1ML with Special Effects

F-ing Generativity Polymorphism

Andreas Rossberg

Google, Germany
`rossberg@mpi-sws.org`

Abstract. We take another look at 1ML, a language in the ML tradition, but with core and modules merged into one unified language, rendering all modules first-class values. 1ML already comes with a simple form of effect system that distinguishes pure from impure computations. Now we enrich it with effect polymorphism: by introducing effect declarations and, more interestingly, abstract effect specifications, effects can be parameterised over, and treated as abstract or concrete in the type system, very much like types themselves. Because type generativity in 1ML is controlled by (im)purity effects, this yields a somewhat exotic novel notion of *generativity polymorphism* – that is, a given functor can be asked to behave as either "generative" or "applicative". And this time, we even get to define an interesting (poly)monad for that!

1 Introduction

In a recent paper [13] we introduced *1ML*, a reboot of ML where modules are first-class values. In this language, there no longer is any distinction between core and modules, records or structures, functions or functors. Morally, every expression denotes a module.

One peculiar feature of 1ML is its distinction between *pure* and *impure* functions and computations, through a – very simple – form of effect system. Although this is primarily motivated by the semantics of functors (as we will see below), it obviously is an interesting distinction to make for "core-like" functions as well. Effect systems [3, 9, 18] are regularly proposed as a complement to functional type systems. They serve a similar role as monads [12, 20], which have been successfully incorporated into languages like Haskell, but with a little extra flexibility – Wadler & Thiemann showed that both are essentially equivalent [21].

Telling pure from impure statically allows much better control over side effects. Like any other form of type discipline, it allows more accurate specification of interfaces, and prevents certain classes of errors. For example, a module can specify that some of its interface functions are free of side effects. Or not. Higher-order functions can shield against unwanted side effects when invoking an argument function (or "callback", as laymen say nowadays). All useful capabilities, and in principle it is possible to put them to good use in 1ML already.

What makes the use of effects cumbersome in 1ML, however, is the lack of support for *effect polymorphism*. In a higher-order language that is a real bummer, because the purity of a higher-order function will typically depend on the purity of its argument. For example, consider

```
map f xs = if null xs then [] else f (head xs) :: map f (tail xs)
```

Is this a pure or impure function? It depends on f. We can imagine (at least) two different signatures for map:

```
map : ∀ a b. (a →_P b) →_P list a →_P list b
map : ∀ a b. (a →_I b) →_P list a →_I list b
```

Here, we use "$\rightarrow_P$" to denote pure function types, and "$\rightarrow_I$" for impure ones. Unfortunately however, map may only have one of these types in 1ML. If we needed both, we would need to write different functions for the different signatures. (Notably, 1ML's pure arrows are subtypes of impure ones, but such subtype polymorphism is insufficient to handle this case.)

Or we introduce the ability to abstract over (im)purity via polymorphism. Both map and its argument could be polymorphic in their (joint) effect:

```
map : ∀ a b e. (a →_e b) →_P list a →_e list b
```

In this signature, we assume that e is a (universally quantified) *effect variable.* By instantiating it with either P or I, we can use the same implementation of map on either a pure or an impure argument, and get a corresponding pure or impure computation in return.

Okay, you might think, that looks like a fairly trivial and standard extension. Others proposed this long ago [9]. Well, what makes it interesting in 1ML is that the (im)purity effect is intimately tied to the notion of *type generativity.*

Functions that return abstract types – "functors" in ML speak – are interesting beasts. There are two traditional schools of semantics: "*generative*" functors (the SML way) and "*applicative*" functors (the OCaml way). In 1ML, both notions coexist, and depend on whether a functor is pure or impure. Consider:

```
F : (a : type) →_P type
```

A pure functor like this is "applicative" [8, 15], meaning that all applications to equivalent types yield equivalent types:

```
type t = F bool
type u = F bool
```

defines types t and u to be equivalent. (Regardless, they are still abstract!)

An impure functor is "generative", however:

```
G : (a : type) →_I type
```

generates a fresh type with every application. So,

```
type v = G bool
type w = G bool
```

are different types! This is important in the face of impurity, since the types produced by an impure computation may e.g. depend on state, and thus equating them would be plain unsound. (More reasons why you all want both applicative *and* generative functors are disclosed in Sections 7 and 8 of [15].)

Now, what would happen if we allowed such a functor to be polymorphic over its effect? Say,

$$\mathsf{H} : (\mathsf{a} : \textbf{type}) \to_{\mathsf{P}} (\mathsf{e} : \textbf{effect}) \to_{\mathsf{e}} \textbf{type}$$

Depending on the choice of e, it would be either applicative or generative: that is, $\mathsf{H\ bool\ P} = \mathsf{H\ bool\ P}$, while $\mathsf{H\ bool\ I}$ is not reflexive (or even a well-formed type expression).

In other words, introducing polymorphism over effects implies a notion of *generativity polymorphism*. Is that actually a thing?

It is! In this paper, we show that this – rather esoteric – notion can actually be defined. And we don't even have to leave the familiar grounds of System F_ω for that. Well, except for one small extension. We also show that generativity polymorphism is the essence of MacQueen & Tofte & Kuan's (arguably equally esoteric) notion of "true" higher-order functors [10, 7, 6].

2 The Language

Figure 1 presents the syntax of 1ML extended with explicit effects, separated into core syntax and various syntactic sugar. For space reasons, we focus on the explicitly typed fragment 1ML_{ex} [13] here (though complementing this with effect inference in full 1ML is an easy addition). The effect-related constructs that are new relative to the original formulation of 1ML_{ex} are highlighted in the figure.

As in the original paper, the syntax is normalised to require named subterms in most constructs. The general forms, as well as many other familiar module or core-level constructs, can easily be defined as syntactic sugar. See [13] for enough sugar to upset your stomach.

2.1 Basic use of effects

The extended language incorporates three main additions.

Effect annotations. First, function types acquire an explicit annotation F, that specifies the effect that is released when calling the function. Because real programming language syntax better works as plain text, we use the notation "F/T" instead of making F a subscript on the arrow, as we did in the introduction.

There are two basic effect constants: **pure** and **impure**. To ease notation in the common cases, we allow abbreviating pure function types with a plain arrow "$\to$", and impure ones with a squiggly arrow "$\leadsto$".[1]

[1] I apologise for any confusion this may cause with the original 1ML paper, where "$\to$" is spelled "$\Rightarrow$" and "$\leadsto$" is spelled "$\to$". I chose to change the syntax for the extended system because pure arrows are the more common case now, and I like them to be represented by their natural operator.

(identifiers)	X		
(types)	T	$::=$	$E \mid \textbf{bool} \mid \{D\} \mid (X{:}T){\to}F/T \mid \textbf{type} \mid \textbf{effect} \mid {=}E \mid T\ \textbf{where}\ (\overline{.X}{:}T)$
(effects)	F	$::=$	$E \mid \textbf{pure} \mid \textbf{impure} \mid F{,}F$
(declarations)	D	$::=$	$X : T \mid \textbf{include}\ T \mid D;D \mid \epsilon$
(expressions)	E	$::=$	$X \mid \textbf{true} \mid \textbf{false} \mid \textbf{if}\ X\ \textbf{then}\ E\ \textbf{else}\ E{:}T \mid \{B\} \mid E.X \mid$
			$\textbf{fun}\ (X{:}T) \Rightarrow E \mid X\ X \mid \textbf{type}\ T \mid \textbf{effect}\ F \mid X{:>}T$
(bindings)	B	$::=$	$X{=}E \mid \textbf{include}\ E \mid B;B \mid \epsilon$

Abbreviations:

(types)

$(X{:}T_1) \to T_2 \quad := \quad (X{:}T_1) \to \textbf{pure}/T_2$

$(X{:}T_1) \rightsquigarrow T_2 \quad := \quad (X{:}T_1) \to \textbf{impure}/T_2$

$T_1 \overset{\to}{\rightsquigarrow} T_2 \quad := \quad (X{:}T_1) \overset{\to}{\rightsquigarrow} T_2$

where: (parameter) $P ::= (X{:}T)$

(declarations)

$\textbf{effect}\ X\ \overline{P} \quad := X : \overline{P \to} \textbf{effect}$

$\textbf{effect}\ X\ \overline{P}{=}F := X : \overline{P \to} ({=}\,\textbf{effect}\ F)$

(bindings)

$\textbf{effect}\ X\ \overline{P}{=}F := X = \textbf{fun}\ \overline{P} \Rightarrow \textbf{effect}\ F$

Fig. 1. Syntax of $1\mathrm{ML}_{\mathrm{ex}}$ with effect polymorphism (see [13] for more abbreviations)

For example, the type of the polymorphic identity function,

```
id = fun a ⇒ fun (x : a) ⇒ a
```

can be denoted as

```
id : (a : type) → pure/(x : a) → pure/a
```

or shorter, as just

```
id : (a : type) → (x : a) → a
```

Similarly, we can add impure operators to the language; for example, an ML-style type ref T of mutable references with operators

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

Note how these types mix 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.

Effect values. Second, what makes the new syntax for function types interesting instead of just verbose is that effects can now be "computed": F can not just refer to the two effect constants, it can also consist of an expression E. This has to be a (pure) expression of the new type **effect**, a type that is inhabited by effect "values". Those values are formed by the corresponding expression **effect** F. Just like types in 1ML, effects can thus be named or passed around as if they were first-class values.

For example, we can write an effect-polymorphic apply combinator:

```
apply = fun (a : type) (b : type) (e : effect) (f : a → e/b) (x : a) ⇒ f x
```

The type of this function is:

```
apply : (a : type) → (b : type) → (e : effect) → (a → e/b) → a → e/b
```

To apply it, we have to provide the right effect argument:

```
t = apply bool bool pure id true
f = apply (ref bool) bool impure (rd bool) (new false)
```

Similarly, the map function from the introduction can be given the (explicitly) polymorphic type

```
map : (a : type) → (b : type) → (e : effect) → (a → e/b) → list a → e/list b
```

Like any other value, effects can also be named by a binding:

```
effect e = pure
t = apply bool bool e id true
```

Analogous to type bindings in 1ML, this effect binding is just sugar for the value binding "e = (**effect** pure)". Admittedly, such bindings are a bit boring with just effect constants, but they become more interesting with the following feature.

Effect composition. Finally, how are we going to type the following function?

```
compose = fun (a : type) (b : type) (c : type) (e1 : effect) (e2 : effect)
              (f : b → e2/c) (g : a → e1/b) (x : a) ⇒ f (g x)
```

To give a type to that, we need the ability to compose effects. That is the role of the effect operator ",":

```
compose : (a : type) → (b : type) → (c : type) → (e1 : effect) → (e2 : effect) →
              (b → e2/c) → (a → e1/b) → a → (e1,e2)/c
```

"F_1,F_2" denotes the least upper bound of effects F_1 and F_2 in a lattice where **pure** is the bottom element and **impure** is top (the direction of this lattice is consistent with the natural notion of subtyping on effects, where pure $\leq$ impure; see Section 3). Consequently, an invocation of compose will be impure if at least one of the effect parameters is instantiated with impure; only if both are pure the result is pure as well. That should come as no surprise.

2.2 Generativity polymorphism

So far, we have only looked at effects of simple functions. If you are already familiar with effect polymorphism, and much of the above looked boring, then you can wake up now – we are now turning to modules and functors.

First-order generativity polymorphism. Let us start with the simplest possible functor – essentially, the identity type constructor:

```
Id = fun (a : type) ⇒ a
```

The most specific type derivable for Id in 1ML is the fully transparent type

```
type ID_TRANS = (a : type) → (= a)
```

where the *singleton type* (= a) indicates that the function returns the argument a itself. This *transparent* signature is a subtype of two possible *opaque* signatures that do not reveal the identity of their resulting type:

```
type ID_PURE = (a : type) → type
type ID_IMPURE = (a : type) ⇝ type
```

We can *seal* Id with either of these signatures:

```
Id_pure = Id :> ID_PURE
Id_impure = Id :> ID_IMPURE
```

Both these functors now return an abstract type, but the first is "applicative", i.e., always returns the same type for equivalent arguments, while the second is "generative", i.e., always returns a fresh type, regardless of the argument.

But in our extended 1ML, there now is a third choice:

```
type ID_POLY (e : effect) = (a : type) → e/type
Id_poly (e : effect) = Id :> ID_POLY e
```

We have just created our first "poly-generative" functor! This one can be applied to our choice of effect: for example, Id_poly pure bool = Id_poly pure bool are equivalent types, while Id_poly impure bool is a generative (and thus impure) expression that cannot be used as a type without binding it to a name. This is exactly the functor H we wondered about in the introduction.

Higher-order generativity polymorphism. Okay, that was a contrived example. Generativity polymorphism becomes more relevant in the higher-order case. Because we can now write the kind of functors that MacQueen always wanted to write, under his slogan of "true higher-order functors" [10, 7, 6]. Recast in 1ML, the problem he is concerned with, as formulated by Kuan & MacQueen [7], boils down to the ability to define a generic Apply functor over types:

```
Apply (F : type → e/type) (a : type) = F a
```

Kuan & MacQueen want to be able to use such a functor transparently (i.e., such that type identities are fully propagated) in both of the following cases:

```
Id (a : type) = a
t = Apply Id bool
f (x : t) = (x : bool)  ;; type-checks because t = bool

Const (a : type) = int
u = Apply Const bool
g (x : u) = (x : int)   ;; type-checks because u = int
```

As it turns out, these examples already type-check in plain 1ML: both applications are well-typed, provided one picks e = pure in the parameter type of F when defining Apply (which is equivalent to a plain 1ML pure function type). In fact, even the application to an abstract functor works, as long as that is pure as well:

```
Abs = Id :> type → type
v = Apply Abs bool
h (x : v) = (x : Abs bool)  ;; type-checks because v = Abs bool
```

However, Kuan & MacQueen didn't deal with a language of (only) applicative functors (and neither do we), so the above isn't quite a fair answer to their challenge. What they really meant in particular is that Apply should also be applicable to a *generative* functor, like here:

```
Gen (a : type) = a :> type
w = Apply Gen bool
```

That also works in plain 1ML, but only if the parameter F is typed as an impure functor in the definition of Apply, equivalent to picking e = impure in our extended language. That is, in plain 1ML, we can write Apply such that either the former three examples type-check, or the last, but not all at the same time.

Effect polymorphism to the rescue! You already guessed it: in extended 1ML we can escape that dilemma by making e into a parameter itself:

```
Apply (e : effect) (F : type → e/type) (a : type) = F a
```

Now all examples are expressible:

```
t = Apply pure Id bool
u = Apply pure Const bool
v = Apply pure Abs bool
w = Apply impure Gen bool
```

The extra argument is a bit more tedious to write than Kuan & MacQueen would want, but it could easily be inferred (though we don't discuss that here).

Existential effect polymorphism. Just for completeness, we mention that effects can also – like types – be *sealed*, introducing the notion of an "abstract effect":

```
M = {effect e = pure; f (g : int → e/bool) = ...} :> {effect e; f : (int → e/bool) → ...}
let g (h : int → M.e/bool) = ... in g M.f
```

Honestly, this does not look like it would be a particularly useful feature, but it falls out from 1ML's design naturally and for free. Ruling it out would be more complicated than allowing it. Maybe there even is some crazy use case that we don't foresee yet... Phantom effects, anyone?

3 Type System

Here's how we build a type system for 1ML_{ex} with generativity polymorphism. (Unfortunately, for the lack of space, we have to focus on the novelties, and refer the interested reader to [13] for many basic details and rules omitted here.)

Semantic Types. Following the F-ing modules approach [15], 1ML's type system [13] is not defined in terms of its own syntactic types. Instead, it is defined by translating those types into *semantic types*. (The short story behind that approach is that syntactic module types are not expressive enough to accurately account for all details of type abstraction and functorisation, especially the problem of local types, a.k.a. the *avoidance problem*. See [15] for the long story.)

$$
\begin{array}{ll}
\text{(computation)} & \varPhi \ ::= \varXi \mid (\varPhi + \varPhi)^{\eta} \\
\text{(abstracted)} & \varXi \ ::= \exists\overline{\alpha}.\varSigma \\
\text{(large)} & \varSigma \ ::= \pi \mid \mathsf{bool} \mid [= \varXi] \mid [= \eta] \mid \{\overline{l{:}\varSigma}\} \mid \forall\overline{\alpha}.\varSigma \rightarrow_{\eta} \varPhi \\
\text{(small)} & \sigma \ ::= \pi \mid \mathsf{bool} \mid [= \sigma] \mid [= \eta] \mid \{\overline{l{:}\sigma}\} \mid \sigma \rightarrow_{\eta} \sigma \\
\text{(paths)} & \pi \ ::= \alpha \mid \pi\,\overline{\sigma} \\
\text{(effects)} & \eta \ ::= \mathtt{P} \mid \mathtt{I} \mid \iota \mid \eta \vee \eta
\end{array}
$$

Notation:

$$
\begin{array}{rrll}
\mathtt{P} \vee \eta := \eta \vee \mathtt{P} := \eta & \varXi\,! := \varXi & (\varPhi_1 \oplus \varPhi_2)^{\mathtt{P}} := \varPhi_1 & \\
\mathtt{I} \vee \eta := \eta \vee \mathtt{I} := \mathtt{I} & (\varPhi_1 + \varPhi_2)^{\mathtt{P}}! := \varPhi_1! & (\varPhi_1 \oplus \varPhi_2)^{\mathtt{I}} := \varPhi_2 & \\
\eta(\varSigma) := \mathtt{P} & & (\varPhi_1 \oplus \varPhi_2)^{\eta} := \varPhi_1 & \text{if } \varPhi_1 = \varPhi_2 \\
\eta(\exists\alpha\overline{\alpha}.\varSigma) := \mathtt{I} & & (\varPhi_1 \oplus \varPhi_2)^{\eta} := (\varPhi_1 + \varPhi_2)^{\eta} & \text{otherwise}
\end{array}
$$

Fig. 2. Semantic Types

Figure 2 shows the grammar of semantic types needed for 1ML_{ex} with effect polymorphism (plus some auxiliary notation we'll get to). They are written in the style of System F types with explicit quantifiers (and as we will see later they actually *are* System F types). Once more, additions to the original 1ML system are highlighted. Type variables can be higher-order in these types, but we assume they are kinded implicitly, and we use the notation κ_α if we need to talk about the kind of α.

Let us recap the main intuitions behind those F-ing module types. The central idea is introducing explicit quantifiers to remove all dependencies within a type.

Following Mitchell & Plotkin [11], signature types containing abstract types (i.e., ADTs) are represented as *existential* types. For example, the signature

$$\{\textbf{type}\ \mathsf{t};\ \textbf{type}\ \mathsf{u};\ \textbf{type}\ \mathsf{v} = \mathsf{t} \rightarrow \mathsf{t};\ \mathsf{f} : \mathsf{t} \rightarrow \mathsf{u}\}$$

where components v and f depend on t and u, corresponds to the semantic type

$$\exists\alpha_1\alpha_2.\{\mathsf{t} : [= \alpha_1], \mathsf{u} : [= \alpha_2], \mathsf{v} : [= \alpha_1 \rightarrow \alpha_1], \mathsf{f} : \alpha_1 \rightarrow \alpha_2\}$$

where α_1 and α_2 are the local names for type components t and u, respectively, and there are no dependencies inside the record. The notation $[= \tau]$ denotes the type of τ reified "as a value", i.e., represents the type **type** (recall that a component specification "**type** t" is just short for "t : **type**" in 1ML).

Functors, on the other hand, correspond to *universal* types. Any abstract type from their domain signature becomes a universal type variable with scope widened to include the codomain. For example, the functor type

$$(\mathsf{X} : \{\textbf{type}\ \mathsf{t};\ \mathsf{v} : \mathsf{t}\}) \to \textbf{pure}/\{\textbf{type}\ \mathsf{u} = \mathsf{X.t} \to \mathsf{X.t};\ \mathsf{x} : \mathsf{X.t}\}$$

with dependencies from its codomain on the type X.t from the domain, maps to

$$\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_{\mathtt{P}} \{\mathsf{u} : [= \alpha \to \alpha], \mathsf{x} : \alpha\}$$

More interesting are cases where an abstract type is bound in the codomain:

$$(\mathsf{X} : \{\textbf{type}\ \mathsf{t};\ \mathsf{v} : \mathsf{t}\}) \to \mathsf{e}/\{\textbf{type}\ \mathsf{u};\ \mathsf{f} : \mathsf{X.t} \to \mathsf{u}\}$$

If e is impure, then this is a generative functor, which is modeled in the semantic types by returning an existential package:

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

If e is pure, however, then the functor is supposed to behave applicatively, i.e., return the same abstract type on each application. That is modeled by lifting the existential out of the function:

$$\exists\beta.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_{\mathtt{P}} \{\mathsf{u} : [= \beta\,\alpha], \mathsf{f} : \alpha \to \beta\,\alpha\}$$

In fact, it as an invariant of our semantic types that a pure arrow never has an existential quantifier to the right – a pure functor cannot generate types. There is one subtlety involved with the above type, though: in the implementation of a functor matching this signature, the definition of u may depend on the parameter type t. Following Biswas [1] and Russo [16], all uses of β in the semantic interpretation are hence skolemised over α accordingly. Consequently, β needs to have higher kind $\kappa_\beta = \Omega \to \Omega$ here (other variables had base kind Ω so far).

As we can see, the structure of the semantic type modeling the functor type above is quite different depending on the choice of e. It differs in terms of where the quantifier goes (inside vs. outside), its variable kind (Ω vs. $\Omega \to \Omega$), and how the abstract type u is denoted (β vs. $\beta\,\alpha$). How can we reconcile these structural differences to support a parametric choice of effect?

We can't. Not really, anyway. We need something new.

The natural trick is to use sums: we generalise function types such that their codomain is a "computation" type Φ that allows arbitrary many alternative results. That is, if e is not statically known to be either pure or impure, then the functor signature above will be represented as

$$\begin{aligned}&\exists\beta_1.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \to_\eta \\ &\qquad(\{\mathsf{u} : [= \beta_1\,\alpha], \mathsf{f} : \alpha \to \beta_1\,\alpha\} + \exists\beta_2.\{\mathsf{u} : [= \beta_2], \mathsf{f} : \alpha \to \beta_2\})^\eta\end{aligned}$$

This encodes both possibilities: the left side of the sum is for the pure choice, the right for the impure one. In this type, η is the representation of the effect e as a semantic type. If η is not a constant P or I, then it either is an *effect variable* ι, or the least upper bound of several of those, represented by the "$\vee$" operator. Since the two effect constants are the top and bottom elements of the effect lattice, least upper bounds that contain those constants can always be simplified

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

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \leadsto e \qquad \Phi! = [= \Xi]}{\Gamma \vdash E \leadsto \Xi}\,\textsc{Tpath}$$

$$\frac{\kappa_\alpha = \Omega}{\Gamma \vdash \textsf{type} \leadsto \exists\alpha.[= \alpha]}\,\textsc{Ttype} \qquad \frac{}{\Gamma \vdash \textsf{effect} \leadsto \exists\iota.[= \iota]}\,\textsc{Teffect}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leadsto \exists\overline{\alpha}_1.\Sigma_1 \\ \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash T_2 \leadsto \exists\overline{\alpha}_2.\Sigma_2 \qquad \Gamma \vdash F \leadsto \eta \qquad \overline{\kappa_{\alpha'_2} = \overline{\kappa_{\alpha_1}} \to \kappa_{\alpha'_2}}\end{array}}{\Gamma \vdash (X{:}T_1) \to F/T_2 \leadsto \exists\overline{\alpha}'_2{}^{\eta \neq \mathtt{I}}.\forall\overline{\alpha}_1.\, \Sigma_1 \to_\eta ((\exists\overline{\alpha}_2.\Sigma_2) \oplus (\Sigma_2[\overline{\alpha'_2\,\overline{\alpha}_1}/\overline{\alpha}_2]))^\eta}\,\textsc{Tfun}$$

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \leadsto e \qquad \Phi! = \Sigma}{\Gamma \vdash (=E) \leadsto \Sigma}\,\textsc{Tsing}$$

Effects $\boxed{\boldsymbol{\Gamma \vdash F \leadsto \eta}}$

$$\frac{\Gamma \vdash E :_{\mathtt{P}} \Phi \leadsto e \qquad \Phi! = [= \eta]}{\Gamma \vdash E \leadsto \eta}\,\textsc{Fpath}$$

$$\frac{}{\Gamma \vdash \textsf{pure} \leadsto [= \mathtt{P}]}\,\textsc{Fpure} \qquad \frac{}{\Gamma \vdash \textsf{impure} \leadsto [= \mathtt{I}]}\,\textsc{Fimpure}$$

$$\frac{\Gamma \vdash F_1 \leadsto \eta_1 \qquad \Gamma \vdash F_2 \leadsto \eta_2}{\Gamma \vdash F_1{,}F_2 \leadsto \eta_1 \vee \eta_2}\,\textsc{Fjoin}$$

Fig. 3. Elaboration of Types and Effects (new and modified rules)

to either ones that don't, or directly to I. We assume that this simplification is always performed implicitly, according to the notational rules given in Figure 2.

The effect η appears twice in the above type: once as an annotation on the arrow, and once as an *index* on the sum, which is written $(\Phi_1 + \Phi_2)^\eta$. The former occurrence tracks the effect of the function, in a generalisation of what we already had in plain 1ML. The latter tracks the choice of the sum: if, at some point, the free effect variables of η get substituted such that the result normalises to an effect constant, then the choice is statically determined – this basically is a binary GADT. We say that a Φ with all effect indices being constants is *unique*. As we will see below, this static extra knowledge is important for figuring out when an expression of type **type** unambiguously denotes a type, such that it is legal to project it.

It may be surprising that we need η twice. The reason is that, in general, the computation type Φ in the codomain of an arrow can consist of many nested sums, indexed by different effects – e.g., when a functor itself invokes several other functors with variable effects. The effect on the arrow then only is an upper bound of all these individual indices (and not necessarily the least); for example, we could construct a functor of type $\Sigma \to_{\eta_1 \vee \eta_2} ((\Xi_1 + \Xi_2)^{\eta_1} + \Xi_3)^{\eta_2}$. In a pure function, however, this upper bound is P, so all effect indices in Φ must be pure as well, such that Φ is already guaranteed to be unique.

Types and Effects. The new and modified rules for translating syntactic types T into semantic types Ξ are collected in Figure 3 (please see [13] for the others). The individual modifications over plain 1ML are again highlighted.

The new rule TEFFECT for the type **effect** is analogous to the rule for **type**, in that it introduces a fresh type variable to name the abstract effect. We use ι to range over type variables that encode effects. We assume that these type variables are a subcategory of general type variables α, such that they can be uniformly written as α wherever we don't care about the distinction.

The rule TFUN for function types effectively merges the previous rules TFUN and TPFUN from 1ML, covering both impure and pure functors, but also effect-polymorphic ones. It refers to the simple effect elaboration judgement also shown in the figure. The auxiliary operator "$\oplus$" (defined in Figure 2) avoids the sum in those cases where the effect is statically known or does not change the type. Likewise, we write "$\exists\overline{\alpha}^{\iota\neq\mathtt{I}}$" to say that a quantifier is to be empty if $\eta = \mathtt{I}$.

Another change is in rules TPATH and TSING (and inherited by FPATH): the notation $\Phi!$ (also defined in Figure 2) requires E's type Φ to be unique – it selects Φ's unique summand and is undefined otherwise. It is here where we rely on the effect index on sums: only sums whose indices are constant are unique, and can be used for unambiguous projection of static information.

Expressions and Bindings. Figure 4 shows selected typing rules for expressions E and bindings B, novelties once more highlighted. Before we go into details, let us recap the main idea of typing modules using the F-ing modules approach.

As we saw before, the main trick in interpreting module 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" 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 $\{\mathsf{x} : \Sigma, \dots\}$; the resulting type would be Σ then. However, if E creates abstract types locally, then its type will be of the form $\exists\overline{\alpha}.\{\mathsf{x} : \Sigma, \dots\}$ instead. The central idea of 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.

As we pointed out in [15], this handling of existential types is akin to a *monad* – the monad of type generation! More precisely, it is a stack of nested monads, one for each generated type. By extending 1ML with generativity polymorphism, however, there no longer necessarily is a unique quantifier sequence for a given expression. Expressions are now classified by "computation" types Φ, which are sums over heterogeneous existentials. Our monad just became more interesting!

Fortunately, the sums we are dealing with are not arbitrary. Ultimately, they all originate, directly or indirectly, from uses of the rule TFUN introducing effect-polymorphic function types. And a quick look at this rule reveals that the inner structure of the type is the same in both cases, up to the presence of quantifiers and the internal naming of the abstract types introduced.

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

$$\frac{\Gamma \vdash T \rightsquigarrow \Xi}{\Gamma \vdash \textsf{type}\ T :_{\mathtt{P}} [= \Xi] \rightsquigarrow [\Xi]}\textsc{EType} \qquad \frac{\Gamma \vdash F \rightsquigarrow \eta}{\Gamma \vdash \textsf{effect}\ F :_{\mathtt{P}} [= \eta] \rightsquigarrow [\eta]}\textsc{EEffect}$$

$$\frac{\Gamma \vdash E :_\eta M\lambda\overline{\alpha}.\{\overline{X':\Sigma'}\} \rightsquigarrow e \qquad X{:}\Sigma \in \overline{X':\Sigma'}}{\Gamma \vdash E.X :_\eta M\lambda\overline{\alpha}.\Sigma \rightsquigarrow \textsf{do}_M\ \overline{\alpha}, y \leftarrow e\ \textsf{in}\ y.X}\textsc{EDot}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma, \overline{\alpha}, X{:}\Sigma \vdash E :_\eta \Phi \rightsquigarrow e}{\Gamma \vdash \textsf{fun}\,(X{:}T) \Rightarrow E :_{\mathtt{P}} \forall\overline{\alpha}.\ \Sigma \rightarrow_\eta \Phi \rightsquigarrow \lambda\overline{\alpha}.\lambda_\eta X{:}\Sigma.e}\textsc{EFun}$$

$$\frac{\begin{array}{ll}\Gamma \vdash X_1 :_{\mathtt{P}} (\forall\overline{\alpha}.\ \Sigma_1 \rightarrow_\eta \Phi) \rightsquigarrow e_1 & \\ \Gamma \vdash X_2 :_{\mathtt{P}} \Sigma_2 \rightsquigarrow e_2 & \Gamma \vdash \Sigma_2 \leq_{\overline{\alpha}} \Sigma_1 \rightsquigarrow \delta; f\end{array}}{\Gamma \vdash X_1\ X_2 :_\eta \delta\Phi \rightsquigarrow (e_1\ (\delta\overline{\alpha})\ (f\ e_2)).\textsf{val}}\textsc{EApp}$$

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

$$\frac{\Gamma \vdash E :_\eta M\lambda\overline{\alpha}.\Sigma \rightsquigarrow e}{\Gamma \vdash X{=}E :_\eta M\lambda\overline{\alpha}.\{X{:}\Sigma\} \rightsquigarrow \textsf{do}_M\ \overline{\alpha}, x \leftarrow e\ \textsf{in}\ \{X{=}x\}}\textsc{BVar}$$

$$\frac{\begin{array}{rl}\Gamma \vdash B_1 :_{\eta_1} M_1\lambda\overline{\alpha}_1.\{\overline{X_1{:}\Sigma_1}\} \rightsquigarrow e_1 & \overline{X}'_1 = \overline{X}_1 - \overline{X}_2 \\ \Gamma, \overline{\alpha}_1, \overline{X_1{:}\Sigma_1} \vdash B_2 :_{\eta_2} M_2\lambda\overline{\alpha}_2.\{\overline{X_2{:}\Sigma_2}\} \rightsquigarrow e_2 & \overline{X'_1{:}\Sigma'_1} \subseteq \overline{X_1{:}\Sigma_1}\end{array}}{\begin{array}{l}\Gamma \vdash B_1;B_2 :_{\eta_1 \vee \eta_2} M_1M_2\lambda\overline{\alpha}_1\overline{\alpha}_2.\{\overline{X'_1{:}\Sigma'_1}, \overline{X_2{:}\Sigma_2}\} \\ \quad \rightsquigarrow \textsf{join}_{M_1M_2}\ \textsf{do}_{M_1}\ \overline{\alpha}_1, y_1 \leftarrow e_1\ \textsf{in let}\ \overline{X_1 = y_1.X_1}\ \textsf{in} \\ \qquad\qquad\qquad \textsf{do}_{M_2}\ \overline{\alpha}_2, y_2 \leftarrow e_2\ \textsf{in}\ \{\overline{X'_1 = y_1.X'_1}, \overline{X_2 = y_2.X_2}\}\end{array}}\textsc{BSeq}$$

Fig. 4. Elaboration of Expressions (selected rules)

That allows to factor computation types Φ such that we separate their inner structure from their quantification scheme. To that end, Figure 5 defines an auxiliary syntactic class of monadic type constructors M. Any computation type Φ can be expressed β-equivalently as an application of such an M to a suitable type constructor defining the inner structure of the result. For example, the effect-polymorphic functor type from earlier can be written equivalently as

$$\exists\beta_1.\forall\alpha.\{\mathsf{t} : [= \alpha], \mathsf{v} : \alpha\} \rightarrow_\eta (\exists[\beta_1\ \alpha] + \exists\beta_2[\beta_2])^\eta\ (\lambda\beta.\{\mathsf{u} : [= \beta], \mathsf{f} : \alpha \rightarrow \beta\})$$

That is, an application of $(M_1 + M_2)^\eta$ with $M_1 = \exists[\beta_1\ \alpha]$ (which has an empty existential quantifier) and $M_2 = \exists\beta_2[\beta_2]$ to the structural template $\lambda\beta.\{\mathsf{u} : [= \beta], \mathsf{f} : \alpha \rightarrow \beta\}$, with β being mapped to either $\beta_1\ \alpha$ or β_2, accordingly.

The set of all monadic types M forms a *polymonad* [5] or *productoid* [19], with $\exists[]$ as its identity element. We can define the composition M_1M_2 as given in Figure 5. We won't go into details here, but leave proving the polymonad laws as an exercise (see also Section 4).

It suffices to observe that we can use this notation to uniformly access the structure of all alternatives of a computation type. Moreover, we can construct a new computation type that lives in the same monadic envelope M. In particular, (monadic type) $M ::= \exists\overline{\alpha}[\overline{\pi}] \mid (M+M)^{\eta}$

with:

$$\begin{array}{lcl}
\exists\overline{\alpha}[\overline{\pi}]_{\overline{\kappa}} & := & \lambda c{:}(\overline{\kappa}\to\Omega).\exists\overline{\alpha}.c\,\overline{\pi} \\
(M_1+M_2)^{\eta}_{\overline{\kappa}} & := & \lambda c{:}(\overline{\kappa}\to\Omega).(M_1c+M_2c)^{\eta}
\end{array}$$

$$(M_1M_2)_{\overline{\kappa}_1\overline{\kappa}_2} := \begin{cases} \exists\overline{\alpha}_1\overline{\alpha}_2[\overline{\pi}_1\overline{\pi}_2] & \text{if } M_1=\exists\overline{\alpha}_1[\overline{\pi}_1] \text{ and } M_2=\exists\overline{\alpha}_2[\overline{\pi}_2] \\ (M_1M_{21}+M_1M_{22})^{\eta_2} & \text{if } M_1=\exists\overline{\alpha}_1[\overline{\pi}_1] \text{ and } M_2=(M_{21}+M_{22})^{\eta_2} \\ (M_{11}M_2+M_{12}M_2)^{\eta_1} & \text{if } M_1=(M_{11}+M_{12})^{\eta_1} \end{cases}$$

$$\mathsf{do}_M\ \overline{\alpha},x\leftarrow e_1\ \mathsf{in}\ e_2 := \begin{cases} \mathsf{unpack}\ \langle\overline{\alpha}',x\rangle=e_1\ \mathsf{in\ pack}\ \langle\overline{\alpha}',e_2\rangle & \text{if } M=\exists\overline{\alpha}'[\overline{\pi}] \\ \mathsf{let}\ f=\lambda\overline{\alpha}.\lambda x.e_2\ \mathsf{in\ case}\ e_1\ \mathsf{of} & \text{if } M=(M_1+M_2)^{\eta} \\ \quad \mathsf{inl}\ x_1.(\mathsf{do}_{M_1}\ \overline{\alpha},x\leftarrow x_1\ \mathsf{in\ inl}\ e_2\,\overline{\alpha}\,x) \mid & \\ \quad \mathsf{inr}\ x_2.(\mathsf{do}_{M_2}\ \overline{\alpha},x\leftarrow x_2\ \mathsf{in\ inr}\ e_2\,\overline{\alpha}\,x) & \end{cases}$$

$$\mathsf{join}_{M_1M_2}\ e := \begin{cases} \mathsf{case}\ e\ \mathsf{of} & \text{if } M_1=(M_{11}+M_{12})^{\eta_1} \\ \quad \mathsf{inl}\ x.\mathsf{inl}\ (\mathsf{join}_{M_{11}M_2}\ x) \mid & \\ \quad \mathsf{inr}\ x.\mathsf{inr}\ (\mathsf{join}_{M_{12}M_2}\ x) & \\ \mathsf{unpack}\ \langle\overline{\alpha}_1,x\rangle=e\ \mathsf{in\ case}\ x\ \mathsf{of} & \text{if } M_1=\exists\overline{\alpha}_1[\overline{\pi}_1] \\ \quad \mathsf{inl}\ y.\mathsf{inl}\ (\mathsf{join}_{M_1M_{21}}\ \mathsf{pack}\ \langle\overline{\alpha}_1,y\rangle) \mid & \text{and } M_2=(M_{21}+M_{22})^{\eta_2} \\ \quad \mathsf{inr}\ y.\mathsf{inr}\ (\mathsf{join}_{M_1M_{22}}\ \mathsf{pack}\ \langle\overline{\alpha}_1,y\rangle) & \\ e & \text{otherwise} \end{cases}$$

Fig. 5. Polymonad notation for computation types

this happens in rules EDOT and BVAR, which project and inject a value from/into a structure, respectively. The notation is put to more interesting use in rule BSEQ, where two nested monadic computations in M_1 and M_2 are lifted to their composition M_1M_2.

On the term level, the polymonad is witnessed by suitably defined do-notation (which expresses a mapping over some M) and a join operator. The definition of these operators, indexed by M, is given in Figure 5.

Subtyping. The existing rules for 1ML subtyping don't change, but subtyping now needs to be generalised to computation types Φ. Figure 6 shows how.

Basically, the six new rules inductively express that $\Phi_1 \leq \Phi_2$ holds if each Ξ_1 from Φ_1 is a subtype of each Ξ_2 from Φ_2. Except that in the case of a constant effect index, the excluded alternative can be ignored.

The most interesting case is rule SR. It coerces a unique type Ξ' into a sum indexed by an effect η. Since η may force later which alternative to pick, the coercion has to perform a case distinction over η. To enable that, effects need to be reified as terms in the elaboration. We refer to the Appendix for details. There, we also explain the operators asl and asr used in the elaboration of rules SLP and SLI, which are akin to a one-armed case over the binary "+" GADT.

Metatheory. For space reasons, we have banished all metatheory to the Appendix, where we define the encoding of semantic types into System F$_\omega$, and state the obvious soundness results for the elaboration.

Subtyping $\quad \Gamma \vdash \Phi' \le \Phi \rightsquigarrow f \;:=\; \Gamma \vdash \Phi' \le_\epsilon \Phi \rightsquigarrow \mathrm{id}; f \qquad \boxed{\boldsymbol{\Gamma \vdash \Phi' \le_\pi \Phi \rightsquigarrow \delta; f}}$

$$\frac{\Gamma \vdash \Phi'_1 \le_\pi \Phi \rightsquigarrow \delta; f_1 \qquad \Gamma \vdash \Phi'_2 \le_\pi \Phi \rightsquigarrow \delta; f_2}{\Gamma \vdash (\Phi'_1 + \Phi'_2)^{\eta'} \le_\pi \Phi \rightsquigarrow \delta; \lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y.f_1 y \mid \mathsf{inr}\ y.f_2 y}\textsc{SL}$$

$$\frac{\Gamma \vdash \Phi'_1 \le_\pi \Phi \rightsquigarrow \delta; f}{\Gamma \vdash (\Phi'_1 + \Phi'_2)^{\mathtt{P}} \le_\pi \Phi \rightsquigarrow \delta; \lambda x.f\ (\mathsf{asl}\ x)}\textsc{SLP} \qquad \frac{\Gamma \vdash \Phi'_2 \le_\pi \Phi \rightsquigarrow \delta; f}{\Gamma \vdash (\Phi'_1 + \Phi'_2)^{\mathtt{I}} \le_\pi \Phi \rightsquigarrow \delta; \lambda x.f\ (\mathsf{asr}\ x)}\textsc{SLI}$$

$$\frac{\Gamma \vdash \Phi' \le_\pi \Phi_1 \rightsquigarrow \delta; f_1 \qquad \Gamma \vdash \Phi' \le_\pi \Phi_2 \rightsquigarrow \delta; f_2}{\Gamma \vdash \Xi' \le_\pi (\Phi_1 + \Phi_2)^{\eta} \rightsquigarrow \delta; \lambda x.\mathsf{case}\ \eta\ \mathsf{of}\ \mathsf{inl}\ y.\mathsf{inl}\ (f_1 x) \mid \mathsf{inr}\ y.\mathsf{inr}\ (f_2 x)}\textsc{SR}$$

$$\frac{\Gamma \vdash \Phi' \le_\pi \Phi_1 \rightsquigarrow \delta; f}{\Gamma \vdash \Xi' \le_\pi (\Phi_1 + \Phi_2)^{\mathtt{P}} \rightsquigarrow \delta; \lambda x.\mathsf{inl}\ (f\ x)}\textsc{SRP} \qquad \frac{\Gamma \vdash \Phi' \le_\pi \Phi_2 \rightsquigarrow \delta; f}{\Gamma \vdash \Xi' \le_\pi (\Phi_1 + \Phi_2)^{\mathtt{I}} \rightsquigarrow \delta; \lambda x.\mathsf{inr}\ (f\ x)}\textsc{SRI}$$

$$\frac{}{\eta \le \mathtt{I}}\textsc{FTop} \qquad \frac{}{\mathtt{P} \le \eta}\textsc{FBot} \qquad \frac{\bar\iota' \supseteq \bar\iota}{\bigvee \bar\iota' \le \bigvee \bar\iota}\textsc{FJoin} \qquad \boxed{\boldsymbol{\eta' \le \eta}}$$

Fig. 6. Elaboration of Subtyping (new rules)

4 Related Work

There has been a broad range of work on effect systems and effect polymorphism, starting from Gifford & Lucassen's original work [3, 9] and Talpin & Jouvelot's refinements [18]. But as noted in the introduction, the implications of effect polymorphism that we have investigated in this paper is rather esoteric – to the best of our knowledge, there is no other work on effect systems for modules, or generativity polymorphism of the kind we introduced here.

"True" higher-order modules. The idea most closely related hence actually is MacQueen's notion of "true" higher-order modules, as originally introduced by MacQueen & Tofte [10], implemented in SML of New Jersey, and later recast by Kuan & MacQueen [7, 6]. In this semantics, every functor type is implicitly "generativity polymorphic" as much as possible.

However, the formal details are rather involved, defining a specialised operational calculus of type name creation, path trees, and explicit environment manipulation (named the "entity calculus" in Kuan & MacQueen's more recent work). This semantics has so far escaped a more type-theoretic treatment, and consequently, none of the other formalisations of higher-order modules on the market [4, 8, 17, 2, 14, 15, 13] has followed its lead.

The system we presented is coming from a completely different angle. Yet, as we show in Section 2, it has similar expressiveness, while maintaining most of the relative simplicity of the 1ML semantics. One could argue that effect polymorphism is what was hiding in MacQueen's system all along, and that our system makes that explicit and gives it a foundation in standard type theory.

Monads, Polymonads and Productoids. Moggi [12] suggested monads as a means for semantic modeling of effectful computations. Wadler [20] recognised their broader value for language design, as an immensely viable user-facing feature, which became a cornerstone of Haskell.

Our paper on F-ing modules [15] already pointed out that existentials behave "like a monad" in our semantics, encapsulating the underlying "effect" of type generation. However, we never formally investigated the connection. A slightly more careful look reveals that it's not really a single monad, but a whole stack of them: one for each abstract type generated.

In the current paper, this interpretation as nested monads is no longer sufficient. Computation types are sums of existentials. In order to maintain this invariant under composition, composition can no longer be just nesting. Consequently, they give rise to a more general, more heterogeneous structure.

Hicks et al. [5] have recently investigated a generalisation of this kind of structure under the name *polymonad.* One way to describe it is as a *set* of monadic type constructors with heterogeneous bind (or join) operators. Independently, Tate [19] introduced a similar, slightly more general notion he calls *productoids.* In both cases, these formal structures were motivated by the desire to model certain forms of effects (though both works only investigate classical term-level effects).

Our computation types Φ, when factored into monadic constructors M, are an instance of this general structure. However, they are higher-kinded: they take a(nother) type constructor as argument, to allow transmitting the choice of type names to the "value" type. We leave a closer investigation of their exact relation to polymonads and productoids, and their formal properties, to future work.

5 Future Work

The current paper is primarily a sketch of a basic system. As always, there are many future roads to go. To mention only a few:

Implementation. We would like to integrate effect polymorphism into our 1ML prototype interpreter (mpi-sws.org/~rossberg/1ml/), to gather some practical experience from more experiments with the system.

Effect Inference. In the current paper we have only investigated the explicitly-typed fragment of 1ML. We believe that it is straightforward to incorporate *implicit functions* over effects to full 1ML, and enable the inference of effect parameters and arguments, just like for types.

More Effects. Our little language provides "impurity" (or partiality, if you prefer) as the only effect. That is as coarse as it can get. While already useful, it would be interesting to refine it to distinguish different concrete effects.

Abstract Effects. We have not yet explored what kind of abstractions might be enabled by the notion of abstract effect that our system introduces. Is it useful?

References

1. S. K. Biswas. Higher-order functors with transparent signatures. In *POPL*, 1995.
2. D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL*, 2003.
3. D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *LFP*, 1986.
4. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.
5. M. Hicks, G. Bierman, N. Guts, D. Leijen, and N. Swamy. Polymonadic programming. In *MSFP*, 2014.
6. G. Kuan. *A True Higher-Order Module System.* PhD thesis, University of Chicago, 2010.
7. G. Kuan and D. MacQueen. Engineering higher-order modules in sml/nj. In *IFL*, volume 6041 of *LNCS*, 2009.
8. X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, 1995.
9. J. Lucassen and D. Gifford. Polymorphic effect systems. In *POPL*, 1988.
10. D. MacQueen and M. Tofte. A semantics for higher-order functors. In *ESOP*, volume 788 of *LNCS*, 1994.
11. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, July 1988.
12. E. Moggi. Computational lambda calculus and monads. In *LICS*, 1989.
13. A. Rossberg. 1ML – Core and modules united. In *ICFP*, 2015.
14. A. Rossberg and D. Dreyer. Mixin' up the ML module system. *ACM TOPLAS*, 35(1), 2013.
15. A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *JFP*, 24(5):529–607, 2014.
16. C. Russo. Types for Modules. *ENTCS*, 60, 2003.
17. Z. Shao. Transparent modules with fully syntactic signatures. In *ICFP*, 1999.
18. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *JFP*, 2(3):245271, 1992.
19. R. Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.
20. P. Wadler. The essence of functional programming. In *POPL*, 1992.
21. P. Wadler and P. Thiemann. The marriage of effects and monads. *TOCL*, 4(1), 2003.

A Elaboration

A.1 Internal Language

As in the the F-ing modules semantics [15] and the original 1ML paper [13], we define the semantics of the extended language by elaborating 1ML_{ex} types and terms into types and terms of a (call-by-value, impredicative) variant of System F_ω with simple record types – see the syntax in Figure 7.

However, this time we need a small extension over plain F_ω: to track generativity effects and the different forms of quantification they can cause in an indexed sum, we require a simple notion of GADT. We add this to F_ω in the simplest possible form: an indexed binary sum $(\tau_1 + \tau_2)^\tau$. Operationally, it is

$$
\begin{array}{lrcl}
\text{(kinds)} & \kappa & ::= & \Omega \mid \kappa \to \kappa \\
\text{(types)} & \tau & ::= & \alpha \mid \tau \to \tau \mid \{\overline{l{:}\tau}\} \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \top \mid \bot \mid (\tau+\tau)^{\tau} \mid \lambda\alpha{:}\kappa.\tau \mid \tau\ \tau \\
\text{(terms)} & e, f & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid \{\overline{l{=}e}\} \mid e.l \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \\
 & & & \mathsf{pack}\ \langle\tau, e\rangle_\tau \mid \mathsf{unpack}\ \langle\alpha, x\rangle{=}e\ \mathsf{in}\ e \mid \\
 & & & \mathsf{inl}_\tau\ e \mid \mathsf{inr}_\tau\ e \mid \mathsf{asl}\ e \mid \mathsf{asr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.e \mid \mathsf{inr}\ x.e
\end{array}
$$

$$
\frac{}{\Gamma \vdash \top : \Omega} \qquad \frac{}{\Gamma \vdash \bot : \Omega} \qquad \frac{\Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau_1 : \Omega \qquad \Gamma \vdash \tau : \Omega}{\Gamma \vdash (\tau_1 + \tau_2)^{\tau} : \Omega}
$$

$$
\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : \Omega}{\Gamma \vdash \mathsf{inl}_{\tau_2}\ e : (\tau_1 + \tau_2)^{\top}} \qquad \frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : \Omega}{\Gamma \vdash \mathsf{inr}_{\tau_1}\ e : (\tau_1 + \tau_2)^{\bot}}
$$

$$
\frac{\Gamma \vdash e : (\tau_1 + \tau_2)^{\top}}{\Gamma \vdash \mathsf{asl}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : (\tau_1 + \tau_2)^{\bot}}{\Gamma \vdash \mathsf{asr}\ e : \tau_2}
$$

$$
\frac{\Gamma \vdash e : (\tau_1 + \tau_2)^{\tau'} \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.e_1 \mid \mathsf{inr}\ x_2.e_2 : \tau}
$$

Fig. 7. Syntax and non-standard typing rules of F_ω with binary GADTs

equivalent to an ordinary sum, with injections $\mathsf{inl}\ e$ and $\mathsf{inr}\ e$ and a case construct for elimination. However, the index τ allows deducing the choice statically: if it is equivalent to the type $\top$, then the value is known to be a left injection; analogously, $\bot$ implies a right injection. Accordingly, there are two additional elimination constructs, $\mathsf{asl}\ e$ and $\mathsf{asr}\ e$, that are only valid on operands with known index, and perform the respective projection (they correspond to a one-armed case in more general forms of GADTs).

Figure 7 shows the syntax for this IL. The non-standard extensions just described are highlighted. The figure also includes the typing and kinding rules for those additional constructs. We omit reduction rules, because they are straightforward. The semantics is otherwise completely standard, and reuses the formulation from [15].

We write $\Gamma \vdash e : \tau$ for the F_ω typing judgement, and let $e \hookrightarrow e'$ denote (one-step) reduction. As one would hope, the language enjoys the standard soundness properties:

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

Theorem 2 (Progress).
If $\cdot \vdash e : \tau$ and e is not a value, then $e \hookrightarrow e'$ for some e'.

To establish soundness of 1ML it suffices to ensure that elaboration always produces well-typed F_ω terms (Section A.3).

A.2 Encoding Semantic Types

Elaboration translates $1\mathrm{ML}_{\mathrm{ex}}$ types directly into "equivalent" System F_ω types. The shape of these *semantic* types was given by the grammar in Figure 2.

(kinds)			(environments)		
Eff	$:=$	$\Omega \to \Omega \to \Omega$	Γ, ι	$:=$	$\Gamma, \alpha_\iota, x_\iota{:}\, \mathsf{eff}\, \alpha_\iota$
$\mathsf{Mon}_{\overline{\kappa}}$	$:=$	$(\overline{\kappa} \to \Omega) \to \Omega$			
(types)			(terms)		
$[= \tau]$	$:=$	$\{\mathsf{typ} : \tau \to \{\}\}$	$[\tau]$	$:=$	$\{\mathsf{typ} = \lambda x{:}\tau.\{\}\}$
$[= \eta]$	$:=$	$\{\mathsf{typ} : \eta \to \{\}\}$	$[\eta]$	$:=$	$\{\mathsf{eff} = \lambda x{:}(\eta \top \bot).\{\}, \mathsf{val} = \eta\}$
$\tau_1 \to_\eta \tau_2$	$:=$	$\tau_1 \to \{\mathsf{val} : \tau_2, \mathsf{eff} : [= \eta]\}$	$\lambda_\eta x{:}\tau.e$	$:=$	$\lambda x{:}\tau.\{\mathsf{val} = e, \mathsf{eff} = [\eta]\}$
$(\Phi_1 + \Phi_2)^\eta$	$:=$	$(\Phi_1 + \Phi_2)^{(\eta \top \bot)}$			
$\forall \iota.\tau$	$:=$	$\forall \alpha_\iota.\mathsf{eff}\, \alpha_\iota \to \tau$	$\lambda \iota.e$	$:=$	$\lambda \alpha_\iota.\lambda x_\iota{:}(\mathsf{eff}\, \alpha_\iota).e$
$\exists \iota.\tau$	$:=$	$\exists \alpha_\iota.\{\mathsf{eff} : \mathsf{eff}\, \alpha_\iota, \mathsf{val} : \tau\}$	$\mathsf{pack}\ \langle \eta, e\rangle$	$:=$	$\mathsf{pack}\ \langle \eta, \{\mathsf{eff} = \eta, \mathsf{val} = e\}\rangle$
eff	$:=$	$\lambda \alpha{:}\mathsf{Eff}.(\{\} + \{\})^{(\alpha \top \bot)}$			
$\mathtt{I}$	$:=$	$\lambda \alpha_1 \alpha_2.\alpha_1$	$\mathtt{I}$	$:=$	$\mathsf{inl}\ \{\}$
$\mathtt{P}$	$:=$	$\lambda \alpha_1 \alpha_2.\alpha_2$	$\mathtt{P}$	$:=$	$\mathsf{inr}\ \{\}$
ι	$:=$	α_ι	ι	$:=$	x_ι
$\eta_1 \vee \eta_2$	$:=$	$\lambda \alpha_1 \alpha_2.\eta_1\, \alpha_1\, (\eta_2\, \alpha_1\, \alpha_2)$	$\eta_1 \vee \eta_2$	$:=$	$\mathsf{case}\ \eta_1\ \mathsf{of}\ \mathsf{inl}\ x.\mathtt{I} \mid \mathsf{inr}\ x.$ $\mathsf{case}\ \eta_2\ \mathsf{of}\ \mathsf{inl}\ x.\mathtt{I} \mid \mathsf{inr}\ x.\mathtt{P}$

Fig. 8. Encoding of Semantic Types in F_ω

Not all the forms in that grammar are unadorned F_ω types, however. Some are auxiliary forms that are definable as syntactic sugar. Figure 8 shows how they can be encoded, along with respective term-level constructs for defining the evidence terms of the elaboration.

Several tricks in the elaboration are new relative to plain 1ML:

- Computation types are represented using the indexed binary sums introduced in the previous section. Their index is (the desugaring of) an effect type applied to $\top$ and $\bot$.
- To this end, effect types are represented as type constructors representing a simple Church-style encoding of Booleans. The kind of effect types hence is $\mathsf{Eff} = \Omega \to \Omega \to \Omega$, and their application behaves like a conditional.
- Effects also need to be reified on the term level, however, in order to enable reflection as needed in rule Sr. We use a binary sum (over units) for that purpose that is indexed by the corresponding type-level effect encoding – terms of type $\mathsf{eff}\ \eta$ are term-level encodings of effect type η.
- Since effects η can contain effect variables ι, this in turn requires those to be represented on both the type and term level. We use the trick of "twinning" each effect variable ι in the environment Γ as both a type variable α_ι and a term variable x_ι, assuming appropriate namespace injections. Likewise, every abstraction and quantification over effect variables consistently happens on both levels.

These encodings make sense, because the following consistency properties hold (note how we overload effects η as notation for both types and terms):

Proposition 1 (Derivable Rules for Effect Encodings).
Let Γ be a well-formed System F_ω environment.

1. $\Gamma \vdash \mathtt{I} : \mathsf{Eff}$.
2. $\Gamma \vdash \mathtt{P} : \mathsf{Eff}$.
3. *If* $\iota \in \Gamma$, *then* $\Gamma \vdash \alpha_\iota : \mathsf{Eff}$.
4. *If* $\Gamma \vdash \eta_1 : \mathsf{Eff}$ *and* $\Gamma \vdash \eta_2 : \mathsf{Eff}$, *then* $\Gamma \vdash \eta_1 \vee \eta_2 : \mathsf{Eff}$
5. $\Gamma \vdash \mathtt{I} : \mathsf{eff}\ \mathtt{I}$.
6. $\Gamma \vdash \mathtt{P} : \mathsf{eff}\ \mathtt{P}$.
7. *If* $\iota \in \Gamma$, *then* $\Gamma \vdash x_\iota : \mathsf{eff}\ \alpha_\iota$.
8. *If* $\Gamma \vdash \eta_1 : \mathsf{eff}\ \eta_1$ *and* $\Gamma \vdash \eta_2 : \mathsf{eff}\ \eta_2$, *then* $\Gamma \vdash \eta_1 \vee \eta_2 : \mathsf{eff}\ (\eta_1 \vee \eta_2)$.

Here, we write "$\iota \in \Gamma$ as a shorthand for "$\Gamma(\alpha_\iota) = \mathsf{Eff} \wedge \Gamma(x_\iota) = \mathsf{eff}\ \alpha_\iota$", in correspondence to the twinning for effect variables explained above.

With the notation $\mathsf{Mon}_{\overline{\kappa}}$ to denote the kind of a monadic computation constructor mentioning abstract types of kinds $\overline{\kappa}$, similar consistency properties can be shown for the polymonad notation from Figure 5:

Proposition 2 (Derivable Rules for Polymonads).
Let Γ be a well-formed System F_ω environment.

1. *If* $\overline{\Gamma, \overline{\alpha} \vdash \pi : \kappa}$, *then* $\Gamma \vdash \exists \overline{\alpha}[\overline{\pi}] : \mathsf{Mon}_{\overline{\kappa}}$.
2. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}}$ *and* $\Gamma \vdash \eta : \mathsf{Eff}$,
   then $\Gamma \vdash (M_1 + M_2)^\eta : \mathsf{Mon}_{\overline{\kappa}}$.
3. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}_1}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}_2}$, *then* $\Gamma \vdash M_1 M_2 : \mathsf{Mon}_{\overline{\kappa}_1 \overline{\kappa}_2}$.
4. *If* $\Gamma \vdash M : \mathsf{Mon}_{\overline{\kappa}_\alpha}$ *and* $\Gamma \vdash e_1 : M \lambda \overline{\alpha}.\tau_1$ *and* $\Gamma, \overline{\alpha}, x{:}\tau_1 \vdash e_2 : \tau_2$,
   then $\Gamma \vdash (\mathsf{do}_M\ \overline{\alpha}, x \leftarrow e_1\ \mathsf{in}\ e_2) : M \lambda \overline{\alpha}.\tau_2$.
5. *If* $\Gamma \vdash M_1 : \mathsf{Mon}_{\overline{\kappa}_1}$ *and* $\Gamma \vdash M_2 : \mathsf{Mon}_{\overline{\kappa}_2}$ *and* $\Gamma \vdash e : M_1 \lambda \overline{\alpha}_1 . M_2 \lambda \overline{\alpha}_2 . \tau$,
   then $\Gamma \vdash \mathsf{join}_{M_1 M_2}\ e : M_1 M_2 \lambda \overline{\alpha}_1 \overline{\alpha}_2 . \tau$.

Note that our use of do-notation is a slight abuse (if you're coming from Haskell, anyway): it actually is a map, not a monadic bind – both are instances of polymonadic binds, however.

A.3 Meta-Theory

With the previous propositions we can verify that elaboration is correct:

Proposition 3 (Correctness of Elaboration).
Let Γ be a well-formed F_ω environment.

1. *If* $\Gamma \vdash T/D \rightsquigarrow \Xi$, *then* $\Gamma \vdash \Xi : \Omega$.
2. *If* $\Gamma \vdash F \rightsquigarrow \eta$, *then* $\Gamma \vdash \eta : \mathsf{Eff}$.
3. *If* $\Gamma \vdash E/B :_\eta \Phi \rightsquigarrow e$, *then* $\Gamma \vdash e : \Phi$ *and* $\Gamma \vdash \eta : \mathsf{Eff}$.
   Furthermore, if $\eta = \mathtt{P}$ *then* $\Phi\,! = \Sigma$.
4. *If* $\Gamma \vdash \Phi' \leq_{\overline{\alpha \overline{\alpha}'}} \Phi \rightsquigarrow \delta; f$ *and* $\Gamma \vdash \Phi' : \Omega$ *and* $\Gamma, \overline{\alpha} \vdash \Phi : \Omega$, *then* $\mathrm{dom}(\delta) = \overline{\alpha}$ *and* $\Gamma \vdash \delta : \Gamma, \overline{\alpha}$ *and* $\Gamma \vdash f : \Phi' \to \delta\Phi$.

Together with the standard soundness result for F_ω we can tell that the extension of $1\mathrm{ML}_{\mathrm{ex}}$ is still sound:

Theorem 3 (Soundness of $1\mathrm{ML}_{\mathrm{ex}}$ with Effect Polymorphism).
If $\cdot \vdash E : \Phi \rightsquigarrow e$, *then either* $e \uparrow$ *or* $e \hookrightarrow^* v$ *such that* $\cdot \vdash v : \Phi$ *and* v *is a value.*