Under consideration for publication in J. Functional Programming

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

ANDREAS ROSSBERG
Dfinity
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 imposes seemingly unnecessary limits on expressiveness because it makes modules second-class citizens of the language. For example, selecting one among several possible modules implementing a given interface 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 and tend to be even more syntactically heavyweight than using second-class modules alone.

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 needed 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: 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.

1 Introduction

The ML family of languages is defined by two splendid innovations: parametric polymorphism with Damas/Milner-style type inference (Milner, 1978; Damas & Milner, 1982), and an advanced module system based on concepts from dependent type theory (MacQueen, 1986). 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 total sense from a historical perspective. Modules were introduced for programming-in-the-large, when the core language already existed. The dependently typed 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 demonstrated that ML-style module types can be readily expressed using only System-F-style quantification (Russo, 1999; Russo, 2003). 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 (Rossberg *et al.*, 2014).

Meanwhile, the second-class nature of modules is increasingly perceived as a practical limitation. The standard example is the desire 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 conventional ML modules. Isn't that a shame!

1.1 Packaged Modules

It comes as 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 (Russo, 2000; Dreyer *et al.*, 2003; Rossberg, 2006; Rossberg & Dreyer, 2013; Garrigue & Frisch, 2010; Rossberg *et al.*, 2014). 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, which implements a simple form of packaged modules, the above example would have to be written as follows:

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

which, arguably, is neither natural nor pretty. Less straightforward examples require even more annotations, because package types cannot always be inferred from context.

But it is not just notation: packaged modules have limited expressiveness as well. One problem is that the subtyping of the module language does not extend to package types, because they are part of the ML core language which does not have subtyping. For example, if HashMap and TreeMap had different signatures HASHMAP and TREEMAP, respectively, both of which are extensions of MAP, then changing the above to

```
let hashmap = (module HashMap : HASHMAP)
let treemap = (module TreeMap : TREEMAP)
module Table = (val (if size > threshold then hashmap else treemap) : MAP)
```

would not type-check. The consumer of the packages would need to unpack and repack them with their use-site types to allow subtyping to kick in, as in the following:

```
module Table = (val (if size > threshold
                     then (module (val hashmap))
                     else (module (val treemap))) : MAP)
```

A more delicate limitation of packaged modules is that sharing types with them is only possible via a detour through core-level polymorphism. For example, with proper modules, we can express the type of a functor that abstracts over *two* modules of a given signature with the requirement that their types t are equivalent:

signature S = {**type** t ...}
F : (X : S) → (Y : S **with type** t = X.t) → U

In the case of a core-level function abstracting over packaged modules the dependency on a parameter is not possible, so it gets a bit more complicated. We need to introduce an auxiliary polymorphic type variable (which is implicitly quantified):

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

Worse, 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 – it would require a type variable of higher kind, which is not supported in ML. In some cases the programmer may be able to work around this limitation resorting to weaker types. But in the general case they are forced to drop the use of packaged modules and raise 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, because all module-level computations are effectively "static".

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

Why not true first-class modules, then? Can we overcome the segregation and make modules more equal citizens of the language? This idea has been explored, of course. The answer from the literature so far has been: no, because first-class modules make type-checking undecidable and type inference infeasible.

The most directly relevant work is the calculus of *translucent sums* of Harper & Lillibridge (1994) (a precursor of later work on *singleton types* (Stone & Harper, 2006)). 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 & Lillibridge prove that type-checking this language is undecidable. Their result applies to any language that (a) has contravariant functions, (b) has both transparent and opaque types, and (c) allows opaque types to be subtyped by arbitrary transparent types. The latter feature usually manifests in a subtyping rule like the following,

$$\frac{\{D_1[\tau/\mathsf{t}]\} \le \{D_2[\tau/\mathsf{t}]\}}{\{\textbf{type}\ \mathsf{t}{=}\tau; D_1\} \le \{\textbf{type}\ \mathsf{t}; D_2\}}\ \text{FORGET}$$

which is, in some variant, at the heart of every definition of signature matching. In the premise the concrete type τ is substituted for the abstract t in the remaining declarations

D_1 and D_2 of the signatures. Obviously, this rule is not inductive. And that is a real problem: 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 (1994):

```
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 (Rossberg, 1999).

Some readers 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. Milner's algorithm *W* (Milner, 1978) is far too weak to handle higher-order types, let alone 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 the work on *F-ing modules* with Russo & Dreyer (Rossberg *et al.*, 2014) 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.

Interestingly, in that paper we assume that the core language on which modules sit is simply System F (respectively, F_ω) itself. That leads to the seemingly paradoxical situation that the core language has *more* expressive types than the module language. But 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" (and decidable) 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 the previous question is: very little! And the present article is all about explaining, exploring and exploiting that answer.

In essence, the F-ing modules semantics reveals that the syntactic stratification between ML core and module language is merely a rather blunt approach to enforce *predicativity* for module types: it prevents abstract types themselves from being instantiated with binders for abstract types. But this blunt *syntactic* restriction can be replaced by a much more surgical *semantic* restriction!

It is in fact enough to introduce a simple universe distinction between *small* and *large* types – reminiscent of Harper & Mitchell's XML (1993) – and limit the equivalent of the FORGET rule shown earlier to allow only small types for substitution, which serves to exclude problematic occurrences of quantifiers:

$$\frac{\{D_1[\tau/\mathrm{t}]\} \le \{D_2[\tau/\mathrm{t}]\} \qquad \tau \text{ small}}{\{\textbf{type } \mathrm{t}{=}\tau; D_1\} \le \{\textbf{type } \mathrm{t}; D_2\}}\,\text{FORGET}$$

A small type is one that does not itself contain any abstract type components or parameters. In particular, a type like $\{\textbf{type } \mathrm{t}; D\}$ is *not* small. The side condition thus prevents the substitution from introducing new occurrences of an opaque type specification, which would necessitate further (potentially never-ending) substitutions. That's all, really.

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. If we limit instantiation of implicit polymorphic type variables (and thus inference and unification) to small types then type inference works almost as usual. The only exception to subtyping coinciding with equivalence on small types is width subtyping on structures – we hence need to be willing to accept that inference is not going to be complete for records (which it already isn't in Standard ML, whereas OCaml does not support inferring types of regular records in the first place).

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 suggestive of 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.
- Conversely, we demonstrate that F_ω can be *embedded* into this language.
- We show how Damas/Milner-style *type inference* can be integrated; it is incomplete, but only in ways that are already present in existing ML implementations.
- Besides theoretic considerations, we develop the basis for a *practical design* of an ML-like language in which the distinction between core and modules is eliminated.

This redesign has several benefits: it produces a language that is more *expressive* and *concise*, and at the same time, more *minimal* and *uniform*. "Modules" become a natural means of expressing all forms of polymorphism, both universal and existential. They can be freely intermixed with "computational" code and data, that is, polymorphism is fully first-class. 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.

Relative to the conference version of this paper, this article adds more extensive explanations of the design and various technicalities of the language, it tweaks some pieces of the formalisation of type inference to make it easier to interpret as an algorithm, it includes all meta-theory that was previously omitted for space reasons — such as the proof of decidability of type checking —, and it devotes an entire new section to the expressiveness of the language (Section 5), including a simple extension with impredicative "packages" and a sound and complete embedding of System F_ω.

2 1ML with Explicit Types

Despite the simplicity of the basic ideas, the dear reader will probably have guessed that the complete story isn't quite as trivial. To separate concerns a little, we will first introduce $1ML_{ex}$, a sublanguage of 1ML proper that is explicitly typed. We hold off fancier features like implicit typing and type inference until Sections 6 and 7.

The kernel syntax of $1ML_{ex}$ is given in Figure 1. This kernel language is intentionally small, but features everything that is semantically relevant. In addition, Figure 2 defines a rich set of syntactic sugar over this kernel that provides many of the syntactic forms familiar from the ML family of languages.[1] In both figures, as well as in other places throughout this article, we use an overbar, $\overline{phrase}$, to stand for an arbitrary number of repetitions of *phrase* (including zero of them). We also conveniently write $\overrightarrow{\to}$ to range over both forms of arrows.

Let us take a little tour of $1ML_{ex}$ by way of examples.

[1] We omit further sugar that is standard and not worth expanding on, such as tuples as records $\{\_1 : T_1 ; \ldots ; \_n : T_n\}$, pattern matching, or multi-argument functions as functions over tuples.

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

Fig. 1. Kernel syntax of $1ML_{ex}$

$$
\begin{array}{lll}
\text{(types)} & \mathsf{let}\ B\ \mathsf{in}\ T & := \{B;\ X = \mathsf{type}\ T\}.X \\
 & T_1 \overset{\Rightarrow}{\to} T_2 & := (X : T_1) \overset{\Rightarrow}{\to} T_2 \\
 & T\ \mathsf{where}\ (\overline{.X}\ \overline{P} : \overline{T} = E) & := T\ \mathsf{where}\ (\overline{.X} : \overline{P \Rightarrow} (= E : \overline{T})) \\
 & T\ \mathsf{where}\ (\mathsf{type}\ \overline{.X}\ \overline{P} = T') & := T\ \mathsf{where}\ (\overline{.X} : \overline{P \Rightarrow} (= \mathsf{type}\ T')) \\
\text{(declarations)} & \mathsf{local}\ B\ \mathsf{in}\ D & := \mathsf{include}\ (\mathsf{let}\ B\ \mathsf{in}\ \{D\}) \\
 & X\ \overline{P} : T & := X : \overline{P \Rightarrow} T \\
 & X\ \overline{P} : \overline{T} = E & := X : \overline{P \Rightarrow} (= E : \overline{T}) \\
 & \mathsf{type}\ X\ \overline{P} & := X : \overline{P \Rightarrow} \mathsf{type} \\
 & \mathsf{type}\ X\ \overline{P} = T & := X : \overline{P \Rightarrow} (= \mathsf{type}\ T) \\
\text{(expressions)} & \mathsf{let}\ B\ \mathsf{in}\ E & := \{B;\ X = E\}.X \\
 & \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 : T & := \mathsf{let}\ X = E_1\ \mathsf{in}\ \mathsf{if}\ X\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 : T \\
 & E_1\ E_2 & := \mathsf{let}\ X_1 = E_1;\ X_2 = E_2\ \mathsf{in}\ X_1\ X_2 \\
 & E\ T & := E\ (\mathsf{type} T) \qquad (\text{if } T \text{ unambiguous}) \\
 & E : T & := (\mathsf{fun} (X : T) \Rightarrow X)\ E \\
 & E :> T & := \mathsf{let}\ X = E\ \mathsf{in}\ X :> T \\
 & \mathsf{fun} \overline{P} \Rightarrow E & := \overline{\mathsf{fun}\ P \Rightarrow} E \\
\text{(bindings)} & \mathsf{local}\ B\ \mathsf{in}\ B' & := \mathsf{include}\ (\mathsf{let}\ B\ \mathsf{in}\ \{B'\}) \\
 & X\ \overline{P}\ \overline{:T'}\ \overline{:>T''} = E & := X = \mathsf{fun}\ \overline{P} \Rightarrow E\ \overline{:T'}\ \overline{:>T''} \\
 & \mathsf{type}\ X\ \overline{P} = T & := X = \mathsf{fun}\ \overline{P} \Rightarrow \mathsf{type}\ T
\end{array}
$$

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

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

Fig. 2. Syntactic abbreviations for $1ML_{ex}$

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 (**true**, **false**, **if**) – in examples that follow, we will often assume the presence of an integer type and respective constructs as well. Then there are records $\{\overline{B}\}$, which consist of a sequence of bindings, and can be accessed via dot notation $E.X$. And of course, it wouldn't be a functional language without functions (**fun**) and their application ($X_1\,X_2$).

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

Like with F-ing modules (Rossberg *et al.*, 2014), 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 given in Figure 2: because we can encode

let-bindings with module projections, we can desugar all expressions into a kind of A-normal form. For example, the application

(**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 without a type annotation. However, in Section 6 we show that this annotation can usually be omitted in full 1ML with type inference.

Other abbreviations defined in Figure 2 are discussed further below.

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*.[2] 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, and 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 two types and returns a type, and effectively defines a type *constructor*. Applied to a reified type it yields a reified type, as a first class value. Again, the implicit projection from "path" expressions 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 2 also defines a bit of syntactic sugar for function and type definitions in "equational" form with parameters on the left, which makes them look more like in traditional ML.[3] For example, the previous functions could equivalently be written as

[2] Ideally, "**type** T" should be written just "T", like in various dependently typed languages. However, that creates syntactic ambiguities, e.g. for phrases like "{}", which we prefer to avoid. The ambiguity could only be addressed by moving to a more artificial syntax for types themselves. Nevertheless, we at least allow writing "$E\ T$" for the application "$E\,(\mathbf{type}\,T)$" if T unambiguously parses as a type.

[3] Some binding forms in Figure 2 allow multiple consecutive type annotations, as in "X : T : U = E". I do not consider this useful, it is merely a shortcut to avoid introducing one-off notation for optional phrases.

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

For now, Figure 2 defines that omitting a type annotation on a parameter, as in the case of a and b above, makes it default to **type** – a rule that we will refine somewhat when we introduce implicit types in Section 6.

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 (Lillibridge, 1997; Dreyer *et al.*, 2003; Rossberg & Dreyer, 2013; Rossberg *et al.*, 2014). 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 $\mathsf{T} = \{\mathbf{type}\ \mathsf{t} = \mathsf{int}\} : \{\mathbf{type}\ \mathsf{t}\}$ and then refer to $\mathsf{T.t}$, we allow injecting types into modules (i.e., values in 1ML) *anonymously*, without wrapping them into a structure; thus $\mathsf{t} = (\mathbf{type}\ \mathsf{int}) : \mathbf{type}$, which can be referred to as just t.

Declarations In the ML module system, *bindings* – i.e., definitions – in the module syntax are mirrored by *declarations* – i.e., specifications – in the signature syntax. That is no different in 1ML, where modules become records, such that bindings define record fields: analogously, signatures become record types, and declarations *specify* record fields.

For example, the previous definitions could be collected in a record with a type as follows:

```
type S =
{
  id : (a : type) ⇒ a → a
  pair : (a : type) ⇒ (b : type) ⇒ type
  second : (a : type) ⇒ (b : type) ⇒ pair a b → b
}
```

This amounts to a module signature, since it contains the type (constructor) pair as a component. And that is referred to in the type of second, 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 4.

As for bindings, Figure 2 defines analogous syntactic sugar for declarations, which enables writing the above declarations in a more familiar style:

```
id a : a → a;
type pair a b;
second a b : pair a b → b
```

In particular, we introduce the same kind of sugar that allows putting parameters on the left of declarations as we do for bindings. And like before, type annotations on parameters can be omitted, defaulting to **type**. This design not only reconstructs the way type specifications are written in ML (as for pair).[4] The left-hand side parameter syntax uniformly also generalises to ordinary value specifications and allows to rid ourselves of the brittle

[4] Minus ML's literally backwards type application syntax.

implicit scoping rules that conventional ML defines for polymorphic type variables (as for id and second).

Translucency The type **type** allows us to classify types *opaquely*: given a value of type **type**, nothing is known about *what* type it is – it is abstract. But for modular programming it is essential that types can selectively be specified *transparently*, with a concrete definition. In particular, that enables expressing the vital concept of *type sharing* (Harper & Lillibridge, 1994; Leroy, 1994; Harper & Pierce, 2005).

As a simple example, consider these type alias definitions:

```
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 in 1ML. 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* (Stone & Harper, 2006; Dreyer *et al.*, 2003) 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 components of type **type**. It allows the type system to infer, for example, that the application pair size size is equivalent to the type {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 declarations in signatures, where according to Figure 2 once more, they can be abbreviated to the more familiar

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

i.e., as in ML, transparent *declarations* look just like the *definitions* given earlier. But the former appear in types while the latter appear in values.

Singletons can be formed over arbitrary values. This gives the ability to express *module sharing* and *aliases*. In the basic semantics described in this article, this is effectively a shorthand for sharing all types contained in the module (including those defined inside transparent functors, see below). For example, given a signature

```
type T = {type t a; A : {type u; x : u}; f : A.u → t bool}
```

and a module $M : T$, the signature

```
type S = {B = M : T}
```

with its transparent module component is equivalent to the more explicit formulation

```
type S = {B : {type t a = M.t a; A : {type u = M.A.u; x : u}; f : A.u → t bool}}
```

which only makes the individual type components transparent, while not implying anything about the identity of its values. This behaviour matches the meaning of structure sharing in Standard ML '97 (Milner *et al.*, 1997).[5]

We leave the extension of 1ML to tracking full value equivalence, like in our F-ing semantics for applicative functors (Rossberg *et al.*, 2014), to future work.

Functors Returning to the 1ML grammar from Figure 1, 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. Here is the unavoidable example of a 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 with an abstract type component t. Similarly, MAP defines a signature with abstract key and map types. 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 (here, 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, with some appropriate sugar defined in Figure 2.

[5] But without disallowing transparent types in the signature of M, which is one of the more controversial limitations of SML

Applicative vs. Generative In this article, we stick to a relatively simple semantics for functor-like functions, in which Map is *generative* (Russo, 2003; Dreyer, 2005; Rossberg *et al.*, 2014). 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 so sealed map functor – which returns an abstract type – has *applicative* semantics (Leroy, 1995; Rossberg *et al.*, 2014), so that

```
type map a = M1.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.[6] 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* (Dreyer *et al.*, 2003). That is, the use of sealing *inside* a functor 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 = F :> type → type
K = 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). Applicative functor types are subtypes of generative ones, so that J turns F into a generative functor. Lastly, K 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 (1999) – 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. This choice is mainly to keep this article focussed on the novelties of 1ML. More fully-featured applicative functors as with F-ing modules (Rossberg *et al.*, 2014), where sealing is pure, could easily be incorporated into 1ML, but complicate elaboration in ways that are largely orthogonal to our present goals.

[6] Not to be confused with the applicative functors later introduced in Haskell by McBride & Paterson (2008).

Effects The ability to distinguish pure from impure, for computations or functions, is motivated by dealing with type abstraction correctly. But even in a traditional "core language" context this distinction is useful, for example, when we consider adding impure constructs to the language, such as an ML-style type ref T of mutable references with the obvious operators:

```
new : (a : type) ⇒ (x : a) → ref a
rd  : (a : type) ⇒ (r : ref a) → a
wr : (a : type) ⇒ (r : ref a) ⇒ a → {}
```

Their types make explicit where effects can occur, by mixing pure and impure arrows. A type parameter – expressing (explicit) polymorphism – is best considered a pure function: "instantiating" a polymorphic type should not have any effect. Similarly, an impure function with curried value parameters (like wr) releases its effect only when the last argument is provided.

Expressing effects in function types is not just useful documentation, it can also aid type inference (see Section 6.1).

The simplistic effect system could naturally be refined to track relevant effects in a manner more fine-grained than with just a black and white distinction. However, we do not explore that space further here. Clearly, there is plenty of room for future work. We discuss one interesting direction, effect polymorphism, in a spin-off paper (Rossberg, 2016), see Section 8.

Transparent vs. Opaque Type Ascription Like Standard ML's modules, our language provides two ways of ascribing a type to an expression or definition:[7]

- *opaquely* (E :> T), which performs *sealing* as described above, that is, any type that is specified abstractly in T will be opaque in the type of the expression;
- *transparently* (E : T), which implicitly refines all abstract specifications in T with their actual definitions from E.

The latter form is convenient when an ascription is used simply to narrow a signature, i.e., limit what is visible to the outside of a module. In Standard ML it is its own primitive with relatively complex semantics. But it turns out that it can simply be encoded as applying the identity functor at type T, as shown in Figure 2.

Higher Polymorphism So far, we have focussed on how 1ML reconstructs features that are 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}
```

The function f takes a function id as argument that is (explicitly) polymorphic, and used with different instantiations inside the body. That is not possible with the prenex polymorphism of core ML.

Similarly, existential types are directly expressible:

[7] The semantics of OCaml's module type ascription, despite being written like the latter form, is that of the former.

```
type SHAPE = {type t; area : t → float}
type shape = {S : SHAPE; v : S.t}
totalArea = List.foldLeft shape float (fun (a : float) (x : shape) ⇒ a + x.S.area x.v)
```

This example uses a two stage construction: SHAPE abstracts an ADT of shapes, which can be implemented e.g. as a circle or rectangle:

```
Circ : SHAPE = {type t = {radius : float}; area c = pi * c.radius * c.radius}
Rect : SHAPE = {type t = {width : float; height : float}; area r = r.width * r.height}
```

The type shape pairs such an ADT with an actual value:

```
aCirc = {S = Circ; v = {radius = 2.0}} :> shape
aRect = {S = Rect; v = {width = 1.8; height = 1.2}} :> shape
```

The trick here is that the ":>" operator corresponds directly to an introduction form for existential types. Because their concrete type is "forgotten" this way, shapes like these can be collected in a heterogeneous list, for which the function totalArea is able to compute the cumulative area:

```
totalArea [aCirc, aRect, aCirc]
```

This works because the computation inside the function is agnostic to the actual type of each shape, and the 1ML typing rules implicitly "open" the existential package. The upshot is that ADTs and existential types are one and the same thing in 1ML – driving home Mitchell & Plotkin's original point (Mitchell & Plotkin, 1988) not just as a theoretical semantic observation, but as very concrete language design choice.

It turns out that the previous examples can still be expressed with packaged modules (Section 1.1), even though they would become fairly verbose. 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 (Wadler & Blott, 1989). It contains two abstract type specifications, which are known as *associated types* in the type class literature (or in the C++ universe). The function entries is parameterised over a corresponding module C – an (explicit) instance of the type class 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.[8]

[8] In OCaml 4, this example can be approximated only 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 = ...
```

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 constructor class, i.e., a type class over a type constructor. As explained in Section 1.1, this kind of polymorphism cannot be expressed in ML, even in dialects 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 $1ML_{ex}$:

```
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 in most cases it can usually be inferred once we add inference to the mix (Section 6).

Predicativity What is the restriction we employ to maintain decidability? It is simple: during subtyping (also known as 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).

The primary place where the distinction between small and large types comes into play is when applying a function that abstracts over a type. The type of its argument naturally has to be a subtype of the functor's parameter. If that includes an abstract type then the argument must contain a "type value". And the crucial restriction is that this requires the type value to denote a small type.

For example, consider these applications of the Map functor from earlier:

```
M1 = Map {type t = {a : int; b : int}; eq (x : t) (y : t) = Int.eq x.a y.a ∧ Int.eq x.b y.b}
M2 = Map {type t = {type u; v : u; f : u → int}; eq (x : t) (y : t) = Int.eq (x.f x.v) (y.f y.v)}
M3 = Map {type t = (u : type) → u → u; eq (x : t) (y : t) = Int.eq (x int 6) (y int 6)}
```

The first application is fine: it passes the type {a : int; b : int}, which is a regular small type with only value components. The second application, however, defines t to be a type that itself has an abstract type component u. Such a definition is a large type, and as such, per the restriction mentioned in Section 1.4, not a subtype of the abstract declaration **type** t in the functor's parameter type EQ. Consequently, this application is rejected by the type system. Similarly, the third line defines t to be a large type, because it is a function that takes an abstract type as a parameter. This application is likewise rejected.

Intuitively, small types characterise what would be "values" while large type characterise what would be "modules" in conventional ML. In 1ML it is possible to abstract over

modules as any other first-class value, but it is not possible to abstract over module *types* – at least not directly; in Section 5.1 we will lift this restriction with a small addition to the language. All of the following define *large* types:

```
type T1 = type;                    type T4 = (x : {}) → type;
type T2 = {type u};                type T5 = (a : type) ⇒ {};
type T3 = {type u = T2};           type T6 = {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, none of the following definitions are well-typed:

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

Notably, the case A with T_1 literally implies **type type** ⁄: **type** – although **type type** itself *is* a well-formed expression, it is the value of T_1 above! But its only type is the transparent (= **type type**); it cannot be abstracted. Preventing a type:type situation is the main challenge with first-class modules, 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. Consequently, the following functor application is fine:

```
M4 = Map {type t = {type u = int; v : u; f : u → int}; eq (x : t) (y : t) = Int.eq x.v y.v}
```

Likewise, the following bindings,

```
type T'1 = (= type int);
type T'2 = {type u = int}
```

are to small types and would hence *not* cause an error when inserted into the above definitions.

Recursion The 1ML_{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.

Recursive functions can be added simply by throwing in a primitive polymorphic fixpoint operator

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

On top of that, it is only a matter of defining suitable syntactic sugar:

$$\textbf{rec}\ X\ \overline{Y}\ (Z{:}T){:}U = E \ \ := \ \ X = \textbf{fun}\ \overline{Y} \Rightarrow \mathsf{fix}\ T\ U\ (\textbf{fun}(X{:}(Z{:}T) \to T') \Rightarrow \textbf{fun}(Z{:}T) \Rightarrow E)$$

Given an appropriate fixpoint operator, this generalises further to mutually recursive functions in the usual manner. Note how the need to specify the result type b (respectively, U for the sugared form) prevents using the operator to construct transparent recursive types,

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

Fig. 3. Syntax of F_ω

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,

rec stream (a : **type**) : **type** = **type** {head : a; tail : 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_ω 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 (Harper & Stone, 2000), is to interpret a recursive datatype like

datatype t = A | B **of** T

as a module defining a primitive ADT with the signature

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

where expose is a case-operator accessed by pattern matching compilation. We refer to Harper & Stone (2000) for more details on this approach. There is one caveat, though: datatypes expressed as ADTs require sealing. With the simple system presented in this article alone they hence could not be defined inside applicative functors. This limitation is removed by pure sealing, which, as mentioned before, is an obvious extension from the F-ing modules paper (Rossberg *et al.*, 2014) that is applicable to 1ML.

3 System F_ω

Before diving deep into the semantics of 1ML in Section 4, a brief detour is in order. Following the *F-ing modules* approach (Rossberg *et al.*, 2014), 1ML's semantics is defined by translation into System F_ω, the higher-order polymorphic λ-calculus (Barendregt, 1992).

Readers who know F_ω inside out should feel free to skip over this section.

Syntax Figure 3 gives the syntax of (impredicative) System F_ω as we will use it here. It includes simple record types $\{\overline{l : \tau}\}$ (where we assume that labels are always disjoint), but is otherwise completely standard. In the grammar, and elsewhere, we liberally use the meta-notation $\overline{A}$ to stand for zero or more repetitions of an object or formula A, sometimes writing ε explicitly for the empty sequence. We also sometimes abuse the notation $\overline{A}$ to actually denote the unordered set $\{\overline{A}\}$.

To differentiate the *external* language (1ML) from the *internal* one (F_ω) later in the paper, we consistently use uppercase letters to range over phrases of the former (X, T, E, etc.), and lowercase for the latter (α, x, τ, e).

Static Semantics The full typing rules of F_ω are given in Figure 4. The judgements are as usual, with $\Gamma \vdash \Box$ denoting well-formedness of typing environments. Type equivalence is defined as full $\beta\eta$-equivalence.

The only point of note is that, unlike in many presentations, environments Γ permit term variables x (but not type variables α) to be shadowed without α-renaming, which is convenient for translating $1ML_{ex}$-bindings later. Thus, we take the notation $\Gamma(x)$ to index the rightmost binding of x in Γ. Finally, $\mathrm{dom}(\Gamma)$ denotes the set of bound variables in Γ, while $\mathrm{fv}(\tau)$ and $\mathrm{fv}(e)$ are the free type and term variables of τ and e, respectively.

Dynamic Semantics We assume a standard left-to-right call-by-value operational semantics, which is defined in Figure 5 via small-step reduction rules and evaluation contexts.

There is nothing unusual about the semantics, so we can move on immediately to the meta-theory.

Properties The calculus as defined enjoys 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 $e \neq v$ for any v, then $e \hookrightarrow e'$ for some e'.

The proofs are entirely standard.

The calculus also has the usual technical properties. The most relevant ones for our purposes are the following:

Lemma 3.3 (*Validity*)

1. If $\Gamma \vdash \tau : \Omega$, then $\Gamma \vdash \Box$.
2. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \Omega$.

Lemma 3.4 (*Weakening*)
Let $\Gamma' \supseteq \Gamma$ with $\Gamma' \vdash \Box$.

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

Lemma 3.5 (*Strengthening*)
Let $\Gamma' \subseteq \Gamma$ with $\Gamma' \vdash \Box$ and $D = \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma')$.

1. If $\Gamma \vdash \tau : \kappa$ and $\mathrm{fv}(\tau) \cap D = \emptyset$, then $\Gamma' \vdash \tau : \kappa$.
2. If $\Gamma \vdash e : \tau$ and $\mathrm{fv}(e) \cap D = \emptyset$, then $\Gamma' \vdash e : \tau$.

Theorem 3.6 (*Uniqueness of types and kinds*)
Assume $\Gamma \vdash \Box$.

1. If $\Gamma \vdash \tau : \kappa_1$ and $\Gamma \vdash \tau : \kappa_2$, then $\kappa_1 = \kappa_2$.
2. If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 \equiv \tau_2$.

Finally, all judgments of the F_ω type system are decidable:

Theorem 3.7 (*Decidability*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. Typing rules for F_ω

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

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

where:

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

Fig. 5. Reduction rules for F_ω

1. It is decidable whether $\Gamma \vdash \Box$.
2. It is decidable whether $\Gamma \vdash \tau : \kappa$.
3. It is decidable whether $\Gamma \vdash e : \tau$.
4. If $\Gamma \vdash \tau_1 : \kappa$ and $\Gamma \vdash \tau_2 : \kappa$, it is decidable whether $\tau_1 \equiv \tau_2$.

Notational shorthands The rest of this article often deals with quantification over whole sequences $\overline{\alpha}$ of type variables, and some other forms of lists of binders. The following (fairly standard) definitions for shortened n-ary notation will come in handy:

$\forall\varepsilon.\tau := \tau$ $\quad \exists\varepsilon.\tau := \tau$ $\quad \lambda\varepsilon.\tau := \tau$ $\quad \tau_0\ \varepsilon := \tau_0$
$\forall\overline{\alpha}.\tau := \forall\alpha_1.\forall\overline{\alpha}'.\tau$ $\quad \exists\overline{\alpha}.\tau := \exists\alpha_1.\exists\overline{\alpha}'.\tau$ $\quad \lambda\overline{\alpha}.\tau := \lambda\alpha_1.\lambda\overline{\alpha}'.\tau$ $\quad \tau_0\ \overline{\tau} := \tau_0\ \tau_1\ \overline{\tau}'$

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

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

Likewise, n-ary forms of other constructs (e.g. term abstraction and applications) are defined in the obvious way.

To ease notation, we often drop type annotations from let, pack, and unpack where clear from context.

Substitution Our semantics will also make considerable use of parallel type substitutions. We write them as $[\overline{\tau}/\overline{\alpha}]$, assuming that both vectors have the same arity. Sometimes we use δ to range over such substitutions.

The following definitions and lemmas are relevant:

Definition 3.8 (*Typing of Type Substitutions*)

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

Desugarings into F_ω:

$$
\begin{array}{llcl@{\qquad}llcl}
\multicolumn{4}{l}{\text{(types)}} & \multicolumn{4}{l}{\text{(terms)}} \\
[= \tau] & & := & \{\mathsf{type} : \tau \to \{\}\} & [\tau] & & := & \{\mathsf{type} = \lambda x{:}\tau.\{\}\} \\
\tau_1 \to_l \tau_2 & & := & \tau_1 \to \{l : \tau_2\} & \lambda_l x{:}\tau.e & & := & \lambda x{:}\tau.\{l : e\}
\end{array}
$$

Notation:

$$
\begin{array}{lcl}
\mathrm{head}(\alpha) & := & \alpha \\
\mathrm{head}(\pi\,\sigma) & := & \mathrm{head}(\pi)
\end{array}
$$

$$
\begin{array}{lll@{\qquad}lll@{\qquad}lll}
\eta \le \eta & & & \eta \vee \eta & := & \eta & & \eta(\Sigma) & := & \mathtt{P} \\
\mathtt{P} \le \mathtt{I} & & & \mathtt{P} \vee \mathtt{I} & := & \mathtt{I} \vee \mathtt{P} := \mathtt{I} & & \eta(\exists\alpha\overline{\alpha}.\Sigma) & := & \mathtt{I}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcll}
\tau.\bar{l} & := & \tau & \tau[.\bar{l}{=}\tau_2] & := & \tau_2 & (\bar{l} = \varepsilon) \\
\{l{:}\tau, \ldots\}.\bar{l} & := & \tau.\bar{l}' & \{l{:}\tau, \ldots\}[.\bar{l}{=}\tau_2] & := & \{l{:}\tau[.\bar{l}'{=}\tau_2], \ldots\} & (\bar{l} = l.\bar{l}')
\end{array}
$$

Fig. 6. Semantic Types

Let $\delta = [\overline{\tau}/\overline{\alpha}]$. We write $\Gamma' \vdash \delta : \Gamma$ if and only if

1. $\Gamma' \vdash \square$,
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 3.9 (*Type Substitutions*)
Let $\Gamma' \vdash \delta : \Gamma$. Then:

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

4 Type System and Elaboration

The general recipe for $1\mathrm{ML}_{\mathrm{ex}}$ is simple: take the semantics from F-ing modules (Rossberg *et al.*, 2014), 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.

4.1 Semantic Types

The entire semantics is defined by *elaborating* $1\mathrm{ML}_{\mathrm{ex}}$ types and terms directly into "equivalent" types and terms of System F_ω as given in the previous section. However, 1ML types do not translate to arbitrary F_ω types – instead, they map to types of very specific shape. The grammar of these *semantic types* is given in Figure 6 ("semantic" as opposed to the "syntactic" types of the surface language which they model). To avoid clutter, we omit all

kind annotations on type variable binders. Where needed, we use the notation κ_α to refer to the kind implicitly associated with α.

Semantic types are the essence of the F-ing elaboration. To understand them, it is probably best to walk through a few explanatory examples that elaborate a concrete 1ML type T into a System F_ω type τ, a relation we write $T \rightsquigarrow \tau$.

- $\{\mathsf{x} : \mathsf{bool},\ \mathsf{y} : \mathsf{int}\} \quad \rightsquigarrow \quad \{\mathsf{x} : \mathsf{bool},\ \mathsf{y} : \mathsf{int}\}$

Elaboration will translate this syntactic type into the semantic type of the same shape. So records map to records, which should be no surprise. We assume an implicit injection from 1ML identifiers X into F_ω labels l (and variables x), so we can conveniently treat any X as a label (or variable).

- $\{\mathsf{t} : (= \mathbf{type}\ \mathsf{bool});\ \mathsf{u} : (= \mathbf{type}\ \mathsf{t});\ \mathsf{x} : \mathsf{t},\ \mathsf{y} : \mathsf{u}\} \quad \rightsquigarrow$
  $\{\mathsf{t} : [= \mathsf{bool}],\ \mathsf{u} : [= \mathsf{bool}],\ \mathsf{x} : \mathsf{bool},\ \mathsf{y} : \mathsf{bool}\}$

Structurally, the elaborated type still looks almost like the original, except that all references to the transparent type specifications are replaced with their definitions.

Of course, one may (and should!) wonder what this "$[= \tau]$" is that represents transparent types. Because we keep saying that semantic types are just System F types, but this does not look like one. The answer can be found in Figure 6: it is simply a notational abbreviation that is encoded into bare System F. Its expansion does not actually matter much, it is merely a coding trick. All that matters is that it is a form of type that (1) uniquely determines τ, and (2) is inhabited by a term that also uniquely determines τ. A function fits the bill.[9] To make the encoding unambiguous, it is wrapped into a record type with a unique label "type" that we assume to be a reserved name disjoint from all valid 1ML identifiers.

- $\{\mathsf{t} : \mathbf{type};\ \mathsf{u} : (= \mathbf{type}\ \mathsf{t});\ \mathsf{x} : \mathsf{t};\ \mathsf{y} : \mathsf{u}\} \quad \rightsquigarrow$
  $\exists\alpha.\{\mathsf{t} : [= \alpha],\ \mathsf{u} : [= \alpha],\ \mathsf{x} : \alpha,\ \mathsf{y} : \alpha\}$

This type is almost the same as before except that t now is an abstract type. The example demonstrates the main trick of elaboration: it inserts appropriate quantifiers to bind abstract types. Following Mitchell & Plotkin (1988), abstract types are represented by existentials. The type variable α provides a "semantic" name for t, and all references to t are replaced by α. This includes the specification of t itself, which can now be viewed as being transparently equal to α. The upshot is that all reified types, whether transparent or opaque, are represented in a uniform manner. It also removes any dependency between fields of the record type – key to elaborating into bare System F without dependent types.

- $\{\mathsf{A} : \{\mathsf{t} : \mathbf{type}\};\ \mathsf{B} : \{\mathsf{u} : \mathbf{type};\ \mathsf{x} : \mathsf{A.t}\};\ \mathsf{y} : \mathsf{B.u}\} \quad \rightsquigarrow$
  $\exists\alpha\beta.\{\mathsf{A} : \{\mathsf{t} : [= \alpha]\},\ \mathsf{B} : \{\mathsf{u} : [= \beta],\ \mathsf{x} : \alpha\},\ \mathsf{y} : \beta\}$

This one shows the handling of nested abstract types: inner quantifiers are collectively hoisted out of records. In particular, this makes them scope over the remainder of any enclosing record, which allows us to reference the type variable across subcomponents –

[9] Because all type constructors are separately represented as "functors" in 1ML, we have no need for types of higher kind in this notation, as opposed to Rossberg *et al.* (2014).

such as A.t and B.u in the example. A type may have arbitrary many abstract components, which all accumulate in an outermost quantifier.

Accordingly, the grammar of semantic types defines an *abstracted* type $\Xi = \exists\overline{\alpha}.\Sigma$ that quantifies over all the abstract types (i.e., components of type **type**) from the underlying *concretised* type Σ, by naming them $\overline{\alpha}$. Σ itself contains no existential quantifiers – except as part of function types. So let us move to functions then.

- $(\mathsf{t} : \mathbf{type}) \to \{\mathsf{u} : \mathbf{type};\ \mathsf{x} : \mathsf{t},\ \mathsf{y} : \mathsf{u}\} \quad \rightsquigarrow$
  $\forall\alpha.[= \alpha] \to_{\mathtt{I}} \exists\beta.\{\mathsf{u} : [= \beta],\ \mathsf{x} : \alpha,\ \mathsf{y} : \beta\}$

Conceptually, all function types map to polymorphic functions in F_ω. Being in negative position, the quantifier for the abstract types from the parameter (here α) turns into a universal quantifier. It scopes over the *whole* arrow type and thereby allows the codomain to refer to it. Like for nested records, this hoisting avoids the need for dependent types.

For an *impure* function type like the above, the abstract types in the result are encapsulated by a local existential quantifier. It binds all those types that will be "generated" when applying the function. This representation of generative functor types as System F types of the form $\forall\overline{\alpha}.\Sigma_1 \to \exists\overline{\beta}.\Sigma_2$ is due to Russo (1999; 2003), and another central ingredient of the F-ing modules semantics.

Arrow types are further annotated by a simple *effect* η, which distinguishes impure ($\to_{\mathtt{I}}$) from pure ($\to_{\mathtt{P}}$) function types. Again, this syntax is encoded into bare F_ω types with a little record trick, see Figure 6 (we assume that record labels used with this form do not overlap with other labels).

- $(\mathsf{t} : \mathbf{type}) \Rightarrow \{\mathsf{u} : \mathbf{type};\ \mathsf{x} : \mathsf{t};\ \mathsf{y} : \mathsf{u}\} \quad \rightsquigarrow$
  $\exists\beta.\forall\alpha.[= \alpha] \to_{\mathtt{P}} \{\mathsf{u} : [= \beta\,\alpha],\ \mathsf{x} : \alpha,\ \mathsf{y} : \beta\,\alpha\}$

This example shows the *pure* variant of the previous function type. Because it does not generate types at applications time, the existential quantifier changes position in the F_ω translation: pure function types encode applicative semantics for the abstract types they return by having their existential quantifiers (here for β) "lifted" over their parameters. Their general form is $\exists\overline{\beta}.\forall\overline{\alpha}.\Sigma_1 \to \Sigma_2$. Conceptually, the abstract types are generated when the function is *defined*, not when it is applied. We impose the syntactic invariant that a pure function type ($\to_{\mathtt{P}}$) never has an existential quantifier immediately right of the arrow, i.e., always has shape $\Sigma_1 \to_{\mathtt{P}} \Sigma_2$.

However, the representation of an abstract type returned by a function may depend on the function's parameters. To capture any potential dependencies, the $\overline{\beta}$ in a pure function type are skolemised over $\overline{\alpha}$ (Biswas, 1995; Russo, 2003; Rossberg *et al.*, 2014). That is, the kinds of $\overline{\beta}$ are of the form $\overline{\kappa_\alpha} \to \kappa$, which is where higher kinds come into play. In return, all their occurrences are applied to $\overline{\alpha}$. In the concrete example, the kind of t's type variable β hence is $\Omega \to \Omega$, and all its references are of the form $\beta\,\alpha$, thereby saturating any use of a type constructor and make sure that they all yield kind Ω. Once the function is applied to a concrete type, say τ, the parameter variable α is substituted, and all occurrences will have the form $\beta\,\tau$.

- $\{\mathsf{t} : \mathbf{type} \Rightarrow \mathbf{type};\ \mathsf{x} : \mathsf{t}\ \mathsf{int};\ \mathsf{y} : \mathsf{t}\ \mathsf{bool}\} \quad \rightsquigarrow$
  $\exists\beta.\{\mathsf{t} : \forall\alpha.[= \alpha] \to_{\mathtt{P}} [= \beta\,\alpha],\ \mathsf{x} : \beta\,\mathsf{int},\ \mathsf{y} : \beta\,\mathsf{bool}\}$

Pure functions over type **type** readily subsume abstract type constructors. As we will see in a short while, applying $(\mathsf{t}\ \mathsf{int})$ for the above substitutes int for α and results in a value of semantic type $[= \beta\,\mathsf{int}]$ – which is the representation of the syntactic type $(= \textbf{type}\ \mathsf{t}\ \mathsf{int})$. Hence, the types of x and y in the example have the desired meaning.

We call the general form of abstract type variables applied to parameters *semantic paths*,[10] ranged over by π. Parameters are (only) introduced through pure function abstraction and the aforementioned kind raising that goes with it. In general, an abstract type that is the result of an application of a pure function F to a value M is represented by the application of the higher-kinded type variable representing the constructor to the concrete types $\overline{\sigma}_M$ from the applied argument, like the type $\beta\,\mathsf{int}$ in the example.

Because we enforce predicativity, these argument types have to be small. Figure 6 defines small types as a subgrammar σ of the general semantic types Σ. The central difference is that small types cannot contain quantifiers. Moreover, small function types are required to be impure, which will simplify type inference (Section 7).

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

$$\begin{array}{rcl} T & \rightsquigarrow & \exists\overline{\alpha}.\Sigma \\ \hline (= \textbf{type}\ T_1) & \rightsquigarrow & [= \exists\overline{\alpha}_1.\Sigma_1] \\ \textbf{type} & \rightsquigarrow & \exists\alpha.[=\alpha] \\ \{X_1{:}T_1; X_2{:}T_2\} & \rightsquigarrow & \exists\overline{\alpha}_1\overline{\alpha}_2.\{X_1{:}\Sigma_1, X_2{:}\Sigma_2\} \\ (X{:}T_1) \to T_2 & \rightsquigarrow & \forall\overline{\alpha}_1.\Sigma_1 \to_1 \exists\overline{\alpha}_2.\Sigma_2 \\ (X{:}T_1) \Rightarrow T_2 & \rightsquigarrow & \exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\Sigma_1 \to_\mathrm{P} \Sigma_2 \\ \mathsf{A.t} & \rightsquigarrow & \alpha_{\mathsf{A.t}} \\ \mathsf{F}(M) & \rightsquigarrow & \alpha_{\mathsf{F}(\_)}\,\overline{\sigma}_M \end{array}$$

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.

Note that all semantic types are F_ω types of kind Ω, even those that are the equivalent of higher kinds, such as $\textbf{type} \Rightarrow \textbf{type}$. This type will elaborate into the polymorphic function $\exists\alpha_2.\forall\alpha_1.[=\alpha_1] \to [=\alpha_2\,\alpha_1]$, where the higher-kindedness is manifest only in the kind of the "output" type variable α_2.

4.2 Elaboration

The elaboration rules for $1\mathrm{ML}_\mathrm{ex}$ are collected in Figures 7 and 8. There is one judgement for each syntactic class with the following regular structure:

$$\Gamma \vdash T \rightsquigarrow \Xi \qquad \Gamma \vdash E :_\eta \Xi \rightsquigarrow e$$
$$\Gamma \vdash D \rightsquigarrow \Xi \qquad \Gamma \vdash B :_\eta \Xi \rightsquigarrow e$$

In addition, there is an additional judgement defining subtyping over semantic types:

$$\Gamma \vdash \Xi_1 \leq_{\overline{\pi}} \Xi_2 \rightsquigarrow \delta; f$$

[10] Because they are the semantic analog to syntactic paths like $\mathsf{F(M).t}$ that appear in some previous type systems for higher-order modules (Leroy, 1995)

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

$$\frac{\Gamma \vdash E :_{\mathrm{P}} [= \Xi] \rightsquigarrow e}{\Gamma \vdash E \rightsquigarrow \Xi}\textsc{Tpath} \qquad \frac{\kappa_\alpha = \Omega}{\Gamma \vdash \mathbf{type} \rightsquigarrow \exists \alpha.[= \alpha]}\textsc{Ttype}$$

$$\frac{}{\Gamma \vdash \mathbf{bool} \rightsquigarrow \mathsf{bool}}\textsc{Tbool} \qquad \frac{\Gamma \vdash D \rightsquigarrow \Xi}{\Gamma \vdash \{D\} \rightsquigarrow \Xi}\textsc{Tstr}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \\ \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2\end{array}}{\Gamma \vdash (X{:}T_1) \to T_2 \rightsquigarrow \forall \overline{\alpha}_1.\Sigma_1 \to_{\mathrm{I}} \exists \overline{\alpha}_2.\Sigma_2}\textsc{Tfun}$$

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

$$\frac{\Gamma \vdash E :_{\mathrm{P}} \Sigma \rightsquigarrow e \qquad E = E'{:}{>}T \vee E = E'{:}T \vee E = \mathbf{type}\ T}{\Gamma \vdash (= E) \rightsquigarrow \Sigma}\textsc{Tsing}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \qquad \overline{\alpha}_1 = \overline{\alpha}_{11} \uplus \overline{\alpha}_{12} \\ \Gamma \vdash T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2 \qquad \Gamma, \overline{\alpha}_{11}, \overline{\alpha}_2 \vdash \Sigma_2 \leq_{\overline{\alpha}_{12}} \Sigma_1.\overline{X} \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash T_1 \ \mathbf{where}\ (\overline{X}{:}T_2) \rightsquigarrow \exists \overline{\alpha}_{11}\overline{\alpha}_2.\delta\Sigma_1[\overline{X}{=}\Sigma_2]}\textsc{Twhere}$$

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

$$\frac{\Gamma \vdash T \rightsquigarrow \exists \overline{\alpha}.\Sigma}{\Gamma \vdash X{:}T \rightsquigarrow \exists \overline{\alpha}.\{X{:}\Sigma\}}\textsc{Dvar} \qquad \frac{\Gamma \vdash T \rightsquigarrow \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}{\Gamma \vdash \mathbf{include}\ T \rightsquigarrow \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}\textsc{Dincl}$$

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

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

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

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

$$\frac{}{\Gamma \vdash \{\overline{l{:}\Sigma'}\} \leq \{\} \rightsquigarrow \lambda x.\{\}}\textsc{Sempty}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma_1' \leq_{\overline{\pi}_1} \Sigma_1 \rightsquigarrow \delta_1; f_1 \\ \Gamma \vdash \{\overline{l'{:}\Sigma'}\} \leq_{\overline{\pi}_2} \{\overline{l{:}\,\delta_1\Sigma}\} \rightsquigarrow \delta_2; f_2 \qquad \delta_2\Sigma_1 = \Sigma_1\end{array}}{\Gamma \vdash \{l_1{:}\Sigma_1', \overline{l'{:}\Sigma'}\} \leq_{\overline{\pi_1\pi_2}} \{l_1{:}\Sigma_1, \overline{l{:}\Sigma}\} \rightsquigarrow \delta_1\delta_2; \lambda x.\{l_1{=}f_1(x.l_1), \overline{l{=}(f_2\,x).l}\}}\textsc{Sstr}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \vdash \Sigma \leq_{\overline{\alpha}'} \Sigma' \rightsquigarrow \delta_1; f_1 \qquad \eta' \leq \eta \\ \Gamma, \overline{\alpha} \vdash \delta_1\Xi' \leq_{\overline{\pi\alpha}} \Xi \rightsquigarrow \delta_2; f_2 \qquad \delta_2\Sigma = \Sigma\end{array}}{\Gamma \vdash (\forall \overline{\alpha}'.\Sigma' \to_{\eta'} \Xi') \leq_{\overline{\pi}} (\forall \overline{\alpha}.\Sigma \to_\eta \Xi) \rightsquigarrow \delta_2; \lambda x.\,\lambda\,\overline{\alpha}.\,\lambda_\eta y{:}\Sigma.\, f_2\,((x\,(\delta_1\overline{\alpha}')\,(f_1\,y)).\eta')}\textsc{Sfun}$$

$$\frac{\Gamma, \overline{\alpha}' \vdash \Sigma' \leq_{\overline{\alpha}} \Sigma \rightsquigarrow \delta; f \qquad \overline{\alpha}'\overline{\alpha} \neq \varepsilon}{\Gamma \vdash \exists \overline{\alpha}'.\Sigma' \leq \exists \overline{\alpha}.\Sigma \rightsquigarrow \lambda x.\,\mathsf{unpack}\ \langle \overline{\alpha}', y\rangle = x\ \mathsf{in\ pack}\ \langle \delta\overline{\alpha}, f\,y\rangle}\textsc{Sabs}$$

Fig. 7. Elaboration of $1\mathrm{ML}_{\mathrm{ex}}$ types

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

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash X :_\mathrm{P} \Sigma \rightsquigarrow X}\textsc{Evar} \qquad \frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \mathsf{type}\ T :_\mathrm{P} [= \Xi] \rightsquigarrow [\Xi]}\textsc{Etype}$$

$$\frac{}{\Gamma \vdash \mathsf{true} :_\mathrm{P} \mathsf{bool} \rightsquigarrow \mathsf{true}}\textsc{Etrue} \qquad \frac{}{\Gamma \vdash \mathsf{false} :_\mathrm{P} \mathsf{bool} \rightsquigarrow \mathsf{false}}\textsc{Efalse}$$

$$\frac{\begin{array}{lll}\Gamma \vdash X :_\mathrm{P} \mathsf{bool} \rightsquigarrow e & \Gamma \vdash E_1 :_{\eta_1} \Xi_1 \rightsquigarrow e_1 & \Gamma \vdash \Xi_1 \le \Xi \rightsquigarrow f_1 \\ \Gamma \vdash T \rightsquigarrow \Xi & \Gamma \vdash E_2 :_{\eta_2} \Xi_2 \rightsquigarrow e_2 & \Gamma \vdash \Xi_2 \le \Xi \rightsquigarrow f_2\end{array}}{\Gamma \vdash \mathsf{if}\ X\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 : T :_{\eta_1 \vee \eta_2 \vee \eta(\Xi)} \Xi \rightsquigarrow \mathsf{if}\ e\ \mathsf{then}\ f_1\ e_1\ \mathsf{else}\ f_2\ e_2}\textsc{Eif}$$

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

$$\frac{\Gamma \vdash T \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X : \Sigma \vdash E :_\eta \Xi \rightsquigarrow e}{\Gamma \vdash \mathsf{fun}\,(X:T) \Rightarrow E :_\mathrm{P} \forall \overline{\alpha}.\Sigma \to_\eta \Xi \rightsquigarrow \lambda \overline{\alpha}.\lambda_\eta X : \Sigma.e}\textsc{Efun}$$

$$\frac{\begin{array}{l}\Gamma \vdash X_1 :_\mathrm{P} \forall \overline{\alpha}.\Sigma_1 \to_\eta \Xi \rightsquigarrow e_1 \\ \Gamma \vdash X_2 :_\mathrm{P} \Sigma_2 \rightsquigarrow e_2 \qquad \Gamma \vdash \Sigma_2 \le_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash X_1\,X_2 :_\eta \delta\Xi \rightsquigarrow (e_1\,(\delta\overline{\alpha})\,(f\,e_2)).\eta}\textsc{Eapp}$$

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

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

$$\frac{\Gamma \vdash E :_\eta \exists \overline{\alpha}.\Sigma \rightsquigarrow e}{\Gamma \vdash X = E :_\eta \exists \overline{\alpha}.\{X : \Sigma\} \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}, x \rangle = e\ \mathsf{in}\ \mathsf{pack}\ \langle \overline{\alpha}, \{X = x\} \rangle}\textsc{Bvar}$$

$$\frac{\Gamma \vdash E :_\eta \exists \overline{\alpha}.\{\overline{X : \Sigma}\} \rightsquigarrow e}{\Gamma \vdash \mathsf{include}\ E :_\eta \exists \overline{\alpha}.\{\overline{X : \Sigma}\} \rightsquigarrow e}\textsc{Bincl} \qquad \frac{}{\Gamma \vdash \varepsilon :_\mathrm{P} \{\} \rightsquigarrow \{\}}\textsc{Bempty}$$

$$\frac{\begin{array}{rl}\Gamma \vdash B_1 :_{\eta_1} \exists \overline{\alpha}_1.\{\overline{X_1 : \Sigma_1}\} \rightsquigarrow e_1 & \overline{X}'_1 = \overline{X}_1 - \overline{X}_2 \\ \Gamma, \overline{\alpha}_1, \overline{X_1 : \Sigma_1} \vdash B_2 :_{\eta_2} \exists \overline{\alpha}_2.\{\overline{X_2 : \Sigma_2}\} \rightsquigarrow e_2 & \overline{X'_1 : \Sigma'_1} \subseteq \overline{X_1 : \Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash B_1; B_2 :_{\eta_1 \vee \eta_2} \exists \overline{\alpha}_1 \overline{\alpha}_2.\{\overline{X'_1 : \Sigma'_1}, \overline{X_2 : \Sigma_2}\} \rightsquigarrow \mathsf{unpack}\ \langle \overline{\alpha}_1, y_1 \rangle = e_1\ \mathsf{in}\ \mathsf{let}\ \overline{X_1 = y_1.X_1}\ \mathsf{in} \\ \qquad\qquad \mathsf{unpack}\ \langle \overline{\alpha}_2, y_2 \rangle = e_2\ \mathsf{in} \\ \qquad\qquad \mathsf{pack}\ \langle \overline{\alpha}_1 \overline{\alpha}_2, \{\overline{X'_1 = y_1.X'_1}, \overline{X_2 = y_2.X_2}\} \rangle\end{array}}\textsc{Bseq}$$

Fig. 8. Elaboration of $1\mathrm{ML}_{\mathrm{ex}}$ expressions

The rules may seem a bit intimidating at first, especially those for expressions and subtyping. But do not despair: that is only because elaboration defines both static semantics (typing) and dynamic semantics (operational meaning) at the same time. The latter is encapsulated in the greyed parts like "$\rightsquigarrow e$", and defines what System F term a construct translates to. A reader who is merely interested in *typing* 1ML can completely ignore those grey parts in all rules – just squint and make them disappear.

Types and Declarations The elaboration rules for types implement the translation scheme we laid out in the previous section. Their main job 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 F-ing modules, except that **type** is a free-standing construct instead of being tied to the syntax

of bindings, and 1ML's "**where**" construct is slightly more general. Most notable are the following rules.

Using an expression E as a type path requires that E is pure and has a type of the form $[= \Xi]$. It denotes the very type Ξ – which may be anything from an abstract type variable to a large type (rule TPATH).

Function types have separate rules for impure (generative) functions (TFUN) and pure (applicative) ones (TPFUN). As explained above, pure function types have the existential quantifier for the $\overline{\alpha}_2$ from the codomain lifted over the parameters, skolemising them accordingly.

Rule TSING handles "singleton" types. It corresponds to rule S-LIKE in Rossberg *et al.* (2014), except that we restrict it to expressions with explicit types, in order to mesh well with type inference later (where it would not otherwise denote a unique type). In return, we can drop the former side condition requiring Σ to be *explicit*. The rule simply infers the (unique) type Σ of the pure expression E. Note that this type is not allowed to have existential quantifiers, i.e., E may not introduce local abstract types. All types occurring in Σ thus are transparent.

The most complex rule is TWHERE. Unlike in conventional ML, where the **where** construct simply concretises one (or several) types via substitution, we allow it to specialise an arbitrary subcomponent of T_1 to an arbitrary subtype. The original semantic type of that subcomponent, denoted by $\Sigma_1.\overline{X}$ in the rule, is replaced by the semantic interpretation of T_2, written $\Sigma_1[.\overline{X} = \Sigma_2]$ (both these notations are defined in Figure 6). If T_2 defines its own abstract types $\overline{\alpha}_2$ then they are added to the list. At the same time, it may remove some of the abstract types $\overline{\alpha}_1$ from the original type. To this end, $\overline{\alpha}_1$ is partitioned into types $\overline{\alpha}_{11}$ that remain abstract and $\overline{\alpha}_{12}$ that are made concrete by the refinement (or just replaced by some of the $\overline{\alpha}_2$). The last premise checks that Σ_2 indeed matches $\Sigma_1.\overline{X}$, using the subtyping judgement explained below. This yields a substitution δ for eliminating the previously abstract types $\overline{\alpha}_{12}$. The remaining $\overline{\alpha}_{11}$ are left abstract, so their quantifiers are kept.

As a rule of thumb, any $1ML_{ex}$ type with an occurrence of type **type** will elaborate to a large type, unless this abstact type is refined to something concrete using the **where** form. Any type without the use of type **type** and no reference to another large type will denote a small type.

Expressions and Bindings The elaboration of expressions closely follows the rules from the first part of Rossberg *et al.* (2014), but adds the tracking of purity as in Section 7 of that paper. However, to keep the current article 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 a relaxed applicative functor semantics. The only other non-editorial change over F-ing mdoules – apart from the addition of Boolean expression forms – is that as for types, "**type** T" is now handled as a first-class value, no longer tied to bindings.

The rules assign effects η and semantic types Ξ to 1ML expressions or bindings, and at the same time translate them to corresponding terms e of System F_ω that define the operational semantics of 1ML. It is an invariant of the judgement that e has type Ξ in System F_ω.

As we saw before, the main trick in interpreting 1ML's module-like types is introducing quantifiers for abstract types. That is reflected in the typing of expressions: an expression

that defines new abstract types will have an "abstracted" semantic type Ξ with existential quantifiers, one quantifier for each new type. When such an expression is nested into a larger expression then the rules have to propagate these quantifiers accordingly. For example, for a projection $E.\mathsf{x}$ to be well-typed, E obviously needs to have a type of the form $\{\mathsf{x} : \Sigma, \ldots\}$; the resulting type would be Σ then. However, if E creates abstract types locally, then its type will be of the form $\exists\overline{\alpha}.\{\mathsf{x} : \Sigma, \ldots\}$ instead. The central idea taken from F-ing modules is to handle such types implicitly by extruding the existential quantifier automatically: that is, the projection $E.\mathsf{x}$ is well-typed and assigned type $\exists\overline{\alpha}.\Sigma$, with the same sequence of quantifiers. And so on for other constructs. On the term level, that corresponds to repeated unpacking and repacking of the respective existentials.

Moreover, the sequencing rule BSEQ combines two (n-ary) existentials into one – and is the only rule that does so, because the kernel syntax allows only variables in most places; the derived forms from Figure 1 all desugar into uses of sequencing.

For the categorically inclined, the existential quantifiers in an expression's type Ξ act like an implicit monad of "abstraction effects". There is no way to "escape" this monad: the sequencing rule BSEQ acts like its bind operator, and the only construct to suspend it is functional abstraction (rule EFUN). The only rules that effectively "run" such a monad are the type formation rules TPATH and TSING, and they require the abstraction effects to be encapsulated in the expression (i.e., no abstract types escape).[11]

It is another invariant of the expression elaboration judgement that $\eta = \mathtt{I}$ in all cases where Ξ does *not* degenerate into a concrete type Σ with empty (i.e., no) quantifier – in other words, abstract type "generation" is indeed 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 $\eta(\Xi)$ defined in Figure 6. In that sense, our notion of purity actually corresponds to the stronger property of *valuability* in the parlance of Dreyer (2005), 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 In two places expression typing uses subtyping: for function application (rule EAPP) and for sealing (rule ESEAL). Subtyping is like signature matching in ML modules: it allows not only for width subtyping on records, it can also substitute abstract types with concrete definitions. It is through that property, that the two occurrences of subtyping subsume universal elimination and existential introduction.

The subtyping judgement is defined on semantic types. It computes a substitution δ for the matched abstract types from the right-hand side, and 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 F-ing modules, merely

[11] More precisely, we actually have an infinite family of monads – one for each possible sequence of variable kinds – and the bind operator is heterogeneous, yielding the composition of two monads with composition being just nesting. See Rossberg (2016) for a more extensive discussion.

adding a straightforward rule for abstract type paths π (rule SPATH), because those can now occur as "module types".

However, we make one structural change: instead of guessing the substitution δ non-deterministically in a separate rule (rule U-MATCH in Rossberg *et al.* (2014)), 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 with the non-determinism before, at least with applicative functors.

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 the rule SFORGET. Those lookups incrementally produce the substitution δ whose domain corresponds to the (root variables of) the abstract paths $\overline{\pi}$. Substitutitions are combined in rule SSTR, where the vector of type paths is partitioned over the record components as needed.

Let us look at a couple of examples. When matching $\{\textbf{type}\ \mathsf{t} = \mathsf{bool};\ \mathsf{v} : \mathsf{t}\}$ against $\{\textbf{type}\ \mathsf{t};\ \mathsf{v} : \mathsf{t}\}$ then first, these types elaborate into $\{\mathsf{t} : [= \mathsf{bool}], \mathsf{v} : \mathsf{bool}\}$ and $\exists\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\}$, respectively. The subtyping judgement then is invoked with an empty path at first, i.e., checking $\{\mathsf{t} : [= \mathsf{bool}], \mathsf{v} : \mathsf{bool}\} \leq_\varepsilon \exists\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\}$. Rule SABS would first eliminate the quantifier and move the variable to the list of paths, thereby invoking $\{\mathsf{t} : [= \mathsf{bool}], \mathsf{v} : \mathsf{bool}\} \leq_\alpha \{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\}$. After applying rule SSTR this would arrive at matching the structure component t via $[= \mathsf{bool}] \leq_\alpha [= \alpha]$. That is where rule SFORGET fires, because the path matches the right-hand side. It produces the substitution $[\mathsf{bool}/\alpha]$, under which the remainder of the structure can be matched successfully.

When matching a type with multiple abstract type components, e.g., $\exists\alpha_1\alpha_2.\{\mathsf{t} : [= \alpha_1], \mathsf{u} : [= \alpha_2], \mathsf{v} : \alpha_2 \to \alpha_1\}$, then it proceeds similarly, but with a list of paths α_1, α_2 that is then split up by rule SSTR. Individual applications of SFORGET yield separate substitutions for each type component which SSTR composes on the way back.

Hence, normally, each path in $\overline{\pi}$ is just a plain abstract type variable that occurs free in Ξ. But as we saw 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. For example, matching $(\mathsf{a} : \textbf{type}) \Rightarrow (= \textbf{type}\ \mathsf{a})$ against $(\mathsf{a} : \textbf{type}) \Rightarrow \textbf{type}$ translates to $\forall\alpha'.[= \alpha'] \to_\mathrm{P} [= \alpha'] \leq \exists\beta.\forall\alpha.[= \alpha] \to_\mathrm{P} [= \beta\,\alpha]$ with $\kappa_\beta = \Omega \to \Omega$. Via rule SABS, this would try to match $\forall\alpha'.[= \alpha'] \to_\mathrm{P} [= \alpha'] \leq_\beta \forall\alpha.[= \alpha] \to_\mathrm{P} [= \beta\,\alpha]$, which involves looking up the higher-kinded β. After contra-variantly matching the function parameters, the second premise of rule SFUN then traverses into the functions' codomains with path $\beta\,\alpha$, trying to match $[= \alpha] \leq_{(\beta\,\alpha)} [= \beta\,\alpha]$. And indeed, that is the only way for SFORGET to "find" the path and yield $\delta = [\lambda\alpha.\alpha/\beta]$.

We assume that applying substitutions implicitly performs $\beta\eta$-normalisation. Because all type applications in semantic paths are saturated, no type lambdas remain in a semantic type after applying a substitution.

No types can actually be looked up from under a(nother) abstracted type Ξ with non-empty quantifier – because $\overline{\pi}$ is required to be empty when entering rule SABS. Hence, for any subtyping derivation $\Xi' \leq_{\overline{\pi}} \Xi$, it is either the vector $\overline{\pi}$ that is empty, or the quantifiers

of both Ξ and Ξ'. Either case can actually occur when invoking the second premise of rule SFUN, depending on whether the function types are pure or not.

The side conditions of the form $\delta\Sigma = \Sigma$ in rules SSTR and SFUN enforce that no variables in the domain of δ occur in Σ (under the usual freshness assumptions for bound variables). That in turn ensures that all abstract variables are properly scoped, i.e., do not occur before a "definitional" occurrence in a type of the form $[= \pi]$ that allows us to look them up.

Finally, we note that rule SFORGET is the single place in the whole set of 1ML typing rules where the predicativity restriction materialises: the rule allows only a small type σ on the left, so that only small types can end up in the image of the substitution δ.

4.3 Meta-Theory

It is relatively straightforward to verify by induction that elaboration is correct, i.e., produces only well-formed F_ω-objects:

Proposition 4.1 (*Correctness of 1ML$_{ex}$ Elaboration*)
Let $\Gamma \vdash \square$.

1. If $\Gamma \vdash T/D \rightsquigarrow \Xi$, then $\Gamma \vdash \Xi : \Omega$.
2. If $\Gamma \vdash E/B :_\eta \Xi \rightsquigarrow e$, then $\Gamma \vdash e : \Xi$, and if $\eta{=}\mathsf{P}$ then $\Xi{=}\Sigma$.
3. If $\Gamma \vdash \Xi' \leq_{\overline{\alpha\alpha'}} \Xi \rightsquigarrow \delta; f$ 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 $\Gamma \vdash f : \Xi' \to \delta\Xi$.

Together with the standard soundness result for System F_ω we can tell that 1ML$_{ex}$ is sound, i.e., a well-typed 1ML$_{ex}$ program will either diverge or terminate with a value of the right type. With the bare definition of 1ML$_{ex}$ we have given divergence cannot even arise, but a more complete ML-like language will surely have it.

Theorem 4.2 (*Soundness of 1ML$_{ex}$*)
If $\cdot \vdash E : \Xi \rightsquigarrow e$, then either $e \uparrow$ or $e \hookrightarrow^* v$ such that $\cdot \vdash v : \Xi$.

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

Theorem 4.3 (*Decidablity of 1ML$_{ex}$ Elaboration*)
All 1ML$_{ex}$ 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. The one odd case is rule TWHERE, 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. With that, 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*. Because the subtyping rules always remove quantifiers before applying a substituion, the overall weight of types involved decreases in all subtyping rules.

One suitable choice of weight functions is mapping to ordered pairs as follows:

$$W[\![\Xi]\!] = \langle Q[\![\Xi]\!], S[\![\Xi]\!]\rangle$$

$Q[\![\mathsf{bool}]\!]$	$= 0$	$S[\![\mathsf{bool}]\!]$	$= 1$
$Q[\![\alpha]\!]$	$= 0$	$S[\![\alpha]\!]$	$= 1$
$Q[\![\pi\,\sigma]\!]$	$= 0$	$S[\![\pi\,\sigma]\!]$	$= S[\![\pi]\!] + S[\![\sigma]\!] + 1$
$Q[\![\lambda\,\overline{\alpha}.\sigma]\!]$	$= 0$	$S[\![\lambda\,\overline{\alpha}.\sigma]\!]$	$= S[\![\sigma]\!] + 1$
$Q[\![[=\Xi]]\!]$	$= 2 \cdot Q[\![\Xi]\!]$	$S[\![[=\Xi]]\!]$	$= S[\![\Xi]\!] + 1$
$Q[\![\{\}]\!]$	$= 0$	$S[\![\{\}]\!]$	$= 1$
$Q[\![\{l_0{:}\Sigma_0, \overline{l{:}\Sigma}\}]\!]$	$= Q[\![\Sigma_0]\!] + Q[\![\{\overline{l{:}\Sigma}\}]\!]$	$S[\![\{l_0{:}\Sigma_0, \overline{l{:}\Sigma}\}]\!]$	$= S[\![\Sigma_0]\!] + S[\![\{\overline{l{:}\Sigma}\}]\!] + 1$
$Q[\![\forall\overline{\alpha}.\Sigma \to_\eta \Xi]\!]$	$= \lvert\overline{\alpha}\rvert + Q[\![\Sigma]\!] + Q[\![\Xi]\!]$	$S[\![\forall\overline{\alpha}.\Sigma \to_\eta \Xi]\!]$	$= S[\![\Sigma]\!] + S[\![\Xi]\!] + 1$
$Q[\![\exists\overline{\alpha}.\Sigma]\!]$	$= \lvert\overline{\alpha}\rvert + Q[\![\Sigma]\!]$	$S[\![\exists\overline{\alpha}.\Sigma]\!]$	$= S[\![\Sigma]\!]$

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.

Intuitively, the first component of a weight pair, $Q[\![]\!]$, measures the number of quantifiers – or more precisely, the number of quantifiers that need to be eliminated in the subtyping rules, thus the seemingly odd definition for the case $Q[\![[=\Xi]]\!]$. The second component, $S[\![]\!]$, is the actual syntactic size of the type expression. We apply addition point-wise to weights $W[\![]\!]$, and impose a lexicographic ordering on them:

$$\begin{array}{lcl} \langle q,s\rangle + \langle q',s'\rangle & := & \langle q+q', s+s'\rangle \\ \langle q,s\rangle < \langle q',s'\rangle & :\Leftrightarrow & q < q' \vee (q = q' \wedge s < s') \end{array}$$

The most important property of this definition of weights is that $Q[\![\sigma]\!] = 0$ for all small types σ, because small types do not contain quantifiers. Consequently, any large type will have more weight then even the biggest small type. 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:

Lemma 4.4 (*Weight Reduction under Small Substitution*)
Let δ be a small substitution.

1. $Q[\![\delta\Xi]\!] = Q[\![\Xi]\!]$.
2. $W[\![\delta\Xi]\!] < W[\![\Xi]\!] + \langle 1,0\rangle$.
3. $W[\![\delta\Xi]\!] \leq W[\![\Xi]\!] + \langle |\mathrm{dom}(\delta)|, 0\rangle$.

In more prose, the first property shows that substitution does not change the number of quantifiers; consequently, the second says that eliminating a quantifier is a strictly larger weight reduction then performing a substitution; and the third says that eliminating a sequence of quantifiers via substitution is a weight reduction or at least keeps the weight unchanged. 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 4.5 (*Termination of Algorithmic Subtyping*)
Let $\Gamma \vdash \Xi' : \Omega$ and $\Gamma, \overline{\alpha} \vdash \Xi : \Omega$ and $\overline{\pi} = \overline{\alpha\ \alpha'}$. 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 $W[\![\Xi']\!] + W[\![\Xi]\!] + \langle|\overline{\pi}|, 0\rangle$ 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 substitution δ is applied in a premise and S increases.

Syntax

(types) $T ::= \ldots \mid \mathsf{wrap}\ T$
(expressions) $E ::= \ldots \mid \mathsf{wrap}\ X{:}T \mid \mathsf{unwrap}\ X{:}T$

Abbreviations:

$\mathsf{wrap}\ E{:}T \quad := \quad \mathsf{let}\ X{=}E\ \mathsf{in}\ \mathsf{wrap}\ X{:}T$
$\mathsf{unwrap}\ E{:}T \quad := \quad \mathsf{let}\ X{=}E\ \mathsf{in}\ \mathsf{unwrap}\ X{:}T$

Semantic Types

(large) $\Sigma ::= \ldots \mid [\Xi]$
(small) $\sigma ::= \ldots \mid [\Xi]$

Desugarings:

(types) $[\Xi] := \{\mathsf{val} : \Xi\}$
(terms) $[e] := \{\mathsf{val} = e\}$

Types

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

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

Expressions

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

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

$$\frac{\Gamma \vdash X :_\mathrm{P} [\Xi'] \rightsquigarrow e \qquad \Gamma \vdash T \rightsquigarrow [\Xi] \qquad \Gamma \vdash \Xi' \le \Xi \rightsquigarrow f}{\Gamma \vdash \mathsf{unwrap}\ X{:}T :_{\eta(\Xi)} \Xi \rightsquigarrow f\,(e.\mathsf{val})}\ \textsc{Eunwrap}$$

Subtyping

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

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

Fig. 9. Extending $1\mathrm{ML}_{\mathrm{ex}}$ with Impredicativity

5 Expressiveness

5.1 Impredicativity Reloaded

Some readers will think that predicativity is a rather severe restriction. And they are right. So it is worth asking: Can we (re)enable impredicative type abstraction without breaking decidability?

Yes we can. One possibility is the common trick of piggy-backing datatypes: we could allow their data constructors to have large parameters. Because datatypes are *nominal* in ML, impredicativity is "hidden away" and does not interfere with subtyping, thereby avoiding the issue described in Section 1.2.

Here, we suggest a more orthogonal solution, namely introducing *structural* impredicative types. Figure 9 reveals how they can be defined as an easy extension to $1\mathrm{ML}_{\mathrm{ex}}$ and the rules we have previously given.

The trick is that a large type has to be injected into the universe of small types *explicitly*, by way of a special type "$\mathsf{wrap}\,T$" (not entirely dissimilar from the bracketted polytypes that occur in the semi-explicit first-class polymorphism of Garrigue & Rémy (1999)). Semantically, this type is represented by a new form "$[\Xi]$" (with the now familiar coding

trick of marking it through a special record) that may appear as a small type even when the inner type is large. The new type comes with respective expression forms for introducing and eliminating it.

The elaboration rules for wrapped types and expressions are straightforward. The crux of this approach is that subtyping does not extend to wrapped types (rule SWRAP): a wrapped type can only be matched by an *equivalent* type, not a subtype. This way, any infinite recursion is avoided; the weight function for the termination proof (Section 4.3) can trivially be extended to the new form of wrapped type as follows:

$$Q[\![[\Xi]]\!] = 0 \qquad S[\![[\Xi]]\!] = 1$$

That goes through because SWRAP is simply an axiom, i.e., always a leaf node in any subtyping derivation.

The syntax we chose for wrapped types 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 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 representation of opt is a large type, so it has to be wrapped in order to "make it small" and allow matching its abstract declaration in the signature.

5.2 Embedding F_ω

The elaboration of $1ML_{ex}$ 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 $1ML_{ex}$.

Doing so is fairly straightforward: Figure 10 gives the canonical translation function $[\![\_]\!]$ for F_ω kinds, types, and terms. It assumes a syntactic injection of F_ω type and term variables α and x into 1ML identifiers X_α and X_x, respectively. 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

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

where:

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

Fig. 10. Embedding of F_ω in 1ML

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.

Notably, all translated F_ω terms become 1ML expressions with a small type. And consequently, ground types translate to small types as well. That is necessary to encompass the impredicativity of F_ω. To achieve that for quantified types, the translation wraps the translation of every quantified type, using the extension from the previous Section 5.1. Wrapped expressions require a type annotation; we simply assume that the types τ_e and τ_{e_1} used in the figure are determined by context – as usual, this could be made more precise via a type-directed translation, but we 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:

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

For each F_ω term binding $x{:}\tau$ this embedding produces a respective 1ML binding $X_x{:}\sigma$, where the 1ML embedding $[\![\tau]\!]$ of the type τ is an *expression* denoting a first-class type (or function over it). This expression has a transparent type by construction. For each F_ω type

binding $\alpha{:}\kappa$ the embedding produces an additional 1ML term binding $X_\alpha{:}\Sigma$ of the type as a first-class value, where Σ is determined by the inverse elaboration of the embedding of the kind κ. For example, the F_ω kind $\Omega \to \Omega$ translates to the 1ML type $\mathsf{type} \Rightarrow \mathsf{type}$, which by the elaboration rules from Figure 7 is represented by the F_ω type $\exists\alpha{:}(\Omega \to \Omega).\, \forall\beta{:}\Omega.\, [= \alpha\,\beta]$. Consequently, a binding $\alpha{:}(\Omega \to \Omega)$ is twinned as $\alpha{:}(\Omega \to \Omega), X_\alpha{:}(\forall\beta{:}\Omega.\, [= \alpha\,\beta])$, thereby representing both the abstract type as well as a term carrying it. It is an invariant of elaboration and embedding that the existential quantifiers produced by the elaboration of $[\![\kappa]\!]$ always coincides with $\alpha{:}\kappa$ – the embedding of kinds is an inverse of the quantifier elaboration of 1ML types.

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

Theorem 5.1 (*Soundness of Embedding*)

1. For all κ, $\cdot \vdash [\![\kappa]\!] \rightsquigarrow \exists\alpha.\Sigma$ with $\kappa_\alpha = \kappa$ and $\cdot \vdash \exists\alpha.\Sigma : \Omega$.
2. For all Γ, if $\Gamma \vdash \square$, then $[\![\Gamma]\!] \vdash \square$.
3. For all τ, if $\Gamma \vdash \tau : \kappa$, then $[\![\Gamma]\!] \vdash [\![\tau]\!] : \Sigma$ and $[\![\Gamma]\!] \vdash [\![\kappa]\!] \rightsquigarrow \Xi$ and $[\![\Gamma]\!] \vdash \Sigma \leq \Xi$ (the latter implies that if $\kappa = \Omega$ then $\Sigma = [= \sigma]$).
4. For all e, if $\Gamma \vdash e : \tau$, then $[\![\Gamma]\!] \vdash [\![e]\!] : \sigma$ and $[\![\Gamma]\!] \vdash [\![\tau]\!] : [= \sigma]$.

The proof of (1) is by induction on the structure of κ, and the rest by (simultaneous) induction on the first derivations, respectively.

For a pedantic proof of correctness we would also need to show that the embedding is adequate, i.e., the produced $1ML_{ex}$ programs are computationally equivalent to the original F_ω terms. One way to achieve that would be through a suitable logical relation, but it would be rather involved, requiring simultaneous reasoning about the embedding just defined *and* the 1ML elaboration back into F_ω. Given the informal "obviousness" of this result we take the liberty to skip over it.

Although we will not show it here either, it is also worth noting that a *predicative* version of System F_ω can be embedded into plain $1ML_{ex}$. All uses of **wrap** and **unwrap** in the embedding of types and terms could simply be dropped – except in the case of existential formation, where it is replaced by sealing, i.e., $[\![\langle\tau_1, e_2\rangle_\tau]\!] = ([\![\tau_1]\!],\, [\![e_2]\!]) :> [\![\tau]\!]$.

5.3 Compositionality

The takeaway from the last two sections is that $1ML_{ex}$ is no more and no less powerful than System F_ω. But if it is all just a matter of a fairly direct translation, why is it so much less compelling to program in F_ω directly? What is the fundamental advantage that a language like 1ML provides, apart from just nifty type inference and a lot of syntax sugar?

The answer is *modular compositionality*. The heart of modularity is the ability to group types and values together into reusable units, typically called modules and signatures. But also, that from these, larger units can be formed in a compositional manner.

Consider the following toy signatures for demonstration:

```
type A = {type t; v : t}
type B = {type u; w : u}
```

They can be freely composed into new types:

type S = {a : A; b : B; c : {v : a.t; w : b.u}}
type F = S → S

Similarly, values of these types can be constructed and composed, for example:

m_1 :> A = {**type** t = int; v = 1}
m_2 :> B = {**type** u = bool; w = true}
m_3 : S = {a = m_1; b = m_2; c = {v = a.v; w = b.w}}

More interestingly, compound types can be abstracted over as they are:

g = **fun** (x : A) (y : B) (f : F) ⇒ f {a = x; b = y}
h = **fun** (p : S) (f : F) ⇒ (f p).a

With raw F_ω such composition is not directly possible. It is not possible to group types and values *together* into composable and reusable units like the signatures and structures above. Type components must instead be represented by quantified types, whose form, placement and scope is dependent on the circumstances.

For example, a naive attempt of translating A and B to F_ω would be to define them as the corresponding existential types, in direct adherence to Mitchell & Plotkin (1988):

$$\begin{array}{rcl} A & = & \exists t.\{\mathsf{v} : t\} \\ B & = & \exists u.\{\mathsf{w} : u\} \end{array}$$

But that's no good! How can we define S in terms of these types? How about functions corresponding to g and h? It is not possible.

The closest one can come to defining reusable named signatures in F_ω is by λ-abstracting the contained abstract types (Russo, 2003):

$$\begin{array}{rcl} A & = & \lambda t.\{\mathsf{v} : t\} \\ B & = & \lambda u.\{\mathsf{w} : u\} \\ S & = & \lambda t\, u.\{\mathsf{a} : A\, t, \mathsf{b} : B\, u, \mathsf{c} : \{\mathsf{v} : t, \mathsf{w} : u\}\} \end{array}$$

That way, they can be associated with the appropriate binders as needed, for example:

$$\begin{array}{rcl} F & = & \forall t\, u.\, S\, t\, u \to \exists t'\, u'.\, S\, t'\, u' \\ g & = & \lambda t\, u.\, \lambda x : A\, t.\, \lambda y : B\, u.\, \lambda f : F.\, f\, t\, u\, \{a = x, b = y\} \\ h & = & \lambda t\, u.\, \lambda p : S\, t\, u.\, \lambda f : F.\, \mathsf{unpack}\, \langle t'\, u', q\rangle = f\, t\, u\, p\, \mathsf{in}\, \mathsf{pack}\, \langle t', q.\mathsf{a}\rangle_{\exists t'.\, A\, t'} \end{array}$$

In general, composition in F_ω requires systematically separating all "static" components of a type (i.e., the types it contains) from its "dynamic" components (i.e., proper values), a transformation known as *phase-splitting* in the module literature (Dreyer *et al.*, 2003). And if the latter definitions look awfully like the result of 1ML elaboration then that is no coincidence! Behind the syntax distraction, 1ML elaboration is primarily a means of automatically and systematically phase-splitting the input program.

Of course, the programmer could try to do that manually, following the sketch above. However, from those simple examples it is already easy to extrapolate that this approach gets very laborious very quickly. Let alone the managing of corresponding existentials on the term level. Harper & Pierce (2005) in fact argue that modular programming based on parameterised signatures (which is what the above is, essentially) leads to a combinatorial explosion of parameters accumulating with each layer of abstraction.

Syntax (types) $T ::= \ldots \mid \_ \mid {}'(X : \mathsf{type}) \Rightarrow T$

Abbreviations:

(expressions)	$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3$	:=	$\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 : \_$
	$\mathsf{fun}\ X \Rightarrow E$	:=	$\mathsf{fun}\,(X : \_) \Rightarrow E$
(types)	${}'X \Rightarrow T$	:=	${}'(X : \mathsf{type}) \Rightarrow T$
(declarations)	$X\ {}'\overline{Y} : T$	:=	$X : \overline{{}'(Y : \mathsf{type}) \Rightarrow T}$

Fig. 11. Extension to Full 1ML

That problem carries over to other proposals that suggest adding or emulating modular structure via manual management of polymorphism or separating type components. The fact that under such approaches it is simply not possible to really build fully composable program units means that such approaches are not really modular. 1ML avoids this problem because it does not treat modular composition as an afterthought, but instead takes it as a starting point at the centre of its design.

6 Full 1ML

A language without type inference is unworthy of being named ML. This section therefore extends $1ML_{ex}$ to full 1ML, which includes implicit typing and recovers ML-style implicit polymorphism. Figure 11 shows the minimal syntactic extension necessary.

6.1 Extensions

Syntactically, full 1ML merely adds two new forms of type expression to the kernel: *wildcards* "_" for types inferred from context, and *implicit* function types, distinguished by a leading tick ' (a choice that will become clear in a moment). Let us walk through a number of examples, and also explain some of the new syntax sugar on the way.

Inferred Types The wildcard "_" stands for a type that is to be inferred from context – similar to OCaml or dependently typed languages like Coq. 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.

For example, with this syntax, the precise argument type of a function or conditional can be left out:

$\mathsf{or}\ (\mathsf{x} : \_)\ (\mathsf{y} : \_) = \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{y} : \_$

The type system will infer that all wildcards in this function are placeholders for bool. As sugar on top of this new piece of kernel syntax we allow a type annotation ": _" on a function parameter or conditional to be omitted entirely, thereby recovering the implicitly typed expression syntax familiar from ML. So the above can be condensed to just

$\mathsf{or}\ \mathsf{x}\ \mathsf{y} = \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{y}$

At this point we drop the $1ML_{ex}$ interpretation of an unannotated parameter as a **type**; we only keep that interpretation in those declaration or binding shorthands that are headed by the **type** keyword, that is:

```
type t a b = a → b
```

still defaults a and b to have type **type**. But in all other places parameter types default to "_" in full 1ML, that is, they can be any small type.

Wildcards can also be used to give partial annotations:

```
projPlus1 b (p : {x : _; y : _}) = (if b then p.x else p.y) + 1
```

In this example, the annotation specifies the outer shape of the parameter p (so that it avoids running into inference problems with records, see below). But the component types are inferred to be int.

Large types cannot be inferred. Consequently, type annotations still have to be given when a parameter type is large (as, e.g., for polymorphic parameters) or when a conditional's type is large (as in the "computed modules" example given in Section 2) – that is, in all cases that the core type system of conventional ML cannot express.

Polymorphic Inference The previous examples all were monomorphic. But of course, the claim to fame of ML's type inference (Damas & Milner, 1982) is that it can assign *polymorphic* types to let-bindings. And of course, that ought to work in 1ML as well:

```
pick b x y = if b then x else y
```

Here, the instantiation of the (omitted) wildcards on the parameters x and y is arbitrary and the type of pick thus polymorphic. This polymorphic type can be expressed syntactically with the other new form of type expression, *implicit function types*:

```
pick : '(a : type) ⇒ bool ⇒ a ⇒ a ⇒ a
```

This corresponds to an ML-style polymorphic type, but makes the binder explicit. For obvious reasons, an implicit function has to be pure. The parameter has to be of type **type**, whose being small fits nicely with the fact that ML can only abstract monotypes, and not type constructors. Because the type annotation must be **type** anyway, we allow to omit it (per Figure 11) and just write the above type as:

```
pick : 'a ⇒ bool ⇒ a ⇒ a ⇒ a
```

Or even moving the implicit parameter to the left-hand side, just like we already allow for explicit parameters (cf. Figure 2):

```
pick 'a : bool ⇒ a ⇒ a ⇒ a
```

Now it is more apparent: the tick becomes a pun on ML's type variable syntax, but without relying on brittle implicit scoping rules.

As the name would suggest, there are no explicit introduction or elimination forms for implicit functions. Instead, implicit parameters are introduced and eliminated implicitly. So, as one would rightly expect from an ML, the previous function can be invoked without worrying about its type argument:

```
x = pick false 2 3
y = pick true (pick false) (pick true)
```

Because implicit parameters always have type **type**, again only small types can be inferred as arguments. So (pick (size > threshold) HashMap TreeMap) would *not* be well-typed, but demands an explicitly polymorphic version of the pick function.

Implicit type generalisation applies to expressions, not just bindings. That is true in some formulations of ML-style polymorphism as well, but the difference usually is not observable. In 1ML it is, because we can have contexts that require a polymorphic value:

```
makeAB (g : '(a : type) ⇒ a ⇒ a ⇒ a) = {a = g 1 2; b = g true false}
```

Given this, the argument function in the invocation

```
makeAB (fun x y ⇒ x)
```

is inferred to have polymorphic type. In fact, we could even apply makeAB (pick true), as we show next.

Purity restriction Regular ML imposes the *value restriction* (Wright, 1995) as an approximation for ensuring the absence of side effects when applying type generalisation, which is needed for soundness. Sine 1ML already tracks purity, it can (once more) replace a syntactic restriction with a more precise semantic one and reuse a concept that we already have introduced. Simply, any *pure* expression can have its type generalised. For example:

```
pickFst = pick true
```

has polymorphic type

```
pickFst : '(a : type) ⇒ a ⇒ a ⇒ a
```

because the fat arrow in (the first explicit argument of) the type of pick indicates that its application has no side effects. Compare that to

```
pickImp1 b x y = if !b then x else y
pickImp2 b = if !b then fun x y ⇒ x else fun x y ⇒ y
```

Assuming (!) : '(a : **type**) ⇒ ref a → a, as the implicitly polymorphic version of the rd operator from Section 2, the previous functions have types as follows:

```
pickImp1 : '(a : type) ⇒ ref bool ⇒ a ⇒ a → a
pickImp2 : '(a : type) ⇒ ref bool → a ⇒ a ⇒ a
```

Respectively, given a reference r, (pickImp$_1$ r) could still be generalised, but (pickImp$_2$ r) could not.

Subtyping The definition and use of expressions is just one way to introduce or eliminate implicit functions. The other is through subtyping: an implicit function type is a subtype of any possible application of this function. If you imagine the two types:

```
type T1 = int → bool
type T2 = 'a ⇒ 'b ⇒ a → b
```

then T_2 is a subtype of T_1. This is another generalisation of a notion that already exists in traditional ML: when matching signatures, a value component can be matched by a more polymorphic value component. Here, it becomes an independent notion.

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

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

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

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

$$\frac{\Gamma, \overline{\alpha} \vdash E :_{\mathtt{P}} \Sigma \rightsquigarrow e \qquad \overline{\kappa_\alpha = \Omega}}{\Gamma \vdash E :_{\mathtt{P}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow \lambda\overline{\alpha}.\lambda_{\mathtt{A}} x{:}\{\}.e}\textsc{Egen}$$

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

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

$$\frac{\overline{\Gamma \vdash \sigma : \kappa_{\alpha'}} \qquad \Gamma \vdash \Sigma'[\overline{\sigma}/\overline{\alpha}'] \leq_{\overline{\pi}} \Sigma \rightsquigarrow \delta; f}{\Gamma \vdash \forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \leq_{\overline{\pi}} \Sigma \rightsquigarrow \delta; \lambda x.\, f\,((x\,\overline{\sigma}\,\{\}).\mathtt{A})}\textsc{Simpll}$$

$$\frac{\Gamma, \overline{\alpha} \vdash \Sigma' \leq_{\overline{\pi}} \Sigma \rightsquigarrow \delta; f \qquad \overline{\mathrm{fv}(\delta\pi) \not\ni \overline{\alpha}}}{\Gamma \vdash \Sigma' \leq_{\overline{\pi}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow \delta; \lambda x.\, \lambda\overline{\alpha}.\lambda_{\mathtt{A}} y{:}\{\}.\, f\,x}\textsc{Simplr}$$

Fig. 12. New elaboration rules for full 1ML

Example With these few extensions, the Map functor from Section 2 can now be written in 1ML very much like it would be 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
}
```

6.2 Semantics

The semantics for the extension of $1\mathrm{ML}_{\mathrm{ex}}$ to Full 1ML can be seen in Figure 12. As is typical for type systems with inference, the formulation of the semantics is purely *declarative*: the typing rules are non-deterministic, they simply "guess" the right types

and quantifiers in various places. We defer the discussion of an actual *algorithm* that implements type inference for this system to Section 7).

Semantic Types Wildcards do not induce any extension to semantic types, since they just stand for a suitable small type σ. But obviously, we have to extend our semantic types with a representation for implicit function types.

The semantic representation of the syntactic type $'(\mathsf{a} : \mathbf{type}) \Rightarrow \{\ldots\}$ is $\forall\alpha.\{\} \to_{\mathtt{A}} \{\ldots\}$. We write these types with an arrow $\to_{\mathtt{A}}$ ("A" for automatic), in order to reuse the notational encoding we already introduced for effects. However, it is a distinct form from $\to_\eta$, i.e., A is not an effect, and not included in η. All implicit functions are pure.

The semantic denotation of an explicitly written, syntactic implicit function type always has exactly one quantified type variable. However, *inferred* implicit functions can quantify multiple type variables at once. For example, the polymorphic function

$\mathsf{f\ x\ y} = \{\mathsf{a} = \mathsf{x};\ \mathsf{b} = \mathsf{y}\}$

can conveniently be typed $\forall\alpha\beta.\{\} \to_{\mathtt{A}} \alpha \to_{\mathtt{P}} \beta \to_{\mathtt{P}} \{\mathsf{a} : \alpha, \mathsf{b} : \beta\}$. As we will see, this type is interchangeable (via subtyping in both directions) with $\forall\alpha.\{\} \to_{\mathtt{A}} \forall\beta.\{\} \to_{\mathtt{A}} \alpha \to_{\mathtt{P}} \beta \to_{\mathtt{P}} \{\mathsf{a} : \alpha, \mathsf{b} : \beta\}$, because all arguments are implicit either way. So this is simply a technical convenience.

The term argument to an implicit function is always $\{\}$, a.k.a. unit. That is so because the function's parameters are simply a sequence of types $\overline{\alpha}$, and there is little value in reifying them as terms $\overline{[= \alpha]}$, since they are implicit anyway.[12]

Typing Rules With regard to elaboration, we first need the formation rules for the new pieces of type syntax. The rule for wildcards simply guesses a (well-formed) small type non-deterministically. The one for implicit functions mirrors the one for regular function types, but with a fixed parameter type. Moreover, implicit functions may not produce abstract types – a restriction that wouldn't strictly be necessary, but precludes questionable examples like the following, where each use of weird could denote a completely different type:[13]

$\mathsf{f}\ (\mathsf{weird} : {'\mathsf{a}} \Rightarrow \mathbf{type})\ (\mathsf{x} : \mathsf{weird})\ (\mathsf{y} : \mathsf{weird}) = \ldots$

Next, the new typing rules for introducing and eliminating implicit functions, EGEN and EINST, match common formulations of ML-style polymorphism (Damas & Milner, 1982). The former rule simply introduces (non-deterministically) a sequence of fresh type variables into the context that can be used to type an expression – which must behave nicely by being pure, however. In the conclusion, the rule turns these variables into implicit function parameters. Conversely, the latter rule can take any expression with implicit function

[12] We could have dropped the term argument entirely, representing implicit function types simply as $\forall_{\mathtt{A}}\overline{\alpha}.\Sigma$, but we opted for keeping the shape of all function types uniform and reusing the $\to_x$ notation. In part, because we foresee nice generalisations of implicit functions.

[13] Our type system still allows *definitions* like "**type** weird = _" to be polymorphic with type $\forall\alpha.\{\} \to_{\mathtt{A}} [= \alpha]$. Those could be ruled out by removing $[= \sigma]$ from the syntax of small types. However, that is a more substantial restriction. More practical experience with the language is probably needed to figure out the most pragmatic trade-off.

type and replaces its type parameters with arbitrary (small) types. The only difference to plain Damas/Milner is that the rule must allow for abstract types $\overline{\alpha}$ being generated by the (possibly impure) expression; as usual, their quantifiers are just hoisted.

Finally, subtyping allows the implicit introduction and elimination of implicit functions as well, via instantiation on the left, or skolemisation on the right (rules SIMPLL and SIMPLR). As mentioned earlier, this closely corresponds to ML's signature matching rules. In combination, the two rules also implement reflexivity for implicit function types. For example, $\forall\alpha.\{\} \to_{\mathtt{A}} [= \alpha]$ can be derived to be a subtype of itself by first applying SIMPLR in reverse, moving the right-hand side α from the quantifier to the context, and then applying SIMPLL to derive $\forall\alpha'.\{\} \to_{\mathtt{A}} [= \alpha'] \leq [= \alpha]$ via instantiation $[\alpha/\alpha']$.

6.3 Meta-Theory Revisited

The additional semantics of 1ML does not introduce any difficulty for the soundness proof from Section 4.3. The following properties extend easily to the new rules:

Proposition 6.1 (*Correctness of Full 1ML Elaboration*)
Let $\Gamma \vdash \square$.

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

Like before, soundness is still a direct corollary:

Theorem 6.2 (*Soundness of Full 1ML*)
If $\cdot \vdash E : \Xi \rightsquigarrow e$, then either $e \uparrow$ or $e \hookrightarrow^* v$ such that $\cdot \vdash v : \Xi$.

What breaks, though, is our previous proof for decidability, because several of the new rules are "guessing". The next section is dedicated to coping with that problem.

7 Type Inference

With the additions from Figure 12 we have turned the deterministic typing and elaboration judgements of $1\mathrm{ML}_{\mathrm{ex}}$ non-deterministic. In a derivation, one has to guess types (in rules TINFER, EINST, SIMPLL) and quantifiers (in rule EGEN). Moreover, one has to decide when to apply rules EGEN and EINST. Clearly, an algorithm is needed.

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

Subtyping sneaking into inference may seem scary at first. But closer inspection of the subtyping rules reveals that, when applied to small types, they almost degenerate to type equivalence! The only exception is width subtyping on records, which we will get back to in Section 7.2. 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* – place holders for small types – and a variation of unification.

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

$$\frac{\Gamma \vdash_\theta^{[=,]} E :_\mathrm{P} [= \Xi]}{\Gamma \vdash_\theta E \rightsquigarrow \Xi}\,\textsc{ItPath} \qquad \frac{\upsilon \text{ fresh} \qquad \Delta_\upsilon = \mathrm{dom}(\Gamma)}{\Gamma \vdash_{[]} \_ \rightsquigarrow \upsilon}\,\textsc{ItInfer}$$

$$\frac{\kappa_\alpha = \Omega}{\Gamma \vdash_{[]} \mathsf{type} \rightsquigarrow \exists \alpha.[= \alpha]}\,\textsc{ItType} \qquad \frac{}{\Gamma \vdash_{[]} \mathsf{bool} \rightsquigarrow \mathsf{bool}}\,\textsc{ItBool}$$

$$\frac{\Gamma \vdash_\theta D \rightsquigarrow \Xi}{\Gamma \vdash_\theta \{D\} \rightsquigarrow \Xi}\,\textsc{ItStr} \qquad \frac{\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \quad \Gamma; \overline{\alpha}_1, X{:}\Sigma_1 \ \theta_1 \vdash_{\theta_2} T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2}{\Gamma \vdash_{\theta_2} (X{:}T_1) \to T_2 \rightsquigarrow \forall \overline{\alpha}_1.\Sigma_1 \to_\mathrm{I} \exists \overline{\alpha}_2.\Sigma_2}\,\textsc{ItFun}$$

$$\frac{\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \quad \Gamma; \overline{\alpha}_1, X{:}\Sigma_1 \ \theta_1 \vdash_{\theta_2} T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2 \quad \overline{\kappa_{\alpha'_2} = \kappa_{\alpha_1} \to \kappa_{\alpha_2}}}{\Gamma \vdash_{\theta_2} (X{:}T_1) \Rightarrow T_2 \rightsquigarrow \exists \overline{\alpha}'_2.\forall \overline{\alpha}_1.\Sigma_1 \to_\mathrm{P} \Sigma_2[\overline{\alpha'_2\ \alpha_1}/\overline{\alpha}_2]}\,\textsc{ItPFun}$$

$$\frac{\Gamma; \alpha, X{:}[= \alpha] \vdash_\theta T \rightsquigarrow \Sigma \qquad \kappa_\alpha = \Omega}{\Gamma \vdash_\theta \text{'}(X{:}\mathsf{type}) \Rightarrow T \rightsquigarrow \forall \alpha.\{\} \to_\mathrm{A} \Sigma}\,\textsc{ItImpl}$$

$$\frac{\Gamma \vdash_\theta T \rightsquigarrow \Xi}{\Gamma \vdash_\theta \mathsf{wrap}\ T \rightsquigarrow [\Xi]}\,\textsc{ItWrap} \qquad \frac{\Gamma \vdash_\theta E :_\mathrm{P} \Sigma \qquad E = E' {:>} T \vee E = E' {:} T \vee E = \mathsf{type}\ T}{\Gamma \vdash_\theta (= E) \rightsquigarrow \Sigma}\,\textsc{ItSing}$$

$$\frac{\Gamma \vdash_{\theta_1} T_1 \rightsquigarrow \exists \overline{\alpha}_1.\Sigma_1 \quad \overline{\alpha}_1 = \overline{\alpha}_{11} \uplus \overline{\alpha}_{12} \quad \Gamma \ \theta_1 \vdash_{\theta_2} T_2 \rightsquigarrow \exists \overline{\alpha}_2.\Sigma_2 \quad \Gamma, \overline{\alpha}_{11}, \overline{\alpha}_2 \ \theta_2 \vdash_\theta \Sigma_2 \leq_{\overline{\alpha}_{12}} \Sigma_1.\overline{X} \rightsquigarrow \delta}{\Gamma \vdash_\theta T_1 \ \mathsf{where}\ (\overline{.X{:}T_2}) \rightsquigarrow \exists \overline{\alpha}_{11}\overline{\alpha}_2.\delta\Sigma'_1[\overline{.X{=}\Sigma_2}]}\,\textsc{ItWhere}$$

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

$$\frac{\Gamma \vdash_\theta T \rightsquigarrow \exists \overline{\alpha}.\Sigma}{\Gamma \vdash_\theta X{:}T \rightsquigarrow \exists \overline{\alpha}.\{X{:}\Sigma\}}\,\textsc{IdVar} \qquad \frac{\Gamma \vdash_\theta T \rightsquigarrow \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}{\Gamma \vdash_\theta \mathsf{include}\ T \rightsquigarrow \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}\,\textsc{IdIncl}$$

$$\frac{\Gamma \vdash_{\theta_1} D_1 \rightsquigarrow \exists \overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \quad \Gamma; \overline{\alpha}_1, \overline{X_1{:}\Sigma_1}\ \theta_1 \vdash_{\theta_2} D_2 \rightsquigarrow \exists \overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \qquad \overline{X}_1 \cap \overline{X}_2 = \emptyset}{\Gamma \vdash_{\theta_2} D_1; D_2 \rightsquigarrow \exists \overline{\alpha}_1\overline{\alpha}_2.\{\overline{X_1{:}\Sigma_1}, \overline{X_2{:}\Sigma_2}\}}\,\textsc{IdSeq} \qquad \frac{}{\Gamma \vdash_{[]} \varepsilon \rightsquigarrow \{\}}\,\textsc{IdEmpty}$$

Fig. 13. Inference for Types

7.1 Algorithm

Figures 13-15 show how the elaboration rules from the previous sections (including the rules for impredicative types) can be turned into an inference algorithm. Because we are just interested in typing, witness *terms* (i.e., the bits that were previously greyed out) are omitted from the rules this time.

In order to treat these rules as a definition of a recursive algorithm, we need to interpret the relational judgements as functions. Therefore, we have to be explicit about what are "inputs" and what are "outputs" to these relations. We mark all outputs with a shaded background in the rule schemata given in the figures, i.e., everything right of a colon ":" or

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

$$\frac{\Gamma(X) = \Sigma}{\Gamma \vdash_{[]} X :_{\mathrm{P}} \Sigma}\,\textsc{IEVar} \qquad \frac{\Gamma \vdash_\theta T \rightsquigarrow \Xi}{\Gamma \vdash_\theta \mathbf{type}\ T :_{\mathrm{P}} [= \Xi]}\,\textsc{IEType}$$

$$\frac{}{\Gamma \vdash_{[]} \mathbf{true} :_{\mathrm{P}} \mathsf{bool}}\,\textsc{IETrue} \qquad \frac{}{\Gamma \vdash_{[]} \mathbf{false} :_{\mathrm{P}} \mathsf{bool}}\,\textsc{IEFalse}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_{\theta_0}^{\mathsf{bool}} X :_{\mathrm{P}} \mathsf{bool} & \Gamma\ \theta_0 \vdash_{\theta_1}^{\forall} E_1 :_{\eta_1} \Xi_1 \qquad \Gamma\ \theta_3 \vdash \theta_4\ \Xi_1 \le \Xi \\ \Gamma\ \theta_2 \vdash \theta_3\ T \rightsquigarrow \Xi & \Gamma\ \theta_1 \vdash_{\theta_2}^{\forall} E_2 :_{\eta_2} \Xi_2 \qquad \Gamma\ \theta_4 \vdash \theta_5\ \Xi_2 \le \Xi\end{array}}{\Gamma \vdash_{\theta_5} \mathbf{if}\ X\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 : T :_{\eta_1 \vee \eta_2 \vee \eta(\Xi)} \Xi}\,\textsc{IEIf}$$

$$\frac{\Gamma \vdash_\theta B :_\eta \Xi}{\Gamma \vdash_\theta \{B\} :_\eta \Xi}\,\textsc{IEStr} \qquad \frac{\Gamma \vdash_\theta^{\{\cdot\}} E :_\eta \exists \overline{\alpha}.\{X{:}\Sigma, \overline{X'{:}\Sigma'}\}}{\Gamma \vdash_\theta E.X :_\eta \exists \overline{\alpha}.\Sigma}\,\textsc{IEDot}$$

$$\frac{\Gamma \vdash_{\theta_1} T \rightsquigarrow \exists \overline{\alpha}.\Sigma \qquad \Gamma; \overline{\alpha}, X{:}\Sigma\ \theta_1 \vdash_{\theta_2} E :_\eta \Xi}{\Gamma \vdash_{\theta_2} \mathbf{fun}\,(X{:}T) \Rightarrow E :_{\mathrm{P}} \forall \overline{\alpha}.\Sigma \to_\eta \Xi}\,\textsc{IEFun}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\theta_1}^{\rightarrow} X_1 :_{\mathrm{P}} \forall \overline{\alpha}.\Sigma_1 \to_\eta \Xi \\ \Gamma\ \theta_1 \vdash \theta_2\ X_2 :_{\mathrm{P}} \Sigma_2 \qquad \Gamma\ \theta_2 \vdash \theta_3\ \Sigma_2 \le_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta\end{array}}{\Gamma \vdash_{\theta_3} X_1\ X_2 :_\eta \delta\Xi}\,\textsc{IEApp}$$

$$\frac{\Gamma \vdash_{\theta_1} X :_{\mathrm{P}} \Sigma_1 \qquad \Gamma\ \theta_1 \vdash_{\theta_2} T \rightsquigarrow \exists \overline{\alpha}.\Sigma_2 \qquad \Gamma\ \theta_2 \vdash \theta_3\ \Sigma_1 \le_{\overline{\alpha}} \Sigma_2 \rightsquigarrow \delta}{\Gamma \vdash_{\theta_3} X{:>}T :_{\eta(\exists \overline{\alpha}.\Sigma_2)} \exists \overline{\alpha}.\Sigma_2}\,\textsc{IESeal}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\theta_1} X :_{\mathrm{P}} \Sigma \\ \Gamma\ \theta_1 \vdash \theta_2\ T \rightsquigarrow [\Xi] \qquad \Gamma\ \theta_2 \vdash \theta_3\ \Sigma \le \Xi\end{array}}{\Gamma \vdash_{\theta_3} \mathbf{wrap}\ X{:}T :_{\mathrm{P}} [\Xi]}\,\textsc{IEWrap} \qquad \frac{\begin{array}{ll}\Gamma \vdash_{\theta_1}^{[\cdot]} X :_{\mathrm{P}} [\Xi'] & \\ \Gamma\ \theta_1 \vdash \theta_2\ T \rightsquigarrow [\Xi] & \Gamma\ \theta_2 \vdash \theta_3\ \Xi' \le \Xi\end{array}}{\Gamma \vdash_{\theta_3} \mathbf{unwrap}\ X{:}T :_{\eta(\Xi)} \Xi}\,\textsc{IEUnwrap}$$

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

$$\frac{\Gamma \vdash_\theta^{\forall} E :_\eta \exists \overline{\alpha}.\Sigma}{\Gamma \vdash_\theta X{=}E :_\eta \exists \overline{\alpha}.\{X{:}\Sigma\}}\,\textsc{IBVar} \qquad \frac{\Gamma \vdash_\theta^{\{\cdot\}} E :_\eta \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}{\Gamma \vdash_\theta \mathbf{include}\ E :_\eta \exists \overline{\alpha}.\{\overline{X{:}\Sigma}\}}\,\textsc{IBIncl}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_{\theta_1} B_1 :_{\eta_1} \exists \overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} & \overline{X}'_1 = \overline{X}_1 - \overline{X}_2 \\ \Gamma; \overline{\alpha}_1, \overline{X_1{:}\Sigma_1}\ \theta_1 \vdash_{\theta_2} B_2 :_{\eta_2} \exists \overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} & \overline{X'_1{:}\Sigma'_1} \subseteq \overline{X_1{:}\Sigma_1}\end{array}}{\Gamma \vdash_{\theta_2} B_1; B_2 :_{\eta_1 \vee \eta_2} \exists \overline{\alpha}_1 \overline{\alpha}_2.\{\overline{X'_1{:}\Sigma'_1}, \overline{X_2{:}\Sigma_2}\}}\,\textsc{IBSeq} \qquad \frac{}{\Gamma \vdash_{[]} \varepsilon :_{\mathrm{P}} \{\}}\,\textsc{IBEmpty}$$

Fig. 14. Inference for Expressions

squiggly arrow "$\rightsquigarrow$". The θ index on the turnstile (explained in a minute) is an additional output in every judgement. All other meta variables occurring in the schemata are inputs.

The basic idea of these algorithmic rules is to introduce a (free) inference variable υ wherever the original declarative rules have to guess a (small) type – for example, in rule ITInfer. Furthermore, the rules are augmented by outputting a substitution θ for *resolved* inference variables: all judgements have the form $\Gamma \vdash_\theta J$, which, roughly, implies the

Subtyping $\quad \Gamma \vdash_\theta \Xi \le \Xi' \quad := \quad \Gamma \vdash_\theta \Xi \le_\varepsilon \Xi' \rightsquigarrow \mathrm{id}$ $\qquad \boxed{\Gamma \vdash_\theta \Xi' \le_{\overline{\pi}} \Xi \rightsquigarrow \delta}$

$$\frac{}{\Gamma \vdash_{[]} \upsilon \le \upsilon}\textsc{IsRefl}$$

$$\frac{\Gamma \vdash^!_\theta \upsilon \approx \Sigma \qquad \Gamma\ \theta \vdash_{\theta'} \upsilon \le \Sigma}{\Gamma \vdash_{\theta'} \upsilon \le \Sigma}\textsc{IsResL} \qquad \frac{\Gamma \vdash^!_\theta \upsilon \approx \Sigma' \qquad \Gamma\ \theta \vdash_{\theta'} \Sigma' \le \upsilon}{\Gamma \vdash_{\theta'} \Sigma' \le \upsilon}\textsc{IsResR}$$

$$\frac{\Gamma \vdash_\theta \pi' = \pi}{\Gamma \vdash_\theta \pi' \le \pi}\textsc{IsPath} \qquad \frac{}{\Gamma \vdash_{[]} \mathbf{bool} \le \mathbf{bool}}\textsc{IsBool}$$

$$\frac{\Gamma \vdash_\theta \Xi' \le \Xi \qquad \Gamma\ \theta \vdash_{\theta'} \Xi \le \Xi'}{\Gamma \vdash_{\theta'} [= \Xi'] \le [= \Xi] \rightsquigarrow []}\textsc{IsType} \qquad \frac{}{\Gamma \vdash_{[]} [= \sigma] \le_{\alpha_0\,\overline{\alpha}} [= \alpha_0\,\overline{\alpha}] \rightsquigarrow [\lambda\overline{\alpha}.\sigma/\alpha_0]}\textsc{IsForget}$$

$$\frac{\Gamma \vdash_\theta \Xi' = \Xi}{\Gamma \vdash_\theta [\Xi'] \le [\Xi]}\textsc{IsWrap} \qquad \frac{}{\Gamma \vdash_{[]} \{\overline{l{:}\Sigma'}\} \le \{\}}\textsc{IsEmpty}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_1} \Sigma_1' \le_{\overline{\pi}_1} \Sigma_1 \rightsquigarrow \delta_1 \qquad \overline{\mathrm{head}(\pi_1)} \subseteq \mathrm{fv}(\Sigma_1) \\ \Gamma\ \theta_1 \vdash_{\theta_2} \{\overline{l'{:}\Sigma'}\} \le_{\overline{\pi}_2} \{\overline{l{:}\delta_1\Sigma}\} \rightsquigarrow \delta_2 \qquad \overline{\mathrm{head}(\pi_2)} \not\ni \mathrm{fv}(\Sigma_1) \qquad \theta_2\delta_2\Sigma_1 = \theta_2\Sigma_1\end{array}}{\Gamma \vdash_{\theta_2} \{l_1{:}\Sigma_1', \overline{l'{:}\Sigma'}\} \le_{\overline{\pi}_1\overline{\pi}_2} \{l_1{:}\Sigma_1, \overline{l{:}\Sigma}\} \rightsquigarrow \delta_1\delta_2}\textsc{IsStr}$$

$$\frac{\begin{array}{c}\Gamma, \overline{\alpha} \vdash_{\theta_1} \Sigma \le_{\overline{\alpha}'} \Sigma' \rightsquigarrow \delta_1 \qquad \eta' \le \eta \\ \Gamma; \overline{\alpha}\ \theta_1 \vdash_{\theta_2} \delta_1\Xi' \le_{\overline{\pi}\overline{\alpha}} \Xi \rightsquigarrow \delta_2 \qquad \theta_2\delta_2\Sigma = \theta_2\Sigma\end{array}}{\Gamma \vdash_{\theta_2} (\forall\overline{\alpha}'.\Sigma' \to_{\eta'} \Xi') \le_{\overline{\pi}} (\forall\overline{\alpha}.\Sigma \to_\eta \Xi) \rightsquigarrow \delta_2}\textsc{IsFun}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_\theta \Sigma' \le_{\overline{\alpha}} \Sigma \rightsquigarrow \delta \qquad \overline{\alpha'}\overline{\alpha} \ne \varepsilon}{\Gamma \vdash_\theta \exists\overline{\alpha}'.\Sigma' \le \exists\overline{\alpha}.\Sigma}\textsc{IsAbs}$$

$$\frac{\Sigma^\circ = \mathbb{S} \qquad \overline{\upsilon\ \mathrm{fresh}} \qquad \overline{\Delta_\upsilon = \mathrm{dom}(\Gamma)} \qquad \Gamma \vdash_\theta \Sigma'[\overline{\upsilon}/\overline{\alpha}'] \le_{\overline{\pi}} \Sigma \rightsquigarrow \delta}{\Gamma \vdash_\theta \forall\overline{\alpha}'.\{\} \to_{\mathtt{A}} \Sigma' \le_{\overline{\pi}} \Sigma \rightsquigarrow \delta}\textsc{IsImplL}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_\theta \Sigma' \le_{\overline{\pi}} \Sigma \rightsquigarrow \delta \qquad \overline{\mathrm{fv}(\theta\delta\pi) \not\ni \overline{\alpha}}}{\Gamma \vdash_\theta \Sigma' \le_{\overline{\pi}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \rightsquigarrow \delta}\textsc{IsImplR}$$

Fig. 15. Inference for Subtyping

respective $\overline{\upsilon}, \theta\Gamma \vdash \theta J$ declaratively, where $\overline{\upsilon}$ in the context binds the unresolved inference variables that still appear free in $\theta\Gamma$ or θJ (as if they were regular type variables).

In rules with multiple premises, the output substitution of one premise needs to be applied to the inputs of the next, and all output substitutions must be composed. We avoid too much notational awkwardness with abbreviations of the form

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

for all inference judgement forms J, where the θ left of the turnstile is effectively an extra input and ${}^\theta J$ is meant to apply θ to J's inputs, as defined above. This notation is used to consistently thread and compose substitutions through all rules that have multiple premises.

Generalisation $\boxed{\Gamma \vdash^{\forall}_{\theta} E :_{\eta} \Xi}$

$$\frac{\Gamma \vdash_{\theta} E :_{\mathtt{I}} \Xi}{\Gamma \vdash^{\forall}_{\theta} E :_{\mathtt{I}} \Xi}\text{IGIMPURE} \qquad \frac{\Gamma \vdash_{\theta} E :_{\mathtt{P}} \Sigma \qquad \overline{\upsilon} = \mathrm{undet}(\theta\Sigma) - \mathrm{undet}(\theta\Gamma) \qquad \overline{\kappa_\alpha = \Omega}}{\Gamma \vdash^{\forall}_{\theta} E :_{\mathtt{P}} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma[\overline{\alpha}/\overline{\upsilon}]}\text{IGPURE}$$

Instantiation $\mathbb{S} \quad ::= \quad \upsilon \mid \alpha\,\cdot \mid \mathsf{bool} \mid [=\cdot] \mid [\cdot] \mid \{\cdot\} \mid \cdot \to \cdot$ $\qquad \boxed{\Gamma \vdash^{\mathbb{S}}_{\theta} E :_{\eta} \Xi}$

$$\frac{\Gamma \vdash_{\theta} E :_{\eta} \Xi}{\Gamma \vdash^{\mathbb{S}}_{\theta} E :_{\eta} \Xi}\text{INREFL} \qquad \frac{\Gamma \vdash^{\mathbb{S}}_{\theta} E :_{\eta} \exists\overline{\alpha}'.\forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma \qquad \overline{\upsilon} \text{ fresh} \qquad \overline{\Delta_\upsilon = \mathrm{dom}(\Gamma)}}{\Gamma \vdash^{\mathbb{S}}_{\theta} E :_{\eta} \exists\overline{\alpha}'.\Sigma[\overline{\upsilon}/\overline{\alpha}]}\text{INIMPL}$$

$$\frac{\Gamma \vdash_{\theta} E :_{\eta} \exists\overline{\alpha}.\upsilon \qquad \Gamma\ \theta \vdash_{\theta'} \upsilon \approx \mathbb{S}}{\Gamma \vdash^{\mathbb{S}}_{\theta'} E :_{\eta} \exists\overline{\alpha}.\upsilon}\text{INRES}$$

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

$$\frac{\upsilon'' \text{ fresh} \qquad \Delta_{\upsilon''} = \Delta_\upsilon \cap \Delta_{\upsilon'}}{\Gamma \vdash_{[\upsilon''/\upsilon,\upsilon''/\upsilon']} \upsilon \approx \upsilon'}\text{IRINFER} \qquad \frac{\kappa_\alpha = \overline{\Omega} \to \Omega \quad \alpha \in \Delta_\upsilon \quad \overline{\upsilon' \text{ fresh}} \quad \overline{\Delta_{\upsilon'} = \Delta_\upsilon}}{\Gamma \vdash_{[\alpha\,\overline{\upsilon}'/\upsilon]} \upsilon \approx \alpha\,\cdot}\text{IRPATH}$$

$$\frac{}{\Gamma \vdash_{[\mathsf{bool}/\upsilon]} \upsilon \approx \mathsf{bool}}\text{IRBOOL} \qquad \frac{\upsilon' \text{ fresh} \qquad \Delta_{\upsilon'} = \Delta_\upsilon}{\Gamma \vdash_{[[=\upsilon']/\upsilon]} \upsilon \approx [=\cdot]}\text{IRTYPE}$$

$$\frac{\upsilon' \text{ fresh} \qquad \Delta_{\upsilon'} = \Delta_\upsilon}{\Gamma \vdash_{[[\upsilon']/\upsilon]} \upsilon \approx [\cdot]}\text{IRWRAP} \qquad \frac{\upsilon_1, \upsilon_2 \text{ fresh} \qquad \Delta_{\upsilon_1} = \Delta_{\upsilon_2} = \Delta_\upsilon}{\Gamma \vdash_{[(\upsilon_1 \to_{\mathtt{I}} \upsilon_2)/\upsilon]} \upsilon \approx \cdot \to \cdot}\text{IRFUN}$$

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

$$\frac{}{\Gamma \vdash_{[]} \upsilon = \upsilon}\text{IUREFL} \qquad \frac{\Gamma \vdash_{\theta} \Xi' = \Xi}{\Gamma \vdash_{\theta} \Xi = \Xi'}\text{IUSYMM}$$

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

$$\frac{\Gamma \vdash_{\theta} \Xi = \Xi'}{\Gamma \vdash_{\theta'} [=\Xi] = [=\Xi']}\text{IUTYPE} \qquad \frac{\Gamma \vdash_{\theta} \Xi = \Xi'}{\Gamma \vdash_{\theta} [\Xi] = [\Xi']}\text{IUWRAP} \qquad \frac{\Gamma \vdash_{\theta} \overline{\sigma} = \overline{\sigma}'}{\Gamma \vdash_{\theta} \alpha\,\overline{\sigma} = \alpha\,\overline{\sigma}'}\text{IUPATH}$$

$$\frac{\Gamma \vdash_{\theta} \overline{\Sigma} = \overline{\Sigma}'}{\Gamma \vdash_{\theta} \{\overline{l{:}\Sigma}\} = \{\overline{l{:}\Sigma'}\}}\text{IUSTR} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\theta_1} \Sigma = \Sigma' \qquad \Gamma; \overline{\alpha}\ \theta_1 \vdash_{\theta_2} \Xi = \Xi'}{\Gamma \vdash_{\theta_2} \forall\overline{\alpha}.\Sigma \to_{\eta} \Xi = \forall\overline{\alpha}.\Sigma' \to_{\eta} \Xi'}\text{IUFUN}$$

$$\frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma = \Sigma'}{\Gamma \vdash_{\theta} \exists\overline{\alpha}.\Sigma = \exists\overline{\alpha}.\Sigma'}\text{IUABS} \qquad \frac{\Gamma; \overline{\alpha} \vdash_{\theta} \Sigma = \Sigma'}{\Gamma \vdash_{\theta} \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma = \forall\overline{\alpha}.\{\} \to_{\mathtt{A}} \Sigma'}\text{IUIMPL}$$

Fig. 16. Generalisation, Instantiation, Resolution, and Unification

The desugared form $\Gamma \vdash_\theta J$ with just an output substitution is equivalent to the degenerate case $\Gamma \,{}_{\mathsf{id}}\!\vdash_\theta J$, i.e., where the input substitution is the identity.

Other than the systematic threading of θ there are fairly few changes relative to the declarative rules. In Figures 13-15, all those other differences are highlighted in red for easier reference.

The basics of the inference algorithm are not that unusual, focussing around generalisation, instantiation, and resolution of inference variables.

Generalisation The declarative typing rules allow generalising the type of an expression at any point. However, there are only two situations under which generalisation actually needs to happen: (1) when a value is bound to a variable, because then it could potentially be used at multiple different types, and (2) when an expression's type needs to match an explicit annotation (directly or indirectly), because that annotation may demand polymorphism.

The first case corresponds to the classical let-rule of Damas/Milner, and occurs in the rule IBVAR: this invokes the auxiliary Generalisation judgement (Figure 16). For pure expressions, it finds all inference variables $\overline{\upsilon}$ in the expression's type that do not also occur in the context (we write $\mathrm{undet}(A)$ to denote the free inference variables of a semantic object A) and quantifies over them – in our case, by introducing an implicit function abstraction.

In fact, this rule already covers almost all cases of the second kind as well, because our kernel grammar is a sort of normal form which requires named variables in most places (Figure 1). In particular, this is the case for function arguments. Consider this example:

$(\mathbf{fun}\ (\mathsf{id} : \mathsf{'a} \Rightarrow \mathsf{a} \to \mathsf{a}) \Rightarrow \{\mathsf{x} = \mathsf{id}\ 3;\ \mathsf{y} = \mathsf{id}\ \mathsf{true}\})\ (\mathbf{fun}\ \mathsf{x} \Rightarrow \mathsf{x})$

Desugaring rewrites this application into an expression that has an explicit binding for the argument $(\mathbf{fun}\ \mathsf{x} \Rightarrow \mathsf{x})$, such that rule IBVAR kicks in. The same observation applies to other places where an expression's type must match an annotation: rules IESEAL, IEWRAP and IEUNWRAP. Similarly, it extends to sugar such as transparent type ascriptions $E{:}T$ as defined in Figure 2.

The only exception is conditionals, because their arms are not variables, yet the type annotation may require them to be polymorphic. Rule IEIF hence also has to invoke the Generalisation judgement.

Instantiation Instantiation is the inverse of generalisation. It takes place in a slightly different location than with Damas/Milner. Since implicit functions are first-class in 1ML, it is not just variables that can have "polymorphic" type. For example, the result of a projection $\mathsf{m.f}$ or of an application $\mathsf{g\ v}$ might have a polymorphic type, too, that potentially requires instantiation.

For that reason, the rules delay instantiation as much as possible, and only perform it in the typing rules of *elimination forms* (e.g. rules IEIF, IEDOT, IEAPP, but also ITPATH). They do so with the help of the auxiliary *Instantiation* judgement, shown in Figure 16.

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 $\upsilon_1 \to_\mathtt{I} \upsilon_2 \leq \beta \to_\mathtt{I} \beta$, resolving easily.

The instantiation judgement may seem non-deterministic because rule INIMPL may or may not be applied after INREFL returns an implicit function type. However, all invoking rules require an output type of a certain shape, so that it is always enforced by context that all implicit functions are actually eliminated, i.e., rule INIMPL is always applied exhaustively. Therefore, the judgement should be read as iterating over the inferred type until all implicit abstractions are eliminated. What is left, modulo possible existential quantifiers, is either an already determined, non-implicit type or an undetermined type signified by an inference variable. Because implicit types are large and inference variables only stand for small types, there is no ambiguity between the three cases.

In the case where the result is an inference variable, rule INRES has to resolve it. What to resolve it to depends on the context. Instantiation is always invoked for elimination forms, and each elimination expects a corresponding type of a specific shape. This information is passed to the Instantiation judgement in the form of an additional index $\mathbb{S}$ that specifies the "shape" in the form of its head constructor. The rule INRES uses this shape to invoke the resolution judgement, which fabricates a fresh type of the expected shape.

Resolution The judgement $\Gamma \vdash_\theta \upsilon \approx \mathbb{S}$ (Figure 16) is responsible for resolving an inference variable to a more specific small type. There is one main complication to this which necessitates turning this step into its own judgement – as opposed to simply using substitution. The reason is that, unlike good old ML, our 1ML allows small types to be intermixed with large ones.

For one, it is necessary to be able to infer small types from large ones via subtyping. For example, we might encounter the inequality

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

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, e.g.:

$$\upsilon \;\leq\; (\forall\alpha.[= \alpha] \to_\mathtt{P} [= \alpha]) \to_\mathtt{I} \mathsf{bool}$$

which can be solved with $\upsilon = ([= \sigma] \to_\mathtt{I} [= \sigma]) \to_\mathtt{I} \mathsf{bool}$.

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 Resolution judgement, which only resolves υ so far as to match the outermost constructor of Σ and inserts fresh inference variables for its subcomponents. For this purpose we define the auxiliary notation Σ° that extracts the

shape from a type. Its obvious definition is:

$$\begin{array}{rcl}
\upsilon^\circ & = & \upsilon \\
(\alpha\,\overline{\sigma})^\circ & = & \alpha\,\cdot \\
\mathsf{bool}^\circ & = & \mathsf{bool} \\
[=\Xi]^\circ & = & [=\cdot] \\
[\Xi]^\circ & = & [\cdot] \\
\{\overline{l{:}\Sigma}\}^\circ & = & \{\cdot\} \\
(\forall\overline{\alpha}_1.\Sigma_1 \to_\eta \exists\overline{\alpha}_2.\Sigma_2)^\circ & = & \cdot \to \cdot \\
(\exists\overline{\alpha}.\Sigma)^\circ & = & \Sigma^\circ
\end{array}$$

With that knowledge, resolution can create a fresh type of the necessary shape. After refining υ that way in the first premise of ISRESL or ISRESR, the rules "try again" in their second premise, and are guaranteed to proceed at least one step further. The occurs check encoded in the shorthand $\Gamma \vdash^!_\theta \upsilon \approx \mathbb{S}$ prevents this process from looping.

There intentionally is no definition of shape for implicit functions, because they must alway be eliminated first. We piggyback on this in the first side condition of the subtyping rule ISIMPLL, which enforces that a shape exists for the right-hand side, and thus, all implicit function abstractions on the right are eliminated (by rule ISIMPLR) before applying this rule.

Variable Scoping The second complication to 1ML inference is that an inference variable υ can be introduced within the scope of arbitrary abstract types (i.e., regular type variables α). 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 all uses of υ. The set is verified when resolving υ (see resolution rule IRPATH in particular). The set also is propagated to any other υ' that 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. For example, consider type-checking the following program:

$\mathsf{f} = \textsf{fun}\ \mathsf{x} \Rightarrow \textsf{fun}\ (\textsf{type}\ \mathsf{t}) \Rightarrow \textsf{fun}\ (\mathsf{g} : \mathsf{t} \to \mathsf{int}) \Rightarrow \mathsf{g}\ \mathsf{x}$

The body of this function is type-checked under some context $\Gamma, \mathsf{x}{:}\upsilon, \alpha, \mathsf{t}{:}[=\alpha], \mathsf{g}{:}(\alpha \to \alpha)$ where $\alpha \notin \Delta_\upsilon$, because υ was created before α was in scope. Because of that, resolving υ with α to make the application type-check will be rejected – which is correct, because otherwise f would be given a malformed type like $\alpha \to \forall\alpha.[=\alpha] \to (\alpha \to \mathsf{int}) \to \mathsf{int}$, where α escapes its scope.

Inference variables also have to be updated when regular type variables go out of scope. That is achieved by employing the following notation (note the semicolon!) in all rules that locally extend Γ with type variables:

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

The net effect is that all local α's from Γ' are stripped from all Δ-sets of inference variable that remain after executing $\Gamma, \Gamma' \vdash J$, thereby disallowing any later resolution with a type that would contain those variables. We omit θ in this notation when it is the identity.

Unification Finally, our algorithm also includes a separate, old-fashioned unification algorithm (Figure 16). However, unification is only needed by two rules: the rule ISPATH for paths and the rule ISWRAP for the wrapped types of the impredicative extension – because those are the only rule where subtyping does not recurse into subcomponents. In their declarative formulation (Figure 12), the respective rules simply require the types on both sides of the relation to be syntactically equivalent (modulo $\alpha\beta$-conversion, as noted). The presence of inference variables in the algorithmic rules makes that formulation insufficient, of course, and necessitates traversing the structure of the two types to unify possible inference variables embedded in them.

This algorithm is much stricter than (mutual) subtyping: there is no quantifier elimination or substitution, all quantifiers, as well as implicit function types, must match up one-to-one. That makes this algorithm trivially decidable even for arbitrary large types.

7.2 Incompleteness

Unfortunately, there are a couple of sources of incompleteness in this algorithm:

Width subtyping Subtyping constraints like $\upsilon \leq \{\overline{l{:}\sigma}\}$ do not determine the shape of the record type υ: the set of its labels can still vary over all possible subsets of $\bar{l}$. Consequently, the Resolution judgement in Figure 16 has no rule for structures – instead a structure type must be determined by the previous context.

This is, in fact, similar to Standard ML (Milner *et al.*, 1997), where record types cannot be inferred either, and require type annotations. 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 in this article and leave out this possible improvement.

The real solution of course would be to incorporate not just row inference but *row polymorphism* (Rémy, 1989), so that width subtyping on structures can be recast as universal and existential quantification over row variables. We leave investigating such an extension for future work. We only note here that it would not solve the problem completely, because of **include**: that extends the current scope with bindings that are determined by a structure type; to handle the case where the structure type is row-polymorphic we somehow would have to allow *environments* Γ to be row-polymorphic, too, or restrict **include** in a suitable manner.

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 the following example, adapted from Example (c) in Dreyer & Blume (2007):

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

Assume id : 'a ⇒ a → a. Because id is impure, the definition of f is impure, and its type cannot be generalised; consequently, G is impure, too, and hence generative. The algorithm will infer G's type as

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

with $\beta \notin \Delta_\upsilon$ – because β goes out of scope the moment we bind it with a local quantifier, while all υ are treated as "global". It 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 it's too late: the solution $\upsilon = \beta$, which would make y well-typed, is already precluded. When typing C, instantiating α with β is not possible either, because β can only come into scope again (by opening the existential) *after* having applied an argument for α already.

Although not well-known, this very problem is already present in conventional ML, as Dreyer & Blume (2007) pointed out: all existing type inference implementations for ML 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 (as for G above). 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 – such as Example (b):

```
F (t : type) = {f = id id};
A = F int; B = F bool;
x = A.f 10; y = B.f "dude";
```

This works because F is polymorphic in 1ML (with type $\forall\alpha.\{\} \to_{\mathtt{A}} \forall\beta.[= \beta] \to_{\mathtt{P}} \{\mathsf{f} : \alpha \to_{\mathtt{I}} \alpha\}$), but that is not enough to handle other examples, like G above.

In other words, our algorithm can infer the types of some programs that regular ML inference unexpectedly can't handle. At the same time, the liberal mixture of modules and core programming in 1ML introduces more of these cases, and some remain incomplete.

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 include only impure functions: the resolution rule IRFUN can always pick `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 (Talpin & Jouvelot, 1992): if effects could be quantified over, they could also be inferred in a complete manner. We explored this solution in a separate paper that extends 1ML with effect polymorphism (Rossberg, 2016), which – interestingly – induces a novel notion of *generativity polymorphism* for functors.

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 valid 1ML without adding a single type annotation.

7.3 Correctness

Algorithmic Soundness The inference algorithm may not be complete, but at least it is sound. That is, we can show the following result:

Theorem 7.1 (*Soundness of 1ML Inference*)
Let $\overline{\upsilon}, \Gamma$ be a well-formed F_ω environment.

1. If $\Gamma \vdash_\theta T/D \rightsquigarrow \Xi$, then $\overline{\upsilon}', \theta\Gamma \vdash T/D \rightsquigarrow \theta\Xi$ for some $\overline{\upsilon}'$.
2. If $\Gamma \vdash_\theta E/B :_\eta \Xi \rightsquigarrow e$, then $\overline{\upsilon}', \theta\Gamma \vdash E/B :_\eta \theta\Xi \rightsquigarrow \theta e$ for some $\overline{\upsilon}'$.
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 $\overline{\upsilon}', \theta\Gamma \vdash \theta\Xi' \leq_{\overline{\pi}} \theta\Xi \rightsquigarrow \theta\delta; \theta f$ for some $\overline{\upsilon}'$.

In order to prove this result, 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 $\overline{\upsilon}$ relative to the regular type variables in their Δ-sets.

To that end, we define a little auxiliary judgement $\overline{\upsilon} \vdash \Gamma \rightsquigarrow \Gamma'$ that intersperses Γ with $\overline{\upsilon}$ such that for each υ the type variables Δ_υ are in scope:

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

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

This assumes that $\mathrm{dom}(\Gamma) \not\cap \overline{\upsilon}$ 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 J, we define

$$\overline{\upsilon}; \Gamma \vdash J \quad :\Leftrightarrow \quad \overline{\upsilon} \vdash \Gamma \rightsquigarrow \Gamma' \wedge \Gamma' \vdash J \text{ (for some } \Gamma')$$

Furthermore, define well-formedness of resolution substitutions:

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

Well-typed substitutions can be composed:

Lemma 7.2 (*Composition of Substitutions*)
If $\overline{\upsilon}';\Gamma' \vdash \theta : \overline{\upsilon};\Gamma$ and $\overline{\upsilon}'';\Gamma'' \vdash \theta' : \overline{\upsilon}';\Gamma'$ and $\mathrm{dom}(\theta) \not\frown \overline{\upsilon}''$, then $\overline{\upsilon}'';\Gamma'' \vdash \theta' \circ \theta : \overline{\upsilon};\Gamma$.

The refined soundness theorem in its full glory is then stated as follows:

Theorem 7.3 (*Soundness of 1ML Inference Refined*)
Let $\mathrm{dom}(\Gamma) \not\frown \overline{\upsilon}$ and $\overline{\upsilon} \vdash \Gamma \rightsquigarrow \Gamma'$. Let J range over the 1ML inference judgements.

1. $\Gamma' \vdash \square$ and $\overline{\upsilon} \subseteq \mathrm{dom}(\Gamma')$.
2. If $\theta\Gamma \vdash_{\theta'} {}^{\theta} J$, then $\theta\Gamma \vdash_{\theta'} \theta J$.
3. If $\Gamma \vdash_{\theta} J$, then $\overline{\upsilon}';\theta\Gamma \vdash \theta : \overline{\upsilon};\Gamma$ with $\overline{\upsilon}' - \overline{\upsilon}$ fresh.
4. If $\Gamma \,{}_{\theta}\!\vdash_{\theta'} J$ 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 $\Gamma_1;\Gamma_2 \,{}_{\theta}\!\vdash_{\theta'} J$ and $\overline{\upsilon}';\theta\Gamma \vdash \theta : \overline{\upsilon};\Gamma$ with $\Gamma = \Gamma_1,\Gamma_2$,
   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 $\Gamma \vdash_{\theta} T/D \rightsquigarrow \Xi$, then $\overline{\upsilon}';\theta\Gamma \vdash T/D \rightsquigarrow \theta\Xi$.
7. If $\Gamma \vdash_{\theta} E/B :_{\eta} \Xi$, then $\overline{\upsilon}';\theta\Gamma \vdash E/B :_{\eta} \theta\Xi \rightsquigarrow \theta e$.
8. If $\Gamma \vdash_{\theta} \Xi \leq_{\overline{\pi}} \Xi' \rightsquigarrow \delta$, and $\overline{\upsilon};\Gamma \vdash \Xi : \Omega$ and $\overline{\upsilon};\Gamma \vdash \Xi' : \Omega$,
   then $\overline{\upsilon}';\theta\Gamma \vdash \theta\Xi \leq_{\overline{\pi}} \theta\Xi' \rightsquigarrow \theta\delta;\theta f$.
9. If $\Gamma \vdash^{\forall}_{\theta} E :_{\eta} \Xi$, then $\overline{\upsilon}';\theta\Gamma \vdash E :_{\eta} \theta\Xi \rightsquigarrow \theta e$.
10. If $\Gamma \vdash^{\mathbb{S}}_{\theta} E :_{\eta} \Xi$, then $\overline{\upsilon}';\theta\Gamma \vdash E :_{\eta} \theta\Xi \rightsquigarrow \theta e$ and $\Xi^{\circ} = \mathbb{S}$.
11. If $\Gamma \vdash_{\theta} \upsilon \approx \mathbb{S}$, then $(\theta\upsilon)^{\circ} = \mathbb{S}$.
12. If $\Gamma \vdash_{\theta} \Xi = \Xi'$, and $\overline{\upsilon};\Gamma \vdash \Xi : \Omega$ and $\overline{\upsilon};\Gamma \vdash \Xi' : \Omega$, then $\theta\Xi = \theta\Xi'$.

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

Termination We can also show that – despite its incompleteness – the inference algorithm at least does not go astray by diverging:

Theorem 7.4 (*Termination of 1ML Inference*)
All 1ML type inference judgements terminate.

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, only subtyping remains as the problem child.

We can prove its termination by extending the prove from Section 4.3. First, we trivially extend the weight function to handle inference variables:

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

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 as well:

Lemma 7.5 (*Weight Reduction under Small Resolution*)
1. $Q[\![\theta\Xi]\!] = Q[\![\Xi]\!]$.
2. $W[\![\theta\Xi]\!] < W[\![\Xi]\!] + \langle 1, 0\rangle$.
3. $W[\![\theta\Xi]\!] \leq W[\![\Xi]\!] + \langle |\mathrm{dom}(\theta)|, 0\rangle$.

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

Lemma 7.6 (*Resolution Progress in Subtyping Inference*)
Let $\Gamma \vdash_\theta J$ an inference derivation and Y the set of fresh inference variables it generates.

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

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

Lemma 7.7 (*Termination of Subtyping Inference*)
Let $\overline{\upsilon} \vdash \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \vdash \Xi' : \Omega$ and $\Gamma', \overline{\alpha} \vdash \Xi : \Omega$ and $\overline{\pi} = \overline{\alpha\ \overline{\alpha'}}$. Then $\Gamma \vdash_\theta \Xi' \leq_{\overline{\pi}} \Xi \rightsquigarrow \delta; f$ 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 a simple case analysis of the Σ in these rules that the temporary variables are resolved immediately and weight will decrease again in the recursive invocation of the subtyping judgement.

8 Related Work

Packaged Modules The first concrete proposal for extending ML with packaged modules was by Russo (2000), and is implemented in Moscow ML. Later work on type systems for modules routinely included them (Dreyer *et al.*, 2003; Dreyer, 2005; Rossberg & Dreyer, 2013; Rossberg *et al.*, 2014), and variations have been implemented in other ML dialects, such as Alice ML (Rossberg, 2006) and OCaml (Garrigue & Frisch, 2010).

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, which forbid 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 (Harper & Mitchell, 1993). It is a dependent type theory modeling modules as terms of Martin-Löf-style Σ and Π types, closely following MacQueen's original ideas (MacQueen, 1986). 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. XML allows both $U_1 : U_2$ and $U_1 \subseteq U_2$, which is similar in 1ML. However, XML lacks any account of either sealing or translucency, which makes it fall short as a foundation for contemporary ML modules.

That gap was closed by Harper & Lillibridge's calculus of *translucent sums* (Harper & Lillibridge, 1994; Lillibridge, 1997), 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 were later deconstructed into the notion of *singleton types* (Stone & Harper, 2006); they formed the foundation of Dreyer et al.'s type theory for higher-order modules (Dreyer *et al.*, 2003). However, to avoid undecidability, this system went back to second-class modules (though it provides packaged modules).

One central 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.* (1990) investigate phase separation for the XML calculus. The beauty of the F-ing approach underlying 1ML is that it enforces phase separation by construction, since it avoids dependent types altogether.

Shields & Peyton Jones (Shields & Peyton Jones, 2002) proposed first-class modules for Haskell. However, these do not provide modular compositionality in the sense discussed in Section 5.3, because they cannot contain type components (only module *types* can), there is no sealing, and type abstraction is manual through raw existential types.

Applicative Functors Leroy proposed applicative semantics for functors (Leroy, 1995), as implemented in OCaml. Russo later combined both generative and applicative functors in one language (Russo, 2003) and implemented them in Moscow ML; others followed (Shao, 1999; Dreyer *et al.*, 2003; Dreyer, 2005; Rossberg *et al.*, 2014).

A system like Leroy's, where *all* functors are applicative, would be incompatible with first-class modules even in a pure language, 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 all generative functors into applicative ones. Like Dreyer (Dreyer, 2005) or F-ing modules (Rossberg *et al.*, 2014), 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 of the conference version of this article (Rossberg, 2015) sketches a similar extension to 1ML.

In the basic language introduced in the present article, an applicative functor can only be created by sealing a fully transparent functor with pure function type, very much like in Shao's system (Shao, 1999).

Type Inference There has been little work that has considered type inference for modules. Russo examined the interplay between core-level inference and modules (Russo, 2003), elegantly dealing with variable scoping via unification under a mixed prefix (which might also form a more elegant basis for 1ML inference than our Δ-sets do).

Dreyer & Blume (2007) investigated how functors interfere with the value restriction. They show that – contrary to popular belief – Damas/Milner type inference (Damas & Milner, 1982) with a value restriction (Wright, 1995), as employed in core ML, is no longer complete when combined with ML-style functors. They propose a way to fix this problem

by allowing functors to become (implicitly) polymorphic as well. However, their solution requires a *destination-passing style* semantics for type abstraction – inspired by work on recursive modules (Dreyer, 2007) – that is well beyond the realms of what System F_ω can express. Consequently, although 1ML can implicitly generalise functors as well and thus is able to handle some of the examples from Dreyer & Blume's paper, it cannot deal with the more interesting ones.

At the same time, there have been ambitious extensions of ML-style type inference with higher-rank or impredicative types (Garrigue & Rémy, 1999; Le Botlan & Rémy, 2003; Vytiniotis *et al.*, 2008; Russo & Vytiniotis, 2009). Unlike those systems, 1ML never tries to infer a polymorphic type annotation: all guessed types are monomorphic; polymorphic parameters always require annotations. The impredicative extension from the Section 5.1 is similar to Garrigue & Rémy's semi-explicit polymorphism (Garrigue & Rémy, 1999), but also allows unboxed polytypes.

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 always remain implicit and are effectively inferred as part of the elaboration process.

9 Conclusion and Future Work

1ML is a coherent, unified redesign of ML in a single language that subsumes both the ML core and the ML module language. It is no more and no less powerful than System F_ω (the higher-order polymorphic lambda calculus), but unlike the latter, supports type inference and modular composition. At the same time, modelling 1ML's type structure in terms of mere System F_ω types trivially establishes *phase separation* between type checking and execution, a property that requires extra effort in approaches based on dependently typed calculi (Harper *et al.*, 1990).

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

Mechanisation Elaboration of $1ML_{ex}$is fairly straightforward and merely a small modification to F-in modules, which has been mechanised in Coq (Rossberg *et al.*, 2014). It should be relatively easy to extend this work to 1ML. The full language and its type inference is far more intricate, however. While we have done the essential parts of the proofs on paper, their details are tedious enough to be almost guaranteed to be wrong. It would be very valuable to mechanise this part of the formalisation and machine-verify the stated properties.

Implementation We have implemented a simple prototype interpreter for 1ML (mpi-sws.org/~rossberg/1ml/) that allows running smallish examples. But it would be great to gather more experience with a "real" implementation, especially regarding the viability and user-friendliness of 1ML type inference.

Usability Pragmatics While 1ML achieves a nice unification of core and module language, it isn't necessarily clear whether this is the most desirable design in practice. A

clearer syntactic separation of module-level features might be a valuable aid to programmers. It might also help to tame more obscure cases of subtyping and type inference semantics and avoid confusing the programmer with the full generality of the system in all places.

Applicative Functors To keep the present paper simple, it adopts a rather basic notion of applicative functor. It is not hard to extend it to *pure sealing* in the style of F-ing modules – as mentioned earlier, a sketch can be found in the Technical Appendix of the conference version of this article (Rossberg, 2015). A greater challenge would be to go a step further and make it properly *abstraction-safe*, by tracking value identities as described in Section 8 of Rossberg *et al.* (2014).

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* (Wadler & Blott, 1989) or Coq-style *canonical structures* (Mahboubi & Tassi, 2013). In particular, generalised implicit functions could be a semantic foundation for modular implicits (White *et al.*, 2014).

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 with some of the more advanced approaches to inference described in literature, some of which we mentioned in Section 8. However, it is non-obvious how well these methods will interact with the implicit quantifier pushing in 1ML's elaboration.

Row Polymorphism Replacing subtyping with a more potent form of polymorphism might increase expressiveness and lead to better inference: in particular, *row polymorphism* (Rémy, 1989) could express width subtyping without compromising completeness of inference. The main issue with such an extension probably is its effect on the small/large type universe distinction, since many current uses of subtyping on small types will require large types once they involve quantification of row variables.

Effect Polymorphism Similarly, the utility of 1MLs effect system and its pure function types could be greatly increased if higher-order functions could vary their effect by employing *effect polymorphism* (Talpin & Jouvelot, 1992). We have recently explored this direction in a separate paper (Rossberg, 2016). Its most interesting implication is that it enables "functors" to be polymorphic over their generativity, i.e., whether they behave applicatively or generatively. Incidentally, this ability finally allows expressing the motivating examples of MacQueen's "truly higher-order modules" (MacQueen & Tofte, 1994; Kuan & MacQueen, 2009), which previously have eluded a nice type-theoretic explanation, and were not supported by any other module type system.

Recursive Modules Recursive modules would add a whole new dimension of expressiveness – and complexity! – to 1ML. With MixML (Rossberg & Dreyer, 2013) 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 than 1ML has. The result of such a cross-breed would presumably share some similarities with Scala (Odersky & Zenger, 2005), but with a more expressive semantics for type equivalences.

Dependent Types Finally, 1ML goes to great lengths 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 (a.k.a. existential types). Incorporating them, along with the quantifier pushing of our elaboration, into a dependent type system poses a thrilling challenge.

References

Barendregt, Henk. (1992). Lambda calculi with types. *Chap. 2, pages 117–309 of:* Abramsky, Samson, Gabbay, Dov, & Maibaum, T.S.E. (eds), *Handbook of logic in computer science*, vol. 2. Oxford University Press.

Biswas, Sandip K. (1995). Higher-order functors with transparent signatures. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Damas, Luis, & Milner, Robin. (1982). Principal type-schemes for functional programs. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Dreyer, Derek. (2005). *Understanding and Evolving the ML Module System*. Ph.D. thesis, Carnegie Mellon University.

Dreyer, Derek. (2007). Recursive type generativity. *Journal of Functional Programming (JFP)*, **17**(4&5), 433–471.

Dreyer, Derek, & Blume, Matthias. (2007). Principal type schemes for modular programs. *European Symposium on Programming (ESOP)*.

Dreyer, Derek, & Rossberg, Andreas. (2008). Mixin' up the ML module system. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Dreyer, Derek, Crary, Karl, & Harper, Robert. (2003). A type system for higher-order modules. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Garrigue, Jacques, & Frisch, Alain. (2010). First-class modules and composable signatures in Objective Caml 3.12. *ACM SIGPLAN Workshop on ML*.

Garrigue, Jacques, & Rémy, Didier. (1999). Semi-explicit first-class polymorphism for ML. *Information and computation*, **155**(1-2).

Harper, Robert, & Lillibridge, Mark. (1994). A type-theoretic approach to higher-order modules with sharing. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Harper, Robert, & Mitchell, John C. (1993). On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **15**(2), 211–252.

Harper, Robert, & Pierce, Benjamin C. (2005). Design considerations for ML-style module systems. *Chap. 8 of:* Pierce, Benjamin C. (ed), *Advanced Topics in Types and Programming Languages*. MIT Press.

Harper, Robert, & Stone, Chris. (2000). A type-theoretic interpretation of Standard ML. *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press.

Harper, Robert, Mitchell, John C., & Moggi, Eugenio. (1990). Higher-order modules and the phase distinction. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Kuan, George, & MacQueen, David. (2009). Engineering higher-order modules in SML/NJ. *International Symposium on the Implementation and Application of Functional Languages (IFL)*. LNCS, vol. 6041.

Le Botlan, Didier, & Rémy, Didier. (2003). MLF: Raising ML to the power of System F. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Leroy, Xavier. (1994). Manifest types, modules, and separate compilation. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Leroy, Xavier. (1995). Applicative functors and fully transparent higher-order modules. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Lillibridge, Mark. (1997). *Translucent sums: A foundation for higher-order module systems*. Ph.D. thesis, Carnegie Mellon University.

MacQueen, David B. (1986). Using dependent types to express modular structure. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

MacQueen, David B., & Tofte, Mads. (1994). A semantics for higher-order functors. *European Symposium on Programming (ESOP)*.

Mahboubi, Assia, & Tassi, Enrico. (2013). Canonical structures for the working Coq user. *Pages 19–34 of:* Blazy, Sandrine, Paulin-Mohring, Christine, & Pichardie, David (eds), *Interactive theorem proving*. LNCS, no. 7998. Springer-Verlag.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of functional programming (jfp)*, **18**(1), 1–13.

Milner, Robin. (1978). A theory of type polymorphism in programming languages. *Journal of computer and system sciences*, **17**, 348–375.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised)*. MIT Press.

Mitchell, John C., & Plotkin, Gordon D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **10**(3), 470–502.

Odersky, Martin, & Zenger, Matthias. (2005). Scalable component abstractions. *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press.

Rémy, Didier. (1989). Records and variants as a natural extension of ML. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Rossberg, Andreas. (1999). *Undecidability of OCaml type checking*. Posting to Caml mailing list, 13 July. `https://sympa.inria.fr/sympa/arc/caml-list/1999-07/msg00027.html`.

Rossberg, Andreas. (2006). The Missing Link – Dynamic components for ML. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Rossberg, Andreas. (2015). *1ML – Core and modules united (Technical Appendix)*. mpi-sws.org/~rossberg/1ml/.

Rossberg, Andreas. (2016). 1ML with special effects. *Pages 336–355 of: A List of Successes That Can Change the World – WadlerFest 2016*. LNCS, no. 9600. Springer-Verlag.

Rossberg, Andreas, & Dreyer, Derek. (2013). Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **35**(1).

Rossberg, Andreas, Russo, Claudio, & Dreyer, Derek. (2014). F-ing modules. *Journal of Functional Programming (JFP)*, **24**(5), 529–607.

Russo, Claudio, & Vytiniotis, Dimitris. (2009). QML: Explicit first-class polymorphism for ML. *ACM SIGPLAN Workshop on ML*.

Russo, Claudio V. (1999). Non-dependent types for Standard ML modules. *International Conference on Principles and Practice of Declarative Programming (PPDP)*.

Russo, Claudio V. (2000). First-class structures for Standard ML. *Nordic Journal of Computing*, **7**(4), 348–374.

Russo, Claudio V. (2003). Types for Modules. *Electronic Notes in Theoretical Computer Science (ENTCS)*, **60**.

Shao, Zhong. (1999). Transparent modules with fully syntactic signatures. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Shields, Mark, & Peyton Jones, Simon. (2002). First-class modules for Haskell. *Pages 28–40 of: International Workshop on Foundations of Object-Oriented Languages (FOOL)*.

Stone, Christopher A., & Harper, Robert. (2006). Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)*, **7**(4), 676–722.

Talpin, Jean-Pierre, & Jouvelot, Pierre. (1992). Polymorphic type, region and effect inference. *Journal of Functional Programming (JFP)*, **2**(3), 245271.

Vytiniotis, Dimitrios, Weirich, Stephanie, & Peyton Jones, Simon. (2008). FPH: First-class polymorphism for Haskell. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

White, Leo, Bour, Frédéric, & Yallop, Jeremy. (2014). Modular implicits. *Pages 22–63 of:* Kiselyov, Oleg, & Garrigue, Jacques (eds), *ACM SIGPLAN ML Family / OCaml Workshops*. EPTCS, no. 198.

Wright, Andrew. (1995). Simple imperative polymorphism. *LISP and Symbolic Computation (LASC)*, **8**(4), 343–356.