Reflections on existential types

Jonathan Sterling

October 3, 2022

Abstract

Existential types are reconstructed in terms of *small reflective subuniverses* and dependent sums. The folklore decomposition detailed here gives rise to a particularly simple account of first-class modules as a mode of use of traditional second-class modules in connection with the modal operator induced by a reflective subuniverse, leading to a semantic justification for the rules of first-class modules in languages like OCaml and MoscowML. Additionally, we expose several constructions that give rise to semantic models of ML-style programming languages with both first-class modules and realistic computational effects, culminating in a model that accommodates higher-order first-class recursive modules *and* higher-order store.

1 Introduction

Ever since the landmark paper *Abstract Types Have Existential Type* by Mitchell and Plotkin (1985), the existential type construct has occupied a central place in the study of programming languages, abstraction, and modularity. The syntactic side of existential types has long been fairly well-understood, as they are characterized by the now-standard rules depicted in Fig. 1 — notable for their *scoped* elimination rule, in which the value of the first component of an existential package is not allowed to "escape":

$$\frac{\exists\text{-ELIM} \qquad C : \mathbf{type} \qquad u : \exists_{x:A} Bx \qquad x : A, y : Bx \vdash vxy : C}{\mathbf{unpack}\ u\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ vxy : C}$$

In light of the Curry–Howard correspondence, these rules could be seen as a proof term assigment for the natural deduction rules of existential quantification in intuitionistic first-order logic. Such a perspective, though meritable, does not seem to lead to an explanation of much definitude; indeed, the competing Brouwer–Heyting–Kolmogorov correspondence would have suggested instead the rules of *dependent sums* (sometimes called "strong sums") rather than the rules for existentials (sometimes called "weak sums") that we have given above. If we are to find answers, therefore, they must come from semantics.

1.1 Existential types *vs.* dependent sums

The difference between existential types and dependent sums is located in the scoped elimination rule that we pointed out above. Existentials are counterposed

$$\frac{A : \mathbf{kind} \qquad x : A \vdash Bx : \mathbf{type}}{\exists_{x:A} Bx : \mathbf{type}}\ \exists\text{-FORM} \qquad \frac{u : A \qquad v : Bu}{\mathbf{pack}\,\langle u, v\rangle : \exists_{x:A} Bx}\ \exists\text{-INTRO}$$

$$\frac{C : \mathbf{type} \qquad u : \exists_{x:A} Bx \qquad x : A, y : Bx \vdash vxy : C}{\mathbf{unpack}\ u\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ vxy : C}\ \exists\text{-ELIM}$$

$$\frac{C : \mathbf{type} \qquad u : K \qquad v : Au \qquad x : A, y : Bx \vdash wxy : C}{\mathbf{unpack}\ (\mathbf{pack}\,\langle u, v\rangle)\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ wxy \equiv wuv : C}\ \exists\text{-COMP}$$

$$\frac{C : \mathbf{type} \qquad z : \exists_{x:A} Bx \vdash uz, vz : C \qquad x : A, y : Bx \vdash u\,(\mathbf{pack}\,\langle x, y\rangle) \equiv v\,(\mathbf{pack}\,\langle x, y\rangle) : C}{z : \exists_{x:A} Bx \vdash uz \equiv vz : C}\ \exists\text{-EXT}$$

Figure 1: The formation, introduction, elimination, computation, and (oft-overlooked) extensionality rules for existential types.

to *dependent sums*, which tend to be written using the $\sum$-symbol; the latter have more flexible unscoped *projection*-style eliminators:

$$\frac{u : \sum_{x:A} Bx}{u.1 : A}\ \sum\text{-ELIM}(1) \qquad \frac{u : \sum_{x:A} Bx}{u.2 : B\,(u.1)}\ \sum\text{-ELIM}(2)$$

In comparison to dependent sums, the existential type interface suffers from some obvious practical disadvantages when it comes to hierarchically structured modular programming, as MacQueen argued forcefully in 1986:

> It appears that when building a collection of interrelated abstractions, the lower the level of the abstraction, the wider the scope in which it must be opened. We thus have the traditional disadvantages of block structured languages where low-level facilities must be given the widest visibility. — MacQueen (1986, p. 279)

On the other hand, the disadvantage of dependent sums is that it is not consistent for $\sum_{\alpha:\mathbf{type}} A\alpha$ to be a type, should α range over *all* types, by a variant of Girard's paradox (Girard, 1972). It is for this reason that MacQueen (1986) and Mitchell and Plotkin (1985) refer to Martin-Löf's theory of *type universes* (Martin-Löf, 1975, 1979) in their respective discussions of dependent sums. Today we might employ the vocabulary of modern module systems (Stone and Harper, 2000) to summarize the situation as follows: the dependent sum of a type family indexed in a kind is not in fact a type, but a **module signature**. In particular, we

have a weaker (but consistent) formation rule for sums:

$$\frac{\sum\text{-FORM (RESTRICTED)} \qquad A : \mathbf{kind} \qquad x : A \vdash Bx : \mathbf{type}}{\sum_{x:A} Bx : \mathbf{sig}}$$

Such dependent sums that live in the "higher" universe of signatures are adequate and, in fact, ideal from the perspective of hierarchical and modular programming "in the large". On the other hand, because these sums are signatures and not types, it means that we cannot pass around their implementations dynamically, *e.g.* by choosing an implementation of a component based on the phase of the moon (Harper, 2016). To support such dynamism is exactly the motivation of the so-called *first-class modules*, which Harper (2016) has advocated to view as existential types in the context of an explicit phase-splitting; complementary to the perspective of *op. cit.*, we will develop here a **modal** construct that derives both existential types and first-class modules from second-class modules simultaneously.

2 Types, impredicativity, and first-class modules

2.1 Introduction to reflective subuniverses

Our analysis begins with the concept of a *reflective subuniverse*, and the similarity of its rules to those of existentials. Although reflective subuniverses were first made explicit in category theory and homotopy type theory (Rijke et al., 2020; Univalent Foundations Program, 2013), they had already appeared covertly within the classical programming languages literature by 1999, as the *protection monads* of Abadi et al.'s *dependency core calculus*.

From a syntactic point of view, a reflective subuniverse of a given universe $\mathcal{U}$ is specified by an additional form of judgment $A\ \mathit{modal}$ such that every type $A : \mathcal{U}$ has a "best approximation" $\bigcirc A$ as a modal type; unfurling what this means, we have a type operator $\bigcirc$ equipped with the following structure:

$$\frac{\bigcirc\text{-FORM} \quad A : \mathcal{U}}{\bigcirc A\ \mathit{modal}} \qquad \frac{\bigcirc\text{-MODAL} \quad a : A}{\eta_A a : \bigcirc A} \qquad \frac{\bigcirc\text{-ELIM} \quad C\ \mathit{modal} \qquad u : \bigcirc A \qquad x : A \vdash vx : C}{\mathbf{bind}\ x = u\ \mathbf{in}\ vx : C}$$

We additionally have a computation rule and extensionality rule for **bind**:

$$\frac{\bigcirc\text{-COMP} \quad C\ \mathit{modal} \qquad a : A \qquad x : A \vdash vx : C}{\mathbf{bind}\ x = \eta_A a\ \mathbf{in}\ vx \equiv va : C}$$

$$\frac{\bigcirc\text{-EXT} \quad C\ \mathit{modal} \qquad x : \bigcirc A \vdash ux, vx : C \qquad x : A \vdash u\,(\eta_A x) \equiv v\,(\eta_A x) : C}{z : \bigcirc A \vdash uz \equiv vz : C}$$

In a reflective subuniverse as described above, the modal operator $\bigcirc$ is called the *reflector* and a type $\bigcirc A$ is called the *reflection* of the type A.

Remark 1 (Comparison to monads)**.** The rules described here are similar to those of Moggi's monadic metalanguage, with some crucial differences. First of all, the **bind** rule targets an arbitrary modal type rather than a type of the form $\bigcirc B$; second of all, the extensionality law that we have imposed is not valid for an arbitrary monad. It follows from the rules above that $\bigcirc$ is a strong monad on $\mathcal{U}$, in fact an *idempotent* one in the sense that $\mu : \bigcirc\bigcirc \to \bigcirc$ is a natural isomorphism.

Remark 2 (Dependency core calculus)**.** The dependency core calculus (DCC) of Abadi et al. (1999) contains exactly the judgmental structure and rules that we have described above, parameterized in a set of security levels $\mathcal{L}$:

1. Our $A\ \mathit{modal}$ judgment corresponds to DCC's A is protected at ℓ.
2. Our reflector $\bigcirc$ corresponds to DCC's T_ℓ.

When first encountering the rules of DCC, it is a rite of passage among programming language theorists to note their deviation from the rules of monads and try to "fix" them (see Choudhury (2022) for a recent and creative example). Nonetheless, these rules are exactly the correct rules of a metalanguage for reflective subuniverses rather than arbitrary strong monads.

We will say that a type $A : \mathcal{U}$ is *essentially modal* when the unit map $\eta_A : A \to \bigcirc A$ is an isomorphism up to β/η-equivalence, *i.e.* it has a left and a right inverse. Of course, any modal type is essentially modal, but the collection of essentially modal types has some remarkable additional closure properties:

- If A and B are essentially modal, then $A \times B$ is essentially modal.
- If B is essentially modal, then $A \to B$ is essentially modal.
- If B is essentially modal and A is a retract of B, then A is essentially modal.[1]

The above justifies adding additional rules for judgmentally modal types, as Abadi et al. (1999) have done for the sake of convenience:

$$\frac{A\ \mathit{modal} \qquad B\ \mathit{modal}}{A \times B\ \mathit{modal}} \qquad\qquad \frac{A : \mathcal{U} \qquad B\ \mathit{modal}}{A \to B\ \mathit{modal}}$$

Note that adding such rules does not change the expressive power of the language, but it can make it more convenient to use.

2.2 Existential types in a *small* reflective subuniverse

We recall both the elimination rule for existential types and the elimination rule for the reflection operator of a reflective subuniverse:

$$\frac{\begin{array}{l}\exists\text{-ELIM} \\ C : \mathbf{type} \qquad u : \exists_{x:A} Bx \qquad x : A, y : Bx \vdash vxy : C\end{array}}{\mathbf{unpack}\ u\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ vxy : C}$$

$$\frac{\begin{array}{l}\bigcirc\text{-ELIM} \\ C\ \mathit{modal} \qquad u : \bigcirc A \qquad x : A \vdash vx : C\end{array}}{\mathbf{bind}\ x = u\ \mathbf{in}\ vx : C}$$

[1] A type A is called a *retract* of B when there exists a section-retraction pair $(s : A \to B, r : B \to A)$, *i.e.* we have $r \circ s = \mathsf{id}_A$.

There is a superficial (syntactical) similarity in the structure of both rules. Let us assume that we have a universe of signatures **sig**, as well as two subuniverses **type**, **kind** $\subseteq$ **sig** such that **type** : **kind**; this is roughly the configuration of ML-family programming languages. Now *additionally* assume that **type** is a reflective subuniverse of **sig**; as **type** is already assumed to be an element of **sig**, this amounts to saying that **type** is a "small" reflective subuniverse. Unraveling these assumptions, we have the following rules, as well as the computation and extensionality rules that we specified earlier:

$$\bigcirc\text{-FORM}\ \frac{A : \mathbf{sig}}{\bigcirc A : \mathbf{type}} \qquad \mathbf{type}\text{-MODAL}\ \frac{A : \mathbf{type}}{A\ \mathit{modal}} \qquad \bigcirc\text{-INTRO}\ \frac{a : A}{\eta_A a : \bigcirc A} \qquad \bigcirc\text{-ELIM}\ \frac{C\ \mathit{modal} \quad u : \bigcirc A \quad x : A \vdash vx : C}{\mathbf{bind}\ x = u\ \mathbf{in}\ vx : C}$$

Now suppose in addition that **sig** is closed under dependent sums of families of types indexed in a kind:

$$\sum\text{-FORM (RESTRICTED)}\ \frac{A : \mathbf{kind} \quad x : A \vdash Bx : \mathbf{type}}{\sum_{x:A} Bx : \mathbf{sig}} \qquad \sum\text{-INTRO}\ \frac{u : A \quad v : Bu}{(u, v) : \sum_{x:A} Bx} \qquad \sum\text{-ELIM(1)}\ \frac{u : \sum_{x:A} Bx}{u.1 : A}$$

$$\sum\text{-ELIM(2)}\ \frac{u : \sum_{x:A} Bx}{u.2 : B\,(u.1)} \qquad \sum\text{-COMP(1)}\ \frac{u : A \quad v : Bu}{(u, v).1 \equiv u : A} \qquad \sum\text{-COMP(2)}\ \frac{u : A \quad v : Bu}{(u, v).2 \equiv v : Bu} \qquad \sum\text{-EXT}\ \frac{u : \sum_{x:A} Bx}{u \equiv (u.1, u.2) : \sum_{x:A} Bx}$$

Then we observe that the existential type $\exists_{x:A} Bx$ could be represented as the "synthetic connective" $\bigcirc\sum_{x:A} Bx$ obtained by taking the reflection of the dependent sum. The introduction and elimination forms of existentials are defined likewise by macro expansion:

$$\begin{aligned} \exists_{x:A} Bx &\rightsquigarrow \bigcirc\textstyle\sum_{x:A} Bx \\ \mathbf{pack}\,\langle u, v\rangle &\rightsquigarrow \eta_{\sum_{x:A} Bx}(u, v) \\ \mathbf{unpack}\ u\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ vxy &\rightsquigarrow \mathbf{bind}\ z = u\ \mathbf{in}\ v\,(z.1)\,(z.2) \end{aligned}$$

Evidence for our encoding is obtained by deriving the rules of existentials from the rules of the reflection operator and the dependent sum:

$$\exists\text{-FORM}\ \cfrac{\cfrac{A : \mathbf{kind} \quad x : A \vdash Bx : \mathbf{type}}{\sum_{x:A} Bx : \mathbf{sig}}}{\underbrace{\bigcirc\textstyle\sum_{x:A} Bx : \mathbf{type}}_{\exists_{x:A} Bx\ :\ \mathbf{type}}} \qquad \exists\text{-INTRO}\ \cfrac{\cfrac{u : A \quad v : Bu}{(u, v) : \sum_{x:A} Bx}}{\underbrace{\eta_{\sum_{x:A} Bx}(u, v) : \bigcirc\textstyle\sum_{x:A} Bx}_{\mathbf{pack}\,\langle u, v\rangle\ :\ \exists_{x:A} Bx}}$$

$$\exists\text{-ELIM}\ \cfrac{C : \mathbf{type} \quad \underbrace{u : \exists_{x:A} Bx}_{u\ :\ \bigcirc\sum_{x:A} Bx} \quad \cfrac{x : A, y : Bx \vdash vxy : C \quad \cdots}{x : \sum_{x:A} Bx \vdash v\,(x.1)\,(x.2) : C}}{\underbrace{\mathbf{bind}\ x = u\ \mathbf{in}\ v\,(x.1)\,(x.2) : C}_{\mathbf{unpack}\ u\ \mathbf{as}\ \langle x, y\rangle\ \mathbf{in}\ vxy\ :\ C}}$$

It is also possible to verify that our encoding satisfies both the computation and extensionality rules of existential types.

$$\begin{aligned}
&\textbf{unpack}\ (\textbf{pack}\ \langle u, v\rangle)\ \textbf{as}\ \langle x, y\rangle\ \textbf{in}\ wxy \\
&\quad \rightsquigarrow \textbf{bind}\ x = \eta_{\sum_{x:A} Bx}(u, v)\ \textbf{in}\ w\,(x.1)\,(x.2) \\
&\quad \equiv w\,(u, v).1\,(u, v).2 \\
&\quad \equiv wuv
\end{aligned}$$

For extensionality, we assume $C : \textbf{type}$ and $z : \exists_{x:A} Bx \vdash uz, vz : C$ such that $x : A, y : Bx \vdash u\,(\textbf{pack}\,\langle x, y\rangle) \equiv v\,(\textbf{pack}\,\langle x, y\rangle) : C$ to check that for each $w : \exists_{x:A} Bx$ we have $uw \equiv vw : C$. Under our encoding, this follows immediately from the extensionality laws for $\bigcirc$ and dependent sums.

2.3 Universal types in a small reflective subuniverse

Just as existential types can be obtained by taking the reflection of the (predicative) dependent sum of signatures, we observe here that universal types arise in the same way from (predicative) dependent products of signatures. In particular, we suppose that **sig** is closed under dependent products of families of types indexed in a kind, as below:

$$\frac{A : \textbf{kind} \qquad x : A \vdash Bx : \textbf{type}}{\prod_{x:A} Bx : \textbf{sig}}\ \prod\text{-FORM (RESTRICTED)} \qquad \frac{x : A \vdash ux : Bx}{\lambda x.ux : \prod_{x:A} Bx}\ \prod\text{-INTRO} \qquad \frac{u : \prod_{x:A} Bx \qquad v : A}{u \cdot v : Bv}\ \prod\text{-ELIM}$$

$$\frac{x : A \vdash ux : Bx \qquad v : A}{(\lambda x.ux) \cdot v \equiv uv : Bv}\ \prod\text{-COMP} \qquad \frac{u : \prod_{x:A} Bx}{u \equiv \lambda x.u \cdot x : \prod_{x:A} Bx}\ \prod\text{-EXT}$$

Then we may close **type** under universal types satisfying the following rules:

$$\frac{A : \textbf{kind} \qquad x : A \vdash Bx : \textbf{type}}{\forall_{x:A} Bx : \textbf{type}}\ \forall\text{-FORM} \qquad \frac{x : A \vdash ux : Bx}{\Lambda x.ux : \forall_{x:A} Bx}\ \forall\text{-INTRO} \qquad \frac{u : \forall_{x:A} Bx \qquad v : A}{u[v] : Bv}\ \forall\text{-ELIM}$$

$$\frac{x : A \vdash ux : Bx \qquad v : A}{(\Lambda x.ux)[v] \equiv uv : Bv}\ \forall\text{-COMP} \qquad \frac{u : \forall_{x:A} Bx}{u \equiv \Lambda x.u[x] : \forall_{x:A} Bx}\ \forall\text{-EXT}$$

In particular, we define universal types via the following macro-expansions:

$$\begin{aligned}
\forall_{x:A} Bx &\rightsquigarrow \bigcirc \prod_{x:A} Bx \\
\Lambda x.ux &\rightsquigarrow \eta_{\prod_{x:A} Bx}(\lambda x.ux) \\
u[v] &\rightsquigarrow \textbf{bind}\ z = u\ \textbf{in}\ z \cdot v
\end{aligned}$$

These macro-expansions do validate the rules of universal types:

$$\dfrac{\dfrac{A : \mathbf{kind} \qquad x : A \vdash Bx : \mathbf{type}}{\prod_{x:A} Bx : \mathbf{sig}}}{\dfrac{\bigcirc \prod_{x:A} Bx : \mathbf{type}}{\forall_{x:A} Bx : \mathbf{type}}}\ \forall\text{-FORM} \qquad \dfrac{\dfrac{\dfrac{x : A \vdash ux : Bx}{\lambda x.ux : \prod_{x:A} Bx}}{\eta_{\prod_{x:A} Bx}(\lambda x.ux) : \bigcirc \prod_{x:A} Bx}}{\Lambda x.ux : \forall_{x:A} Bx}\ \forall\text{-INTRO}$$

$$\dfrac{\dfrac{x : A \vdash Bx : \mathbf{type} \qquad v : A}{\dfrac{Bv : \mathbf{type}}{Bv\ \mathit{modal}}} \qquad \dfrac{u : \forall_{x:A} Bx}{u : \bigcirc \prod_{x:A} Bx} \qquad \dfrac{\overline{z : \prod_{x:A} Bx \vdash z : \prod_{x:A} Bx} \qquad v : A}{z : \prod_{x:A} Bx \vdash z \cdot v : Bv}}{\dfrac{\mathbf{bind}\ z = u\ \mathbf{in}\ z \cdot v : Bv}{u[v] : Bv}}\ \forall\text{-ELIM}$$

Next we check the computation rule:

$$\begin{aligned}(\Lambda x.ux)[v] &\rightsquigarrow \mathbf{bind}\ z = \eta_{\prod_{x:A} Bx}(\lambda x.ux)\ \mathbf{in}\ z \cdot v \\ &\equiv (\lambda x.ux) \cdot v \\ &\equiv uv\end{aligned}$$

For the extensionality law, we fix $u : \bigcirc \prod_{x:A} Bx$ to check that $u \equiv \Lambda x.u[x]$ under our encoding. By the extensionality law for $\bigcirc$, to check that $u \equiv \Lambda x.u[x]$ for all u is the same as to check that for all $f : \prod_{x:A} Bx$ we have $\eta_{\prod_{x:A} Bx} f \equiv \Lambda x.(\eta_{\prod_{x:A} Bx} f)[x]$. We proceed by macro-expansion and computation:

$$\begin{aligned}\Lambda x.(\eta_{\prod_{x:A} Bx} f)[x] &\rightsquigarrow \eta_{\prod_{x:A} Bx}(\lambda x.\mathbf{bind}\ z = \eta_{\prod_{x:A} Bx} f\ \mathbf{in}\ z \cdot x) \\ &\equiv \eta_{\prod_{x:A} Bx}(\lambda x.f \cdot x) \\ &\equiv \eta_{\prod_{x:A} Bx} f\end{aligned}$$

2.4 Modal decomposition of first-class modules

In Section 2.2 we have only explicitly assumed the closure of **sig** under dependent sums of the form $\sum_{x:A} Bx$ when $A : \mathbf{kind}$ and $x : A \vdash Bx : \mathbf{type}$. This restriction echoes the explicit phase splitting of Harper et al. (1990), but it is by no means forced. Sterling and Harper (2021) have advocated to replace explicit phase splittings (in which every signature is *literally* the dependent sum of a family of types indexed in a kind) with a *synthetic* phase splitting, in which the type theory of modules is simply Martin-Löf type theory extended by a (modal) phase distinction between compiletime and runtime data. In *op. cit.*, module signatures are naturally closed under arbitrary dependent sums and products.

The modal reflection $\bigcirc$ can be thought of as a connective that singlehandedly takes a module signature to the type of *runtime packages* of that signature. In the case of a signature of the form $\sum_{x:A} Bx$ where A is a kind and each Bx is a type, the package type $\bigcirc \sum_{x:A} Bx$ encodes the existential type $\exists_{x:A} Bx$ as we have seen in Section 2.2, but it is of course possible to apply the reflection to any signature we want regardless of the way it was formed: the result is a form of *first-class modules*.

Remark 3. The modal decomposition of the phase distinction by Sterling and Harper (2021) can be thought of as the first stage of a longer and more involved process of recasting *every* last peculiarity of type theories for module systems as a completely orthodox modal extension of Martin-Löf type theory. The present paper tackles a different aspect of modules, namely the relationship between types, modules, and impredicative polymorphism. Our approach can be viewed as a more modular alternative to the F-ing Modules tradition (Rossberg et al., 2014), in which the peculiarities of module systems are reduced to the type structure of System F via a highly non-trivial syntactic translation.

2.4.1 Existential and universal quantification over *modules*

We now assume that signatures are closed under arbitrary dependent sums and products, lifting the old restriction to families of types indexed in a kind:

$$\frac{A : \mathbf{sig} \qquad x : A \vdash Bx : \mathbf{sig}}{\sum_{x:A} Bx : \mathbf{sig}}\ \sum\text{-FORM} \qquad \frac{A : \mathbf{sig} \qquad x : A \vdash Bx : \mathbf{sig}}{\prod_{x:A} Bx : \mathbf{sig}}\ \prod\text{-FORM}$$

With these more flexible rules in hand, we may use arguments identical to those of Sections 2.2 and 2.3 to establish the closure of **type** under existential and universal types that take their indices in signatures rather than only kinds:

$$\frac{A : \mathbf{sig} \qquad \dfrac{x : A \vdash Bx : \mathbf{type}}{x : A \vdash Bx : \mathbf{sig}}}{\dfrac{\sum_{x:A} Bx : \mathbf{sig}}{\bigcirc\sum_{x:A} Bx : \mathbf{type}}}\ \exists\text{-FORM (EXTENDED)} \qquad \frac{A : \mathbf{sig} \qquad \dfrac{x : A \vdash Bx : \mathbf{type}}{x : A \vdash Bx : \mathbf{sig}}}{\dfrac{\prod_{x:A} Bx : \mathbf{sig}}{\bigcirc\prod_{x:A} Bx : \mathbf{type}}}\ \forall\text{-FORM (EXTENDED)}$$

$$\exists_{x:A} Bx : \mathbf{type} \qquad \forall_{x:A} Bx : \mathbf{type}$$

A consequence of the closure under types of universal quantification over elements of a given *signature* is that we may write programs that take modules as arguments without resorting (directly) to module functors; of course, unraveling the encoding, such programs are ultimately packages of module functors. We explore a concrete example of this phenomenon in Section 2.4.2

2.4.2 Programs that take modules as arguments

In Standard ML and languages with similar "second-class" module systems, a program that takes a module as an argument must be implemented as a *module functor*; thus such a program has a signature rather than a type, and it cannot be (*e.g.*) stored in the heap, *etc.* To give an example, suppose we have a module signature for an output interface, where α `eff` is some ambient effect monad:

```
signature OUTPUT_IO =
sig
  type outstream
  val openOut : filepath -> outstream eff
  val write : oustream * string -> unit eff
  (* ... *)
end
```

```
signature DATABASE_IO =
sig
  type db
  val openDb : filepath -> db eff
  val closeDb : db -> unit eff
  val execSql : db -> sql -> callback -> int eff
  (* ... *)
end

structure MockDatabaseIO : DATABASE_IO = (*...*)
structure RealDatabaseIO : DATABASE_IO = (*...*)
```

Figure 2: A module signature for a database interface, together with two implementations.

We can imagine a program that opens a file and writes "Hello world" to it, and returns the open handle. Such a program in Standard ML must be written as a module functor, taking a module `O : OUTPUT_IO` to a new module that implements a function returning `O.outstream eff`:

```
functor Hello (O : OUTPUT_IO) :
sig val hello : filepath -> O.outstream eff end =
struct
  fun hello path =
    Eff.bind (O.openOut path) (fn h =>
    Eff.bind (O.write (h, "Hello world")) (fn _ =>
    Eff.return h))
end
```

Given our encoding of program-level universal quantification over *modules* from Section 2.4.1, the `hello` program could be written much more succinctly without resorting to module functors:

```
fun hello (O : OUTPUT_IO) (path : filepath) : O.outstream eff =
  Eff.bind (O.openOut path) (fn h =>
  Eff.bind (O.write (h, "Hello world")) (fn _ =>
  Eff.return h))
```

2.4.3 Programs that return modules

In Section 2.4.2 we have used the fact that reflective subuniverses are closed under dependent products to simplify the definition of programs that take entire modules as arguments. On the other hand, it is also possible to define programs that *return* entire modules, perhaps even computed using runtime inputs and/or computational effects.

Consider for example the case of an interface to a database, whose signature we depict in Fig. 2; can we write a program that will return either a *mocked* or *real* database interface depending on the value of an environment variable? It so happens that such a program can easily be typed and implemented using

the reflection operator ○, turning the module signature DATABASE_IO into a first-class module type ○ DATABASE_IO:

```
val chooseDbInterface : (○ DATABASE_IO) eff =
  Eff.bind (getEnvFlag "USE_MOCK_DB")
  (fn true =>
      Eff.return (η_DATABASE_IO MockDatabaseIO)
    | false =>
      Eff.return (η_DATABASE_IO RealDatabaseIO))
```

Then a function that opens and closes a given database, using either the mock or the real database interface, can be written using a combination of the **bind** construct for ○ and the monadic bind of the ambient effect monad:

```
val main : unit eff =
  Eff.bind chooseDbInterface (fn dbPkg =>
  bind DbInterface = dbPkg in
  Eff.bind (DbInterface.openDb "mydb.sql") (fn db =>
  DbInterface.closeDb db))
```

2.4.4 Modules with initialization effects

A programming pattern identical to the one described in Section 2.4.3 can be used to explain modules that exhibit initialization effects, *e.g.* binding an abstract type to a given runtime state. For instance, we can consider the interface of a monotone symbol table as follows:

```
signature SYMBOL_TABLE =
sig
  type symbol
  val fresh : symbol eff
  val symbolEq : symbol * symbol -> bool
end
```

To allocate a new symbol table, we define a function that returns a *package* of type ○ SYMBOL_TABLE under the effect monad:

```
val initSymbolTable : (○ SYMBOL_TABLE) eff =
  Eff.bind (Ref.new 0) (fn r =>
  Eff.return
   (η_SYMBOL_TABLE
    (struct
       type symbol = int
       val fresh =
         Eff.bind (Ref.get r) (fn i =>
         Eff.bind (Ref.set r (i + 1)) (fn _ =>
         Eff.ret i))
       val symbolEq = Int.eq
     end)))
```

2.4.5 Comparison to first-class modules in OCaml and MoscowML

Both OCaml and MoscowML feature a form of first-class modules, and their rules are similar in the broad strokes to those arising here through the encoding

This Paper	OCaml	MoscowML
$\bigcirc A$	`(module A)`	`[A]`
$\eta_A u$	`(module u : A)`	`[structure u as A]`
bind $x = u$	`module x = (val u : A)`	`structure x as A = u`

Table 1: A summary of the different notations for first-class modules compared with our modal account in terms of the reflective subuniverse.

of first-class modules as the *reflections* of second-class modules; in Table 1 we display a "Rosetta stone" that relates the different notations for first-class modules to the modal account given here. Our principled account of first-class modules in terms of a reflective subuniverse can be seen, therefore, as a theoretical justification for the rules of OCaml and MoscowML — which were chosen on practical rather than semantical grounds.

2.4.6 Comparison to first-class modules in 1ML

Rossberg (2018) has argued that the so-called "first-class" modules of OCaml and MoscowML should be referred to as *packaged modules* rather than first-class modules on the grounds that actual modules support a *type sharing* mechanism that is not expressible for the former without a detour through core-level polymorphism. The example of *op. cit.* involves a signature containing an abstract type, and a function that aims to take two implementations of that signature that share the same type component as in the following code:

```
signature S = sig type t (* ... *) end
fun example (X : S) (Y : S with type t = X.t) = (* ... *)
```

To encode the above in OCaml or MoscowML, it would be necessary take `X` and `Y` as *packaged* arguments, but then the dependency between the two packages would not be expressible; Rossberg (2018) explains that in systems like OCaml and MoscowML, one must instead use an explicit universal quantification over the type components of both modules and impose two type sharing constraints in order to simulate the dependency:

```
fun example
  (α : type)
  (X : S with type t = α)
  (Y : S with type t = α) =
  (* ... *)
```

1ML's elaboration process ultimately addresses this defect of OCaml and MoscowML's packaged modules, but we point out that our own account does not suffer from this defect in the first place: we have observed in Section 2.4.2 that the universe of types is *automatically* closed under dependent products of families over types indexed in a signature as soon as the signature layer is closed under these dependent products (this is a general fact about reflective subuniverses). Thus the troublesome example depicted at the beginning of this section is already expressible in our language without any arduous encoding. Of course, it follows immediately from our analysis that the limitation of OCaml

and MoscowML's packaged modules criticized by Rossberg (2018) is only a superficial one, *i.e.* a deficiency of syntax rather than of semantics.

3 Semantic analysis of first-class modules

So far we have investigated the purely syntactical aspects of existential types, first-class modules, and their derivation from (small) reflective subuniverses. Here we turn to semantics in order to justify to ourselves that such scenarios are not merely a syntax-induced fantasy.

3.1 A type theoretic metalanguage

The metalanguage for our semantic investigations is intensional type theory with dependent product types $\prod_{x:A} Bx$, dependent sum types $\sum_{x:A} Bx$, and identification types $u =_A v$ such that dependent product types satisfy the function extensionality principle in relation to the identification types. Although we use the vocabulary of univalent foundations (*e.g.* propositions, sets, *etc.*) we do not assume the univalence principle. Thus our results are compatible with the *usual* settings for models of programming languages, as well as the future ones that we anticipate in the world of univalent mathematics.

Definition 4 (Propositions and sets)**.** A type A is called a *proposition* when for every $x, y : A$ we may choose an element of $x =_A y$. A type A is called a *set* when for each $x, y : A$, the identification type $x =_A y$ is a proposition.

Definition 5 (Contractibility)**.** A type A is *contractible* when we have an element $a : A$ such that every other element $x : A$ can be identified with a. In other words, when we have an element of the type $\sum_{a:A} \prod_{x:A} x =_A a$.

Definition 6 (Fibers)**.** Given a function $f : A \to B$ and an element $b : B$, we will write $\mathsf{fib}_f b$ for the *fiber* of f at b defined to be the dependent sum $\mathsf{fib}_f b :\equiv \sum_{a:A} fa =_B b$.

Definition 7 (Embeddings and equivalences)**.** A function $f : A \to B$ is called an *embedding* when each of its fibers is a proposition; it is called an *equivalence* when each of its fibers is contractible.

Definition 8 (Universes)**.** In this paper, a universe is simply a type $\mathcal{U}$ equipped with a dependent type $A : \mathcal{U} \vdash \mathsf{El}_{\mathcal{U}} A$. As a notational abuse, for each $A : \mathcal{U}$ we will again abbreviate $\mathsf{El}_{\mathcal{U}} A$ by A.

We do *not* assume here that each universe is closed under the connectives of type theory, as it will be useful for us to be able to make such assumptions on a more granular level. Likewise, we do not assume that each universe is univalent.

Definition 9 (Closure under a type)**.** A universe $\mathcal{U}$ is said to be *closed under* a type A when there exists an element $\lfloor A \rfloor : \mathcal{U}$ and an equivalence $\alpha_A : \mathsf{El}_{\mathcal{U}} \lfloor A \rfloor \simeq A$. As an abuse of notation, we will simply write $A \in \mathcal{U}$ to assert that $\mathcal{U}$ is closed under A, in this scenario we will write A for $\lfloor A \rfloor$ and leave the equivalence $\alpha_A : \mathsf{El}_{\mathcal{U}} \lfloor A \rfloor \simeq A$ implicit.

Definition 10 (Reflection of a type)**.** A universe $\mathcal{U}$ is said to *reflect* a type A when there exists an element $A_{|\mathcal{U}} : \mathcal{U}$ and a function $\eta_A : A \to A_{|\mathcal{U}}$ such that for every $B : \mathcal{U}$ the map B^{η_A} determined by precomposition with η_A depicted below is an equivalence:

$$\begin{aligned} B^{\eta_A} &: \left(A_{|\mathcal{U}} \to B\right) \to (A \to B) \\ B^{\eta_A} f &= f \circ \eta_A \end{aligned} \tag{1}$$

We will write $\mathsf{rec}^B_{A|\mathcal{U}} : (A \to B) \to \left(A_{|\mathcal{U}} \to B\right)$ for the inverse to B^{η_A} so-determined; $A_{|\mathcal{U}}$ is called the *reflection* of A in $\mathcal{U}$, η_A is called its *unit*, and $\mathsf{rec}_{A|\mathcal{U}}$ is its *recursion principle*.

Note that the reflection of a type in a given universe is unique up to unique isomorphism when it exists. It is not difficult to see that a universe reflects any type A that it is closed under, setting $A_{|\mathcal{U}} :\equiv A$ and $\eta_A :\equiv \mathsf{id}_A$.

Definition 11 (Subuniverses)**.** A *subuniverse* $\mathcal{U} \subseteq \mathcal{V}$ is given by a predicate on $\mathcal{V}$ that sends each $A : \mathcal{V}$ to the proposition $(A \in \mathcal{U})$, which is moreover *replete* in the sense that when $f : A \to B$ is an equivalence where $A : \mathcal{V}$ and $B \in \mathcal{U}$, then $A \in \mathcal{U}$.

If $\mathcal{V}$ were assumed to be univalent, then any predicate on $\mathcal{V}$ would be replete in the sense of Definition 11 and thus give rise to a subuniverse.

Definition 12 (Reflective subuniverse)**.** A subuniverse $\mathcal{U} \subseteq \mathcal{V}$ is called *reflective* in $\mathcal{V}$ when every $A : \mathcal{V}$ is reflected in $\mathcal{U}$.

Notation 13. In the case of a reflective subuniverse $\mathcal{U} \subseteq \mathcal{V}$, we will write $\bigcirc : \mathcal{V} \to \mathcal{U}$ for the operation that sends each type $A : \mathcal{V}$ to its reflection $\bigcirc A :\equiv A_{|\mathcal{U}}$ in $\mathcal{U}$; this operation is called the *reflector*. In this scenario, we will often write $\mathcal{V}_\bigcirc \subseteq \mathcal{V}$ for the reflective subuniverse for which $\bigcirc$ is the reflector.

A reflective subuniverse $\mathcal{U}_\bigcirc \subseteq \mathcal{U}$ is closed under all *limit* type constructors that exist in $\mathcal{U}$; for instance, if for some $A, B : \mathcal{U}_\bigcirc$ we have $(A \times B) \in \mathcal{U}$, then it follows that $(A \times B) \in \mathcal{U}_\bigcirc$. This holds even for dependent products: for any $A : \mathcal{U}$ and $B : A \to \mathcal{U}_\bigcirc$ such that $\left(\prod_{x:A} Bx\right) \in \mathcal{U}$, we furthermore have $\left(\prod_{x:A} Bx\right) \in \mathcal{U}_\bigcirc$. Likewise, given $A : \mathcal{U}_\bigcirc$ and $a, b : A$, if it happens that $(a =_A b) \in \mathcal{U}$, then so shall we also have $(a =_A b) \in \mathcal{U}_\bigcirc$.

3.2 Small reflective subuniverses and impredicativity

Let $\mathcal{U}, \mathcal{V}$ be two universes such that $\mathcal{U} \in \mathcal{V}$ and $\mathcal{V}$ is closed under dependent products; note that we have not yet assumed that $\mathcal{U}$ is reflective in $\mathcal{V}$ nor even that $\mathcal{U} \subseteq \mathcal{V}$.

Definition 14 (Universal types)**.** The nested universe $\mathcal{U} \in \mathcal{V}$ has *universal types* when for all $A : \mathcal{V}$ and $B : A \to \mathcal{U}$, we have $\left(\prod_{x:A} Bx\right) \in \mathcal{U}$.

The following falls out directly from the results of Awodey et al. (2018).

Lemma 15 (Universal types *vs.* reflectivity)**.** *Suppose that $\mathcal{U} \in \mathcal{V}$ and $\mathcal{U} \subseteq \mathcal{V}$ such every $A : \mathcal{U}$ is a set and $\mathcal{U}$ is moreover closed under identification types. Then $\mathcal{U} \in \mathcal{V}$ has universal types if and only if $\mathcal{U}$ is reflective in $\mathcal{V}$.*

Remark 16. Further observations of Shulman (2017, 2018) concerning the splitting of idempotents in intensional type theory seem to suffice to generalize Lemma 15 to avoid the assumption that every $A : \mathcal{U}$ is a set.

4 Concrete models of first-class modules

In this section, we instantiate the results of the preceding sections to produce several concrete semantic models of first-class modules.

4.1 Models of total functional programming

For any small subuniverse $\mathcal{U} \in \mathcal{V}$, Lemma 15 establishes that closure under universal types is equivalent to $\mathcal{U}$ being reflective in $\mathcal{V}$. Thus in light of our characterization of first-class modules in terms of small reflective subuniverses (Section 2.4), if we seek models of first-class modules, it suffices to search for models of universal types.

Famously, Reynolds (1984) observed that ***"polymorphism is not set-theoretic"*** by showing that the "naïve" model of universal types in set theory, previously conjectured by Reynolds (1983), is ill-defined due to insurmountable size issues. Category theorists would have anticipated Reynolds' negative result, being aware of Freyd's 1964 observation in passing that any *complete* small category is necessarily a preorder (Freyd, 2003). Freyd's observation pertains specifically to categories that are defined using *sets* as raw materials, *i.e.* categories internal to the (large) category of sets. Thus it did not contradict the results of Freyd and Reynolds when Pitts proclaimed that ***"polymorphism is set-theoretic, constructively"*** (Pitts, 1987).

The meaning of Pitts' shocking slogan should be understood as follows: when "complete small category" is interpreted in a *different* category of set-like objects, it is possible to find a complete small category and, in particular, examples of non-trivial universes $\mathcal{U} \in \mathcal{V}$ closed under universal types. The observations of Pitts (1987) rely on the results of Hyland (1988), who showed that categories of assemblies contain non-trivial universes $\mathcal{U} \in \mathcal{V}$ closed under universal types.

Assumption 17. *We assume a* partial combinatory algebra $\mathbb{A}$*, which is a certain kind of computational model of untyped computation (van Oosten, 2008). Given $u, v \in \mathbb{A}$ we will write $u \cdot v$ for the* partial application *operator.*

For example, $\mathbb{A}$ could be the collection of Turing machines, or it could be the syntax of λ-calculus under its α/β-congruence, or it could be a universal domain after Scott. It is not necessary to understand the details of partial combinatory algebras in order to grasp the results of this section.

Definition 18. A *partial equivalence relation* is defined to be a relation $R \subseteq \mathbb{A} \times \mathbb{A}$ that is both symmetric and transitive. A *morphism* of partial equivalence relations $R \to S$ is given by an element $u \in \mathbb{A}$ such that for all $x\ R\ y$ we have both $u \cdot x, u \cdot y$ are defined and moreover $u \cdot x\ S\ u \cdot y$.

Definition 19. An *assembly* X is defined to be a set $\mathbf{\Gamma} X$ together with a family of non-empty subsets $\boldsymbol{E}_X : \mathbf{\Gamma} X \to \mathcal{P}_{ne}\mathbb{A}$ sending each element of $\mathbf{\Gamma} X$ to its collection of "realizers". A *morphism* of assemblies $X \to Y$ is given by a function $f : \mathbf{\Gamma} X \to \mathbf{\Gamma} Y$ for which there exists an element $u \in \mathbb{A}$ such that for each $x \in \mathbf{\Gamma} X$ and $v \in \boldsymbol{E}_X x$, $u \cdot v$ is defined and lies in $\boldsymbol{E}_Y(f\,x)$.

Definition 20. An assembly $X = (\mathbf{\Gamma} X, \boldsymbol{E}_X)$ is called a *modest set* when for each $x, y \in \mathbf{\Gamma} X$ if the intersection $\boldsymbol{E}_X x \cap \boldsymbol{E}_X y$ is inhabited, then $x = y$.

Every partial equivalence relation R gives rise to a modest set M_R: we take $\mathbf{\Gamma} M_R$ to be the quotient of $\{u \in \mathbb{A} \mid u\ R\ u\}$ by the (total) equivalence relation induced by R. Then $\boldsymbol{E}_{M_R} x$ is defined to be the equivalence class $\{v \in \mathbb{A} \mid [v]_R = x\}$. It turns out that the mapping from partial equivalence relations to modest sets is an equivalence of categories, whence we obtain a full embedding of PERs in assemblies.

Construction 21 (Universes of assemblies (Luo, 1994))**.** For each universe V of the ambient set theory, we may define an assembly $\mathcal{V}$ of V-small assemblies by taking $\mathbf{\Gamma}\mathcal{V}$ to be the set of assemblies X such that $\mathbf{\Gamma} X \in V$, and taking $\boldsymbol{E}_{\mathcal{V}} X$ to be all of $\mathbb{A}$. The dependent assembly $X : \mathcal{V} \vdash \mathsf{El}_{\mathcal{V}} X$ is defined to be X itself.

Construction 22 (Universe of partial equivalence relations)**.** We define an assembly $\mathcal{U}$ of all partial equivalence relations by taking $\mathbf{\Gamma}\mathcal{U}$ to be the set of partial equivalence relations and $\boldsymbol{E}_{\mathcal{U}} R$ to be all of $\mathbb{A}$. The dependent assembly $R : \mathcal{U} \vdash \mathsf{El}_{\mathcal{U}} R$ is defined to be the modest set M_R determined by R.

Fact 23. *It follows that $\mathcal{U} \in \mathcal{V}$ and $\mathcal{U} \subseteq \mathcal{V}$ for any $\mathcal{V}$ built according to Construction 21. Moreover, $\mathcal{U} \in \mathcal{V}$ is closed under universal types.*

Corollary 24. *As $\mathcal{U}$ is closed under equality types, it follows from Lemma 15 that $\mathcal{U} \in \mathcal{V}$ is a small reflective subuniverse and thus a model of first-class modules.*

Summary. Module signatures are modeled in $[\![\mathbf{sig}]\!] :\equiv \mathcal{V}$ and types are modeled in $[\![\mathbf{type}]\!] :\equiv \mathcal{U} \in \mathcal{V}$. First-class modules are accommodated using the reflection $\bigcirc : \mathcal{V} \to \mathcal{U}$ which we have by Corollary 24.

4.2 Models with recursive types

The results of Section 4.1 suffice to produce semantic models of first-class modules in the setting of *total functional programming*; in this section, we refine the model of Section 4.1 to account for general recursion using the methods of **synthetic domain theory** (Hyland, 1991).

In the category of assemblies, we have a (univalent) universe $\mathbb{P}$ consisting of *all* $\neg\neg$-closed propositions; we can isolate the subuniverse $\Sigma \subseteq \mathbb{P}$ spanned by *recursively enumerable* propositions (Rosolini, 1986), which we will use to define a subuniverse of $\mathcal{U}$ whose types support general recursion. It so happens that Σ lies in $\mathcal{U}$, so we may define a lifting monad $L : \mathcal{U} \to \mathcal{U}$ for Σ-partial elements:

$$LA :\equiv \textstyle\sum_{p:\Sigma}(p \to A)$$

Let F, I be the *final coalgebra* and *initial algebra* respectively for the endofunctor $L : \mathcal{U} \to \mathcal{U}$; these can be computed as (co)inductive types. We should think of I as a type of "generalized" natural numbers, and F as the extension of I by an infinite element. There is a canonical embedding $\iota : \mathsf{I} \hookrightarrow \mathsf{F}$ that sends every "finite generalized natural number" to itself, induced via Lambek's lemma by either the universal property of the final coalgebra or the initial algebra.

Definition 25 (Longley (1995))**.** A type $A : \mathcal{U}$ is called *complete* if the precomposition map $A^\iota : A^{\mathsf{F}} \to A^{\mathsf{I}}$ is an equivalence. The type A is called *well-complete* when the lifted type LA is complete.

Fact 26. *The subuniverse $\mathcal{U}_{wc} \subseteq \mathcal{U}$ of well-complete PERs is reflective in $\mathcal{U}$, and moreover closed under L.*

Corollary 27. *As $\mathcal{U} \in \mathcal{V}$ is reflective and we have $\mathcal{U}_{wc} \in \mathcal{V}$, the universe of well-complete PERs is a small reflective subuniverse of each $\mathcal{V}$.*

Well-completeness is a synthetic analogue of closure under suprema of ω-chains. In particular, the lifting LA of any $A : \mathcal{U}_{wc}$ is closed under general recursive definitions; furthemore, it even happens that $\mathcal{U}_{wc}$ is closed under recursive types.

Summary. Module signatures are modeled in $[\![\mathbf{sig}]\!] :\equiv \mathcal{V}$ and types are modeled in $[\![\mathbf{type}]\!] :\equiv \mathcal{U}_{wc} \in \mathcal{V}$. First-class modules are accommodated via the composite reflection $\mathcal{V} \to \mathcal{U} \to \mathcal{U}_{wc}$. Moreover, we can model an effect monad on **type** for general recursion, and we may even model recursive types of the form $\mu\alpha.A\alpha$ for any $\alpha : \mathbf{type} \vdash A\alpha : \mathbf{type}$.

4.3 Models with recursive *modules*

An alternative to the methods of synthetic domain theory discussed in Section 4.2 is given by *synthetic **guarded** domain theory* (Birkedal et al., 2011) or SGDT, whose standard model takes place in the *topos of trees*, which is presheaves on the poset ω of natural numbers with their standard order. SGDT centers around the use of type connective $\blacktriangleright$ satisfying the rules of an *applicative functor* in the sense of McBride and Paterson (2008) to stratify general recursive definitions in their finite unrollings. Palombi and Sterling (2022) have shown that all the main results of SGDT continue to apply when working in a different topos, such as a *realizability* topos into which the category of assemblies from Section 4.1 embeds.

The model is constructed by replacing the poset of natural numbers from set theory with the *internal* poset ω of natural numbers from the category of assemblies. Then we obtain a category of *internal presheaves of assemblies*, which is stratified by universes $\mathcal{V}$ each obtained by taking the Hofmann–Streicher lifting (Hofmann and Streicher, 1997) of the corresponding universe of ordinary assemblies. Then we have a subuniverse $\mathcal{U} \subseteq \mathcal{V}$ of internal presheaves of PERs, which (as before) lies in $\mathcal{V}$ and is moreover reflective in $\mathcal{V}$ as established by Sterling et al. (2022). Every universe is closed under a *guarded lifting monad*, which enables a form of general recursive definition as a computational effect — formally quite different from the lifting monad L from Section 4.2, but achieving much the same goal.

Remark 28 (Recursive modules)**.** In contrast to Section 4.2, even the higher universes $\mathcal{V}$ are closed under recursive types and recursive functions — thus we model not only *recursively defined modules* in the sense of Dreyer (2007) but also recursively defined module signatures.

4.4 Models with recursive modules and higher-order store

The guarded recursive model described in Section 4.3 can be upgraded à la Sterling et al. (2022) to a model with all the same formal properties, that *additionally* closes **type** under an effect monad T for higher-order store. The idea of *op. cit.* is to iterate the presheaf construction, defining a preorder $\mathbb{W}$ of semantic

Kripke worlds simultaneously with its collection of $\mathcal{U}$-small co-presheaves by solving the guarded recursive domain equation $\mathbb{W} = (\mathbb{N} \rightharpoonup_{fin} \blacktriangleright[\mathbb{W}, \mathcal{U}])$.

In particular, *op. cit*. have shown how to model a type constructor $\mathsf{ref} : \mathbf{type} \to \mathbf{type}$ for general (unrestricted) reference types. Combined with the reflection, it becomes possible to store module packages in the heap.

Acknowledgments

I'm thankful to Robert Harper for many productive and enlightening conversations on the subject of modules, existential types, and reflective subuniverses; thanks to Dan Licata and Michael Shulman as well for helpful discussions and suggestions. Thanks to El Pin Al and Asta Halkjær From for typographical corrections. This work is funded by the European Union under the Marie Skłodowska-Curie Actions Postdoctoral Fellowship project *TypeSynth: synthetic methods in program verification*. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

References

Abadi, M., Banerjee, A., Heintze, N. & Riecke, J. G. (1999) A core calculus of dependency. Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Antonio, Texas, USA. Association for Computing Machinery. pp. 147–160.

Awodey, S., Frey, J. & Speight, S. (2018) Impredicative encodings of (higher) inductive types. Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. Oxford, United Kingdom. Association for Computing Machinery. pp. 76–85.

Birkedal, L., Møgelberg, R. E., Schwinghammer, J. & Støvring, K. (2011) First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science. Washington, DC, USA. IEEE Computer Society. pp. 55–64.

Choudhury, P. (2022) *Monadic and comonadic aspects of dependency analysis*. To appear in SPLASH 2022.

Dreyer, D. (2007) A type system for recursive modules. Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. Freiburg, Germany. Association for Computing Machinery. pp. 289–302.

Freyd, P. (2003) *Abelian categories*. vol. 3 of *Reprints in Theory and Applications of Categories*. Originally published by: Harper and Row, New York, 1964.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis. Université Paris VII.

Harper, R. (2016) *Practical Foundations for Programming Languages*. Cambridge University Press. New York, NY, USA. second edition.

Harper, R., Mitchell, J. C. & Moggi, E. (1990) Higher-order modules and the phase distinction. Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Francisco, California, USA. Association for Computing Machinery. pp. 341–354.

Hofmann, M. & Streicher, T. (1997) Lifting Grothendieck universes. Unpublished note.

Hyland, J. M. E. (1988) A small complete category. *Annals of Pure and Applied Logic*. **40**(2), 135–165.

Hyland, J. M. E. (1991) First steps in synthetic domain theory. Category Theory. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 131–156.

Longley, J. (1995) *Realizability Toposes and Language Semantics*. Ph.D. thesis. Edinburgh University.

Luo, Z. (1994) *Computation and Reasoning: A Type Theory for Computer Science*. vol. 11 of *International Series of Monographs on Computer Science*. Oxford Science Publications.

MacQueen, D. B. (1986) Using dependent types to express modular structure. Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. St. Petersburg Beach, Florida. Association for Computing Machinery. pp. 277–286.

Martin-Löf, P. (1975) An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, Rose, H. E. & Shepherdson, J. C. (eds). vol. 80 of *Studies in Logic and the Foundations of Mathematics*. Elsevier. pp. 73–118.

Martin-Löf, P. (1979) Constructive mathematics and computer programming. 6th International Congress for Logic, Methodology and Philosophy of Science. Hanover. pp. 153–175. Published by North Holland, Amsterdam. 1982.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *Journal of Functional Programming*. **18**(1), 1–13.

Mitchell, J. C. & Plotkin, G. D. (1985) Abstract types have existential types. Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New Orleans, Louisiana, USA. Association for Computing Machinery. pp. 37–51.

Palombi, D. & Sterling, J. (2022) Classifying topoi in synthetic guarded domain theory. Proceedings 38th Conference on Mathematical Foundations of Programming Semantics, MFPS 2022. To appear.

Pitts, A. M. (1987) Polymorphism is set theoretic, constructively. Category Theory and Computer Science. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 12–39.

Reynolds, J. C. (1983) Types, abstraction, and parametric polymorphism. Information Processing.

Reynolds, J. C. (1984) Polymorphism is not set-theoretic. Semantics of Data Types. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 145–156.

Rijke, E., Shulman, M. & Spitters, B. (2020) Modalities in homotopy type theory. *Logical Methods in Computer Science*. **Volume 16, Issue 1**.

Rosolini, G. (1986) *Continuity and effectiveness in topoi*. Ph.D. thesis. University of Oxford.

Rossberg, A. (2018) 1ML – core and modules united. *Journal of Functional Programming*. **28**, e22.

Rossberg, A., Russo, C. & Dreyer, D. (2014) F-ing modules. *Journal of Functional Programming*. **24**(5), 529–607.

Shulman, M. (2017) Idempotents in intensional type theory. *Logical Methods in Computer Science*. **12**.

Shulman, M. (2018) Impredicative encodings, part 3. Blog post.

Sterling, J., Gratzer, D. & Birkedal, L. (2022) Denotational semantics of general store and polymorphism. Unpublished manuscript.

Sterling, J. & Harper, R. (2021) Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM*. **68**(6).

Stone, C. A. & Harper, R. (2000) Deciding type equivalence in a language with singleton kinds. Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Boston, MA, USA. Association for Computing Machinery. pp. 214–227.

Univalent Foundations Program, T. (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`. Institute for Advanced Study.

van Oosten, J. (2008) *Realizability: An Introduction to its Categorical Side*. Elsevier Science, San Diego.

A Appendix

Let $\mathcal{U}$ be a universe closed under dependent sums, function types, and identification types such that every $A : \mathcal{U}$ is a set in the sense of Definition 4; let $\mathcal{V}$ be another universe such that $\mathcal{U} \in \mathcal{V}$ and moreover $\mathcal{U} \in \mathcal{V}$ is closed under universal types in the sense of Definition 14.

Lemma 29. *Let A be a type such that for each $C : \mathcal{U}$, the function space $A \to C$ lies in $\mathcal{U}$; then A is reflected by $\mathcal{U}$.*

The proof is an immediate application of the method of Awodey et al. (2018).

Proof. First we define the "wild" reflection $\tilde{A}$ as follows:

$$\tilde{A} :\equiv \textstyle\prod_{C:\mathcal{U}}((A \to C) \to C)$$

The wild reflection lies in $\mathcal{U}$ because $\mathcal{U} \in \mathcal{V}$ is assumed to have universal types, and moreover each $A \to C$ is assumed to lie in $\mathcal{U}$. We define a family of types indexed in $u : \tilde{A}$ governing "naturality data" for u:

$$\begin{aligned} &\mathsf{isNatural} : \tilde{A} \to \mathcal{U} \\ &\mathsf{isNatural}\, u :\equiv \textstyle\prod_{C,D:\mathcal{U}} \prod_{f:C\to D} \prod_{h:A\to C} u\, D\, (f \circ h) =_D f\, (u\, C\, h) \end{aligned}$$

Because each $D : \mathcal{U}$ is assumed to be a set, it follows that each $\mathsf{isNatural}\, u$ is a proposition. Thus we define the actual reflection $A_{|\mathcal{U}}$ to be the following subset:

$$A_{|\mathcal{U}} :\equiv \{u : \tilde{A} \mid \mathsf{isNatural}\, u\}$$

The unit $\eta_A : A \to A_{|\mathcal{U}}$ is defined to take $x : A$ to $\lambda C\, k.kx$; this function can be seen to be valued in $A_{|\mathcal{U}}$ by definition. It remains to argue that for any $B : \mathcal{U}$, the precomposition function $B^{\eta_A} : \left(A_{|\mathcal{U}} \to B\right) \to (A \to B)$ is an equivalence. Fixing $f : A \to B$, we must show that the fiber of B^{η_A} over f is contractible. First we inhabit this fiber by exhibiting the extension $f^\sharp\, u :\equiv u\, B\, f$; to see that $f^\sharp$ does in fact extend f along η_A, we compute:

$$f^\sharp\, (\eta_A a) \equiv \eta_A a\, B\, f \equiv f a$$

Thus the fiber of B^{η_A} over f is inhabited by $(f^\sharp, \mathsf{refl})$. Next we fix any $(h, p) : \mathsf{fib}_{(B^{\eta_A})} f$ to check that $(h, p) =_{\mathsf{fib}_{(B^{\eta_A})} f} (f^\sharp, \mathsf{refl})$. Because B is assumed to be a set, it suffices to check only that $h =_{A_{|\mathcal{U}} \to B} f^\sharp$. Fixing $u : A_{|\mathcal{U}}$, we must check: $hu = f^\sharp u$, or equivalently, $hu = u\, B\, f$. Because u is assumed natural, we have $u\, B\, f =_B h\left(u\, A_{|\mathcal{U}}\, \eta_A\right)$, so it suffices to check that $u\, A_{|\mathcal{U}}\, \eta_A =_{A_{|\mathcal{U}}} u$, but this also follows from naturality. □

With this in hand, Lemma 15 is an immediate corollary.

Lemma 15 (Universal types *vs*. reflectivity)**.** *Suppose that $\mathcal{U} \in \mathcal{V}$ and $\mathcal{U} \subseteq \mathcal{V}$ such every $A : \mathcal{U}$ is a set and $\mathcal{U}$ is moreover closed under identification types. Then $\mathcal{U} \in \mathcal{V}$ has universal types if and only if $\mathcal{U}$ is reflective in $\mathcal{V}$.*