[= \tau] & := & \{\mathsf{typ} : \tau \to \{\}\} & [\tau] & := & \{\mathsf{typ} = \lambda x{:}\tau.\{\}\} \\
\tau_1 \to_l \tau_2 & := & \tau_1 \to \{l : \tau_2\} & \lambda_l x{:}\tau.e & := & \lambda x{:}\tau.\{l : e\}
\end{array}$$

$$\begin{array}{llll}
\text{Notation:} & \iota \leq \iota & \iota \vee \iota := \iota & \iota(\Sigma) \qquad\quad = \mathtt{P} \\
 & \mathtt{P} \leq \mathtt{I} & \mathtt{P} \vee \mathtt{I} := \mathtt{I} \vee \mathtt{P} := \mathtt{I} & \iota(\exists\alpha\overline{\alpha}.\Sigma) = \mathtt{I} \\
\tau.\bar{l} := \tau & & \tau[.\bar{l}{=}\tau_2] := \tau_2 & (\bar{l} = \epsilon) \\
\{l{:}\tau, ...\}.\bar{l} := \tau.\bar{l}' & & \{l{:}\tau, ...\}[.\bar{l}{=}\tau_2] := \{l{:}\tau[.\bar{l}'{=}\tau_2], ...\} & (\bar{l} = l.\bar{l}')
\end{array}$$

Figure 3. Semantic Types

We write $\Gamma \vdash e : \tau$ for the F$_\omega$ typing judgement, and let $e \hookrightarrow e'$ denote (one-step) reduction. Then System F$_\omega$ is well-known to enjoy the standard soundness properties:

THEOREM 3.1 (Preservation).
If $\cdot \vdash e : \tau$ and $e \hookrightarrow e'$, then $\cdot \vdash e' : \tau$.

THEOREM 3.2 (Progress).
If $\cdot \vdash e : \tau$ and e is not a value, then $e \hookrightarrow e'$ for some e'.

To establish soundness of 1ML it suffices to ensure that elaboration always produces well-typed F$_\omega$ terms (Section 3.3).

We assume obvious encodings of let-expressions and n-ary universal and existential types in F$_\omega$. To ease notation we often drop type annotations from let, pack, and unpack where clear from context. We will also omit kind annotations on type variables, and where necessary, use the notation κ_α to refer to the kind implicitly associated with α.

Semantic Types Elaboration translates 1ML$_{\text{ex}}$ types directly into "equivalent" System F$_\omega$ types. The shape of these *semantic* types is given by the grammar in Figure 3.

The main magic of the elaboration is that it inserts appropriate quantifiers to bind abstract types. Following Mitchell & Plotkin [20], abstract types are represented by existentials: an *abstracted* type $\Xi = \exists\overline{\alpha}.\Sigma$ quantifies over all the abstract types (i.e., components of type **type**) from the underlying *concretised* type Σ, by naming them $\overline{\alpha}$. Inside Σ they can hence be represented as transparent types, equal to those $\overline{\alpha}$'s.

A sketch of the mapping between syntactic types T and semantic types Ξ is as follows:

$$\begin{array}{rcl}
T & \leadsto & \exists\overline{\alpha}.\Sigma \\
\hline
(= \textbf{type}\ T_1) & \leadsto & [= \exists\overline{\alpha}_1.\Sigma_1] \\
\textbf{type} & \leadsto & \exists\alpha.[= \alpha] \\
\{X_1{:}T_1; X_2{:}T_2\} & \leadsto & \exists\overline{\alpha}_1\overline{\alpha}_2.\{X_1{:}\Sigma_1, X_2{:}\Sigma_2\} \\
(X{:}T_1) \to T_2 & \leadsto & \forall\overline{\alpha}_1.\Sigma_1 \to_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2 \\
(X{:}T_1) \Rightarrow T_2 & \leadsto & \exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\Sigma_1 \to_{\mathsf{P}} \Sigma_2 \\
\mathsf{A.t} & \leadsto & \alpha_{\mathsf{A.t}} \\
\mathsf{F}(M) & \leadsto & \alpha_{\mathsf{F}(\_)}\,\overline{\sigma}_M
\end{array}$$

Here, we assume that each constituent type T_i on the left-hand side is recursively mapped to a corresponding $\exists\overline{\alpha}_i.\Sigma_i$ appearing on the right-hand side.

Walking through these in turn, (transparent) reified types are represented as $[= \Xi]$, which is expressed in System F using a simple coding trick [25] – cf. the desugaring of $[= \tau]$ and $[\tau]$ given in Figure 3, assuming a reserved label "typ". Because all type constructors are represented as functors, we have no need for reified types of higher kind (as was the case in [25]).

With all abstract types being named, they always appear as transparent types $[= \alpha]$ as well, albeit quantified as necessary.

Records, no surprise, map to records. We assume an implicit injection from 1ML identifiers X into both F_ω variables x and labels l, so we can conveniently treat any X as a variable or label. The abstract type names from all record components (here, the $\overline{\alpha}_1$ from T_1 and the $\overline{\alpha}_2$ from T_2) are collectively hoisted outside the record; within, the components all have concretised types, respectively. In particular, this makes $\overline{\alpha}_1$ scope over Σ_2, thereby allowing possible dependencies of T_2 on (abstract types from) T_1 without requiring actual dependent types.

Function types map to polymorphic functions in F_ω. Being in negative position, the existential quantifier for the abstract types $\overline{\alpha}_1$ from the parameter type Σ_1 turns into a universal quantifier, scoping over the whole type, and allowing the result type Σ_2 to refer to the parameter types. Like for records, this hoisting avoids the need for dependent types. Functions are also annotated by a simple *effect* ι, which distinguishes impure ($\to$) from pure ($\Rightarrow$) function types, and thus, generative from applicative functors.

Pure function types encode applicative semantics for the abstract types they return by having their existential quantifiers $\overline{\alpha}_2$ "lifted" over their parameters. To capture potential dependencies, the $\overline{\alpha}_2$ are skolemised over $\overline{\alpha}_1$ [2, 28, 25]. That is, the kinds of $\overline{\alpha}_2$ are of the form $\overline{\kappa_{\alpha_1}} \to \kappa$ for pure functors, which is where higher kinds come into play. We impose the syntactic invariant that a pure function type never has an existential quantifier right of the arrow.

Abstract types are denoted by their type variables – e.g. some $\alpha_{\mathsf{A.t}}$ introduced for $\mathsf{A.t}$ – but may generally take the form of a *semantic path* π if they have parameters. Parameters are (only) introduced through pure function abstraction and the aforementioned kind lifting that goes along with it. An abstract type that is the result of an application of a pure function (applicative functor, or type constructor) F to a value (module) M becomes the application of a higher-kinded type variable representing the constructor to the concrete types $\overline{\sigma}_M$ from the argument, corresponding to the abstract types $\overline{\alpha}_1$ in F's parameter. Because we enforce predicativity, these argument types have to be small. For example, the type constructor map (Section 2) has semantic type $\forall\alpha.[= \alpha] \to_{\mathsf{P}} [= \alpha_{\mathsf{map}}(\alpha)]$, and the application $\mathsf{map\ int}$ translates to $\alpha_{\mathsf{map}}(\mathsf{int})$.

The latter forms can appear in arbitrary combination: for instance, an abstract type projected from a functor application, $\mathsf{G}(M).\mathsf{t}$, would map to $\alpha_{\mathsf{G}(\_).\mathsf{t}}\,\overline{\sigma}_M$ accordingly.

Figure 3 also defines the subgrammar of small types, which cannot have quantifiers in them. Moreover, small functions are required to be impure, which will simplify type inference (Section 5).

3.2 Elaboration

The complete elaboration rules for $1\mathrm{ML}_{\mathrm{ex}}$ are collected in Figure 4. There is one judgement for each syntactic class, plus an auxiliary judgement for subtyping. If you are merely interested in typing 1ML then you can ignore the greyed out parts "$\leadsto e$" in the rules – they are concerned with the translation of terms, and are only relevant to define the operational semantics of the language.

Types and Declarations The main job of the elaboration rules for types is to name all abstract type components with type variables, collect them, and bind them hoisted to an outermost existential (or universal, in the case of functions) quantifier. The rules are mostly identical to [25], except that **type** is a free-standing construct instead of being tied to the syntax of bindings, and 1ML's "**where**" construct requires a slightly more general rule.

Rule TSING corresponds to rule S-LIKE in [25] and handles "singleton" types. It simply infers the (unique) type Σ of the expression E. Note that this type is not allowed to have existential quantifiers, i.e., E may not introduce local abstract types. All types $[= \Xi]$ occurring in Σ thus are transparent. As explained below, we dropped the side condition for Σ to be *explicit* in this rule.

Expressions and Bindings The elaboration of expressions closely follows the rules from the first part of [25], but adds the tracking of purity as in Section 7 of that paper. However, to keep the current paper simple, we left out the ability to perform pure sealing, or to create pure functions around it. That avoids some of the notational contortions necessary for the applicative functor semantics from [25]. An extension of $1\mathrm{ML}_{\mathrm{ex}}$ with pure sealing can be found in the Technical Appendix [23].

The only other non-editorial changes over [25] are that "**type** T" is now handled as a first-class value, no longer tied to bindings, and that Booleans have been added as representatives of the core.

The rules collect all abstract types generated by an expression (e.g. by sealing or by functor application) into an existential package. This requires repeated unpacking and repacking of existentials created by constituent expressions. Moreover, the sequencing rule BSEQ combines two (n-ary) existentials into one.

It is an invariant of the expression elaboration judgement that $\iota = \mathtt{I}$ if Ξ is not a concrete type Σ – i.e., abstract type "generation" is impure. Without this invariant, rule EFUN might form an invalid function type that is marked pure but yet has an inner existential quantifier (i.e., is "generative"). To maintain the invariant, both sealing (rule ESEAL) and conditionals (rule EIF) have to be deemed impure if they generate abstract types – enforced by the notation $\iota(\Xi)$ defined in Figure 3. In that sense, our notion of purity actually corresponds to the stronger property of *valuability* in the parlance of Dreyer [4], which also implies *phase separation*, i.e., the ability to separate static type information from dynamic computation, key to avoiding the need for dependent types.

Subtyping The subtyping judgement is defined on semantic types. It generates a coercion function f as computational evidence of the subtyping relation. The domain of that function always is the left-hand type Ξ'; to avoid clutter, we omit its explicit annotation from the λ-terms produced by the rules. The rules mostly follow the structure from [25], merely adding a straightforward rule for abstract type paths π, which now may occur as "module types".

However, we make one structural change: instead of guessing the substitution for the right-hand side's abstract types non-deterministically in a separate rule (rule U-MATCH in [25]), the current formulation looks them up algorithmically as it goes, using the new rule SFORGET to match an individual abstract type. The reason for this change is merely a technical one: it eliminates the need for any significant meta-theory about decidability, which was somewhat non-trivial before, at least with applicative functors.

Types $\boxed{\Gamma \vdash T \rightsquigarrow \Xi}$

$$\frac{\Gamma \vdash E :_{\mathrm{P}} [= \Xi] \rightsquigarrow e}{\Gamma \vdash E \rightsquigarrow \Xi}\textsc{TPath} \qquad \frac{\kappa_\alpha = \Omega}{\Gamma \vdash \textbf{type} \rightsquigarrow \exists\alpha.[= \alpha]}\textsc{TType} \qquad \frac{}{\Gamma \vdash \textbf{bool} \rightsquigarrow \mathsf{bool}}\textsc{TBool} \qquad \frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi}\textsc{TStr}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2\end{array}}{\Gamma \vdash (X{:}T_1) \to T_2 \rightsquigarrow \forall\overline{\alpha}_1.\, \Sigma_1 \to_{\mathrm{I}} \exists\overline{\alpha}_2.\Sigma_2}\textsc{TFun} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2 \qquad \overline{\kappa_{\alpha'_2} = \overline{\kappa_{\alpha_1}} \to \kappa_{\alpha_2}}\end{array}}{\Gamma \vdash (X{:}T_1) \Rightarrow T_2 \rightsquigarrow \exists\overline{\alpha}'_2.\forall\overline{\alpha}_1.\, \Sigma_1 \to_{\mathrm{P}} \Sigma_2[\overline{\alpha'_2\,\overline{\alpha}_1}/\overline{\alpha}_2]}\textsc{TPfun}$$

$$\frac{\Gamma \vdash E :_{\mathrm{P}} \Sigma \rightsquigarrow e}{\Gamma \vdash (= E) \rightsquigarrow \Sigma}\textsc{TSing} \qquad \frac{\begin{array}{ll}\Gamma \vdash T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 & \overline{\alpha}_1 = \overline{\alpha}_{11} \uplus \overline{\alpha}_{12} \\ \Gamma \vdash T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2 & \Gamma, \overline{\alpha}_{11}, \overline{\alpha}_2 \vdash \Sigma_2 \leq_{\overline{\alpha}_{12}} \Sigma_1.\overline{X} \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash T_1 \ \textbf{where}\ (.\overline{X}{:}T_2) \rightsquigarrow \exists\overline{\alpha}_{11}\overline{\alpha}_2.\delta\Sigma_1[.\overline{X}{=}\Sigma_2]}\textsc{TWhere}$$

Declarations $\boxed{\Gamma \vdash D \rightsquigarrow \Xi}$

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma}{\Gamma \vdash X{:}T \rightsquigarrow \exists\overline{\alpha}.\{X{:}\Sigma\}}\textsc{DVar} \qquad \frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\}}{\Gamma \vdash \textbf{include}\ T \rightsquigarrow \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\}}\textsc{DIncl}$$

$$\frac{\begin{array}{c}\Gamma \vdash D_1 \rightsquigarrow \exists\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash D_2 \rightsquigarrow \exists\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \qquad \overline{X}_1 \cap \overline{X}_2 = \emptyset\end{array}}{\Gamma \vdash D_1;D_2 \rightsquigarrow \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X_1{:}\Sigma_1}, \overline{X_2{:}\Sigma_2}\}}\textsc{DSeq} \qquad \frac{}{\Gamma \vdash \epsilon \rightsquigarrow \{\}}\textsc{DEmpty}$$

Expressions $\boxed{\Gamma \vdash E :_\iota \Xi \rightsquigarrow e}$

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X :_{\mathrm{P}} \Sigma \rightsquigarrow X}\textsc{EVar} \qquad \frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \textbf{type}\ T :_{\mathrm{P}} [= \Xi] \rightsquigarrow [\Xi]}\textsc{EType} \qquad \frac{}{\Gamma \vdash \textbf{true} :_{\mathrm{P}} \mathsf{bool} \rightsquigarrow \mathsf{true}}\textsc{ETrue}$$

$$\frac{}{\Gamma \vdash \textbf{false} :_{\mathrm{P}} \mathsf{bool} \rightsquigarrow \mathsf{false}}\textsc{EFalse} \qquad \frac{\begin{array}{lll}\Gamma \vdash X :_{\mathrm{P}} \mathsf{bool} \rightsquigarrow e & \Gamma \vdash E_1 :_{\iota_1} \Xi_1 \rightsquigarrow e_1 & \Gamma \vdash \Xi_1 \leq \Xi \rightsquigarrow f_1 \\ \Gamma \vdash T \rightsquigarrow \Xi & \Gamma \vdash E_2 :_{\iota_2} \Xi_2 \rightsquigarrow e_2 & \Gamma \vdash \Xi_2 \leq \Xi \rightsquigarrow f_2\end{array}}{\Gamma \vdash \textbf{if}\ X\ \textbf{then}\ E_1\ \textbf{else}\ E_2 : T :_{\iota_1 \vee \iota_2 \vee \iota(\Xi)} \Xi \rightsquigarrow \mathsf{if}\ e\ \mathsf{then}\ f_1\ e_1\ \mathsf{else}\ f_2\ e_2}\textsc{EIf}$$

$$\frac{\Gamma \vdash B :_\iota \Xi \rightsquigarrow e}{\Gamma \vdash \{B\} :_\iota \Xi \rightsquigarrow e}\textsc{EStr} \qquad \frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\{\overline{X'{:}\Sigma'}\} \rightsquigarrow e \qquad X{:}\Sigma \in \overline{X'{:}\Sigma'}}{\Gamma \vdash E.X :_\iota \exists\overline{\alpha}.\Sigma \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, y.X\rangle}\textsc{EDot}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash E :_\iota \Xi \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\,(X{:}T) \Rightarrow E :_{\mathrm{P}} \forall\overline{\alpha}.\, \Sigma \to_\iota \Xi \rightsquigarrow \lambda\overline{\alpha}.\lambda_\iota X{:}\Sigma.e}\textsc{EFun} \qquad \frac{\begin{array}{ll}\Gamma \vdash X_1 :_{\mathrm{P}} \forall\overline{\alpha}.\, \Sigma_1 \to_\iota \Xi \rightsquigarrow e_1 & \\ \Gamma \vdash X_2 :_{\mathrm{P}} \Sigma_2 \rightsquigarrow e_2 & \Gamma \vdash \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash X_1\, X_2 :_\iota \delta\Xi \rightsquigarrow (e_1\ (\delta\overline{\alpha})\ (f\ e_2)).\iota}\textsc{EApp}$$

$$\frac{\Gamma \vdash X :_{\mathrm{P}} \Sigma_1 \rightsquigarrow e \qquad \Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma_2 \qquad \Gamma \vdash \Sigma_1 \leq_{\overline{\alpha}} \Sigma_2 \rightsquigarrow \delta; f}{\Gamma \vdash X{:>}T :_{\iota(\exists\overline{\alpha}.\Sigma_2)} \exists\overline{\alpha}.\Sigma_2 \rightsquigarrow \mathsf{pack}\ \langle\delta\overline{\alpha}, f\ e\rangle}\textsc{ESeal}$$

Bindings $\boxed{\Gamma \vdash B :_\iota \Xi \rightsquigarrow e}$

$$\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\Sigma \rightsquigarrow e}{\Gamma \vdash X{=}E :_\iota \exists\overline{\alpha}.\{X{:}\Sigma\} \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \{X{=}x\}\rangle}\textsc{BVar} \qquad \frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \rightsquigarrow e}{\Gamma \vdash \textbf{include}\ E :_\iota \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \rightsquigarrow e}\textsc{BIncl}$$

$$\frac{\begin{array}{rl}\Gamma \vdash B_1 :_{\iota_1} \exists\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \rightsquigarrow e_1 & \overline{X}'_1 = \overline{X}_1 - \overline{X}_2 \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash B_2 :_{\iota_2} \exists\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \rightsquigarrow e_2 & \overline{X'_1{:}\Sigma'_1} \subseteq \overline{X_1{:}\Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash B_1;B_2 :_{\iota_1 \vee \iota_2} \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X'_1{:}\Sigma'_1}, \overline{X_2{:}\Sigma_2}\} \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in}\ \mathsf{let}\ \overline{X_1 = y_1.X_1}\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{unpack}\ \langle\overline{\alpha}_2, y_2\rangle = e_2\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{pack}\ \langle\overline{\alpha}_1\overline{\alpha}_2, \{\overline{X'_1 = y_1.X'_1}, \overline{X_2 = y_2.X_2}\}\rangle\end{array}}\textsc{BSeq} \qquad \frac{}{\Gamma \vdash \epsilon :_{\mathrm{P}} \{\} \rightsquigarrow \{\}}\textsc{BEmpty}$$

Subtyping $\Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f \ := \ \Gamma \vdash \Xi \leq_\epsilon \Xi' \rightsquigarrow \mathrm{id}; f$ $\boxed{\Gamma \vdash \Xi' \leq_{\overline{\pi}} \Xi \rightsquigarrow \delta; f}$

$$\frac{}{\Gamma \vdash \pi \leq \pi \rightsquigarrow \lambda x.x}\textsc{SPath} \qquad \frac{}{\Gamma \vdash \mathsf{bool} \leq \mathsf{bool} \rightsquigarrow \lambda x.x}\textsc{SBool}$$

$$\frac{\Gamma \vdash \Xi' \leq \Xi \rightsquigarrow f \qquad \Gamma \vdash \Xi \leq \Xi' \rightsquigarrow f'}{\Gamma \vdash [= \Xi'] \leq [= \Xi] \rightsquigarrow \lambda x.[\Xi]}\textsc{SType} \qquad \frac{\pi = \alpha\,\overline{\alpha}'}{\Gamma \vdash [= \sigma] \leq_\pi [= \pi] \rightsquigarrow [\lambda\overline{\alpha}'.\sigma/\alpha]; \lambda x.x}\textsc{SForget}$$

$$\frac{}{\Gamma \vdash \{\overline{l{:}\Sigma'}\} \leq \{\} \rightsquigarrow \lambda x.\{\}}\textsc{SEmpty} \qquad \frac{\begin{array}{c}\Gamma \vdash \Sigma'_1 \leq_{\overline{\pi}_1} \Sigma_1 \rightsquigarrow \delta_1; f_1 \\ \Gamma \vdash \{\overline{l'{:}\Sigma'}\} \leq_{\overline{\pi}_2} \{\overline{l{:}\,\delta_1\Sigma}\} \rightsquigarrow \delta_2; f_2 \qquad \delta_2\Sigma_1 = \Sigma_1\end{array}}{\Gamma \vdash \{l_1{:}\Sigma'_1, \overline{l'{:}\Sigma'}\} \leq_{\overline{\pi}_1\overline{\pi}_2} \{l_1{:}\Sigma_1, \overline{l{:}\Sigma}\} \rightsquigarrow \delta_1\delta_2; \lambda x.\{l_1{=}f_1(x.l_1), \overline{l{=}(f_2\,x).l}\}}\textsc{SStr}$$

$$\frac{\begin{array}{ll}\Gamma, \overline{\alpha} \vdash \Sigma \leq_{\overline{\alpha}'} \Sigma' \rightsquigarrow \delta_1; f_1 & \iota' \leq \iota \\ \Gamma, \overline{\alpha} \vdash \delta_1\Xi' \leq_{\overline{\pi\overline{\alpha}}} \Xi \rightsquigarrow \delta_2; f_2 & \delta_2\Sigma = \Sigma\end{array}}{\begin{array}{l}\Gamma \vdash (\forall\overline{\alpha}'.\Sigma' \to_{\iota'} \Xi') \leq_{\overline{\pi}} (\forall\overline{\alpha}.\Sigma \to_\iota \Xi) \\ \quad \rightsquigarrow \delta_2; \lambda x.\,\lambda\overline{\alpha}.\,\lambda_\iota y{:}\Sigma.\, f_2\,((x\,(\delta_1\overline{\alpha}')\,(f_1\,y)).\iota')\end{array}}\textsc{SFun} \qquad \frac{\Gamma, \overline{\alpha}' \vdash \Sigma' \leq_{\overline{\alpha}} \Sigma \rightsquigarrow \delta; f \qquad \overline{\alpha}'\overline{\alpha} \neq \epsilon}{\Gamma \vdash \exists\overline{\alpha}'.\Sigma' \leq \exists\overline{\alpha}.\Sigma \rightsquigarrow \lambda x.\,\mathsf{unpack}\ \langle\overline{\alpha}', y\rangle = x\ \mathsf{in}\ \mathsf{pack}\ \langle\delta\overline{\alpha}, f\ y\rangle}\textsc{SAbs}$$

Figure 4. Elaboration of $1\mathrm{ML}_{\mathrm{ex}}$

To this end, the judgement is indexed by a vector $\overline{\pi}$ of abstract paths that correspond to the abstract types from the right-hand Ξ. The counterparts of those types have to be looked up in the left-hand Ξ', which happens one at a time in rule SFORGET. And that's where the predicativity restriction materialises: the rule only allows a small type on the left. Lookup produces a substitution δ whose domain corresponds to the root variables of the abstract paths $\overline{\pi}$. Normally, each of $\overline{\pi}$ is just a plain abstract type variable (which occur free in Ξ in this judgement). But in the formation rule TPFUN for pure function types, lifting produces more complex paths. So when subtyping goes inside a pure functor in rule SFUN, the same abstract paths with skolem parameters have to be formed for lookup, so that rule SFORGET can match them accordingly.

The move to deterministic subtyping allows us to drop the auxiliary notion of *explicit* types, which was present in [25] to ensure that non-deterministic lookup can be made deterministic. There is one side effect from dropping the "explicitness" side condition from rule TSING, though: subtyping is no longer reflexive. There are now "monster" types that cannot be matched, not even by themselves. For example, take $\{\} \rightarrow_{\mathtt{I}} \exists\alpha.\alpha$, which is created by

```
(= (fun (x : {}) ⇒ ({type t = int; v = 0} :> {type t; v : t}).v))
```

and is not a subtype of itself (it only contains a *use* of the abstract type α, no "binding" of the form $[= \alpha]$; consequently, when recursively matching $\exists\alpha'.\alpha' \leq \exists\alpha.\alpha$, rule SFORGET is never invoked to introduce the necessary substitution $[\alpha'/\alpha]$ of α by (the renamed version of) itself). However, this does not break anything else, so we make that simplification anyway – if desired, explicitness could easily be revived.

3.3 Meta-Theory

It is relatively straightforward to verify that elaboration is correct:

PROPOSITION 3.3 (Correctness of 1ML$_{\text{ex}}$ Elaboration).
Let Γ be a well-formed F_ω environment.

1. *If* $\Gamma \vdash T/D \rightsquigarrow \Xi$, *then* $\Gamma \vdash \Xi : \Omega$.
2. *If* $\Gamma \vdash E/B :_\iota \Xi \rightsquigarrow e$, *then* $\Gamma \vdash e : \Xi$, *and if* $\iota{=}\mathtt{P}$ *then* $\Xi{=}\Sigma$.
3. *If* $\Gamma \vdash \Xi' \leq_{\overline{\alpha\overline{\alpha}'}} \Xi \rightsquigarrow \delta; f$ *and* $\Gamma \vdash \Xi' : \Omega$ *and* $\Gamma, \overline{\alpha} \vdash \Xi : \Omega$, *then* $\mathrm{dom}(\delta) = \overline{\alpha}$ *and* $\Gamma \vdash \delta : \Gamma, \overline{\alpha}$ *and* $\Gamma \vdash f : \Xi' \rightarrow \delta\Xi$.

Together with the standard soundness result for F$_\omega$ we can tell that 1ML$_{\text{ex}}$ is sound, i.e., a well-typed 1ML$_{\text{ex}}$ program will either diverge or terminate with a value of the right type:

THEOREM 3.4 (Soundness of 1ML$_{\text{ex}}$). *If* $\cdot \vdash E : \Xi \rightsquigarrow e$, *then either* $e \uparrow$ *or* $e \hookrightarrow^* v$ *such that* $\cdot \vdash v : \Xi$ *and* v *is a value.*

More interestingly, the 1ML$_{\text{ex}}$ type system is also decidable:

THEOREM 3.5 (Decidablity of 1ML$_{\text{ex}}$ Elaboration).
All 1ML$_{\text{ex}}$ elaboration judgements are decidable.

This is immediate for all but the subtyping judgement, since they are syntax-directed and inductive, with no complicated side conditions. The rules can be read directly as an inductive algorithm. (In the case of **where**, it seems necessary to find a partitioning $\overline{\alpha}_1 = \overline{\alpha}_{11} \uplus \overline{\alpha}_{12}$, but it is not hard to see that the subtyping premise can only possibly succeed when picking $\overline{\alpha}_{12} = \mathrm{fv}(\Sigma_1) \cap \overline{\alpha}_1$.)

The only tricky judgement is subtyping. Although it is syntax-directed as well, the rules are not actually inductive: some of their premises apply a substitution δ to the inspected types. Alas, that is exactly what can cause undecidability (see Section 1.2).

The restriction to substituting small types saves the day. We can define a weight metric over semantic types such that a quantified type variable has more weight than any possible substitution of that variable *with a small type*. We can then show that the overall weight of types involved decreases in all subtyping rules. For space reasons, the details appear in the Technical Appendix [23].

4. Full 1ML

A language without type inference is not worth naming ML. Because that is so, Figure 5 shows the minimal extension to 1ML$_{\text{ex}}$ necessary to recover ML-style implicit polymorphism. Syntactically, there are merely two new forms of type expression.

First, "_" stands for a type that is to be inferred from context. The crucial restriction here is that this can only be a *small* type. This fits nicely with the notion of a *monotype* in core ML, and prevents the need to infer polymorphic types in an analogous manner.

On top of this new piece of kernel syntax we allow a type annotation ": _" on a function parameter or conditional to be omitted, thereby recovering the implicitly typed expression syntax familiar from ML. (At the same time we drop the 1ML$_{\text{ex}}$ sugar interpreting an unannotated parameter as a type; we only keep that interpretation in **type** declarations or bindings.)

Second, there is a new type of *implicit* function, distinguished by a leading tick ' (a choice that will become clear in a moment). This corresponds to an ML-style polymorphic type. The parameter has to be of type **type**, whose being small fits nicely with the fact that ML can only abstract monotypes, and no type constructors. For obvious reasons, an implicit function has to be pure. We write the semantic type of implicit functions with an arrow $\rightarrow_{\mathtt{A}}$, in order to reuse notational convention. It is distinct from $\rightarrow_\iota$, however, and we do not consider $\mathtt{A}$ an actual effect; i.e., $\mathtt{A}$ is not included in ι.

As the name would suggest, there are no explicit introduction or elimination forms for implicit functions. Instead, they are introduced and eliminated implicitly. The respective typing rules (EGEN and EINST) match common formulations of ML-style polymorphism [3]. Any pure expression can have its type generalised, which is more liberal than ML's *value restriction* [35] (recall that purity also implies that no abstract types are produced).

Subtyping allows the implicit elimination of implicit functions as well, via instantiation on the left, or skolemisation on the right (rules SIMPLL and SIMPLR). This closely corresponds to ML's signature matching rules, which allow any value to be matched by a value of more polymorphic type. However, this behaviour can now be intermixed with proper "module" types. In particular, that means that we allow looking up types from an implicit function, similar to other pure functions. For example, the following subtyping holds, by implicitly instantiating the parameter a with int:

```
'(a : type) ⇒ {type t = a; f : a → t}  ≤  {type t; f : int → int}
```

With these few extensions, the Map functor from Section 2 can now be written in 1ML very much like in traditional ML:

```
type MAP =
{
  type key;
  type map a;
  empty 'a : map a;
  lookup 'a : key → map a → opt a;
  add 'a : key → a → map a → map a
};
Map (Key : EQ) :> MAP where (type .key = Key.t) =
{
  type key = Key.t;
  type map a = key → opt a;
  empty = fun x ⇒ none;
  lookup x m = m x;
  add x y m = fun z ⇒ if Key.eq z x then some y else m z
}
```

The MAP signature here uses one last bit of syntactic sugar defined in Figure 5, which is to allow implicit parameters on the left-hand side of declarations, like we already do for explicit parameters (cf. Figure 1), The tick becomes a pun on ML's type variable syntax, but without relying on brittle implicit scoping rules.

Syntax

$$\text{(types)} \quad T \quad ::= \quad \ldots \mid \_ \mid {}'(X{:}\textbf{type}) \Rightarrow T$$

$$\begin{array}{llcl}
\text{(expressions)} & \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 & := & \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 {:}\ \_ \\
 & \textbf{fun}\, X \Rightarrow E & := & \textbf{fun}\,(X{:}\ \_\,) \Rightarrow E \\
\text{(types)} & {}'X \Rightarrow T & := & {}'(X{:}\textbf{type}) \Rightarrow T \\
\text{(declarations)} & X\,\overline{{}'Y}{:}T & := & X : \overline{{}'(Y{:}\,\textbf{type})} \Rightarrow T
\end{array}$$

Semantic Types

$$\text{(large signatures)} \quad \Sigma \quad ::= \quad \ldots \mid \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma$$

Types $\boxed{\Gamma \vdash T \rightsquigarrow \Xi}$

$$\frac{\Gamma \vdash \sigma : \Omega}{\Gamma \vdash \_ \rightsquigarrow \sigma}\textsc{TInfer} \qquad \frac{\Gamma, \alpha, X{:}[=\alpha] \vdash T \rightsquigarrow \Sigma \qquad \kappa_\alpha = \Omega}{\Gamma \vdash {}'(X{:}\textbf{type}) \Rightarrow T \rightsquigarrow \forall\alpha.\{\} \to_{\mathtt{A}} \Sigma}\textsc{TImpl}$$

Expressions $\boxed{\Gamma \vdash E :_\iota \Xi \rightsquigarrow e}$

$$\frac{\Gamma, \overline{\alpha} \vdash E :_{\mathtt{P}} \Sigma \rightsquigarrow e \qquad \overline{\kappa_\alpha = \Omega}}{\Gamma \vdash E :_{\mathtt{P}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow \lambda\overline{\alpha}.\lambda_{\mathtt{A}} x{:}\{\}.e}\textsc{EGen} \qquad \frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow e \qquad \overline{\Gamma, \overline{\alpha} \vdash \sigma : \kappa_{\alpha'}}}{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\Sigma[\overline{\sigma}/\overline{\alpha}'] \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, (x\,\overline{\sigma}\,\{\}).\mathtt{A}\rangle}\textsc{EInst}$$

Subtyping $\boxed{\Gamma \vdash \Xi' \leq_\pi \Xi \rightsquigarrow \delta; f}$

$$\frac{\overline{\Gamma \vdash \sigma : \kappa_{\alpha'}} \qquad \Gamma \vdash \Sigma'[\overline{\sigma}/\overline{\alpha}'] \leq_\pi \Sigma \rightsquigarrow \delta; f}{\Gamma \vdash \forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \leq_\pi \Sigma \rightsquigarrow \delta; \lambda x.\, f\,((x\,\overline{\sigma}\,\{\}).\mathtt{A})}\textsc{SImplL} \qquad \frac{\Gamma, \overline{\alpha} \vdash \Sigma' \leq_\pi \Sigma \rightsquigarrow \delta; f \qquad \overline{\mathrm{fv}(\delta\pi) \not\cap \overline{\alpha}}}{\Gamma \vdash \Sigma' \leq_\pi \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow \delta; \lambda x.\, \lambda\overline{\alpha}.\lambda_{\mathtt{A}} y{:}\{\}.f\,x}\textsc{SImplR}$$

Figure 5. Extension to Full 1ML

Space reasons forbid more extensive examples, but it should be clear from the rules that there is nothing preventing the use of implicit functions as first-class values, given sufficient annotations for their (large) types. For example:

(**fun** (id : 'a ⇒ a → a) ⇒ {x = id 3; y = id true}) (**fun** x ⇒ x)

The type of the argument expression is generalised implicitly and matches the implicitly polymorphic parameter via subtyping.

5. Type Inference

With the additions from Figure 5 we have turned the deterministic typing and elaboration judgements of 1ML$_{\text{ex}}$ non-deterministic. They have to guess types (in rules TINFER, EINST, SIMPLL) and quantifiers (in rule EGEN). Moreover, we have decide when to apply rules EGEN and EINST. Clearly, an algorithm is needed.

Fortunately, what's going on is not fundamentally different from core ML. Where core ML would require type equivalence (and type inference would use unification), the 1ML rules require subtyping.

That may seem scary at first, but a closer inspection of the subtyping rules reveals that, when applied to small types, subtyping almost degenerates to type equivalence! The only exception is width subtyping on records. The 1ML type system only promises to infer small types, so we are not far away from conventional ML. That is, we can still formulate an algorithm based on *inference variables* (which we write υ) holding place for small types.

5.1 Algorithm

Figure 6 shows the essence of this algorithm, formulated via inference rules. The basic idea is to modify the declarative typing rules such that wherever they have to guess a (small) type, we simply introduce a (free) inference variable. Furthermore, the rules are augmented with outputting a substitution θ for resolved inference variables: all judgements have the form $\Gamma \vdash_\theta \mathcal{J}$, which, roughly, implies the respective declarative judgement $\overline{\upsilon}, \theta\Gamma \vdash \theta\mathcal{J}$, where $\overline{\upsilon}$ binds the unresolved inference variables that still appear free in $\theta\Gamma$ or $\theta\mathcal{J}$. Notation is simplified by abbreviations of the form

$$\Gamma \ {}_\theta{\vdash}_{\theta'} \mathcal{J} \quad := \quad \theta\Gamma \vdash_{\theta''} {}^\theta\mathcal{J} \ \wedge \theta' = \theta'' \circ \theta$$

where ${}^\theta\mathcal{J}$ is meant to apply θ to $\mathcal{J}$'s "inputs". It's used to thread and compose substitutions through multiple premises (e.g. rule IEIF).

There are two main complications, both due to the fact that, unlike in old ML, small types can be intermixed with large ones.

First, it may be necessary to infer a small type from a large one via subtyping. For example, we might encounter the inequation

$$\forall\alpha.[=\alpha] \to_{\mathtt{P}} [=\alpha] \quad \leq \quad \upsilon$$

which can be solved just fine with $\upsilon = [=\sigma] \to_{\mathtt{I}} [=\sigma]$ for any σ; through contravariance, similar situations can arise with an inference variable on the left. Because of this, it is not enough to just consider the cases $\upsilon \leq \sigma$ or $\sigma \leq \upsilon$ for resolving υ. Instead, when the subtyping algorithm hits $\upsilon \leq \Sigma$ or $\Sigma \leq \upsilon$ (rules ISRESL and ISRESR, where Σ may or may not be small) it invokes the auxiliary *Resolution* judgement $\Gamma \vdash_\theta \upsilon \approx \Sigma$, which only resolves υ so far as to match the shape of Σ and inserts fresh inference variables for its subcomponents. After that, subtyping "tries again".

Second, an inference variable υ can be introduced in the scope of abstract types (i.e., regular type variables). In general, it would be incorrect to resolve υ to a type containing type variables that are *not* in scope for *all* occurrences of υ in a derivation. To prevent that, each υ is associated with a set Δ_υ of type variables that are known to be in scope for υ everywhere. The set is verified when resolving υ (see rule IRPATH in particular). The set also is propagated to any other υ' the original υ is unified with, by intersecting $\Delta_{\upsilon'}$ with Δ_υ – or more precisely, by introducing a new variable υ'' with the intersected $\Delta_{\upsilon''}$, and replacing both υ and υ' with it (see e.g. rule IRINFER); that way, we can treat Δ_υ as a globally fixed set for each υ, and do not need to maintain those sets separately. Inference variables also have to be updated when type variables go out of scope. That is achieved by employing the following notation in rules locally extending Γ with type variables (we write $\mathrm{undet}(\Xi)$ to denote the free inference variables of Ξ):

$$\begin{array}{rcl}
\Gamma; \Gamma' \ {}_\theta{\vdash}_{\theta'} \mathcal{J} & := & \Gamma, \Gamma' \ {}_\theta{\vdash}_{\theta''} \mathcal{J} \ \wedge \theta' = [\overline{\upsilon}'/\overline{\upsilon}] \circ \theta'' \\
 & & \text{where } \overline{\upsilon} = \mathrm{undet}(\theta''\mathcal{J}) \\
 & & \overline{\overline{\upsilon}' \text{ fresh with } \Delta_{\upsilon'} = \Delta_\upsilon \cap \mathrm{dom}(\Gamma)}
\end{array}$$

The net effect is that all local α's from Γ' are removed from all Δ-sets of inference variable remaining after executing $\Gamma, \Gamma' \vdash \mathcal{J}$. We omit θ in this notation when it is the identity.

Implicit functions work mostly like in ML. Like with let-polymorphism, generalisation is deferred to the point where an expression is bound – in this case, in rule IBPVAR. This works despite 1ML's first-class polymorphism, thanks to the desugaring into a kernel syntax requiring named variables in most places (Figure 1). Consider the example from the previous section:

Types $\boxed{\Gamma \vdash_{\theta} T \leadsto \Xi}$

$$\frac{\Gamma \vdash^{!}_{\theta} E :_{\mathtt{P}} [= \Xi]}{\Gamma \vdash_{\theta} E \leadsto \Xi}\textsc{ITpath} \qquad \frac{\upsilon \text{ fresh} \qquad \Delta_{\upsilon} = \mathrm{dom}(\Gamma)}{\Gamma \vdash_{[]} \_ \leadsto \upsilon}\textsc{ITinfer} \qquad \frac{}{\Gamma \vdash_{[]} \textbf{type} \leadsto \exists\alpha.[= \alpha]}\textsc{ITtype} \qquad \frac{\Gamma \vdash_{\theta} E : \Sigma}{\Gamma \vdash_{\theta} (= E) \leadsto \Sigma}\textsc{ITsing}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} T_1 \leadsto \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma; \overline{\alpha}_1, X{:}\Sigma_1 \;{}_{\theta_1}{\vdash}_{\theta_2} T_2 \leadsto \exists\overline{\alpha}_2.\Sigma_2 \qquad \overline{\kappa_{\alpha'_2} = \overline{\kappa_{\alpha_1} \to \kappa_{\alpha_2}}}\end{array}}{\Gamma \vdash_{\theta_2} (X{:}T_1) \Rightarrow T_2 \leadsto \exists\overline{\alpha}'_2.\forall\overline{\alpha}_1.\,\Sigma_1 \to_{\mathtt{P}} \Sigma_2[\overline{\alpha'_2\,\overline{\alpha}_1}/\overline{\alpha}_2]}\textsc{ITpfun} \qquad \frac{\Gamma; \alpha, X{:}[= \alpha] \vdash_{\theta} T \leadsto \Sigma \qquad \kappa_{\alpha} = \Omega}{\Gamma \vdash_{\theta} {}'(X{:}\textbf{type}) \Rightarrow T \leadsto \forall\alpha.\{\} \to_{\mathtt{A}} \Sigma}\textsc{ITimpl}$$

Expressions $\boxed{\Gamma \vdash_{\theta} E :_{\iota} \Xi}$

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash_{[]} X :_{\mathtt{P}} \Sigma}\textsc{IEvar} \qquad \frac{\begin{array}{ccc}\Gamma \vdash^{!}_{\theta_0} X :_{\mathtt{P}} \mathsf{bool} & \Gamma \;{}_{\theta_0}{\vdash}_{\theta_1} E_1 :_{\iota_1} \Xi_1 & \Gamma \;{}_{\theta_3}{\vdash}_{\theta_4} \Xi_1 \leq \Xi \\ \Gamma \;{}_{\theta_2}{\vdash}_{\theta_3} T \leadsto \Xi & \Gamma \;{}_{\theta_1}{\vdash}_{\theta_2} E_2 :_{\iota_2} \Xi_2 & \Gamma \;{}_{\theta_4}{\vdash}_{\theta_5} \Xi_2 \leq \Xi\end{array}}{\Gamma \vdash_{\theta_5} \textbf{if } X \textbf{ then } E_1 \textbf{ else } E_2 : T :_{\iota_1 \vee \iota_2 \vee \iota(\Xi)} \Xi}\textsc{IEif} \qquad \frac{\Gamma \vdash^{!}_{\theta} E :_{\iota} \exists\overline{\alpha}.\{X{:}\Sigma, \overline{X'{:}\Sigma'}\}}{\Gamma \vdash_{\theta} E.X :_{\iota} \exists\overline{\alpha}.\Sigma}\textsc{IEdot}$$

$$\frac{\Gamma \vdash_{\theta_1} T \leadsto \exists\overline{\alpha}.\Sigma \qquad \Gamma; \overline{\alpha}, X{:}\Sigma \;{}_{\theta_1}{\vdash}_{\theta_2} E :_{\iota} \Xi}{\Gamma \vdash_{\theta_2} \textbf{fun}\,(X{:}T) \Rightarrow E :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma \to_{\iota} \Xi}\textsc{IEfun} \qquad \frac{\begin{array}{c}\Gamma \vdash^{!}_{\theta_1} X_1 :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma_1 \to_{\iota} \Xi \\ \Gamma \;{}_{\theta_1}{\vdash}_{\theta_2} X_2 :_{\mathtt{P}} \Sigma_2 \qquad \Gamma \;{}_{\theta_2}{\vdash}_{\theta_3} \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \leadsto \delta\end{array}}{\Gamma \vdash_{\theta_3} X_1\, X_2 :_{\iota} \delta\Xi}\textsc{IEapp}$$

Bindings $\boxed{\Gamma \vdash_{\theta} B :_{\iota} \Xi}$

$$\frac{\Gamma \vdash_{\theta} E :_{\mathtt{I}} \exists\overline{\alpha}.\Sigma}{\Gamma \vdash_{\theta} X{=}E :_{\mathtt{I}} \exists\overline{\alpha}.\{X : \Sigma\}}\textsc{IBvar} \qquad \frac{\Gamma \vdash_{\theta} E :_{\mathtt{P}} \Sigma \qquad \overline{\upsilon} = \mathrm{undet}(\theta\Sigma) - \mathrm{undet}(\theta\Gamma) \qquad \overline{\kappa_{\alpha} = \Omega}}{\Gamma \vdash_{\theta} X{=}E :_{\mathtt{P}} \{X : \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma[\overline{\alpha}/\overline{\upsilon}]\}}\textsc{IBpvar}$$

Subtyping $\boxed{\Gamma \vdash_{\theta} \Xi \leq_{\overline{\pi}} \Xi' \leadsto \delta}$

$$\frac{}{\Gamma \vdash_{[]} \upsilon \leq \upsilon}\textsc{ISrefl} \qquad \frac{\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma \qquad \Gamma \;{}_{\theta}{\vdash}_{\theta'} \upsilon \leq \Sigma}{\Gamma \vdash_{\theta'} \upsilon \leq \Sigma}\textsc{ISresl} \qquad \frac{\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma' \qquad \Gamma \;{}_{\theta}{\vdash}_{\theta'} \Sigma' \leq \upsilon}{\Gamma \vdash_{\theta'} \Sigma' \leq \upsilon}\textsc{ISresr}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \vdash_{\theta_1} \Sigma \leq_{\overline{\alpha}'} \Sigma' \leadsto \delta_1 \qquad \iota' \leq \iota \\ \Gamma; \overline{\alpha} \;{}_{\theta_1}{\vdash}_{\theta_2} \delta_1\Xi' \leq_{\overline{\pi\overline{\alpha}}} \Xi \leadsto \delta_2 \qquad \theta_2\delta_2\Sigma = \theta_2\Sigma\end{array}}{\Gamma \vdash_{\theta_2} (\forall\overline{\alpha}'.\Sigma' \to_{\iota'} \Xi') \leq_{\overline{\pi}} (\forall\overline{\alpha}.\Sigma \to_{\iota} \Xi) \leadsto \delta_2}\textsc{ISfun} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma' \leq_{\overline{\alpha}} \Sigma \leadsto \delta \qquad \overline{\alpha}'\overline{\alpha} \neq \epsilon}{\Gamma \vdash_{\theta} \exists\overline{\alpha}'.\Sigma' \leq \exists\overline{\alpha}.\Sigma}\textsc{ISabs}$$

$$\frac{\overline{\upsilon} \text{ fresh} \qquad \overline{\Delta_{\upsilon} = \mathrm{dom}(\Gamma)} \qquad \Gamma \vdash_{\theta} \Sigma'[\overline{\upsilon}/\overline{\alpha}'] \leq_{\overline{\pi}} \Sigma \leadsto \delta}{\Gamma \vdash_{\theta} \forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \leq_{\overline{\pi}} \Sigma \leadsto \delta}\textsc{ISimpll} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma' \leq_{\overline{\pi}} \Sigma \leadsto \delta; f \qquad \overline{\alpha} \not\cap \mathrm{fv}(\theta\delta)}{\Gamma \vdash_{\theta} \Sigma' \leq_{\overline{\pi}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \leadsto \delta}\textsc{ISimplr}$$

Resolution $\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma \quad := \quad \upsilon \notin \mathrm{undet}(\Sigma) \wedge \Gamma \vdash_{\theta} \upsilon \approx \Sigma$ $\boxed{\Gamma \vdash_{\theta} \upsilon \approx \Sigma}$

$$\frac{\upsilon'' \text{ fresh} \qquad \Delta_{\upsilon''} = \Delta_{\upsilon} \cap \Delta_{\upsilon'}}{\Gamma \vdash_{[\upsilon''/\upsilon, \upsilon''/\upsilon']} \upsilon \approx \upsilon'}\textsc{IRinfer} \qquad \frac{\alpha \in \Delta_{\upsilon} \qquad \overline{\upsilon' \text{ fresh}} \qquad \overline{\Delta_{\upsilon'} = \Delta_{\upsilon}}}{\Gamma \vdash_{[\alpha\,\overline{\upsilon}'/\upsilon]} \upsilon \approx \alpha\,\overline{\sigma}}\textsc{IRpath}$$

$$\frac{}{\Gamma \vdash_{[\mathsf{bool}/\upsilon]} \upsilon \approx \mathsf{bool}}\textsc{IRbool} \qquad \frac{\upsilon' \text{ fresh} \qquad \Delta_{\upsilon'} = \Delta_{\upsilon}}{\Gamma \vdash_{[[=\upsilon']/\upsilon]} \upsilon \approx [= \Xi]}\textsc{IRtype} \qquad \frac{\upsilon_1, \upsilon_2 \text{ fresh} \qquad \Delta_{\upsilon_1} = \Delta_{\upsilon_2} = \Delta_{\upsilon}}{\Gamma \vdash_{[(\upsilon_1 \to_{\mathtt{I}} \upsilon_2)/\upsilon]} \upsilon \approx \forall\overline{\alpha}.\Sigma \to_{\iota} \Xi}\textsc{IRfun}$$

Instantiation $\Gamma \vdash^{!}_{\theta} E :_{\iota} \Xi \quad := \quad \Gamma \vdash_{\theta} E :_{\iota} \Xi' \wedge \Gamma \;{}_{\theta}{\vdash}_{\theta'} \Xi' \preceq \Xi$ $\boxed{\Gamma \vdash_{\theta} \Xi \preceq \Xi'}$

$$\frac{}{\Gamma \vdash_{\theta} \Xi \preceq \Xi}\textsc{INrefl} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\theta} \upsilon \approx \Sigma}{\Gamma \vdash_{\theta} \exists\overline{\alpha}.\upsilon \preceq \exists\overline{\alpha}.\Sigma}\textsc{INres} \qquad \frac{\overline{\upsilon} \text{ fresh} \qquad \overline{\Delta_{\upsilon} = \mathrm{dom}(\Gamma, \overline{\alpha})} \qquad \Gamma \vdash_{\theta} \exists\overline{\alpha}.\Sigma[\overline{\upsilon}/\overline{\alpha}'] \preceq \exists\overline{\alpha}.\Sigma'}{\Gamma \vdash_{\theta} \exists\overline{\alpha}.\forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma \preceq \exists\overline{\alpha}.\Sigma'}\textsc{INimpl}$$

Figure 6. Type Inference for 1ML (Excerpt)

(**fun** (id : 'a ⇒ a → a) ⇒ {x = id 3; y = id true}) (**fun** x ⇒ x)

Desugaring rewrites this application into an expression that has an explicit binding for the argument (**fun** x ⇒ x). The same observations applies to other relevant forms. Hence, generalising bindings in the kernel syntax is still enough.

Similarly, instantiation is deferred to rules corresponding to elimination forms (e.g. IEIF, IEDOT, IEAPP, but also ITPATH). There, the auxiliary *Instantiation* judgement is invoked (as part of the notation $\Gamma \vdash^{!}_{\theta} \mathcal{J}$.). This does not only instantiate implicit functions (possibly under existential binders), it also may resolve inference variables to create a type whose shape matches the shape that is expected by the invoking rule.

Instantiation can also happen implicitly as part of subtyping (rule ISIMPLL), which covers the case where a polymorphic value is matched against a monomorphic (or other polymorphic) parameter. For example, $\forall\alpha_1\alpha_2.\{\} \to_{\mathtt{A}} \alpha_1 \to_{\mathtt{I}} \alpha_2 \leq \forall\beta.\{\} \to_{\mathtt{A}} \beta \to_{\mathtt{I}} \beta$ will be checked by first applying ISIMPLR, turning the right type monomorphic, and then instantiating the left with ISIMPLL, so that the check is down to $\upsilon_1 \to_{\mathtt{I}} \upsilon_2 \leq \beta \to_{\mathtt{I}} \beta$, unifying easily.

5.2 Incompleteness

There are a couple of sources of incompleteness in this algorithm:

Width subtyping Subtyping like $\upsilon \leq \{\overline{l{:}\sigma}\}$ does not determine the shape of the record type υ: the set of labels can still vary. Consequently, the Resolution judgement has no rule for structures – instead a structure type must be determined by the previous context.

This is, in fact, similar to Standard ML [19], where record types cannot be inferred either, and require type annotation. However, SML implementations typically ensure that type inference is still order-independent, i.e., the information may be supplied *after* the point of use. They do so by employing a simple form of row inference. A similar approach would be possible for 1ML, but subtyping would still make more programs fail to type-check. For the sake of presentation, we decided to err on the side of simplicity.

The real solution of course would be to incorporate not just row inference but *row polymorphism* [21], so that width subtyping on structures can be recast as universal and existential quantification. We leave investigating such an extension for future work (though we note that **include** would still represent a challenge).

Type Scoping Tracking of the sets Δ_υ is conservative: after leaving the scope of a type variable α, we exclude any solution for υ that would still involve α, even if υ only appears inside a type binder for α. Consider, for example [5]:

```
G (x : int) = {M = {type t = int; v = x} :> {type t; v : t}; f = id id};
C = G 3;
x = C.f (C.M.v);
```

and assume $\mathsf{id} : {}'(\mathsf{a} : \textbf{type}) \Rightarrow \mathsf{a} \to \mathsf{a}$. Because id is impure, the definition of f is impure, and its type cannot be generalised; moreover, G is impure too. The algorithm will infer G's type as

$$\mathsf{int} \to \exists\beta.\{\mathsf{M} : \{\mathsf{t} : [= \beta], \mathsf{v} : \beta\}, \mathsf{f} : \upsilon \to_{\mathtt{I}} \upsilon\}$$

with $\beta \notin \Delta_\upsilon$ (because β goes out of scope the moment we bind it with a local quantifier), and then generalises to

$$\mathsf{G} : \forall\alpha.\{\} \to_{\mathtt{A}} \mathsf{int} \to \exists\beta.\{\mathsf{M} : \{\mathsf{t} : [= \beta], \mathsf{v} : \beta\}, \mathsf{f} : \alpha \to_{\mathtt{I}} \alpha\}$$

But its too late, the solution $\upsilon = \beta$, which would make x well-typed, is already precluded. When typing C, instantiating α with β is not possible either, because β can only come into scope again *after* having applied an argument for α already.

Although not well-known, this very problem is already present in good old ML, as Dreyer & Blume point out [5]: existing type inference implementations are incomplete, because combinations of functors and the value restriction (like above) do not have principal types. Interestingly, a variation of the solution suggested by Dreyer & Blume (implicitly generalising the types of functors) is implied by the 1ML typing rules: since functors are just functions, their types can already be generalised. However, generalisation happens outside the abstraction, which is more rigid than what they propose (but which is not expressible in System F_ω). Consequently, 1ML can type some examples from their paper, but not all.

Purity Annotations Due to effect subtyping, a function type as an upper bound does not determine the purity of a smaller type. Technically, that does not affect completeness, because we defined small types to only include impure functions: the resolution rule IRFUN can always pick $\mathtt{I}$. But arguably, that is cheating a little by side-stepping the issue. In particular, it makes an extension of the notion of (im)purity to other effects, as suggested in Section 2, somewhat inconvenient, because pure function types could not be inferred in parameter positions.

Again, the solution would be more polymorphism, in this case a simple form of effect polymorphism [32]. That will be future work.

Despite these limitiations, we found 1ML inference quite usable. In practice, MLs have long given up on complete type inference: various limitations exist in both SML and OCaml (and the extended language family including Haskell), necessitating type annotations or declarations. In our limited experience with a prototype, 1ML is not substantially worse, at least not when used in the same manner as traditional ML. In fact, we conjecture that any SML program not using features omitted from 1ML – but including both modules and Damas/Milner polymorphism – can be directly transliterated into 1ML without adding type annotations.

5.3 Metatheory

If the inference algorithm isn't complete, then at least it is sound. That is, we can show the following result:

THEOREM 5.1 (Correctness of 1ML Inference).
Let $\overline{\upsilon}, \Gamma$ be a well-formed F_ω environment.

1. *If $\Gamma \vdash_\theta T/D \rightsquigarrow \Xi$, then $\overline{\upsilon}', \theta\Gamma \vdash T/D \rightsquigarrow \theta\Xi$.*
2. *If $\Gamma \vdash_\theta E/B :_\iota \Xi \rightsquigarrow e$, then $\overline{\upsilon}', \theta\Gamma \vdash E/B :_\iota \theta\Xi \rightsquigarrow \theta e$.*
3. *If $\Gamma \vdash_\theta \Xi' \leq_{\overline{\pi}} \Xi \rightsquigarrow \delta; f$ and $\overline{\upsilon}, \Gamma \vdash \Xi' : \Omega$ and $\overline{\upsilon}, \Gamma, \overline{\alpha} \vdash \Xi : \Omega$, then $\overline{\upsilon}', \theta\Gamma \vdash \theta\Xi' \leq_{\overline{\pi}} \theta\Xi \rightsquigarrow \theta\delta; \theta f$.*

THEOREM 5.2 (Termination of 1ML Inference).
All 1ML type inference judgements terminate.

We have to defer the details to the Technical Appendix [23].

6. Related Work

Packaged Modules The first concrete proposal for extending ML with packaged modules was by Russo [27], and is implemented in Moscow ML. Later work on type systems for modules routinely included them [6, 4, 24, 25], and variations have been implemented in other ML dialects, such as Alice ML [22] and OCaml [7].

To avoid soundness issues in the combination with applicative functors, Russo's original proposal conservatively allowed unpacking a module only local to core-level expressions, but this restriction has been lifted in later systems, restricting only the occurrence of unpacking inside applicative functors.

First-Class Modules The first to unify ML's stratified type system into one language was Harper & Mitchell's XML calculus [10]. It is a dependent type theory modeling modules as terms of Martin-Löf-style Σ and Π types, closely following MacQueen's original ideas [17]. The system enforces predicativity through the introduction of two universes U_1 and U_2, which correspond directly to our notion of small and large type, and both systems allow both $U_1 : U_2$ and $U_1 \subseteq U_2$. XML lacks any account of either sealing or translucency, which makes it fall short as a foundation for modern ML.

That gap was closed by Harper & Lillibridge's calculus of *translucent sums* [9, 16], which also was a dependently typed language of first-class modules. Its main novelty were records with both opaque and transparent type components, directly modeling ML structures. However, unlike XML, the calculus is impredicative, which renders it undecidable.

Translucent sums where later superseded by the notion of *singleton types* [31]; they formed the foundation of Dreyer et al.'s type theory for higher-order modules [6]. However, to avoid undecidability, this system went back to second-class modules.

One concern in dependently typed theories is *phase separation*: to enable compile-time checking without requiring core-level computation, such theories must be sufficiently restricted. For example, Harper et al. [11] investigate phase separation for the XML calculus. The beauty of the F-ing approach is that it enjoys phase separation by construction, since it does not use dependent types.

Applicative Functors Leroy proposed applicative semantics for functors [15], as implemented in OCaml. Russo later combined both generative and applicative functors in one language [28] and implemented them in Moscow ML; others followed [30, 6, 4, 25].

A system like Leroy's, where *all* functors are applicative, would be incompatible with first-class modules, because the application in type paths like F(A).t needs to be phase-separable to enable type checking, but not all functions are. Russo's system has similar problems, because it allows converting generative functors into applicative ones. Like Dreyer [4] or F-ing modules [25], 1ML hence combines applicative (pure) and generative (impure) functors such that applicative semantics is only allowed for functors whose body is both pure *and* separable. In F-ing modules, applicativity is even inferred from purity, and sealing itself not considered impure; the Technical Appendix [23] shows a similar extension to 1ML.

In the version of 1ML shown in the main paper, an applicative functor can only be created by sealing a fully transparent functor with pure function type, very much like in Shao's system [30].

Type Inference There has been little work that has considered type inference for modules. Russo examined the interplay between core-level inference and modules [28], elegantly dealing with variable scoping via unification under a mixed prefix. Dreyer & Blume investigated how functors interfere with the value restriction [5].

At the same time, there have been ambitious extensions of ML-style type inference with higher-rank or impredicative types [8, 14, 33, 29]. Unlike those systems, 1ML never tries to infer a polymorphic type annotation: all guessed types are monomorphic and polymorphic parameters require annotation.

On the other hand, 1ML allows bundling types and terms together into structures. While it is necessary to explicitly annotate terms that contain types, associated type *quantifiers* (both universal and existential) and their actual introduction and elimination are implicit and effectively inferred as part of the elaboration process.

7. Future Work

1ML, as shown here, is but a first step. There are many possible improvements and extensions.

Implementation We have implemented a simple prototype interpreter for 1ML (mpi-sws.org/~rossberg/1ml/), but it would be great to gather more experience with a "real" implementation.

Applicative Functors We would like to extend 1ML's rather basic notion of applicative functor with *pure sealing* à la F-ing modules (see the Technical Appendix [23]), but more importantly, make it properly *abstraction-safe* by tracking value identities [25].

Implicits The domain of implicit functions in 1ML is limited to type **type**. Allowing richer types would be a natural extension, and might provide functionality like Haskell-style *type classes* [34].

Type Inference Despite the ability to express first-class and higher-order polymorphism, inference in 1ML is rather simple. Perhaps it is possible to combine 1ML elaboration with some of the more advanced approaches to inference described in literature.

More Polymorphism Replacing more of subtyping with polymorphism might lead to better inference: *row polymorphism* [21] could express width subtyping, and simple *effect polymorphism* [32] would allow more extensive use of pure function types.

Recursive Modules In [24] we gave a fully general design for recursive modules, elaborating into an extension of System F. It would be interesting (but complicated) to redo it 1ML-style, in order to achieve a more uniform treatment of recursion for 1ML.

Dependent Types Finally, 1ML goes to length to push the boundaries of non-dependent typing. It's a legitimate question to ask, what for? Why not go fully dependent? Well, even then sealing necessitates some equivalent of weak sums (existential types). Incorporating them, along with the quantifier pushing of our elaboration, into a dependent type system might pose an interesting challenge.

Acknowledgements

I thank Scott Kilpatrick, Claudio Russo, Gabriel Scherer, and the anonymous reviewers for their careful and helpful comments.

References

[1] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science,* vol. 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[2] S. K. Biswas. Higher-order functors with transparent signatures. In *POPL*, 1995.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[4] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, 2005.

[5] D. Dreyer and M. Blume. Principal type schemes for modular programs. In *ESOP*, 2007.

[6] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL*, 2003.

[7] J. Garrigue and A. Frisch. First-class modules and composable signatures in Objective Caml 3.12. In *ML*, 2010.

[8] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1-2), 1999.

[9] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

[10] R. Harper and J. C. Mitchell. On the type structure of Standard ML. In *ACM TOPLAS*, volume 15(2), 1993.

[11] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *POPL*, 1990.

[12] R. Harper and B. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–346. MIT Press, 2005.

[13] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[14] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System F. In *ICFP*, 2003.

[15] X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, 1995.

[16] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, CMU, 1997.

[17] D. MacQueen. Using dependent types to express modular structure. In *POPL*, 1986.

[18] R. Milner. A theory of type polymorphism in programming languages. *JCSS*, 17:348–375, 1978.

[19] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[20] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, July 1988.

[21] D. Rémy. Records and variants as a natural extension of ML. In *POPL*, 1989.

[22] A. Rossberg. The Missing Link – Dynamic components for ML. In *ICFP*, 2006.

[23] A. Rossberg. 1ML – Core and modules as one (Technical Appendix), 2015. mpi-sws.org/~rossberg/1ml/.

[24] A. Rossberg and D. Dreyer. Mixin' up the ML module system. *ACM TOPLAS*, 35(1), 2013.

[25] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *JFP*, 24(5):529–607, 2014.

[26] C. Russo. Non-dependent types for Standard ML modules. In *PPDP*, 1999.

[27] C. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.

[28] C. Russo. Types for Modules. *ENTCS*, 60, 2003.

[29] C. Russo and D. Vytiniotis. QML: Explicit first-class polymorphism for ML. In *ML*, 2009.

[30] Z. Shao. Transparent modules with fully syntactic signatures. In *ICFP*, 1999.

[31] C. A. Stone and R. Harper. Extensional equivalence and singleton types. *ACM TOCL*, 7(4):676–722, 2006.

[32] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *JFP*, 2(3):245271, 1992.

[33] D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH: First-class polymorphism for Haskell. In *ICFP*, 2008.

[34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL*, 1989.

[35] A. Wright. Simple imperative polymorphism. *LASC*, 8:343–356, 1995.

A. System F_ω

A.1 Syntax

Figure 2 gave the syntax of the variant of impredicative System F_ω that we use as the target of our elaboration translation. It includes record types (where we assume that labels are always disjoint), but is otherwise completely standard.

The syntactic sugar for n-ary pack's and unpack's, that introduce/eliminate existential types $\exists\overline{\alpha}.\tau$ quantifying over several type variables at once, is defined as follows:

$$
\begin{array}{lcl}
\exists\epsilon.\tau & := & \tau \\
\exists\overline{\alpha}.\tau & := & \exists\alpha_1.\exists\overline{\alpha}'.\tau \\
\\
\mathsf{pack}\ \langle\epsilon, e\rangle_{\exists\epsilon.\tau_0} & := & e \\
\mathsf{pack}\ \langle\overline{\tau}, e\rangle_{\exists\overline{\alpha}.\tau_0} & := & \mathsf{pack}\ \langle\tau_1, \\
 & & \quad \mathsf{pack}\ \langle\overline{\tau}', e\rangle_{\exists\overline{\alpha}'.\tau_0[\tau_1/\alpha_1]}\rangle_{\exists\overline{\alpha}.\tau_0} \\
\mathsf{unpack}\ \langle\epsilon, x{:}\tau\rangle = e_1\ \mathsf{in}\ e_2 & := & \mathsf{let}\ x{:}\tau = e_1\ \mathsf{in}\ e_2 \\
\mathsf{unpack}\ \langle\overline{\alpha}, x{:}\tau\rangle = e_1\ \mathsf{in}\ e_2 & := & \mathsf{unpack}\ \langle\alpha_1, x_1\rangle = e_1\ \mathsf{in} \\
 & & \quad \mathsf{unpack}\ \langle\overline{\alpha}', x{:}\tau\rangle = x_1\ \mathsf{in}\ e_2 \\
\mathsf{let}\ \overline{x{:}\tau = e_1}\ \mathsf{in}\ e_2 & := & (\lambda\overline{x{:}\tau}.e_2)\,\overline{e_1}
\end{array}
$$

(where $\overline{\tau} = \tau_1\overline{\tau}'$ and $\overline{\alpha} = \alpha_1\overline{\alpha}'$)

Likewise, n-ary forms of other constructs (e.g. universals, lambdas, or applications) are defined in all instances in the obvious way.

A.2 Static Semantics

The only point of note about the static semantics is that, unlike in most presentations, typing environments Γ permit shadowing of bindings for value variables x (but not for type variables α). We write $\Gamma(x)$ to refer to the type in the right-most binding of x in Γ.

Environments $\boxed{\Gamma \vdash \Box}$

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

Types $\boxed{\Gamma \vdash \tau : \kappa}$

$$
\frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \tau_1 \to \tau_2 : \Omega}
\qquad
\frac{\overline{\Gamma \vdash \tau : \Omega}}{\Gamma \vdash \{\overline{l{:}\tau}\} : \Omega}
$$

$$
\frac{\Gamma \vdash \Box}{\Gamma \vdash \alpha : \Gamma(\alpha)}
\qquad
\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \Omega}{\Gamma \vdash \forall\alpha{:}\kappa.\tau : \Omega}
\qquad
\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \Omega}{\Gamma \vdash \exists\alpha{:}\kappa.\tau : \Omega}
$$

$$
\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda\alpha{:}\kappa.\tau : \kappa \to \kappa'}
\qquad
\frac{\Gamma \vdash \tau_1 : \kappa' \to \kappa \qquad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1\ \tau_2 : \kappa}
$$

Terms $\boxed{\Gamma \vdash e : \tau}$

$$
\frac{\Gamma \vdash \Box}{\Gamma \vdash x : \Gamma(x)}
\qquad
\frac{\Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash e : \tau}
$$

$$
\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}
\qquad
\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}
$$

$$
\frac{\overline{\Gamma \vdash e : \tau}}{\Gamma \vdash \{\overline{l{=}e}\} : \{\overline{l{:}\tau}\}}
\qquad
\frac{\Gamma \vdash e : \{l{:}\tau, \overline{l'{:}\tau'}\}}{\Gamma \vdash e.l : \tau}
$$

$$
\frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\tau}
\qquad
\frac{\Gamma \vdash e : \forall\alpha{:}\kappa.\tau' \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\ \tau : \tau'[\tau/\alpha]}
$$

$$
\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash e : \tau'[\tau/\alpha] \qquad \Gamma \vdash \exists\alpha{:}\kappa.\tau' : \Omega}{\Gamma \vdash \mathsf{pack}\ \langle\tau, e\rangle_{\exists\alpha:\kappa.\tau'} : \exists\alpha{:}\kappa.\tau'}
$$

$$
\frac{\Gamma \vdash e_1 : \exists\alpha{:}\kappa.\tau' \qquad \Gamma, \alpha{:}\kappa, x{:}\tau' \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash \mathsf{unpack}\ \langle\alpha, x\rangle{=}e_1\ \mathsf{in}\ e_2 : \tau}
$$

Type Equivalence $\boxed{\tau \equiv \tau'}$

$$
\frac{}{\tau \equiv \tau}
\qquad
\frac{\tau' \equiv \tau}{\tau \equiv \tau'}
\qquad
\frac{\tau \equiv \tau' \qquad \tau' \equiv \tau''}{\tau \equiv \tau''}
$$

$$
\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}
\qquad
\frac{\overline{\tau \equiv \tau'}}{\{\overline{l{:}\tau}\} \equiv \{\overline{l{:}\tau'}\}}
$$

$$
\frac{\tau \equiv \tau'}{\forall\alpha{:}\kappa.\tau \equiv \forall\alpha{:}\kappa.\tau'}
\qquad
\frac{\tau \equiv \tau'}{\exists\alpha{:}\kappa.\tau \equiv \exists\alpha{:}\kappa.\tau'}
$$

$$
\frac{\tau \equiv \tau'}{\lambda\alpha{:}\kappa.\tau \equiv \lambda\alpha{:}\kappa.\tau'}
\qquad
\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}
$$

$$
\frac{}{(\lambda\alpha{:}\kappa.\tau_1)\ \tau_2 \equiv \tau_1[\tau_2/\alpha]}
\qquad
\frac{\alpha \notin \mathrm{fv}(\tau)}{(\lambda\alpha{:}\kappa.\tau\ \alpha) \equiv \tau}
$$

A.3 Dynamic Semantics

We assume a standard left-to-right call-by-value evaluation order:

Reduction $\boxed{e \hookrightarrow e'}$

$$
\begin{array}{rcl}
(\lambda x{:}\tau.e)\ v & \hookrightarrow & e[v/x] \\
\{\overline{l_1{=}v_1}, l{=}v, \overline{l_2{=}v_2}\}.l & \hookrightarrow & v \\
(\lambda\alpha{:}\kappa.e)\ \tau & \hookrightarrow & e[\tau/\alpha] \\
\mathsf{unpack}\ \langle\alpha, x\rangle = \mathsf{pack}\ \langle\tau, v\rangle_{\tau'}\ \mathsf{in}\ e & \hookrightarrow & e[\tau/\alpha][v/x]
\end{array}
$$

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

where:

$$
\begin{array}{llcl}
\text{(values)} & v & ::= & \lambda x{:}\tau.e \mid \{\overline{l{=}v}\} \mid \lambda\alpha{:}\kappa.e \mid \mathsf{pack}\ \langle\tau, v\rangle_\tau \\
\text{(contexts)} & C & ::= & [] \mid C\ e \mid v\ C \mid \{\overline{l_1{=}v}, l{=}C, \overline{l_2{=}e}\} \mid C.l \mid \\
 & & & C\ \tau \mid \mathsf{pack}\ \langle\tau, C\rangle_\tau \mid \mathsf{unpack}\ \langle\alpha, x\rangle{=}C\ \mathsf{in}\ e
\end{array}
$$

A.4 Properties

The calculus as defined enjoys the standard soundness properties:

THEOREM A.1 (Preservation).
If $\cdot \vdash e : \tau$ and $e \hookrightarrow e'$, then $\cdot \vdash e' : \tau$.

THEOREM A.2 (Progress).
If $\cdot \vdash e : \tau$ and $e \neq v$ for any v, then $e \hookrightarrow e'$ for some e'.

The proofs are entirely standard.

A.5 Substitution

Our semantics makes fair use of parallel type substitutions. We write them as $[\overline{\tau}/\overline{\alpha}]$, assuming that both vectors have the same arity, and use δ to range over them. The following definitions and lemmas are relevant:

DEFINITION A.3 (Typing of Type Substitutions). *Let $\delta = [\overline{\tau}/\overline{\alpha}]$. We write $\Gamma' \vdash \delta : \Gamma$ if and only if*

1. *$\Gamma' \vdash \Box$,*
2. *$\overline{\alpha} \subseteq \mathrm{dom}(\Gamma)$,*
3. *for all $\alpha \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash \delta\alpha : \Gamma(\alpha)$,*
4. *for all $x \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash x : \delta(\Gamma(x))$.*

LEMMA A.4 (Type Substitutions). *Let $\Gamma' \vdash \delta : \Gamma$. Then:*

1. *If $\Gamma \vdash \tau : \kappa$, then $\Gamma' \vdash \delta\alpha : \kappa$.*
2. *If $\Gamma \vdash e : \tau$, then $\Gamma' \vdash \delta e : \delta\tau$.*

B. Decidability of Subtyping

In order to prove that subtyping terminates, we define the following weight functions over semantic types:

$$\begin{array}{lcl}
S\llbracket \mathsf{bool} \rrbracket & = & 1 \\
S\llbracket \alpha \rrbracket & = & 1 \\
S\llbracket \pi\,\sigma \rrbracket & = & S\llbracket \pi \rrbracket + S\llbracket \sigma \rrbracket + 1 \\
S\llbracket \lambda\overline{\alpha}.\sigma \rrbracket & = & S\llbracket \sigma \rrbracket + 1 \\
S\llbracket [= \Xi] \rrbracket & = & S\llbracket \Xi \rrbracket + 1 \\
S\llbracket \{\} \rrbracket & = & 1 \\
S\llbracket \{l_0{:}\Sigma_0, \overline{l{:}\Sigma}\} \rrbracket & = & S\llbracket \Sigma_0 \rrbracket + S\llbracket \{\overline{l{:}\Sigma}\} \rrbracket + 1 \\
S\llbracket \forall\overline{\alpha}.\Sigma \to_\iota \Xi \rrbracket & = & S\llbracket \Sigma \rrbracket + S\llbracket \Xi \rrbracket + 1 \\
S\llbracket \exists\overline{\alpha}.\Sigma \rrbracket & = & S\llbracket \Sigma \rrbracket \\
 & & \\
Q\llbracket \mathsf{bool} \rrbracket & = & 0 \\
Q\llbracket \alpha \rrbracket & = & 0 \\
Q\llbracket \pi\,\sigma \rrbracket & = & 0 \\
Q\llbracket \lambda\overline{\alpha}.\sigma \rrbracket & = & 0 \\
Q\llbracket [= \Xi] \rrbracket & = & 2 \cdot Q\llbracket \Xi \rrbracket \\
Q\llbracket \{\} \rrbracket & = & 0 \\
Q\llbracket \{l_0{:}\Sigma_0, \overline{l{:}\Sigma}\} \rrbracket & = & Q\llbracket \Sigma_0 \rrbracket + Q\llbracket \{\overline{l{:}\Sigma}\} \rrbracket \\
Q\llbracket \forall\overline{\alpha}.\Sigma \to_\iota \Xi \rrbracket & = & |\overline{\alpha}| + Q\llbracket \Sigma \rrbracket + Q\llbracket \Xi \rrbracket \\
Q\llbracket \exists\overline{\alpha}.\Sigma \rrbracket & = & |\overline{\alpha}| + Q\llbracket \Sigma \rrbracket \\
 & & \\
W\llbracket \Xi \rrbracket & = & \langle Q\llbracket \Xi \rrbracket, S\llbracket \Xi \rrbracket \rangle
\end{array}$$

The definitions assume $\beta\eta$-normal form; in the case of paths and structures, they are inductive over the vector of arguments and components, respectively.

We apply addition point-wise to weight pairs $W\llbracket\rrbracket$, and impose a lexicographic ordering on them:

$$\langle q, s\rangle < \langle q', s'\rangle \quad :\Leftrightarrow \quad q < q' \vee (q = q' \wedge s < s')$$

Small types don't contain quantifiers, so $Q\llbracket\sigma\rrbracket = 0$. We extend the notion of small type to (higher-order) type constructors and substitutions: a type constructor is small if it is of the form $\lambda\overline{\alpha}.\sigma$; a substitution δ is small if $\delta(\alpha)$ is small for all α. Then, abbreviating $|\mathrm{dom}(\delta)|$ to $|\delta|$:

LEMMA B.1 (Weight Reduction under Small Substitution). *Let δ be a small substitution.*

1. $Q\llbracket\delta\Xi\rrbracket = Q\llbracket\Xi\rrbracket$.
2. $W\llbracket\delta\Xi\rrbracket < \langle|1, 0\rangle + W\llbracket\Xi\rrbracket$.
3. $W\llbracket\delta\Xi\rrbracket \leq \langle|\delta|, 0\rangle + W\llbracket\Xi\rrbracket$.

Note how this lemma crucially relies on δ being small. The properties are essential in proving the case of rule SFUN for the termination lemma:

LEMMA B.2 (Termination of Algorithmic Subtyping). *Let Γ be a well-formed context, $\Gamma \vdash \Xi' : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Xi : \Omega$ and $\overline{\pi} = \overline{\alpha}\ \overline{\alpha'}$. Then $\Gamma \vdash \Xi' \leq_{\overline{\pi}} \Xi \rightsquigarrow \delta; f$ terminates.*

The proof is by case analysis on the algorithm. In each rule, the weight $W\llbracket\Xi'\rrbracket + W\llbracket\Xi\rrbracket + \langle|\overline{\pi}|, 0\rangle$ gets strictly smaller for each premise: either its Q component shrinks, because a quantifier is peeled, or its S, because the structure is otherwise simplified. In particular, Q shrinks in all cases where a non-empty substitution δ is applied in a premise.

C. Impredicativity

Figure 7 shows shows an extension of 1ML$_{\mathrm{ex}}$ with (structural) impredicative types. The trick is that a large type has to be injected into the universe of small types explicitly, marked by the form **wrap** T (this is rather similar to the bracketted polytypes in Garrigue & Rémy's semi-explicit first-class polymorphism [8]). Subtyping, then, does not apply to wrapped types (rule SWRAP). This way, any infinite recursion is avoided; the weight function for the termination proof (Appendix B) can trivially be extended to the additional form of type as $S\llbracket[\Xi]\rrbracket = 1$ and $Q\llbracket[\Xi]\rrbracket = 0$.

Syntax

$$\begin{array}{lrcl}
\text{(types)} & T & ::= & \ldots \mid \textbf{wrap}\ T \\
\text{(expressions)} & E & ::= & \ldots \mid \textbf{wrap}\ X{:}T \mid \textbf{unwrap}\ X{:}T
\end{array}$$

$$\begin{array}{lcl}
\textbf{wrap}\ E : T & := & \textbf{let}\ X{=}E\ \textbf{in}\ \textbf{wrap}\ X : T \\
\textbf{unwrap}\ E : T & := & \textbf{let}\ X{=}E\ \textbf{in}\ \textbf{unwrap}\ X : T
\end{array}$$

Semantic Types

$$\begin{array}{lrcl}
\text{(large)} & \Sigma & ::= & \ldots \mid [\Xi] \\
\text{(small)} & \sigma & ::= & \ldots \mid [\Xi]
\end{array}$$

Desugarings into F$_\omega$:

$$\begin{array}{lclclcl}
\text{(types)} & & & \qquad & \text{(terms)} & & \\
[\Xi] & := & \{\mathsf{val} : \Xi\} & \qquad & [e] & := & \{\mathsf{val} = e\}
\end{array}$$

Types $\boxed{\Gamma \vdash T \rightsquigarrow \Xi}$

$$\frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \textbf{wrap}\ T \rightsquigarrow [\Xi]}\ \textsc{Twrap}$$

Expressions $\boxed{\Gamma \vdash E :_\iota \Xi \rightsquigarrow e}$

$$\frac{\Gamma \vdash X :_{\mathrm{P}} \Sigma \rightsquigarrow e \qquad \Gamma \vdash T \rightsquigarrow [\Xi] \qquad \Gamma \vdash \Sigma \leq \Xi \rightsquigarrow f}{\Gamma \vdash \textbf{wrap}\ X{:}T :_{\mathrm{P}} [\Xi] \rightsquigarrow [f\ e]}\ \textsc{Ewrap}$$

$$\frac{\Gamma \vdash X :_{\mathrm{P}} [\Xi'] \rightsquigarrow e \qquad \Gamma \vdash T \rightsquigarrow [\Xi] \qquad \Gamma \vdash \Xi' \leq \Xi \rightsquigarrow f}{\Gamma \vdash \textbf{unwrap}\ X{:}T :_{\iota(\Xi)} \Xi \rightsquigarrow f\ (e.\mathsf{val})}\ \textsc{Eunw}$$

Subtyping $\boxed{\Gamma \vdash \Xi' \leq \Xi \rightsquigarrow \delta; f}$

$$\frac{}{\Gamma \vdash [\Xi] \leq [\Xi] \rightsquigarrow \lambda x{:}[\Xi].x}\ \textsc{Swrap}$$

Figure 7. Extension with Impredicativity

The syntax we chose here is reminiscent of packaged modules (Section 1.1), but the construct has far more specialised use cases in 1ML. In particular, wrapping is never needed if one merely wants to abstract over a *value* of large type (like is the case in conventional ML with packaged modules). It is only needed in the rare case where one wants to *type*-abstract over a large type.

As an example, consider a Church encoding of the option type:

```
type OPT =
{
  type opt a
  none a : opt a
  some a : a → opt a
  caseopt a b : opt a → b → (a → b) → b
}
```

```
Opt :> OPT =
{
  type opt a = wrap (b : type) ⇒ b → (a → b) → b
  none a = wrap (fun b (n : b) (s : a → b) ⇒ n) : opt a
  some a (x : a) = wrap (fun b (n : b) (s : a → b) ⇒ s x) : opt a
  caseopt a b (o : opt a) = (unwrap o : opt a) b
}
```

The implementation of opt is a large type, so it has to be wrapped in order to "make it small" and allow matching the abstract declaration in the signature.

Impredicativity in this manner is also easily integrated with type inference. The respective rules can be seen in Appendix D.

D. Type Inference Rules

Abbreviations

$$\begin{array}{rcl}
\Gamma\ {}_{\theta}\!\vdash_{\theta'} \mathcal{J} & := & \theta\Gamma \vdash_{\theta''} {}^{\theta}\mathcal{J}\ \wedge \theta' = \theta'' \circ \theta \\
\Gamma;\Gamma'\ {}_{\theta}\!\vdash_{\theta'} \mathcal{J} & := & \Gamma,\Gamma'\ {}_{\theta}\!\vdash_{\theta''} \mathcal{J}\ \wedge \theta' = [\overline{\upsilon}'/\overline{\upsilon}] \circ \theta'' \\
 & & \text{where } \overline{\upsilon} = \mathrm{undet}(\theta''\mathcal{J}) \\
 & & \qquad \overline{\upsilon}' \text{ fresh with } \overline{\Delta_{\upsilon'} = \Delta_{\upsilon} \cap \mathrm{dom}(\Gamma)}
\end{array}$$

Types

$$\boxed{\Gamma \vdash_{\theta} T \rightsquigarrow \Xi}$$

$$\frac{\Gamma \vdash^{!}_{\theta} E :_{\mathtt{P}} [= \Xi] \rightsquigarrow e}{\Gamma \vdash_{\theta} E \rightsquigarrow \Xi}\ \text{IT}\textsc{path}$$

$$\frac{\upsilon \text{ fresh} \qquad \Delta_{\upsilon} = \mathrm{dom}(\Gamma)}{\Gamma \vdash_{[]} \_ \rightsquigarrow \upsilon}\ \text{IT}\textsc{infer}$$

$$\frac{}{\Gamma \vdash_{[]} \textbf{type} \rightsquigarrow \exists\alpha.[= \alpha]}\ \text{IT}\textsc{type}$$

$$\frac{}{\Gamma \vdash_{[]} \textbf{bool} \rightsquigarrow \mathsf{bool}}\ \text{IT}\textsc{bool}$$

$$\frac{\Gamma \vdash_{\theta} D \rightsquigarrow \Xi}{\Gamma \vdash_{\theta} \{D\} \rightsquigarrow \Xi}\ \text{IT}\textsc{str}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma;\overline{\alpha}_1, X{:}\Sigma_1\ {}_{\theta_1}\!\vdash_{\theta_2} T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2\end{array}}{\Gamma \vdash_{\theta_2} (X{:}T_1) \rightarrow T_2 \rightsquigarrow \forall\overline{\alpha}_1.\,\Sigma_1 \rightarrow_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2}\ \text{IT}\textsc{fun}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma;\overline{\alpha}_1, X{:}\Sigma_1\ {}_{\theta_1}\!\vdash_{\theta_2} T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2 \qquad \overline{\kappa_{\alpha'_2} = \overline{\kappa_{\alpha_1} \rightarrow \kappa_{\alpha_2}}}\end{array}}{\Gamma \vdash_{\theta_2} (X{:}T_1) \Rightarrow T_2 \rightsquigarrow \exists\overline{\alpha}'_2.\forall\overline{\alpha}_1.\,\Sigma_1 \rightarrow_{\mathtt{P}} \Sigma_2[\overline{\alpha'_2\,\overline{\alpha}_1}/\overline{\alpha}_2]}\ \text{IT}\textsc{pfun}$$

$$\frac{\Gamma;\alpha, X{:}[= \alpha] \vdash_{\theta} T \rightsquigarrow \Sigma \qquad \kappa_{\alpha} = \Omega}{\Gamma \vdash_{\theta} \text{'}(X{:}\textbf{type}) \Rightarrow T \rightsquigarrow \forall\alpha.\{\} \rightarrow_{\mathtt{A}} \Sigma}\ \text{IT}\textsc{impl}$$

$$\frac{\Gamma \vdash_{\theta} T \rightsquigarrow \Xi}{\Gamma \vdash_{\theta} \textbf{wrap}\ T \rightsquigarrow [\Xi]}\ \text{IT}\textsc{wrap}$$

$$\frac{\Gamma \vdash_{\theta} E :_{\mathtt{P}} \Sigma \rightsquigarrow e}{\Gamma \vdash_{\theta} (= E) \rightsquigarrow \Sigma}\ \text{IT}\textsc{sing}$$

$$\frac{\begin{array}{cc}\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 & \overline{\alpha}_1 = \overline{\alpha}_{11} \uplus \overline{\alpha}_{12} \\ \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} T_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2 & \Gamma, \overline{\alpha}_{11}, \overline{\alpha}_2\ {}_{\theta_2}\!\vdash_{\theta} \Sigma_2 \leq_{\overline{\alpha}_{12}} \Sigma_1.\overline{X} \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash_{\theta} T_1\ \textbf{where}\ (\overline{.X}{:}T_2) \rightsquigarrow \exists\overline{\alpha}_{11}\overline{\alpha}_2.\delta\Sigma'_1[\overline{.X}{=}\Sigma_2]}\ \text{IT}\textsc{where}$$

Declarations

$$\boxed{\Gamma \vdash_{\theta} D \rightsquigarrow \Xi}$$

$$\frac{\Gamma \vdash_{\theta} T \rightsquigarrow \exists\overline{\alpha}.\Sigma}{\Gamma \vdash_{\theta} X{:}T \rightsquigarrow \exists\overline{\alpha}.\{X{:}\Sigma\}}\ \text{ID}\textsc{var}$$

$$\frac{\Gamma \vdash_{\theta} T \rightsquigarrow \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\}}{\Gamma \vdash_{\theta} \textbf{include}\ T \rightsquigarrow \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\}}\ \text{ID}\textsc{incl}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} D_1 \rightsquigarrow \exists\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \\ \Gamma;\overline{\alpha}_1, \overline{X_1{:}\Sigma_1}\ {}_{\theta_1}\!\vdash_{\theta_2} D_2 \rightsquigarrow \exists\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \qquad \overline{X}_1 \cap \overline{X}_2 = \emptyset\end{array}}{\Gamma \vdash_{\theta_2} D_1{;}D_2 \rightsquigarrow \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X_1{:}\Sigma_1}, \overline{X_2{:}\Sigma_2}\}}\ \text{ID}\textsc{seq}$$

$$\frac{}{\Gamma \vdash_{[]} \epsilon \rightsquigarrow \{\}}\ \text{ID}\textsc{empty}$$

Expressions

$$\boxed{\Gamma \vdash_{\theta} E :_{\iota} \Xi \rightsquigarrow e}$$

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash_{[]} X :_{\mathtt{P}} \Sigma \rightsquigarrow X}\ \text{IE}\textsc{var}$$

$$\frac{\Gamma \vdash_{\theta} T \rightsquigarrow \Xi}{\Gamma \vdash_{\theta} \textbf{type}\ T :_{\mathtt{P}} [= \Xi] \rightsquigarrow [\Xi]}\ \text{IE}\textsc{type}$$

$$\frac{}{\Gamma \vdash_{[]} \textbf{true} :_{\mathtt{P}} \mathsf{bool} \rightsquigarrow \mathsf{true}}\ \text{IE}\textsc{true} \qquad \frac{}{\Gamma \vdash_{[]} \textbf{false} :_{\mathtt{P}} \mathsf{bool} \rightsquigarrow \mathsf{false}}\ \text{IE}\textsc{false}$$

$$\frac{\begin{array}{ll}\Gamma \vdash^{!}_{\theta_0} X :_{\mathtt{P}} \mathsf{bool} \rightsquigarrow e_X & \Gamma\ {}_{\theta_2}\!\vdash_{\theta_3} T \rightsquigarrow \Xi \\ \Gamma\ {}_{\theta_0}\!\vdash_{\theta_1} E_1 :_{\iota_1} \Xi_1 \rightsquigarrow e_1 & \Gamma\ {}_{\theta_3}\!\vdash_{\theta_4} \Xi_1 \leq \Xi \rightsquigarrow f_1 \\ \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} E_2 :_{\iota_2} \Xi_2 \rightsquigarrow e_2 & \Gamma\ {}_{\theta_4}\!\vdash_{\theta_5} \Xi_2 \leq \Xi \rightsquigarrow f_2\end{array}}{\begin{array}{c}\Gamma \vdash_{\theta_5} \textbf{if}\ X\ \textbf{then}\ E_1\ \textbf{else}\ E_2 : T :_{\iota_1 \vee \iota_2 \vee \iota(\Xi)} \Xi \\ \rightsquigarrow \mathsf{if}\ e_X\ \mathsf{then}\ f_1\ e_1\ \mathsf{else}\ f_2\ e_2\end{array}}\ \text{IE}\textsc{if}$$

$$\frac{\Gamma \vdash_{\theta} B :_{\iota} \Xi \rightsquigarrow e}{\Gamma \vdash_{\theta} \{B\} :_{\iota} \Xi \rightsquigarrow e}\ \text{IE}\textsc{str}$$

$$\frac{\Gamma \vdash^{!}_{\theta} E :_{\iota} \exists\overline{\alpha}.\{X{:}\Sigma, \overline{X'{:}\Sigma'}\} \rightsquigarrow e}{\begin{array}{l}\Gamma \vdash_{\theta} E.X :_{\iota} \exists\overline{\alpha}.\Sigma \\ \quad \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, y.X\rangle\end{array}}\ \text{IE}\textsc{dot}$$

$$\frac{\Gamma \vdash_{\theta_1} T \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma;\overline{\alpha}, X{:}\Sigma\ {}_{\theta_1}\!\vdash_{\theta_2} E :_{\iota} \Xi \rightsquigarrow e}{\Gamma \vdash_{\theta_2} \textbf{fun}\,(X{:}T) \Rightarrow E :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma \rightarrow_{\iota} \Xi \rightsquigarrow \lambda\overline{\alpha}.\lambda X{:}\Sigma.e}\ \text{IE}\textsc{fun}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{!}_{\theta_1} X_1 :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma_1 \rightarrow_{\iota} \Xi \rightsquigarrow e_1 \\ \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} X_2 :_{\mathtt{P}} \Sigma_2 \rightsquigarrow e_2 \qquad \Gamma\ {}_{\theta_2}\!\vdash_{\theta_3} \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash_{\theta_3} X_1\,X_2 :_{\iota} \delta\Xi \rightsquigarrow (e_1\,(\delta\overline{\alpha})\,(f\,e_2)).\iota}\ \text{IE}\textsc{app}$$

$$\frac{\Gamma \vdash_{\theta_1} X :_{\mathtt{P}} \Sigma \rightsquigarrow e \qquad \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} T \rightsquigarrow [\Xi] \qquad \Gamma\ {}_{\theta_2}\!\vdash_{\theta_3} \Sigma \leq \Xi \rightsquigarrow f}{\Gamma \vdash_{\theta_3} \textbf{wrap}\ X{:}T :_{\mathtt{P}} [\Xi] \rightsquigarrow [f\,e]}\ \text{IE}\textsc{wrap}$$

$$\frac{\Gamma \vdash^{!}_{\theta_1} X :_{\mathtt{P}} [\Xi'] \rightsquigarrow e \quad \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} T \rightsquigarrow [\Xi] \quad \Gamma\ {}_{\theta_2}\!\vdash_{\theta_3} \Xi' \leq \Xi \rightsquigarrow f}{\Gamma \vdash_{\theta_3} \textbf{unwrap}\ X{:}T :_{\mathtt{P}} \Xi \rightsquigarrow (f\,e).\mathsf{val}}\ \text{IE}\textsc{unw}$$

Bindings

$$\boxed{\Gamma \vdash_{\theta} B :_{\iota} \Xi \rightsquigarrow e}$$

$$\frac{\Gamma \vdash_{\theta} E :_{\mathtt{I}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}{\begin{array}{l}\Gamma \vdash_{\theta} X{=}E :_{\mathtt{I}} \exists\overline{\alpha}.\{X : \Sigma\} \\ \quad \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \{X{=}x\}\rangle\end{array}}\ \text{IB}\textsc{var}$$

$$\frac{\Gamma \vdash_{\theta} E :_{\mathtt{P}} \Sigma \rightsquigarrow e \qquad \overline{\upsilon} = \mathrm{undet}(\theta\Sigma) - \mathrm{undet}(\theta\Gamma) \qquad \overline{\kappa_{\alpha} = \Omega}}{\Gamma \vdash_{\theta} X{=}E :_{\mathtt{P}} \{X : \forall\overline{\alpha}.\{\} \rightarrow_{\mathtt{A}} \Sigma[\overline{\alpha}/\overline{\upsilon}]\} \rightsquigarrow \{X{=}\lambda\overline{\alpha}.\lambda_{\mathtt{A}}x{:}\{\}.e\}}\ \text{IB}\textsc{pvar}$$

$$\frac{\Gamma \vdash^{!}_{\theta} E :_{\iota} \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \rightsquigarrow e}{\Gamma \vdash_{\theta} \textbf{include}\ E :_{\iota} \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \rightsquigarrow e}\ \text{IB}\textsc{incl}$$

$$\frac{\begin{array}{rl}\Gamma \vdash_{\theta_1} B_1 :_{\iota_1} \exists\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \rightsquigarrow e_1 & \overline{X}'_1 = \overline{X}_1 - \overline{X}_2 \\ \Gamma;\overline{\alpha}_1, \overline{X_1{:}\Sigma_1}\ {}_{\theta_1}\!\vdash_{\theta_2} B_2 :_{\iota_2} \exists\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \rightsquigarrow e_2 & \overline{X}'_1{:}\Sigma'_1 \subseteq \overline{X_1{:}\Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash_{\theta_2} B_1{;}B_2 :_{\iota_1 \vee \iota_2} \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X'_1{:}\Sigma'_1}, \overline{X_2{:}\Sigma_2}\} \\ \quad \rightsquigarrow \mathsf{unpack}\ \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in}\ \mathsf{let}\ \overline{X_1 = y_1.X_1}\ \mathsf{in} \\ \qquad \mathsf{unpack}\ \langle\overline{\alpha}_2, y_2\rangle = e_2\ \mathsf{in} \\ \qquad \mathsf{pack}\ \langle\overline{\alpha}_1\overline{\alpha}_2, \{\overline{X'_1 = y_1.X'_1}, \overline{X_2 = y_2.X_2}\}\rangle\end{array}}\ \text{IB}\textsc{seq}$$

$$\frac{}{\Gamma \vdash_{[]} \epsilon :_{\mathtt{P}} \{\} \rightsquigarrow \{\}}\ \text{IE}\textsc{empty}$$

Subtyping $\boxed{\Gamma \vdash_{\theta} \Xi' \leq_{\overline{\pi}} \Xi \leadsto \delta; f}$

$$\Gamma \vdash_{\theta} \Xi \leq \Xi' \leadsto f \quad := \quad \Gamma \vdash_{\theta} \Xi \leq_{\epsilon} \Xi' \leadsto \mathrm{id}; f$$

$$\frac{}{\Gamma \vdash_{[]} \upsilon \leq \upsilon \leadsto \lambda x.x}\text{IS}\textsc{Refl}$$

$$\frac{\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma \qquad \Gamma\ {}_{\theta}\!\vdash_{\theta'} \upsilon \leq \Sigma}{\Gamma \vdash_{\theta'} \upsilon \leq \Sigma \leadsto \lambda x.x}\text{IS}\textsc{ResL}$$

$$\frac{\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma' \qquad \Gamma\ {}_{\theta}\!\vdash_{\theta'} \Sigma' \leq \upsilon}{\Gamma \vdash_{\theta'} \Sigma' \leq \upsilon \leadsto \lambda x.x}\text{IS}\textsc{ResR}$$

$$\frac{\Gamma \vdash_{\theta} \pi' = \pi}{\Gamma \vdash_{\theta} \pi' \leq \pi \leadsto \lambda x.x}\text{IS}\textsc{Path}$$

$$\frac{}{\Gamma \vdash_{[]} [= \sigma] \leq_{\alpha_0\,\overline{\alpha}} [= \alpha_0\,\overline{\alpha}] \leadsto [\lambda\overline{\alpha}.\sigma/\alpha_0]; \lambda x.x}\text{IS}\textsc{Forget}$$

$$\frac{\Gamma \vdash_{\theta} \Xi' \leq \Xi \leadsto f \qquad \Gamma\ {}_{\theta}\!\vdash_{\theta'} \Xi \leq \Xi' \leadsto f'}{\Gamma \vdash_{\theta'} [= \Xi'] \leq [= \Xi] \leadsto []; \lambda x.[\Xi]}\text{IS}\textsc{Type}$$

$$\frac{\Gamma \vdash_{\theta} \Xi' = \Xi}{\Gamma \vdash_{\theta} [\Xi'] \leq [\Xi] \leadsto \lambda x.x}\text{IS}\textsc{Wrap}$$

$$\frac{}{\Gamma \vdash_{[]} \{\overline{l{:}\Sigma'}\} \leq \{\} \leadsto \lambda x.\{\}}\text{IS}\textsc{Empty}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} \Sigma'_1 \leq_{\overline{\pi}_1} \Sigma_1 \leadsto \delta_1; f_1\\ \Gamma\ {}_{\theta_1}\!\vdash_{\theta_2} \{\overline{l'{:}\Sigma'}\} \leq_{\overline{\pi}_2} \delta_1\{\overline{l{:}\Sigma}\} \leadsto \delta_2; f_2 \qquad \delta_2\Sigma_1 = \Sigma_1\end{array}}{\begin{array}{c}\Gamma \vdash_{\theta_2} \{l_1{:}\Sigma'_1, \overline{l'{:}\Sigma'}\} \leq_{\overline{\pi}_1\overline{\pi}_2} \{l_1{:}\Sigma_1, \overline{l{:}\Sigma}\}\\ \leadsto \delta_1\delta_2; \lambda x.\{l_1{=}f_1(x.l_1), \overline{l{=}(f_2\,x).l}\}\end{array}}\text{IS}\textsc{Str}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \vdash_{\theta_1} \Sigma \leq_{\overline{\alpha}'} \Sigma' \leadsto \delta_1; f_1 \qquad \iota' \leq \iota\\ \Gamma; \overline{\alpha}\ {}_{\theta_1}\!\vdash_{\theta_2} \delta_1\Xi' \leq_{\overline{\pi\overline{\alpha}}} \Xi \leadsto \delta_2; f_2 \qquad \theta_2\delta_2\Sigma = \theta_2\Sigma\end{array}}{\begin{array}{c}\Gamma \vdash_{\theta_2} (\forall\overline{\alpha}'.\Sigma' \to_{\iota'} \Xi') \leq_{\overline{\pi}} (\forall\overline{\alpha}.\Sigma \to_{\iota} \Xi)\\ \leadsto \delta_2; \lambda x.\,\lambda\overline{\alpha}.\,\lambda_{\iota}y{:}\Sigma.\,f_2\,((x\,(\delta_1\overline{\alpha}')\,(f_1\,y)).\iota')\end{array}}\text{IS}\textsc{Fun}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma' \leq_{\overline{\alpha}} \Sigma \leadsto \delta; f \qquad \overline{\alpha}'\overline{\alpha} \neq \epsilon}{\begin{array}{l}\Gamma \vdash_{\theta} \exists\overline{\alpha}'.\Sigma' \leq \exists\overline{\alpha}.\Sigma\\ \quad \leadsto \lambda x.\,\mathsf{unpack}\ \langle\overline{\alpha}', y\rangle = x\ \mathsf{in}\ \mathsf{pack}\ \langle\delta\overline{\alpha}, f\,y\rangle\end{array}}\text{IS}\textsc{Abs}$$

$$\frac{\overline{\upsilon\text{ fresh}} \qquad \overline{\Delta_{\upsilon} = \mathrm{dom}(\Gamma)} \qquad \Gamma \vdash_{\theta} \Sigma'[\overline{\upsilon}/\overline{\alpha}'] \leq_{\overline{\pi}} \Sigma \leadsto \delta; f}{\Gamma \vdash_{\theta} \forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \leq_{\overline{\pi}} \Sigma \leadsto \delta; \lambda x.\,f\,((x\,\overline{\upsilon}\,\{\}).\mathtt{A})}\text{IS}\textsc{ImplL}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma' \leq_{\overline{\pi}} \Sigma \leadsto \delta; f \qquad \overline{\alpha} \not\cap \mathrm{fv}(\theta\delta)}{\Gamma \vdash_{\theta} \Sigma' \leq_{\overline{\pi}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \leadsto \delta; \lambda x.\,\lambda\overline{\alpha}.\lambda_{\mathtt{A}}y{:}\{\}.f\,x}\text{IS}\textsc{ImplR}$$

Resolution $\boxed{\Gamma \vdash_{\theta} \upsilon \approx \Sigma}$

$$\Gamma \vdash^{!}_{\theta} \upsilon \approx \Sigma \quad := \quad \upsilon \notin \mathrm{undet}(\Sigma) \wedge \Gamma \vdash_{\theta} \upsilon \approx \Sigma$$

$$\frac{\upsilon''\text{ fresh} \qquad \Delta_{\upsilon''} = \Delta_{\upsilon} \cap \Delta_{\upsilon'}}{\Gamma \vdash_{[\upsilon''/\upsilon,\upsilon''/\upsilon']} \upsilon \approx \upsilon'}\text{IR}\textsc{Infer}$$

$$\frac{\alpha \in \Delta_{\upsilon} \qquad \overline{\upsilon'\text{ fresh}} \qquad \overline{\Delta_{\upsilon'} = \Delta_{\upsilon}}}{\Gamma \vdash_{[\alpha\,\overline{\upsilon}'/\upsilon]} \upsilon \approx \alpha\,\overline{\sigma}}\text{IR}\textsc{Path}$$

$$\frac{}{\Gamma \vdash_{[\mathsf{bool}/\upsilon]} \upsilon \approx \mathsf{bool}}\text{IR}\textsc{Bool}$$

$$\frac{\upsilon'\text{ fresh} \qquad \Delta_{\upsilon'} = \Delta_{\upsilon}}{\Gamma \vdash_{[[=\upsilon']/\upsilon]} \upsilon \approx [= \Xi]}\text{IR}\textsc{Type}$$

$$\frac{\upsilon'\text{ fresh} \qquad \Delta_{\upsilon'} = \Delta_{\upsilon}}{\Gamma \vdash_{[[\upsilon']/\upsilon]} \upsilon \approx [\Xi]}\text{IR}\textsc{Wrap}$$

$$\frac{\upsilon_1, \upsilon_2\text{ fresh} \qquad \Delta_{\upsilon_1} = \Delta_{\upsilon_2} = \Delta_{\upsilon}}{\Gamma \vdash_{[(\upsilon_1 \to_{\mathtt{I}} \upsilon_2)/\upsilon]} \upsilon \approx \forall\overline{\alpha}.\Sigma \to_{\iota} \Xi}\text{IR}\textsc{Fun}$$

Unification $\boxed{\Gamma \vdash_{\theta} \Xi = \Xi'}$

$$\frac{}{\Gamma \vdash_{[]} \upsilon = \upsilon}\text{IU}\textsc{Refl}$$

$$\frac{\Gamma \vdash_{\theta} \Xi' = \Xi}{\Gamma \vdash_{\theta} \Xi = \Xi'}\text{IU}\textsc{Symm}$$

$$\frac{\dfrac{\overline{\upsilon}' = \mathrm{undet}(\sigma)}{\overline{\upsilon''\text{ fresh}}} \qquad \dfrac{\upsilon \notin \overline{\upsilon}' \qquad \Delta_{\upsilon} \supseteq \mathrm{fv}(\sigma)}{\overline{\Delta_{\upsilon''} = \Delta_{\upsilon'} \cap \Delta_{\upsilon}}}}{\Gamma \vdash_{[\overline{\upsilon}''/\overline{\upsilon}']\circ[\sigma/\upsilon]} \upsilon = \sigma}\text{IU}\textsc{Bind}$$

$$\frac{\Gamma \vdash_{\theta} \Xi = \Xi'}{\Gamma \vdash_{\theta'} [= \Xi] = [= \Xi']}\text{IU}\textsc{Type}$$

$$\frac{\Gamma \vdash_{\theta} \Xi = \Xi'}{\Gamma \vdash_{\theta} [\Xi] = [\Xi']}\text{IU}\textsc{Wrap}$$

$$\frac{\Gamma \vdash_{\theta} \overline{\sigma} = \overline{\sigma}'}{\Gamma \vdash_{\theta} \alpha\,\overline{\sigma} = \alpha\,\overline{\sigma}'}\text{IU}\textsc{Path}$$

$$\frac{\Gamma \vdash_{\theta} \overline{\Sigma} = \overline{\Sigma}'}{\Gamma \vdash_{\theta} \{\overline{l{:}\Sigma}\} = \{\overline{l{:}\Sigma'}\}}\text{IU}\textsc{Str}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta_1} \Sigma = \Sigma' \qquad \Gamma; \overline{\alpha}\ {}_{\theta_1}\!\vdash_{\theta_2} \Xi = \Xi'}{\Gamma \vdash_{\theta_2} \forall\overline{\alpha}.\Sigma \to_{\iota} \Xi = \forall\overline{\alpha}.\Sigma' \to_{\iota} \Xi'}\text{IU}\textsc{Fun}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma = \Sigma'}{\Gamma \vdash_{\theta} \exists\overline{\alpha}.\Sigma = \exists\overline{\alpha}.\Sigma'}\text{IU}\textsc{Abs}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma = \Sigma'}{\Gamma \vdash_{\theta} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma = \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma'}\text{IU}\textsc{Impl}$$

Instantiation $\boxed{\Gamma \vdash_{\theta} \Xi' \preceq \Xi \leadsto e[\_]}$

$$\begin{array}{rl}\Gamma \vdash^{!}_{\theta} E :_{\iota} \Xi \leadsto e'[e] \ := & \Gamma \vdash_{\theta} E :_{\iota} \Xi' \leadsto e \wedge\\ & \Gamma\ {}_{\theta}\!\vdash_{\theta'} \Xi' \preceq \Xi \leadsto e'[\_]\end{array}$$

$$\frac{}{\Gamma \vdash_{[]} \Xi \preceq \Xi \leadsto [\_]}\text{IN}\textsc{Refl}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \upsilon \approx \Sigma}{\Gamma \vdash_{\theta} \exists\overline{\alpha}.\upsilon \preceq \exists\overline{\alpha}.\Sigma \leadsto [\_]}\text{IN}\textsc{Res}$$

$$\frac{\begin{array}{c}\overline{\upsilon}\text{ fresh} \qquad \overline{\Delta_{\upsilon} = \mathrm{dom}(\Gamma, \overline{\alpha})}\\ \Gamma \vdash_{\theta} \exists\overline{\alpha}.\Sigma'[\overline{\upsilon}/\overline{\alpha}'] \preceq \exists\overline{\alpha}.\Sigma \leadsto e[\_]\end{array}}{\begin{array}{l}\Gamma \vdash_{\theta} \exists\overline{\alpha}.\forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \preceq \exists\overline{\alpha}.\Sigma\\ \quad \leadsto e[\mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = [\_]\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, x\,\overline{\upsilon}\,\{\}).\mathtt{A})\rangle]\end{array}}\text{IN}\textsc{Impl}$$

E. Correctness of Type Inference

E.1 Soundness

In order to formulate and prove correctness of the type inference rules (Theorem 5.1) properly, we need to refine the statement a little bit. In particular, instead of just bundling together $\overline{\upsilon}, \Gamma$ as an environment, we want to construct an environment that respects the scoping constraints on the free inference variables $\overline{\upsilon}$.

To that end, we define a judgement $\overline{\upsilon} \vdash \Gamma \leadsto \Gamma'$ as follows:

$$\frac{}{\epsilon \vdash \cdot \leadsto \cdot} \qquad \frac{\upsilon \in \overline{\upsilon} \qquad \overline{\upsilon} - \upsilon \vdash \Gamma \leadsto \Gamma' \qquad \Delta_\upsilon \subseteq \mathrm{dom}(\Gamma)}{\overline{\upsilon} \vdash \Gamma \leadsto \Gamma', \upsilon}$$

$$\frac{\overline{\upsilon} \vdash \Gamma \leadsto \Gamma' \qquad \alpha \notin \mathrm{dom}(\Gamma')}{\overline{\upsilon} \vdash \Gamma, \alpha \leadsto \Gamma', \alpha} \qquad \frac{\overline{\upsilon} \vdash \Gamma \leadsto \Gamma' \qquad \Gamma' \vdash \Sigma : \Omega}{\overline{\upsilon} \vdash \Gamma, X{:}\Sigma \leadsto \Gamma', X{:}\Sigma}$$

This assumes that $\mathrm{dom}(\Gamma) \not\pitchfork \overline{\upsilon}$ initially. Γ' is an extension of Γ that includes bindings for $\overline{\upsilon}$, treated as ordinary type variables, and placed such that they adhere to the scoping constraints encoded in Δ. Note that this is a relation: there may be many possible Γ' for a given pair of Γ and $\overline{\upsilon}$ (although they are all equivalent in the sense that they all admit the same set of F_ω typing judgments).

With that, for any F_ω or 1ML typing judgement $\mathcal{J}$, we define

$$\overline{\upsilon}; \Gamma \vdash \mathcal{J} \quad :\Leftrightarrow \quad \exists \Gamma', \overline{\upsilon} \vdash \Gamma \leadsto \Gamma' \wedge \Gamma' \vdash \mathcal{J}$$

Furthermore, define

$$\begin{aligned} \overline{\upsilon}'; \Gamma' \vdash \theta : \overline{\upsilon}; \Gamma \quad :\Leftrightarrow \quad & \overline{\upsilon} \vdash \Gamma \leadsto \Gamma'' \wedge \overline{\upsilon}'; \Gamma' \vdash \theta : \Gamma'' \wedge \\ & \theta \text{ small} \wedge \mathrm{dom}(\theta) \not\pitchfork \overline{\upsilon}' \wedge \\ & \forall \upsilon \in \overline{\upsilon}, \forall \upsilon'' \in \mathrm{undet}(\theta\upsilon), \Delta_{\upsilon''} \subseteq \Delta_\upsilon \end{aligned}$$

Well-typed substitutions can be composed:

LEMMA E.1 (Composition of Substitutions).
If $\overline{\upsilon}'; \Gamma' \vdash \theta : \overline{\upsilon}; \Gamma$ and $\overline{\upsilon}''; \Gamma'' \vdash \theta' : \overline{\upsilon}'; \Gamma'$ and $\mathrm{dom}(\theta) \not\pitchfork \overline{\upsilon}''$, *then $\overline{\upsilon}''; \Gamma'' \vdash \theta' \circ \theta : \overline{\upsilon}; \Gamma$.*

The correctness theorem in its full beauty is stated as follows:

THEOREM E.2 (Soundness of 1ML Inference).
Let $\mathrm{dom}(\Gamma) \not\pitchfork \overline{\upsilon}$ *and $\overline{\upsilon} \vdash \Gamma \leadsto \Gamma'$. Let $\mathcal{J}$ range over the 1ML inference judgements.*

1. *Γ' is well-formed and a permutation of $\overline{\upsilon}, \Gamma$.*
2. *If $\theta\Gamma \vdash_{\theta'} {}^{\theta}\mathcal{J}$, then $\theta\Gamma \vdash_{\theta'} \theta\mathcal{J}$.*
3. *If $\Gamma \vdash_\theta \mathcal{J}$, then $\overline{\upsilon}'; \theta\Gamma \vdash \theta : \overline{\upsilon}; \Gamma$ with $\overline{\upsilon}' - \overline{\upsilon}$ fresh.*
4. *If $\Gamma\ {}_\theta{\vdash}_{\theta'}\ \mathcal{J}$ and $\overline{\upsilon}'; \theta\Gamma \vdash \theta : \overline{\upsilon}; \Gamma$, then $\overline{\upsilon}''; \theta'\Gamma \vdash \theta' : \overline{\upsilon}; \Gamma$ with $\overline{\upsilon}'' - \overline{\upsilon}' - \overline{\upsilon}$ fresh.*
5. *If $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_1; \Gamma_2\ {}_\theta{\vdash}_{\theta'}\ \mathcal{J}$ and $\overline{\upsilon}'; \theta\Gamma \vdash \theta : \overline{\upsilon}; \Gamma$, then $\overline{\upsilon}''; \theta'\Gamma \vdash \theta' : \overline{\upsilon}; \Gamma$ with $\overline{\upsilon}'' - \overline{\upsilon}' - \overline{\upsilon}$ fresh and $\Delta_{\overline{\upsilon}''} \subseteq$* $\mathrm{dom}(\Gamma_1)$.
6. *If $\Gamma \vdash_\theta T/D \leadsto \Xi$, then $\overline{\upsilon}'; \theta\Gamma \vdash T/D \leadsto \theta\Xi$.*
7. *If $\Gamma \vdash_\theta E/B : \Xi \leadsto e$, then $\overline{\upsilon}'; \theta\Gamma \vdash E/B : \theta\Xi \leadsto \theta e$.*
8. *If $\Gamma \vdash_\theta \Xi \leq_{\overline{\pi}} \Xi' \leadsto \delta; f$, and $\overline{\upsilon}; \Gamma \vdash \Xi : \Omega$ and $\overline{\upsilon}; \Gamma \vdash \Xi' : \Omega$, then $\overline{\upsilon}'; \theta\Gamma \vdash \theta\Xi \leq_{\overline{\pi}} \theta\Xi' \leadsto \theta\delta; \theta f$.*
9. *If $\Gamma \vdash_\theta \Xi \preceq \Xi' \leadsto e'[\_]$, and $\overline{\upsilon}; \Gamma \vdash \Xi : \Omega$ and $\overline{\upsilon}; \Gamma \vdash \Xi' : \Omega$ and $\overline{\upsilon}; \Gamma \vdash E : \Xi \leadsto e$, then $\overline{\upsilon}'; \theta\Gamma \vdash E : \theta\Xi' \leadsto \theta e'[\theta e]$.*
10. *If $\Gamma \vdash_\theta \Xi = \Xi'$, and $\overline{\upsilon}; \Gamma \vdash \Xi : \Omega$ and $\overline{\upsilon}; \Gamma \vdash \Xi' : \Omega$, then $\theta\Xi = \theta\Xi'$.*
11. *If $\Gamma \vdash_\theta \upsilon \approx \Sigma$, then $\theta\upsilon$ has the same outer shape as Σ.*

The proof of the first part is by induction on the derivation of $\overline{\upsilon} \vdash \Gamma \leadsto \Gamma'$, for the others by simultaneous induction on the (first) derivation. We omit the gory details.

E.2 Termination

As for the $1\mathrm{ML}_{\mathrm{ex}}$ typing rules, termination is obvious for most the 1ML inference judgements: the main ones are inductive on the syntactic structure of the language; for the auxiliary Unification judgement, termination can be proved in the usual manner, Instantiation is inductive on the right-hand side type, and Resolution is not even recursive. Again, it only remains subtyping as the problem child.

We can prove its termination by extending the prove from Appendix B. First, we trivially extend the weight function to handle inference variables:

$$S[\![\upsilon]\!] \;=\; 1 \qquad\qquad Q[\![\upsilon]\!] \;=\; 0$$

The Weight Reduction lemma still holds for δ-substitutions on type variables, but we can formulate it analogously for θ-substitutions on inference variables (which are always small):

LEMMA E.3 (Weight Reduction under Small Resolution).

1. *$Q[\![\theta\Xi]\!] = Q[\![\Xi]\!]$.*
2. *$W[\![\theta\Xi]\!] < \langle 1, 0\rangle + W[\![\Xi]\!]$.*
3. *$W[\![\theta\Xi]\!] \leq \langle |\theta|, 0\rangle + W[\![\Xi]\!]$.*

The following key lemma postulates that a subtyping derivation either resolves more inference variables than it introduces, or the amount of excess variables (which will come from rule ISIMPLL) is bounded by the number of quantifiers in the types involved.

LEMMA E.4 (Resolution Progress in Subtyping Inference).
Let $\Gamma \vdash_\theta \mathcal{J}$ an inference derivation and Y the set of fresh inference variables generated by it.

1. *If $\mathcal{J}$ is $\Xi' = \Xi$, then either $|Y| = |\theta| = 0$ or $|Y| < |\theta|$.*
2. *If $\mathcal{J}$ is $\Xi' \leq_{\overline{\pi}} \Xi \leadsto \delta; f$, then either $|Y| = |\theta| = 0$ or $|Y| < |\theta|$ or $|Y| - |\theta| \leq Q[\![\Xi']\!] + Q[\![\Xi]\!]$.*
   Furthermore, if $\Xi \neq \Xi'$ and $\Xi = \upsilon$ or $\Xi' = \upsilon$, then $\upsilon \in$ $\mathrm{dom}(\theta)$.

With this, the termination proof can proceed similar to before, except that we have to take θ into account.

LEMMA E.5 (Termination of Subtyping Inference).
Let $\overline{\upsilon} \vdash \Gamma \leadsto \Gamma'$ and $\Gamma' \vdash \Xi' : \Omega$ and $\Gamma', \overline{\alpha} \vdash \Xi : \Omega$ and $\overline{\pi} = \overline{\alpha\, \overline{\alpha}'}$. Then $\Gamma \vdash_\theta \Xi' \leq_{\overline{\pi}} \Xi \leadsto \delta; f$ terminates.

The proof is again by case analysis on the algorithm, mostly as before, but this time using the weight $W[\![\Xi']\!] + W[\![\Xi]\!] + \langle |\overline{\pi}| + |\overline{\upsilon}'|, 0\rangle$, with $\overline{\upsilon}' = \mathrm{undet}(\Xi') \cup \mathrm{undet}(\Xi)$. In each rule, this weight gets smaller for all premises. The only exceptions are the rules ISRESL and ISRESR, which invoke the Resolution judgment that may introduce additional temporary inference variables. That locally increases the weight of the judgement, but we can show by case analysis of the Σ in these rules that the temporary variables are resolved immediately and weight will decrease again in the recursive invocation. For example, consider rule ISRESL for the case $\Sigma = \forall\overline{\alpha}.\Sigma_1 \to_\iota \Xi_2$ as one representative case:

- the weight is $w = \langle 0, 1\rangle + W[\![\Sigma]\!] + \langle |\overline{\upsilon}'|\rangle = W[\![\Sigma_1]\!] + W[\![\Xi_2]\!] + \langle |\overline{\alpha}| + |\overline{\upsilon}'|, 2\rangle$
- by inverting the first premise of ISRESL, (1) $\upsilon \notin \mathrm{undet}(\Sigma_1) \cup \mathrm{undet}(\Xi_2)$ and (2) $\Gamma \vdash_\theta \upsilon \approx \Sigma$
- the only rule that applies for (2) is IRFUN
- hence, $\theta = [(\upsilon_1 \to_{\mathrm{I}} \upsilon_2)/\upsilon]$ for fresh υ_1, υ_2
- because of (1), $\theta\Sigma = \Sigma$
- hence, the second premise of ISRESL is invoked as $\theta\Gamma \vdash_{\theta''} (\upsilon_1 \to_{\mathrm{I}} \upsilon_2) \leq (\forall\overline{\alpha}.\Sigma_1 \to_\iota \Xi_2)$
- because of (1), $\Sigma \neq \upsilon_1 \to_{\mathrm{I}} \upsilon_2$, so only rule ISFUN applies

- its first premise is invoked as $\Gamma, \overline{\alpha} \vdash \Sigma_1 \leq_{\epsilon} \upsilon_1$, producing θ_1 (3)
- because of (1), $\upsilon_1, \upsilon_2 \notin \mathrm{undet}(\Sigma_1) = \mathrm{undet}(\theta\Sigma_1)$
- so $\overline{\upsilon}_1 = \mathrm{undet}(\Sigma_1) \cup \mathrm{undet}(\upsilon_1) \subseteq (\overline{\upsilon}' - \upsilon) \cup \upsilon_1$
- that is, $|\overline{\upsilon}_1| \leq |\overline{\upsilon}'|$
- the weight for the first premise of ISFUN is $w_1 = W[\![\Sigma_1]\!] + \langle 0, 1\rangle + \langle |\overline{\upsilon}_1|, 0\rangle \leq W[\![\Sigma_1]\!] + \langle |\overline{\upsilon}'|, 1\rangle$
- $w_1 < w$, so the first premise terminates (with empty δ_1)
- let Y_1 be the set of fresh inference variables generated by it
- the second premise of ISFUN is invoked as $\theta_1\Gamma, \overline{\alpha} \vdash \upsilon_2 \leq_{\epsilon} \theta_1\Xi_2$ (after expanding the judgement abbreviation), yielding θ_2
- by Lemma E.4, $\upsilon_1 \in \mathrm{dom}(\theta_1)$, and thus, $\upsilon_1 \notin \mathrm{undet}(\theta_1\Xi_2)$
- so $\overline{\upsilon}_2 = \mathrm{undet}(\upsilon_2) \cup \mathrm{undet}(\theta_1\Xi_2) \subseteq \upsilon_2 \cup (\overline{\upsilon}' - \upsilon) \cup Y_1 - \mathrm{dom}(\theta_1)$
- that is, $|\overline{\upsilon}_2| \leq |\overline{\upsilon}'| + |Y_1| - |\theta_1|$
- the weight for the second premise of ISFUN is $w_2 \leq \langle 0, 1\rangle + W[\![\theta_1\Xi_2]\!] + \langle |\overline{\upsilon}_2|, 0\rangle \leq W[\![\theta_1\Xi_2]\!] + \langle |\overline{\upsilon}'| + |Y_1| - |\theta_1|, 1\rangle$
- split by Lemma E.4 applied to (3):
  - if $|Y_1| = |\theta_1| = 0$, then $w_2 \leq W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'|, 1\rangle < w$
  - if $|Y_1| < |\theta_1|$, then with Weight Reduction, $w_2 < \langle 1, 0\rangle + W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'| + |Y_1| - |\theta_1|, 1\rangle \leq W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'|, 1\rangle < w$
  - if $|Y_1| - |\theta_1| < Q[\![\Sigma_1]\!] + Q[\![\upsilon_1]\!]$, then with Weight Reduction, $w_2 < \langle 1, 0\rangle + W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'| + |Y_1| - |\theta_1|, 1\rangle \leq W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'|, 1\rangle + Q[\![\Sigma_1]\!] < W[\![\Xi_2]\!] + \langle |\overline{\upsilon}'|, 1\rangle + W[\![\Sigma_1]\!] < w$
- in all cases, $w_2 < w$, so the second premise of ISFUN terminates as well

Abbreviations for Environment Abstraction

$$
\begin{array}{lll}
 & \text{(kinds)} & \begin{array}{lcl} (\cdot) \to \kappa & := & \kappa \\ (\Gamma,\alpha) \to \kappa & := & \Gamma \to \kappa_\alpha \to \kappa \\ (\Gamma, x{:}\tau) \to \kappa & := & \Gamma \to \kappa \end{array}
\quad \text{(types)} \begin{array}{lcl} \lambda(\cdot).\tau & := & \tau \\ \lambda(\Gamma,\alpha).\tau & := & \lambda\Gamma.\lambda\alpha.\tau \\ \lambda(\Gamma, x{:}\tau').\tau & := & \lambda\Gamma.\tau \end{array}
\quad \text{(terms)} \begin{array}{lcl} \lambda(\cdot).e & := & e \\ \lambda(\Gamma,\alpha).e & := & \lambda\Gamma.\lambda\alpha.e \\ \lambda(\Gamma,x{:}\tau).e & := & \lambda\Gamma.\lambda_{\mathtt{P}}x{:}\tau.e \end{array} \\
\begin{array}{lcl} \Gamma^{\mathtt{I}} & := & \cdot \\ \Gamma^{\mathtt{P}} & := & \Gamma \end{array} & \text{(types)} & \begin{array}{lcl} \forall(\cdot).\tau & := & \tau \\ \forall(\Gamma,\alpha).\tau & := & \forall\Gamma.\forall\alpha.\tau \\ \forall(\Gamma,x{:}\tau').\tau & := & \forall\Gamma.\tau' \to_{\mathtt{P}} \tau \end{array}
\quad \phantom{\text{(types)}} \begin{array}{lcl} \tau\ (\cdot) & := & \tau \\ \tau\ (\Gamma,\alpha) & := & \tau\ \Gamma\ \alpha \\ \tau\ (\Gamma, x{:}\tau') & := & \tau\ \Gamma \end{array}
\quad \phantom{\text{(terms)}} \begin{array}{lcl} e\ (\cdot) & := & e \\ e\ (\Gamma,\alpha) & := & e\ \Gamma\ \alpha \\ e\ (\Gamma,x{:}\tau) & := & (e\ \Gamma\ x).\mathtt{P} \end{array}
\end{array}
$$

Types $\boxed{\Gamma \vdash T \leadsto \Xi}$

$$
\frac{\Gamma \vdash E :_{\mathtt{P}} \exists\overline{\alpha}.[=\Xi] \leadsto e \qquad \overline{\alpha} \not\cap \mathrm{fv}(\Xi)}{\Gamma \vdash E \leadsto \Xi}\,\textsc{TPath}
\qquad
\frac{\Gamma \vdash E :_{\mathtt{P}} \exists\overline{\alpha}.\Sigma \leadsto e \qquad \overline{\alpha} \not\cap \mathrm{fv}(\Sigma)}{\Gamma \vdash (=E) \leadsto \Sigma}\,\textsc{TSing}
$$

Expressions $\boxed{\Gamma \vdash E :_\iota \Xi \leadsto e}$

$$
\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X :_{\mathtt{P}} \Sigma \leadsto \lambda\Gamma.X}\,\textsc{EVar}
\qquad
\frac{\Gamma \vdash T \leadsto \Xi}{\Gamma \vdash \textbf{type}\ T :_{\mathtt{P}} [=\Xi] \leadsto \lambda\Gamma.[\Xi]}\,\textsc{EType}
\qquad
\frac{}{\Gamma \vdash \textbf{true} :_{\mathtt{P}} \mathsf{bool} \leadsto \lambda\Gamma.\mathsf{true}}\,\textsc{ETrue}
$$

$$
\frac{}{\Gamma \vdash \textbf{false} :_{\mathtt{P}} \mathsf{bool} \leadsto \lambda\Gamma.\mathsf{false}}\,\textsc{EFalse}
\qquad
\frac{\begin{array}{lll} \Gamma \vdash X :_{\mathtt{P}} \mathsf{bool} \leadsto e & \Gamma \vdash E_1 :_{\iota_1} \Xi_1 \leadsto e_1 & \Gamma \vdash \Xi_1 \leq \Xi \leadsto f_1 \\ \Gamma \vdash T \leadsto \Xi & \Gamma \vdash E_2 :_{\iota_2} \Xi_2 \leadsto e_2 & \Gamma \vdash \Xi_2 \leq \Xi \leadsto f_2 \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{if}\ X\ \textbf{then}\ E_1\ \textbf{else}\ E_2 : T :_{\iota_1 \vee \iota_2 \vee \iota(\Xi)} \Xi \\ \qquad \leadsto \lambda\Gamma^{\iota_1 \vee \iota_2 \vee \iota(\Xi)}.\ \mathsf{if}\ e\,\Gamma\ \mathsf{then}\ f_1\ (e_1\,\Gamma^{\iota_1})\ \mathsf{else}\ f_2\ (e_2\,\Gamma^{\iota_2}) \end{array}}\,\textsc{EIf}
$$

$$
\frac{\Gamma \vdash B :_\iota \Xi \leadsto e}{\Gamma \vdash \{B\} :_\iota \Xi \leadsto e}\,\textsc{EStr}
\qquad
\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\{\overline{X'{:}\Sigma'}\} \leadsto e \qquad X{:}\Sigma \in \overline{X'{:}\Sigma'}}{\Gamma \vdash E.X :_\iota \exists\overline{\alpha}.\Sigma \leadsto \mathsf{unpack}\ \langle\overline{\alpha}, y\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \lambda\Gamma^\iota.\ (y\,\Gamma^\iota).X\rangle}\,\textsc{EDot}
$$

$$
\frac{\Gamma \vdash T \leadsto \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash E :_{\mathtt{I}} \Xi \leadsto e}{\Gamma \vdash \textbf{fun}\,(X{:}T) \Rightarrow E :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma \to_{\mathtt{I}} \Xi \leadsto \lambda\Gamma.\lambda\overline{\alpha}.\lambda_\iota X{:}\Sigma.e}\,\textsc{EFun}
\qquad
\frac{\Gamma \vdash T \leadsto \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash E :_{\mathtt{P}} \exists\overline{\alpha}_2.\Sigma_2 \leadsto e}{\Gamma \vdash \textbf{fun}\,(X{:}T) \Rightarrow E :_{\mathtt{P}} \exists\overline{\alpha}_2.\forall\overline{\alpha}.\,\Sigma \to_{\mathtt{P}} \Sigma_2 \leadsto e}\,\textsc{EPfun}
$$

$$
\frac{\begin{array}{ll} \Gamma \vdash X_1 :_{\mathtt{P}} \forall\overline{\alpha}.\,\Sigma_1 \to_\iota \Xi \leadsto e_1 & \\ \Gamma \vdash X_2 :_{\mathtt{P}} \Sigma_2 \leadsto e_2 & \Gamma \vdash \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \leadsto \delta; f \end{array}}{\Gamma \vdash X_1\ X_2 :_\iota \delta\Xi \leadsto \lambda\Gamma^\iota.\ (e_1\,\Gamma\ (\delta\overline{\alpha})\ (f\ (e_2\ \Gamma))).\iota}\,\textsc{EApp}
$$

$$
\frac{\Gamma \vdash X :_{\mathtt{P}} \Sigma_1 \leadsto e \qquad \Gamma \vdash T \leadsto \exists\overline{\alpha}.\Sigma_2 \qquad \Gamma \vdash \Sigma_1 \leq_{\overline{\alpha}} \Sigma_2 \leadsto \delta; f \qquad \overline{\kappa_{\alpha'} = \Gamma \to \kappa_\alpha}}{\Gamma \vdash X{:>}T :_{\mathtt{P}} \exists\overline{\alpha}'.\Sigma_2[\overline{\alpha'\,\Gamma/\alpha}] \leadsto \mathsf{pack}\ \langle\lambda\Gamma.\delta\overline{\alpha}, \lambda\Gamma.f\ (e\,\Gamma)\rangle}\,\textsc{ESeal}
$$

$$
\frac{\Gamma \vdash X :_{\mathtt{P}} \Sigma \leadsto e \qquad \Gamma \vdash T \leadsto [\Xi] \qquad \Gamma \vdash \Sigma \leq \Xi \leadsto f}{\Gamma \vdash \textbf{pack}\ X{:}T :_{\mathtt{P}} [\Xi] \leadsto \lambda\Gamma.[f\ (e\,\Gamma)]}\,\textsc{EPack}
\qquad
\frac{\Gamma \vdash X :_{\mathtt{P}} [\Xi'] \leadsto e \qquad \Gamma \vdash T \leadsto [\Xi] \qquad \Gamma \vdash \Xi' \leq \Xi \leadsto f}{\Gamma \vdash \textbf{unpack}\ X{:}T :_{\iota(\Xi)} \Xi \leadsto \lambda\Gamma^{\iota(\Xi)}.\ f\ ((e\,\Gamma).\mathsf{val})}\,\textsc{EUnp}
$$

$$
\frac{\Gamma, \overline{\alpha}' \vdash E :_{\mathtt{P}} \exists\overline{\alpha}.\Sigma \leadsto e \qquad \overline{\kappa_{\alpha'} = \Omega}}{\Gamma \vdash E :_{\mathtt{P}} \exists\overline{\alpha}.\forall\overline{\alpha}'.\Sigma \leadsto e}\,\textsc{EGen}
\qquad
\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\forall\overline{\alpha}'.\Sigma \leadsto e \qquad \overline{\Gamma, \overline{\alpha} \vdash \sigma : \kappa_{\alpha'}}}{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\Sigma[\overline{\sigma}/\overline{\alpha}'] \leadsto \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \lambda\Gamma^\iota.x\,\Gamma^\iota\,\overline{\sigma}\rangle}\,\textsc{EInst}
$$

Bindings $\boxed{\Gamma \vdash B :_\iota \Xi \leadsto e}$

$$
\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\Sigma \leadsto e}{\Gamma \vdash X{=}E :_\iota \exists\overline{\alpha}.\{X{:}\Sigma\} \leadsto \mathsf{unpack}\ \langle\overline{\alpha}, x\rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle\overline{\alpha}, \lambda\Gamma^\iota.\{X{=}x\,\Gamma^\iota\}\rangle}\,\textsc{BVar}
\qquad
\frac{\Gamma \vdash E :_\iota \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \leadsto e}{\Gamma \vdash \textbf{include}\ E :_\iota \exists\overline{\alpha}.\{\overline{X{:}\Sigma}\} \leadsto e}\,\textsc{BIncl}
$$

$$
\frac{\begin{array}{rl} \Gamma \vdash B_1 :_{\iota_1} \exists\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \leadsto e_1 & \overline{X}_1' = \overline{X}_1 - \overline{X}_2 \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash B_2 :_{\iota_2} \exists\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \leadsto e_2 & \overline{X_1'{:}\Sigma_1'} \subseteq \overline{X_1{:}\Sigma_1} \end{array}}{\begin{array}{l} \Gamma \vdash B_1{;}B_2 :_{\iota_1 \vee \iota_2} \exists\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X_1'{:}\Sigma_1'}, \overline{X_2{:}\Sigma_2}\} \\ \quad \leadsto \mathsf{unpack}\ \langle\overline{\alpha}_1, y_1\rangle = e_1\ \mathsf{in} \\ \qquad \mathsf{unpack}\ \langle\overline{\alpha}_2, y_2\rangle = (\mathsf{let}\ \overline{X_1 = \lambda\Gamma^{\iota_1 \vee \iota_2}.\ (y_1\,\Gamma^{\iota_1}).X_1}\ \mathsf{in}\ e_2)\ \mathsf{in} \\ \qquad \mathsf{pack}\ \langle\overline{\alpha}_1\overline{\alpha}_2, \lambda\Gamma^{\iota_1 \vee \iota_2}.\ \mathsf{let}\ \overline{X_1 = (y_1\,\Gamma^{\iota_1}).X_1}\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ \overline{X_2 = (y_2\,(\Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1})^{\iota_2}).X_2}\ \mathsf{in}\ \{\overline{X_1' = X_1'}, \overline{X_2 = X_2}\}\rangle \end{array}}\,\textsc{BSeq}
\qquad
\frac{}{\Gamma \vdash \epsilon :_{\mathtt{P}} \{\} \leadsto \lambda\Gamma.\{\}}\,\textsc{BEmpty}
$$

Figure 8. Elaboration of 1ML with Pure Sealing (rules not shown are unchanged)

F. 1ML with Pure Sealing

Figure 8 shows how to modify the 1ML elaboration rules from Figures 4, 7, and 5 to support a semantics where sealing is a *pure* operation. The actual changes are highlighted.

The new rules are mostly straight out of [25], Section 7. Regarding typing, the main change over plain 1ML is the modification to rule ESEAL for sealing and the addition of rule FPFUN for pure functions. ESEAL now skolemises the abstract variables over the whole environment Γ. This is so that for any use of sealing occurring inside a function, the abstract types are already skolemised over that function's parameters. Rule EPFUN can thus lift the existential quantifier for the local abstract types $\overline{\alpha}_2$ from the body.

The translation invariant for expression and bindings changes so that $\Gamma \vdash E/B :_\iota \exists\overline{\alpha}.\Sigma \leadsto e$ implies $\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$ if $\iota = \mathtt{I}$ (as before), but $\cdot \vdash e : \exists\overline{\alpha}.\forall\Gamma.\Sigma$ if $\iota = \mathtt{P}$ (see [25] for an extensive explanation of this approach). Consequently, the gist of the changes is in the term translation. which now has to abstract over Γ for all pure expressions, and systematically apply and reabstract Γ to get to the actual value where necessary – for pure expressions, the $\forall\Gamma$ becomes part of the "abstraction monad".

This is possible because sealing is the only pure operation that can create quantifiers in the modified system: as motivated in Section 3, all other expressions introducing quantifiers (conditionals in rule EIF, application in rule RAPP, and also unwrapping in rule RUNW) continue to be treated as impure if quantifiers are present. The reason for this is that they are not *phase-separable*, and no translation to System F_ω could be given that would allow them to be treated as pure. The new rules render (only) sealing pure, exactly by making it separable, which allows maintaining the crucial invariant that "purity" implies phase separation.

Rules TPATH and TSING now have to allow for the possibility of pure expressions creating quantifiers through sealing. That is fine, as long as the abstract names are purely local and do not escape from E. Similarly, quantifiers may occur in rule EGEN, where they just remain, in the same spirit as for ordinary pure functions.

The only practical complication of this system over what is discussed in [25], Section 7, is the interaction of sealing with implicit generalisation: the type derived by rule ESEAL depends on Γ, which includes all variables $\overline{\alpha}'$ that are possibly introduced by uses of rule EGEN in scope. However, both the use of the latter rule, and the sequence of variables it introduces are non-deterministic. In practice, a type inference algorithm will only determine both *after* type-checking an actual expression. If that expression uses sealing, the inference algorithm would hence need to be able to defer the exact choice of Γ in rule ESEAL. This is certainly doable, but we leave the formal refinement of this idea to future work.

The rules from Figure 8 define a semantics for sealing that is not *abstraction-safe* when used locally inside a function – see [25], Section 8, for a discussion of the issue, and an approach for solving it in conventional ML. Unfortunately, the solution is not trivial to integrate with type inference of the form required for 1ML, because it necessitates existential quantifiers to occur in many additional types, including ones we'd like to consider small. We leave solving this to future work as well.

$$
\begin{array}{lcl}
\text{(kinds)} & & \\
\llbracket \Omega \rrbracket & = & \textbf{type} \\
\llbracket \kappa_1 \to \kappa_2 \rrbracket & = & \llbracket \kappa_1 \rrbracket \Rightarrow \llbracket \kappa_2 \rrbracket \\
\text{(types)} & & \\
\llbracket \alpha \rrbracket & = & X_\alpha \\
\llbracket \tau_1 \to \tau_2 \rrbracket & = & \textbf{type}\,(\llbracket \tau_1 \rrbracket \to \llbracket \tau_2 \rrbracket) \\
\llbracket \tau_1 \times \tau_2 \rrbracket & = & \textbf{type}\,(\llbracket \tau_1 \rrbracket,\ \llbracket \tau_2 \rrbracket) \\
\llbracket \forall \alpha{:}\kappa.\tau \rrbracket & = & \textbf{type}\,(\textbf{wrap}\,((X_\alpha : \llbracket \kappa \rrbracket) \Rightarrow \llbracket \tau \rrbracket)) \\
\llbracket \exists \alpha{:}\kappa.\tau \rrbracket & = & \textbf{type}\,(\textbf{wrap}\,(X_\alpha = \llbracket \kappa \rrbracket,\ \llbracket \tau \rrbracket)) \\
\llbracket \lambda \alpha{:}\kappa.\tau \rrbracket & = & \textbf{fun}\ (X_\alpha : \llbracket \kappa \rrbracket) \Rightarrow \llbracket \tau \rrbracket \\
\llbracket \tau_1\, \tau_2 \rrbracket & = & \llbracket \tau_1 \rrbracket\, \llbracket \tau_2 \rrbracket \\
\text{(expressions)} & & \\
\llbracket x \rrbracket & = & X_x \\
\llbracket \lambda x{:}\tau.e \rrbracket & = & \textbf{fun}\ (X_x : \llbracket \tau \rrbracket) \Rightarrow \llbracket e \rrbracket \\
\llbracket \langle e_1, e_2 \rangle \rrbracket & = & (\llbracket e_1 \rrbracket,\ \llbracket e_2 \rrbracket) \\
\llbracket \lambda \alpha{:}\kappa.e \rrbracket & = & \textbf{wrap}\,(\textbf{fun}\ (X_\alpha : \llbracket \kappa \rrbracket) \Rightarrow \llbracket e \rrbracket) \\
 & & \qquad : (X_\alpha : \llbracket \kappa \rrbracket) \Rightarrow T_e \\
\llbracket \langle \tau_1, e_2 \rangle_\tau \rrbracket & = & \textbf{wrap}\,(\llbracket \tau_1 \rrbracket,\ \llbracket e_2 \rrbracket) : \llbracket \tau \rrbracket \\
\llbracket e_1\, e_2 \rrbracket & = & \llbracket e_1 \rrbracket\, \llbracket e_2 \rrbracket \\
\llbracket e_1\, \tau_2 \rrbracket & = & (\textbf{unwrap}\,\llbracket e_1 \rrbracket : T_1)\, \llbracket \tau_2 \rrbracket \\
\llbracket e.i \rrbracket & = & \llbracket e \rrbracket.i \\
\llbracket \mathsf{unpack}\ \langle \alpha, x \rangle {=} e_1\ \mathsf{in}\ e_2 \rrbracket & = & \textbf{let}\,(X_\alpha, X_x) = \textbf{unwrap}\,\llbracket e_1 \rrbracket : T_1 \\
 & & \textbf{in}\ \llbracket e_2 \rrbracket
\end{array}
$$

where:

$$
\begin{array}{lcl}
(X = T_1, T_2) & := & \{\_1 : T_1;\ \_2 : \textbf{let}\ X = \_1\ \textbf{in}\ T_2\} \\
(E_1, E_2) & := & \{\_1 = E_1;\ \_2 = E_2\} \\
E.i & := & E.\_i \\
\textbf{let}\,(X_1, X_2) = E_1\ \textbf{in}\ E_2 & := & \textbf{let}\ X = E_1; \\
 & & \quad X_1 = X.\_1;\ X_2 = X.\_2\ \textbf{in}\ E_2
\end{array}
$$

Figure 9. Embedding of F_ω in 1ML

G. Embedding F_ω

The elaboration of 1ML embeds the language into System F_ω, showing that it is no more expressive than that calculus. We can also show that it is no *less* expressive, by doing the inverse: embedding F_ω into 1ML.

Doing so is fairly straightforward: Figure 9 gives the canonical translation function $\llbracket \_ \rrbracket$ for F_ω kinds, types, and terms.

F_ω kinds are translated to suitable 1ML types. F_ω types are translated to 1ML expressions; in the case that the type has ground kind Ω, the resulting expression will have a type of the form $[= \sigma]$, so that it can be used as a 1ML type; otherwise, it will be a function returning such a value. Finally, F_ω terms are translated into 1ML expressions, as is to be expected.

All translated terms become 1ML expressions with a small type, and consequently, ground types translate to small types as well. This is necessary to encompass impredicativity in F_ω. To achieve that for quantified types, the translation wraps each of them, using the extension from Section C – if we were to translate only a predicative version of F_ω, then the uses of **wrap** in the figure could be dropped (and the use in existential formation would be replaced by sealing). Unfortunately wrapped expressions require a type annotation; we assume that the types T_e and T_1 used in the figure are determined by context (as usual, this could be made more precise via a type-directed translation, but we chose to gloss over this detail for the sake of simple exposition).

To state correctness of the embedding, we also need to define an embedding of F_ω environments. It modifies variable bindings in such a way that they match the requirements of the 1ML typing rules:

$$
\begin{array}{lcll}
\llbracket\, \cdot\, \rrbracket & = & \cdot & \\
\llbracket \Gamma, x{:}\tau \rrbracket & = & \llbracket \Gamma \rrbracket, X_x{:}\sigma & \text{if } \llbracket \Gamma \rrbracket \vdash \llbracket \tau \rrbracket : [= \sigma] \\
\llbracket \Gamma, \alpha{:}\kappa \rrbracket & = & \llbracket \Gamma \rrbracket, \alpha{:}\kappa, X_\alpha{:}\Sigma & \text{if } \llbracket \Gamma \rrbracket \vdash \llbracket \kappa \rrbracket \rightsquigarrow \exists \alpha{:}\kappa.\Sigma
\end{array}
$$

With this, we can show that the embedding is sound, i.e., produces well-formed 1ML programs from well-formed F_ω terms:

THEOREM G.1 (Soundness of Embedding). *Let* $\Gamma \vdash \Box$.

1. $\llbracket \Gamma \rrbracket \vdash \Box$.
2. $\cdot \vdash \llbracket \kappa \rrbracket \rightsquigarrow \exists \alpha{:}\kappa.\Sigma$ *with* $\cdot \vdash \exists \alpha.\Sigma \leq \exists \alpha.\Sigma$.
3. *If* $\Gamma \vdash \tau : \kappa$, *then* $\llbracket \Gamma \rrbracket \vdash \llbracket \tau \rrbracket : \Sigma$ *and* $\llbracket \Gamma \rrbracket \vdash \llbracket \kappa \rrbracket \rightsquigarrow \Xi$ *and* $\llbracket \Gamma \rrbracket \vdash \Sigma \leq \Xi$ *(the latter implies that if* $\kappa = \Omega$ *then* $\Sigma = [= \sigma]$*).*
4. *If* $\Gamma \vdash e : \tau$, *then* $\llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket : \sigma$ *and* $\llbracket \Gamma \rrbracket \vdash \llbracket \tau \rrbracket : [= \sigma]$.

The proof is by simultaneous induction on (1) the derivations of $\Gamma \vdash \Box$, (2) the structure of κ, and the derivations of (3) $\Gamma \vdash \tau : \kappa$ and (4) $\Gamma \vdash e : \tau$, respectively.

For a fully pedantic proof of correctness we would also need to show that the embedding is adequate, i.e., the produced 1ML programs are computationally equivalent to the original F_ω terms. This should be possible through a suitable logical relation, but would be involved, requiring simultaneous reasoning about the embedding just defined *and* the 1ML elaboration back into F_ω. Given the limited relevance of this result, and its informal “obviousness”, it seems acceptable to punt.