1ML — core and modules as one
or: F-ing first-class modules

Andreas Rossberg
rossberg@mpi-sws.org

1. Introduction

ML is two languages in one: there is the core, with types and expressions, and there are modules, with signatures, structures and functors. ML modules form a separate, higher-order functional language on top of the ML core language. Both practical and theoretical reasons led to this stratification (as well as historical ones). Yet, it creates substantial duplication in syntax and semantics, and it limits the expressiveness of the language; for example, selecting a module for a given signature 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 [9, 2, 8, 3]. However, that approach arguably is a kludge. It remedies the expressiveness issue only to some extent, does not alleviate the redundancy in the language, and is even more syntactically heavyweight than using second-class modules alone.

In this presentation, we propose a redesign of ML in which modules are truly first-class values. We call it 1ML, because it combines the core and the module layer into one unified language (and also because it is a "*1st-class module language*"). In 1ML, functions, functors, and even type constructors are one and the same construct. Likewise, no distinction is made between structures, records, or tuples, including tuples over types. Or viewed the other way round, everything is just ("a mode of use of") modules.

Yet, 1ML does not required dependent types, and its type structure is expressible as sugar for plain System F_ω, in a minor variation of our F-ing modules approach [8]. How is that possible? Hasn't the module literature taught us that first-class modules cause the disease of undecidable type-checking [5]? And wouldn't we loose any hope for ever providing the super-convenient Hindley/Milner-style type inference feature that we love so much about ML?

As it turns out, neither has to be true. We first show how decidable type-checking can be maintained in an explicitly typed version of 1ML. All that is necessary is a surgical restriction on the definition of signature matching, which amounts to a simple universe distinction into small and large types. We then introduce a relatively simple extension to support Hindley/Milner-style type inference. Reusing the aforementioned universe distinction, only small types can be inferred. This inference is necessarily incomplete, but, we argue, no more so than in existing practical MLs.

A prototype of a toy 1ML interpreter is available on request.

2. 1ML with Explicit Types

We start out by introducing a version of 1ML that is explicitly typed – let's call it $1\mathrm{ML}_{\mathrm{ex}}$. The kernel syntax of this language is given in Figure 1. It mostly consists of fairly conventional functional language constructs: as a representative for a base type we have Booleans, there are records, which consist of a sequence of bindings, and of course, functions. These forms are mirrored on the type level as one would expect, except that for functions we distinguish two forms of arrow type, pure functions ($\Rightarrow$) and impure ones ($\rightarrow$), with purity being inferred for terms. Like with F-ing modules, most other interesting constructs are definable as syntactic sugar [8].

What makes this language able to express modules is the ability to embed types in a first-class manner: the expression **type** T denotes the type T *as a value*. Such an expression has type **type**, and thereby can be abstracted over. For example,

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

defines a polymorphic identity function id, very similar to how it would be written in System F_ω (or in dependent type theories); a "type constructor" pair, which, when applied to a type, yields another type as a first-class value; this type can be implicitly projected from the value using the *path* form E as a type, as demonstrated with the type pair a for parameter p of the function second. We can easily define a bit of syntactic sugar to make function and type definitions look more like traditional ML (taking a function parameter x with no annotation to be shorthand for (x : **type**)):

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

More interestingly, our language can also express real modules. Here is a function (i.e., a "functor") that defines a simple set type:

```
type EQ =
{
  type t;  (* short for t : type *)
  eq : t → t → bool
};
Set (Elem : EQ) =
{
  type elem = Elem.t;
  type set = elem → bool;
  empty = fun (x : elem) ⇒ false;
  mem (x : elem) (s : set) = s x;
  add (x : elem) (s : set) =
    if mem x s then s
    else (fun (y : elem) ⇒ Elem.eq x y or mem y s) : set
}
```

The record type EQ amounts to a signature, since it contains a nested abstract type component t. Further, note how the **if**-construct requires a type annotation in $1\mathrm{ML}_{\mathrm{ex}}$, so that the type system does not need to find a least upper bound for the types of both branches (which is not always unique for module types).

Following the F-ing modules elaboration semantics, we define the 1ML type system using *semantic* types, a subset of System F_ω types with the following shape:

$$\Sigma ::= \alpha\ \overline{\sigma} \mid \mathsf{bool} \mid [= \Sigma] \mid \{\overline{X{:}\Sigma}\} \mid \forall\overline{\alpha}_1.\Sigma_1 \rightarrow \exists\overline{\alpha}_2.\Sigma_2$$

where $[= \Sigma]$ is notation for a known first-class type (and defined as syntactic sugar over F_ω types). Unlike in the original F-ing modules

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

Figure 1. $\mathrm{1ML_{ex}}$syntax

semantics, there is no distinction between values and modules, hence type variables can now represent abstract "signatures".

But we have to be careful – losing the *syntactic* distinction between types and signatures is where we would run into a decidability issue akin to the one identified by Harper & Lillibridge [5]. We avoid the type:type sort of situation underlying this problem by *semantically* restricting what is substitutable for a type variable. To that end, we define the subset of *small* semantic types:

$$\sigma ::= \alpha\ \overline{\sigma} \mid \mathsf{bool} \mid [= \sigma] \mid \{\overline{X{:}\sigma}\} \mid \sigma_1 \rightarrow_{\mathrm{I}} \sigma_2$$

It contains all Σ that are free of quantifiers. This distinction is reminiscent of the universes of XML [6]. Given this definition, the only tweak necessary to keep the F-ing rules decidable is to restrict the one rule deriving substitutions: signature matching, such that abstract types can only be matched by small types.

So far so good. However, with this alone, we would limit ourselves to predicative polymorphism. That is, we would not be able to type-abstract over general "signatures" (i.e., types with abstract type components), or other forms of polymorphic types. To address this limitation, we add *package* types to the system, that allow wrapping up (values of) large types as (values of) small types:

$$
\begin{array}{llcl}
\text{(types)} & T & ::= & \ldots \mid \textbf{pack } T \\
\text{(expressions)} & E & ::= & \ldots \mid \textbf{pack } E \mid \textbf{unpack } E{:}T \\
\end{array}
\qquad
\Sigma ::= \ldots \mid [\Sigma] \qquad \sigma ::= \ldots \mid [\Sigma]
$$

This looks very similar to existing modules-as-first-class-values, and in ways it is. But in 1ML, these packages play a different, more fringe role: packaging is *only* necessary to *type*-abstract over a signature, *not* to use a module with a *known* signature in a first-class manner. In that sense, 1ML packages are more reminiscent of various approaches to "boxed" polymorphism [4, 11, 10].

3. 1ML with Type Inference

To support type inference, we add two pieces of type syntax:

$$T ::= \ldots \mid \_ \mid {'X} \Rightarrow E$$

An underbar is a type expression whose denotation is to be inferred by the type system; in addition, as MLish syntax sugar, we redefine omitted type annotations to be ": _" (except in **type** bindings).

The second production is a new type of *implicit* functions. It abstracts over values of type **type**, and application is always implicit. Moreover, such types can be introduced implicitly at (pure) bindings, by generalising over free variables of type **type**. In other words, we are reintroducing ML-style implicit polymorphism.

With additional syntax sugar, we can now write, for example:

```
type MAP =
{
  type key;
  type map a;  (* map : (a : type) ⇒ type *)
  empty 'a : 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 x z then Some y else lookup a y m
}
```

The crucial restriction we impose in this system is that an underbar can only denote a *small* type. The trick is, then, that for small types, subtyping – i.e., "signature matching" – (almost) degenerates to type equivalence. We can hence overlay the usual matching algorithm for large types with type unification on undetermined small types. The overall type system, when specialised to small types, will then largely resemble Hindley/Milner. Yet, it seamlessly extends to (explicit) large types, and all the quantifier introduction and elimination machinery involved in the F-ing semantics.

There are only three rules in the system for which small type inference will be incomplete: width subtyping on records, the **include** form, and one rather obscure problem with the value restriction and principality of "functors" [1]. This is, of course, unfortunate. However, interestingly, all of these limitations already exist in a language like Standard ML (or OCaml): record typing is not principal and generally requires type annotations, an equivalent to **include** does not even exist for records, and the principality issue with functors is exactly the same. In fact, we conjecture that the 1ML type inference algorithm could actually type all programs that conventional SML implementations can handle (under a 1-to-1 mapping of the syntax, with no type annotations added).

Nevertheless, we would like to extend the type system with row polymorphism [7] to overcome at least the first of the aforementioned sources of incompleteness (and perhaps the second). It might also be worth investigating how 1ML could be integrated with various approaches to type inference for first-class polymorphism, in order to allow omitting annotations in cases where traditional ML modules would not. Finally, extending implicit functions to richer domains could provide some of the convenience of type classes.

References

[1] Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. In *ESOP*, 2007.

[2] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL*, 2003.

[3] Jacques Garrigue and Alain Frisch. First-class modules and composable signatures in Objective Caml 3.12. In *ML*, 2010.

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

[5] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

[6] Robert Harper and John C. Mitchell. On the type structure of Standard ML. In *TOPLAS*, volume 15(2), 1993.

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

[8] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. In *TLDI*, 2010. Extended version: mpi-sws.org/~rossberg/f-ing/.

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

[10] Claudio Russo and Dimitrios Vytiniotis. QML: explicit first-class polymorphism for ML. In *ML*, 2009.

[11] Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. FPH: First-class polymorphism for Haskell. In *ICFP*, 2008.