On the Expressive Power of User-Defined Effects:

Effect Handlers, Monadic Reflection, Delimited Control

YANNICK FORSTER, Saarland University, Germany and University of Cambridge, England
OHAD KAMMAR, University of Oxford, England and University of Cambridge, England
SAM LINDLEY, University of Edinburgh, Scotland
MATIJA PRETNAR, University of Ljubljana, Slovenia

We compare the expressive power of three programming abstractions for user-defined computational effects: Plotkin and Pretnar’s effect handlers, Filinski’s monadic reflection, and delimited control without answer-type-modification. This comparison allows a precise discussion about the relative expressiveness of each programming abstraction. It also demonstrates the sensitivity of the relative expressiveness of user-defined effects to seemingly orthogonal language features.

We present three calculi, one per abstraction, extending Levy’s call-by-push-value. For each calculus, we present syntax, operational semantics, a natural type-and-effect system, and, for effect handlers and monadic reflection, a set-theoretic denotational semantics. We establish their basic metatheoretic properties: safety, termination, and, where applicable, soundness and adequacy. Using Felleisen’s notion of a macro translation, we show that these abstractions can macro-express each other, and show which translations preserve typeability.

We use the adequate finitary set-theoretic denotational semantics for the monadic calculus to show that effect handlers cannot be macro-expressed while preserving typeability either by monadic reflection or by delimited control. Our argument fails with simple changes to the type system such as polymorphism and inductive types. We supplement our development with a mechanised Abella formalisation.

CCS Concepts: • Theory of computation → Control primitives; Functional constructs; Type structures; Denotational semantics; Operational semantics; Categorical semantics;

Additional Key Words and Phrases: algebraic effects and handlers, monads, delimited control, computational effects, shift and reset, monadic reflection, reify and reflect, macro expressiveness, type-and-effect systems, denotational semantics, language extension, call-by-push-value, lambda calculus

ACM Reference Format:
https://doi.org/http://dx.doi.org/10.1145/3110257

1 INTRODUCTION
How should we compare abstractions for user-defined effects?

The use of computational effects, such as file, terminal, and network I/O, random-number generation, and memory allocation and mutation, is controversial in functional programming. While languages like Scheme and ML allow these effects to occur everywhere, pure languages like Haskell restrict the use of effects. One reason to be wary of incorporating computational effects into a language is that doing so can mean giving up some of the most basic properties of the lambda
calculus, like $\beta$-equality, referential transparency, and confluence. The loss of these properties leads to unpredictable behaviour in lazy languages, makes it harder to reason about program behaviour, and limits the applicability of correctness preserving transformations like common subexpression elimination or code motion.

Monads [Moggi 1989; Spivey 1990; Wadler 1990] are the established abstraction for incorporating effects into pure languages. Recently, Bauer and Pretnar [2015] proposed the use of algebraic effects and handlers [Plotkin and Pretnar 2009] to structure programs with user-defined effects. In this approach, the programmer first declares algebraic operations as the syntactic constructs she will use to cause the effects, in analogy with declaring new exceptions. Then, she defines effect handlers that describe how to handle these operations, in analogy with exception handlers. While exceptions immediately transfer control to the enclosing handler without resumption, a computation may continue in the same position following an effect operation. In order to support resumption, an effect handler has access to the continuation at the point of effect invocation. Thus algebraic effects and handlers provide a form of delimited control. Delimited control operators have long been used to encode effects [Danvy 2006]. There are many variants of such control operators, and their inter-relationships are subtle [Shan 2007], and often appear only in folklore. Here we focus on a specific pair of operators: shift-zero and dollar [Materzok and Biernacki 2012] without answer-type-modification, whose operational semantics and type system are the closest to effect handlers and monads.

We study the three different abstractions for user-defined effects: effect handlers, monads, and delimited control operators. Our goal is to enable language designers to conduct a precise and informed discussion about the relative expressiveness of each abstraction. In order to compare them, we build on an idealised calculus for functional-imperative programming, namely call-by-push-value [Levy 2004], and extend it with each of the three abstractions and their corresponding natural type systems. We then assess the expressive power of each abstraction by rigorously comparing and analysing these calculi.

We use Felleisen’s notion of macro expressibility [Felleisen 1991]: when a programming language $L$ is extended by some feature, we say that the extended language $L_+$ is macro expressible when there is a local syntax-directed translation (a macro translation) from $L_+$ to $L$ that keeps the features in $L$ fixed. Felleisen introduces this notion to study the relative expressive power of Turing-complete calculi, as macro expressivity is more sensitive in these contexts than notions of expressivity based on computability. We adapt Felleisen’s approach to the situation where one extension $L_1^+$ of a base calculus $L$ is macro expressible in another extension $L_2^+$ of the same base calculus $L$. Doing so allows us to formally compare the expressive power of each of the different abstractions for user-defined effects.

In the first instance, we show that, disregarding types, all three abstractions are macro-expressible in terms of one another, giving six macro translations. Some of these translations are known in less rigorous forms, either published, or in folklore. One translation, macro-expressing effect-handlers in delimited control, improves on previous concrete implementations [Kammar et al. 2013], which rely on the existence of a global higher-order memory cell storing a stack of effect-handlers. The translation from monadic reflection to effect handlers is new.

We also examine whether these translations preserve typeability: the translations of some well-typed programs are untypeable. This untypeability is sensitive to the precise choice of features of the type system. We show that the translation from delimited control to monadic reflection preserves typeability. A potential difference between the expressive power of handler type systems and between monadic reflection and delimited control type systems was recently suggested by Kammar and Pretnar [2017], who give a straightforward typeability preserving macro-translation of delimited dynamic state into a calculus of effect handlers, whereas existing translations using...
monads and delimited control require more sophistication [Kiselyov et al. 2006]. We show that there exists no macro translation from effect handlers to monadic reflection that preserves typeability. The proof relies on the denotational semantics for the monadic calculus. This set-theoretic denotational semantics and its adequacy for Filinski’s multi-monadic metalanguage [2010] is another piece of folklore which we formalise here. We conjecture that a similar proof, though with more mathematical sophistication, can be used to prove the non-existence of a typeability-preserving macro translation from the monadic calculus to effect handlers. To this end, we give adequate set-theoretic semantics to the effect handler calculus with its type-and-effect system, and highlight the critical semantic invariant a monadic calculus will invalidate.

Fig. 1 summarises our contributions and conjectured results. Untyped calculi appear on the left and their typed fragments on the right. Unlabelled arrows between the typed calculi signify that the corresponding macro translation between the untyped calculi preserves typeability. Arrows labelled by \(*\) are new direct untyped direct translations. Arrows labelled by \(\notin\) signify that no macro translation exists between the calculi, not even a partial macro translation that is only defined for well-typed programs.

The non-expressivity results are sensitive to the precise collection of features in each calculus. For example, extending the base calculus with inductive types and primitive recursion would create gaps in our non-existence arguments, and we conjecture that extending the calculi with various forms of polymorphism would make our untyped translations typeability-preserving. Adding more features to each calculus blurs the distinction between each abstraction. This sensitivity means that in a realistic programming language, such as Haskell, OCaml, or Scheme, the different abstractions are often practically equivalent [Schrijvers et al. 2016]. It also teaches us that meaningful relative expressivity results must be stated within a rigorous framework such as a formal calculus, where the exact assumptions and features are made explicit. The full picture is still far from complete, but our work lays the foundation for drawing it.

We supplement our pencil-and-paper proofs with a mechanised formalisation in the Abella proof assistant [Gacek 2008, 2009] of the more syntactic aspects of our work. Specifically, for each calculus, we formalise a Wright and Felleisen style progress-and-preservation safety theorem [1994], and correctness theorems for our translations.

We make the following contributions:

- syntax and semantics of formal calculi for effect handlers, monadic reflection, and delimited control, where each calculus extends a shared call-by-push-value core, and their metatheory:
  - set-theoretic denotational semantics for effect handlers and monadic reflection;
  - denotational soundness and adequacy proofs for effect handlers and monadic reflection;
  - a termination proof for monadic reflection (termination proofs for the other calculi appear in existing work);
- six macro-translations between the three untyped calculi, and variations on three of those translations;
• formally mechanised metatheory in Abella\footnote{https://github.com/matijapretnar/user-defined-effects-formalization} comprising:
  – progress and preservation theorems;
  – the translations between the untyped calculi; and
  – their correctness proofs in terms of formal simulation results;
• typeability preservation of the macro translation from delimited control to monadic reflection; and
• a proof that there exists no typeability-preserving macro translation from effect handlers to either monadic reflection or delimited control.

We structure the remainder of the paper as follows. Sections 2–5 present the core calculus and its extensions with effect handlers, monadic reflection, and delimited control, in this order, along with their metatheoretic properties. Section 6 presents the macro translations between these calculi, their correctness, and typeability-preservation. Our positive translation results appear in § 6.1–6.6, which only depend on §1–4 of Section 2–5. Section 7 concludes and outlines further work.

2 THE CORE-CALCULUS: mam

We are interested in a functional-imperative calculus where effects and higher-order features interact well. Levy’s call-by-push-value (CBPV) calculus fits the bill [2004]. It allows us to uniformly deal with call-by-value and call-by-name evaluation strategies, making the theoretical development relevant to both ML-like and Haskell-like languages. In CBPV evaluation order is explicit, and the way it combines computational effects with higher-order features yields simpler program logic reasoning principles [Plotkin and Pretnar 2008; Kammar and Plotkin 2012]. We extend it with a type-and-effect system. It is a variant of Kammar and Plotkin’s multi-adjunctive intermediate language [2012] without effect operations or coercions. We call the resulting calculus the multi-adjunctive metalanguage (mam).

2.1 Syntax

Fig. 2 presents mam’s raw term syntax, which distinguishes between values (data) and computations (programs). We assume a countable set of variables ranged over by \(x, y, \ldots\), and a countable set of variant constructor literals ranged over by \(\ell\). The unit value, products, and finite variants/sums are standard. A computation can be suspended as a thunk \(\{M\}\), which may be passed around. Products and variants are eliminated with standard pattern matching constructs. Thunks can be forced to resume their execution. A computation may simply return a value, and two computations can be sequenced, as in Haskell’s do notation. A function computation abstracts over values to which it may be applied. In order to pass a function \(\lambda x.M\) as data, it must first be suspended as a thunk \(\{\lambda x.M\}\). For completeness, we also include CBPV’s binary computation products, which subsume projections from products in call-by-name languages.

![Fig. 2. mam syntax](image-url)
On the Expressive Power of User-Defined Effects

Frames and contexts

\[ \mathcal{P} ::= x \leftarrow [ ] ; N | [ ] V | \text{prj}_{i} [ ] \] pure frames

\[ \mathcal{F} ::= \mathcal{P} \] computation frames

\[ \mathcal{C} ::= [ ] | C[\mathcal{F} [ ] ] \] evaluation context

\[ \mathcal{H} ::= [ ] | \mathcal{H}[\mathcal{P} [ ] ] \] pure context

Beta reduction \[ M \xrightarrow{\beta} M' \]

\[ (\times) \text{ case } (V_1, V_2) \text{ of } (x_1, x_2) \rightarrow M \xrightarrow{\beta} M[V_1/x_1, V_2/x_2] \] (U) \{ M \}! \xrightarrow{\beta} M

\[ (+) \text{ case inj}_{\ell} V \text{ of } \{ \ldots \text{inj}_{\ell} x \rightarrow M \ldots \} \xrightarrow{\beta} M[V/x] \] (\rightarrow) \((\lambda x. M) V \xrightarrow{\beta} M[V/x]\)

\[ (F) \quad x \leftarrow \text{return } V; M \xrightarrow{\beta} M[V/x] \] (&) \text{prj}_{i} (M_1, M_2) \xrightarrow{\beta} M_i

Fig. 3. mam operational semantics

Example 2.1. Using the boolean values \text{inj}_{\text{True}()} \text{ and \text{inj}_{\text{False}()}}, we define a logical not operation:

\[ \text{not} = \{ \lambda b. \text{case } b \text{ of } \{ \text{inj}_{\text{True}()} x \rightarrow \text{return } \text{inj}_{\text{False}()} \} \text{ inj}_{\text{False}()} x \rightarrow \text{return } \text{inj}_{\text{True}()} \} \}

2.2 Operational semantics

Fig. 3 presents mam’s standard structural operational semantics, in the style of Felleisen and Friedman [1987]. In order to reuse the core definitions as much as possible, we refactor the semantics into \( \beta \)-reduction rules and a single congruence rule. As usual, a \( \beta \)-reduction reduces a matching pair of introduction and elimination forms.

We factor the definition of evaluation contexts through computation frames. In mam these consist of pure frames, the elimination frames for pure computation. For each extension we will add another kind of effectful computation frame. We use [ ] to denote the hole in each frame or context, which signifies which term should evaluate first, and define substitution frames and terms for holes (\( C[\mathcal{F} [ ] ] \), \( C[M] \)) in the standard way. Later, in each calculus we will make use of pure contexts in order to capture continuations, stacks of pure frames, extending from a control operator to the nearest delimiter. A reducible term can be decomposed into at most one pair of evaluation context and \( \beta \)-reducible term, making the semantics deterministic.

Example 2.2. With this semantics we have not! (\text{inj}_{\text{True}()} \) \xrightarrow{+} \text{return } \text{inj}_{\text{False}()}\).

We use the following standard syntactic sugar. We use nested patterns in our pattern matching constructs. We abbreviate the variant constructors to their labels, and omit the unit value, e.g., True desugars to inj\text{True}(). We allow the application of functions and the elimination constructs to apply to arbitrary computations, and not just values, by setting for example \( MN ::= x \leftarrow N ; M x \) for some fresh \( x \), giving a more readable, albeit call-by-value, appearance.

Example 2.3. As a running example, we express boolean state in each of our calculi. Fig. 4(a) shows the code, which toggles the state and returns the value of the original state, as we would like to write it. Fig. 4(b) shows how we do so in mam, via a standard state-passing transformation. We may then run toggle with the initial value True to get the expected result runState! toggle True \xrightarrow{\ast} (True, False). This transformation is not a macro translation. In addition to the definition of put and get, it globally threads the state through toggle’s structure. Each user-defined effect abstraction in Sections 3–5 provides a different means for macro-expressing state.
2.3 Type-and-effect system

Fig. 5 presents \textsc{mam}'s types and effects. As a core calculus for three calculi with different notions of effect, \textsc{mam} is pure, and the only shared effect is the empty effect $\emptyset$.

We include a kind system, unneeded in traditional CBPV where a context-free distinction between values and computations forces types to be well-formed. The two points of difference from CBPV are the kind of effects, and the refinement of the computation kind by well-kinded effects $E$. The other available kinds are the standard value kind and a kind for well-formed environments (without type dependencies).

Our type system includes value-type variables (which in Section 4 we use for defining monads parametrically). The simple types, finite products and variants, are the standard CBPV value types. Thunk types are annotated with effect annotations. Computation types include returners $FA$, which are computations that return a value of type $A$, similar to the monadic type $\text{Monad} m \Rightarrow m a$ in Haskell. Functions are computations and only take values as arguments. We include CBPV’s computation products, which account for product elimination via projection in call-by-name languages.

To ensure well-kindedness of types, which may contain type variables, we use type environments in a list notation that denotes sets of type variables. Similarly, we use a list notation for value environments, which are functions from a finite set of variable names to the set of value types.

Example 2.4. The type of booleans $\text{bit}$ is given by $\{\text{inj}_{\text{False}} 1, \text{inj}_{\text{True}} 1\}$.

Fig. 6 presents the kind and type systems. The only effect ($\emptyset$) is well-kinded. Type variables must appear in the current type environment, and they are always value types. The remaining value and computation types and environments have straightforward structural kinding conditions. Thunks of $E$-computations of type $C$ require the type $C$ to be well-kinded, which includes the side-condition that $E$ is a well-kinded effect. This kind system has the property that each valid kinding judgement has a unique derivation. Value type judgements assert that a value term has a well-formed value type under a well-formed environment in some type variable environment.

The rules for simple types are straightforward. Observe how the effect annotation moves between the $E$-computation type judgement and the type of $E$-thunks. The side condition for computation

\[
\text{toggle} = \{ x \leftarrow \text{get}!; \}
\]

\[
\text{get} = \{ \lambda s. (s, s) \}
\]

\[
\text{toggle} = \{ \lambda s. (x, s) \leftarrow \text{get}! s; \}
\]

\[
y \leftarrow \text{not}! x; \}
\]

\[
\text{put} = \{ \lambda s'. \lambda _. (() , s') \}
\]

\[
y \leftarrow \text{not}! x; \}
\]

\[
\text{runState} = \lambda c. \lambda s. c! s
\]

\[
(\_, s) \leftarrow \text{put}! y s;
\]

\[
(x, s)
\]

(a) Direct style

(b) State-passing style

Fig. 4. User-defined boolean state

\[
E ::= \text{effects} \quad A, B ::= \text{value types} \quad C, D ::= \text{computation types}
\]

\[
\emptyset \quad \text{pure effect} \quad \alpha \quad \text{type variable} \quad FA \quad \text{returners}
\]

\[
K ::= \text{kinds} \quad | 1 \quad \text{unit} \quad | A \to C \quad \text{functions}
\]

\[
| \text{Eff} \quad \text{effects} \quad | A_1 \times A_2 \quad \text{products} \quad | C_1 & C_2 \quad \text{products}
\]

\[
| \text{Val} \quad \text{values} \quad | \{ \text{inj}_{\ell_i} A_i \}_i \quad \text{variants} \quad \text{Environments:}
\]

\[
| \text{Comp}_E \quad \text{computations} \quad | U_{E C} \quad \text{thunks} \quad \Theta ::= a_1, \ldots, a_n
\]

\[
\Gamma, \Delta ::= x_1 : A_1, \ldots, x_n : A_n
\]

Fig. 5. \textsc{mam} kinds and types
Effect kinding \( \Theta \vdash_k E : \text{Eff} \)

\[ \Theta \vdash_k \emptyset : \text{Eff} \]

Value kinding \( \Theta \vdash_k A : \text{Val} \)

\[ \alpha \in \Theta \]
\[ \Theta \vdash_k \alpha : \text{Val} \]
\[ \Theta \vdash_k 1 : \text{Val} \]
\[ [\Theta \vdash_k a_i : \text{Val}]_i \]
\[ \Theta \vdash_k \{\text{inj}_{\ell_i} A_i\}_i : \text{Val} \]

Computation kinding \( \Theta \vdash_k C : \text{Comp}_E \)

\[ \Theta \vdash_k A : \text{Val} \]
\[ \Theta \vdash_k FA : \text{Comp}_E \]
\[ \Theta \vdash_k A \rightarrow C : \text{Comp}_E \]
\[ \Theta \vdash_k C : \text{Comp}_E \]
\[ \Theta \vdash_k C_1 \& C_2 : \text{Comp}_E \]

Value typing \( \Theta ; \Gamma \vdash V : A \)

\[ (x : A) \in \Gamma \]
\[ \Theta ; \Gamma \vdash x : A \]
\[ \Theta ; \Gamma \vdash () : 1 \]
\[ \Theta ; \Gamma \vdash V : A_i \]
\[ \Theta ; \Gamma \vdash \text{inj}_{\ell_i} V : \{\text{inj}_{\ell_i} A_i\}_i \]

Computation typing \( \Theta ; \Gamma \vdash E M : C \)

\[ \Theta ; \Gamma \vdash V : A_1 \times A_2 \]
\[ \Theta ; \Gamma \vdash \text{case } V \text{ of } (x_1, x_2) : M : C \]
\[ \Theta ; \Gamma \vdash V = U_E C \]
\[ \Theta ; \Gamma \vdash \text{return } V : FA \]
\[ \Theta ; \Gamma, \lambda x : A + E M : A \rightarrow C \]
\[ \Theta ; \Gamma \vdash E M_1 : C_1 \]
\[ \Theta ; \Gamma \vdash E M_2 : C_2 \]
\[ \Theta ; \Gamma \vdash E \langle M_1, M_2 \rangle : C_1 \& C_2 \]
\[ \Theta ; \Gamma \vdash E \text{prj}_1 : M : C_i \]

Context kinding \( \Theta \vdash_k \Gamma : \text{Context} \)

\[ [\Theta \vdash_k \Gamma(x) : \text{Val}]_{x \in \text{Dom}(\Gamma)} \]
\[ \Theta \vdash_k \Gamma : \text{Context} \]

\[ \Theta \vdash_k A_1 : \text{Val} \]
\[ \Theta \vdash_k A_2 : \text{Val} \]
\[ \Theta \vdash_k A_1 \times A_2 : \text{Val} \]
\[ \Theta \vdash_k C : \text{Comp}_E \]
\[ \Theta \vdash_k U_E C : \text{Val} \]

\[ \Theta ; \Gamma \vdash V_1 : A_1 \]
\[ \Theta ; \Gamma \vdash V_2 : A_2 \]
\[ \Theta ; \Gamma \vdash (V_1, V_2) : A_1 \times A_2 \]
\[ \Theta ; \Gamma \vdash M : C \]
\[ \Theta ; \Gamma \vdash \{M\} : U_E C \]

\[ \Theta ; \Gamma \vdash \text{case } V \text{ of } \{\text{inj}_{\ell_i} A_i\}_i : C \]
\[ \Theta ; \Gamma \vdash \text{of } A \rightarrow E N : C \]
\[ \Theta ; \Gamma \vdash x = M : N : C \]

\[ \Theta ; \Gamma \vdash E M_1 : C_1 \]
\[ \Theta ; \Gamma \vdash E M_2 : C_2 \]
\[ \Theta ; \Gamma \vdash E \langle M_1, M_2 \rangle : C_1 \& C_2 \]

\[ \Theta ; \Gamma \vdash E \text{prj}_1 : M : C_i \]

Fig. 6. \textit{maa} kind and type system
choice to omit any form of effect coercion, subeffecting, or effect polymorphism: the three effect annotations must agree. More sophisticated effect systems allow greater flexibility [Katsumata 2014]. We leave the precise treatment of such extensions to later work.

Example 2.5. The values from Fig. 4(b) have the following types:

\[
\begin{align*}
\text{not} & : U_0(\text{bit} \to F(\text{bit} \times \text{bit})) \\
\text{get} & : U_0(\text{bit} \to F(\text{bit} \times \text{bit})) \\
\text{put} & : U_0(\text{bit} \to \text{bit} \to F(\text{bit} \times \text{bit})) \\
\text{toggle} & : U_0(\text{bit} \to F(\text{bit} \times \text{bit})) \\
\text{runState} & : U_0(U_0(\text{bit} \to F(\text{bit} \times \text{bit})) \to \text{bit} \to F(\text{bit} \times \text{bit}))
\end{align*}
\]

2.4 Operational metatheory

We establish the basic properties of \textsc{mam}.

Theorem 2.6 (\textsc{mam} safety). Well-typed programs don’t go wrong: for all closed \textsc{mam} returners \(\Theta;\tau_0 M : FA\), either \(M \leadsto N\) for some \(\Theta;\tau_0 N : FA\) or else \(M = \text{return } V\) for some \(\Theta; \vdash V : A\).

The standard inductive progress-and-preservation proof is in the Abella formalisation.

We extend existing termination results for CBPV [Doczkal 2007; Doczkal and Schwinghammer 2009]. We say that a term \(M\) diverges, and write \(M \leadsto^{\omega}\) if for every \(n \in \mathbb{N}\) there exists some \(N\) such that \(M \leadsto^n N\). We say that \(M\) does not diverge when \(M \not\leadsto^{\omega}\).

Theorem 2.7 (\textsc{mam} termination). There are no infinite reduction sequences: for all \textsc{mam} terms \(\Theta;\tau_0 M : FA\), we have \(M \not\leadsto^{\omega}\), and there exists some unique \(\Theta; \vdash V : A\) such that \(M \leadsto^{*} \text{return } V\).

The proof uses Tait’s method [1967] to establish totality. Explicitly, we define a (unary) relational interpretation to types and establish a basic lemma. To interpret returners \(FA\), we need a monadic lifting. We use the lifting from Hermida’s thesis [1993], defined to contains the returners that reduce to a return value for all closed substitutions. The remainder of the proof is immediate as the semantics is deterministic.

We now define contextual equivalence of \textsc{mam} terms. We define the subclass of ground types:

\[G ::= 1 \mid G_1 \times G_2 \mid \{\text{inj}_{f_i} G_i\}_i\]

The standard next step is to define well-typed program contexts \(X[\ ]\) — terms with zero, one, or more occurrences of a hole, denoted by \([\ ]\), not to be confused with evaluation contexts \(C[\ ]\), which always contain exactly one hole. Defining program contexts and their type judgements directly is straightforward but tedious and lengthy, with four kinds of judgements, and so we take a different approach. Informally, given two computation terms \(M_1\) and \(M_2\), in order to define their contextual equivalence, we need to quantify over the set of all the pairs of contexts plugged with \(M_1\) and \(M_2\):

\[\Xi[M_1, M_2] := \langle X[M_1], X[M_2] \rangle \mid X[\ ]\text{ is a well-typed enclosing context}\]

Once we define this set, we do not need contexts, their type system, nor their semantics in the remainder of the development, and so we will define this set directly.

When defining this set, we need to know the form of the typing judgement, and so it will contain, apart from the two terms \(\langle X[M_1], X[M_2] \rangle\), their shared environments, \(\Theta'\) and \(\Gamma'\), and shared type, \(A\) or \(C\). When this shared type is a computation type \(C\), we also need to know the effect annotation \(E\). So this set will contain quintuples \(\langle \Theta', \Gamma', V_1, V_2, A \rangle\) representing the simultaneous value judgement \(\Theta'; \Gamma' \vdash V_1, V_2 : A\) and sextuples \(\langle \Theta', \Gamma', E', N_1, N_2, C \rangle\) representing the simultaneous computation judgement \(\Theta'; \Gamma' \vdash E; N_1, N_2 : C\). Because values can contain computations through thunking, and vice versa through forcing, this set will contain both such quintuples and sextuples. Because the program context can introduce identifiers through function abstraction, we should also allow for environment extension.
The full definition is as follows. We say that an environment $\Gamma'$ extends an environment $\Gamma$, and write $\Gamma' \geq \Gamma$ if $\Gamma'$ extends $\Gamma$ as a partial function from identifiers to value types. Consider any two computations with the same type $C_0$ under the same environments $\Theta_0, C_0$, that is, $\Theta_0; \Gamma_0 + E_0 \mid M_1 : C_0$ and $\Theta_0; \Gamma_0 + E_0 \mid M_2 : C_0$. Define the set $\Xi[\Theta_0; \Gamma_0 + E_0 \mid M_1, M_2 : C_0]$ to be the smallest set of tuples $(\Theta', \Gamma', V_1, V_2, A)$ and $(\Theta', \Gamma', E', N_1, N_2, C)$ that is compatible with the typing rules and contains all the tuples $(\Theta, \Gamma, E_0, M_1, M_2, C_0)$, where $\Theta \supseteq \Theta_0$ and $\Gamma \geq \Gamma_0$. The compatibility with the rules means, for example, that if $\Theta' \Gamma' = \Theta \Gamma$, then $\Theta' \Gamma' = \Theta \Gamma$.

For uniformity’s sake, we let types $X$ range over both value and $E$-computation types, and phrases $P$ range over both value and computation terms. Judgements of the form $\Theta; \Gamma \vdash P : X$ are meta judgements, ranging over value judgements $\Theta; \Gamma \vdash P : X$ and $E$-computation judgement $\Theta; \Gamma \vdash P : X$.

Let $\Theta; \Gamma \vdash E \mid P, Q : X$ be two $\text{mam}$ phrases. We say that $P$ and $Q$ are contextually equivalent and write $\Theta; E \vdash P \equiv Q : X$ when, for all pairs of plugged closed ground-returner pure contexts $\langle \emptyset, \emptyset, MP, MQ, FG \rangle$ in $\Xi[\Theta; \Gamma \vdash E \mid P, Q : X]$ and for all closed ground value terms $; \vdash V : G$, we have $MP \sim^* V$ if and only if $MQ \sim^* V$.

### 2.5 Denotational semantics

$mam$ has a straightforward set-theoretic denotational semantics. Presenting the semantics for the core calculus will simplify our later presentation. To do so, we first recall the following established facts about monads, specialised and concretised to the set-theoretic setting.

A monad is a triple $(T, \text{return}, \ggg)$ where $T$ assigns to each set $X$ a set $TX$, $\text{return}$ assigns to each set $X$ a function $\ggg_X : X \to TX$ and $\ggg$ assigns to each function $f : X \to Y$ its Kleisli extension: a function $\ggg f : TX \to TY$ and the three assignments satisfy the monad laws:

$$(\text{return } x) \ggg f = f(x), \quad a \ggg \text{return } x = a, \quad (a \ggg f) \ggg g = a \ggg (\lambda x. (f x \ggg g))$$

for all $f : X \to TY$, $x \in X$, $a \in TX$, and $g : Y \to TZ$. A $T$-algebra for a monad $(T, \text{return}, \ggg)$, following Marmolejo and Wood [2010], is a pair $C = \langle |C|, \ggg_C \rangle$ where $|C|$ is a set, called the carrier, and $\ggg_C$ assigns to every function $f : X \to |C|$ its Kleisli extension $\ggg_C f : TX \to |C|$ satisfying:

$$(\text{return } x) \ggg_C f = f(x), \quad (a \ggg g) \ggg_C f = a \ggg_C (\lambda y. (gy \ggg_C f))$$

for all $x \in X$, $f : X \to |C|$, $a \in TX$, and $g : Y \to TX$. For each set $X$, the pair $FX := \langle TX, \ggg \rangle$ forms a $T$-algebra called the free $T$-algebra over $X$.

We parameterise $mam$’s semantics function $\langle \emptyset \vdash E : \text{Eff} \rangle$ by an assignment $\theta$ of sets $\theta(\alpha)$ to each of the type variables $\alpha$ in $\Theta$. Given such a type variable assignment $\theta$, we assign to each

- effect: a monad $\langle \emptyset \vdash E : \text{Eff} \rangle_\theta$, denoted by $\langle T_E \rangle_\theta, \text{return}_E, \ggg_E \rangle_\theta$;
- value type: a set $\langle \emptyset \vdash A : \text{Val} \rangle_\theta$;
We develop the basic properties of our denotational semantics. Our goal is to establish the adequacy of the semantics: terms with equivalent denotations are observationally equivalent. In our set-theoretic setting, the proof-recipe is well-established, using the following compositionality and adequacy:

- E-computation type: a $T_{E\theta}$-algebra $[[\Theta \vdash_E C : \text{Comp}_E]]_{\theta}$; and
- context: the set $[[\Theta \vdash_E \Gamma : \text{Context}]]_{\theta} := \prod_{x \in \text{Dom}(\Gamma)}[[\Gamma(x)]]_{\theta}$.

Fig. 7 defines the standard set-theoretic semantics function over the structure of types. The pure effect denotes the identity monad, which sends each set to itself, and extends a function by doing nothing. The extended languages in the following sections will assign more sophisticated monads to other effects. The semantics of type variables uses the type assignment given as parameter. The unit type always denotes the singleton set. Product types and variants denote the corresponding set-theoretic operations of cartesian product and disjoint union, and thus the empty variant type $0 := \{\}$ denotes the empty set. The type of thunked $E$-computations of type $C$ denotes the carrier of the $T_{E\theta}$-algebra $[[C]]_{\theta}$. The E-computation type of $A$ returners denotes the free $[E]_{\theta}$-algebra. Function and product types denote well-known algebra structures over the sets of functions and pairs, respectively [Barr and Wells 1985, Theorem 4.2].

Terms can have multiple types, for example the function $\lambda x. \text{return} \ x$ has the types $1 \rightarrow F1$ and $0 \rightarrow F0$, and type judgements can have multiple type derivations. We thus give a Curry-style semantics [Reynolds 2009] by defining the semantic function for type judgement derivations rather than for terms. To increase readability, we write $[[P]]$ and omit typing derivation for $P$.

The semantic function for terms is parameterised by an assignment $\theta$ of sets to type variables. It assigns to each well-typed derivation for a:

- value term: a function $[[\Theta ; \Gamma \vdash V : A]]_{\theta} : [[A]]_{\theta} \rightarrow [[A]]_{\theta}$; and
- E-computation term: a function $[[\Theta ; \Gamma \vdash_E M : C]]_{\theta} : [[C]]_{\theta} \rightarrow [[C]]_{\theta}$.

Fig. 8 defines the standard set-theoretic semantics over the structure of derivations. The semantics of sequencing uses the Kleisli extended function $(\gg [C] f) : TX \rightarrow [[C]]$ for functions into non-free algebras $f : X \rightarrow [[C]]$, given by the algebra structure.

2.6 Denotational metatheory

We develop the basic properties of our denotational semantics. Our goal is to establish the adequacy of the semantics: terms with equivalent denotations are observationally equivalent. In our set-theoretic setting, the proof-recipe is well-established, using the following compositionality and soundness theorems:

**Theorem 2.8 (MAM Compositionality).** The meaning of a term depends only on the meaning of its sub-terms: for all pairs of well-typed plugged MAM contexts $M_P, M_Q$ in $\Xi[\Theta ; \Gamma \vdash_E P, Q : X]$, if $[[P]] = [[Q]]$ then $[[M_P]] = [[M_Q]]$. 

The proof is a straightforward induction on the set of plugged contexts.

To phrase our simulation results in later development, we adopt a relaxed variant of simulation: let $\sim_{\text{cong}}$ be the smallest relation containing $\sim_\beta$ that is closed under the term formation constructs, and so contains $\sim$ as well, and let $\approx_{\text{cong}}$ be the smallest congruence relation containing $\sim_\beta$.

**Theorem 2.9 (mam soundness).** *Reduction preserves the semantics: for every pair of well-typed *mam* terms $\Theta; \Gamma \vdash_P P, Q : X$, if $P \approx_{\text{cong}} Q$ then $[P] = [Q]$. In particular, for every well-typed closed term of ground type $\Theta; \Gamma \vdash_0 P : FG$, if $P \overset{*}{\rightarrow} \text{return } V$ then $[P] = [V]$.

The proof is standard: check that $\sim_\beta$ preserves the semantics via calculation, and appeal to compositionality. It now follows that the semantics is adequate:

**Theorem 2.10 (mam adequacy).** *Denotational equivalence implies contextual equivalence: for all well-typed *mam* terms $\Theta; \Gamma \vdash_P P, Q : X$, if $[P] = [Q]$ then $P \approx Q$.

To see how the different pieces fit together, consider any two denotationally equivalent terms $P$ and $Q$, a closed ground context plugged with them $X[P], X[Q]$, and assume $X[P] \overset{*}{\rightarrow} \text{return } V$. By the Compositionality Theorem 2.8, $X[P]$ and $X[Q]$ have equal denotations. By the Safety Theorem 2.6 and Termination Theorem 2.7, $X[Q] \overset{*}{\rightarrow} \text{return } V'$ for some value $\text{return } V'$. And so by the Soundness Theorem 2.9:

$[\text{return } V] = [X[P]] = [X[Q]] = [\text{return } V']$

Conclude by verifying that, for ground returners, denotational equality implies syntactic equality.

As a consequence, we deduce that our operational semantics is very well-behaved: for all well-typed computations $\Theta; \Gamma \vdash_0 M, M' : C$, if $M \overset{\text{cong}}{\rightarrow} M'$ then $M \approx M'$.

### 3 Effect Handlers: $\textbf{eff}$

Algebraic effects and handlers provide a basis for modular programming with user-defined effects [Kammar et al. 2013; Kiselyov et al. 2013; Bauer and Pretnar 2015; Hillerström and Lindley 2016; Leijen 2017; Lindley et al. 2017]. Programmable effect handlers arose as part of Plotkin and Power’s denotational theory of computational effects [2002], which investigates the consequences of using the additional structure in algebraic presentations of monadic models of effects. This account refines Moggi’s monadic account [1989] by incorporating into the theory the syntactic constructs that generate effects as *algebraic operations for a monad* [Plotkin and Power 2003]: each monad is accompanied by a collection of syntactic operations, whose interaction is specified by a collection of equations, i.e., an algebraic theory, which fully determines the monad. To fit exception handlers into this account, Plotkin and Pretnar [2009] generalised exception handlers to effect handlers, handling arbitrary algebraic effects and, following Levy’s CBPV, give a computational interpretation of algebras for a monad. By allowing the user to declare operations, effects can be described in a composable manner. Bauer and Pretnar [2015] demonstrate how, by defining algebras for the free monad with these operations, users can give the abstract operations different meanings, in similar fashion to Swierstra’s use of free monads [2008].

#### 3.1 Syntax

Fig. 9(a) presents the extension $\textbf{eff}$, Kammar et al.’s core calculus of effect handlers [2013]. We assume a countable set of elements of a separate syntactic class, called *operation names* and ranged over by $\text{op}$. For each operation name $\text{op}$, $\textbf{eff}$’s operation call construct allows the programmer to invoke the effect associated with $\text{op}$ by passing it a value as an argument. Operation names are the only interface to effects the language has. The handling construct allows the programmer to use a handler to interpret the operation calls of a given returner computation. As the given computation
may call thunks returned by functions, the decision which handler will handle a given operation call is dynamic. Handlers are specified by two kinds of clauses. A return clause describes how to proceed when returning a value. An operation clause describes how to proceed when invoking an operation \( op \). The body of an operation clause can access the value passed in the operation call using the first bound variable \( p \), which is similar to the bounding occurrence of an exception variable when handling exceptions. But unlike exceptions, we expect arbitrary effects like reading from or writing to memory to resume. Therefore the body of an operation clause can also access the continuation \( k \) at the operation’s calling point.

Example 3.1. The left column of Fig. 10 expresses user-defined boolean state in \textit{eff}. The handler \( H_{ST} \) is parameterised by the current state. When the computation terminates, we discard this state. When the program calls \textit{get}, the handler returns the current state and leaves it unchanged. When the program calls \textit{put}, the handler returns the unit value, and instates the newly given state.

### 3.2 Operational semantics

Fig. 9(b) presents \textit{eff}’s extension to \textit{mam}’s operational semantics. Computation frames \( F \) now include the handling construct, whereas the pure frames \( P \) do not, allowing a handled computation to \( \beta \)-reduce under the handler. We add two \( \beta \)-reduction cases for the added construct. When the returner computation inside a handler is fully evaluated, the return clause proceeds with the return value. When the returner computation inside a handler needs to evaluate an operation call, the definition of pure contexts \( \mathcal{H} \) ensures \( \mathcal{H} \) is precisely the continuation of the operation call delimited by the handler. Put differently, it ensures that the handler in the root of the reduct is the closest handler to the operation call in the call stack. The operation clause corresponding to the operation called then proceeds with the supplied parameter and current continuation. Rewrapping the handler around this continuation ensures that all operation calls invoked in the continuation are handled in the same way.
Kinds and types

\[ E ::= \ldots \text{ effects} \]
\[ \{ \text{op} : A \rightarrow B \} \uplus E \text{ arity assignment} \]
\[ K ::= \ldots \text{ kinds} \]
\[ \text{Handler} \text{ handlers} \]
\[ R ::= A \Rightarrow E' C \text{ handler types} \]

Computation typing

\[ (\text{op} : A \rightarrow B) \in E \Theta; \Gamma \vdash V : A \]
\[ \Theta; \Gamma \vdash E \text{ op } : FB \]
\[ \Theta; \Gamma \vdash E M : FA \]
\[ \Theta; \Gamma \vdash H : A \Rightarrow E' C \]
\[ \Theta; \Gamma \vdash E' \text{ handle } M \text{ with } H : C \]

Handler typing

\[ \Theta; \Gamma \vdash H : R \] (\( \Theta \vdash_k \Gamma : \text{Context}, R : \text{Handler} \))
\[ \Theta; \Gamma, x : A \Rightarrow E M : C \]
\[ \Theta; \Gamma, p : A, k : U E(B_i \rightarrow C) \vdash E N_i : C \]
\[ \Theta; \Gamma \vdash \{ \text{return } x \mapsto M \} \uplus \{ \text{op}_i, p, k \mapsto N_i \} : A \Rightarrow E' C \]

Effect kinding

\[ \Theta \vdash_k A : \text{Val} \]
\[ \Theta \vdash_k B : \text{Val} \]
\[ \Theta \vdash_k \text{op} \notin E \]
\[ \Theta \vdash_k E : \text{Eff} \]

Example 3.2. With this semantics, the user-defined state from Fig. 10 behaves as expected:

\[ \text{runState! toggle True } \sim^* (\text{handle True with } H_{ST}) \text{ False } \sim^* \text{ True} \]

More generally, the handler \( H_{ST} \) expresses dynamically scoped state [Kammar and Pretnar 2017]. For additional handlers for state and other effects, see Pretnar’s tutorial [2015].

3.3 Type-and-effect system

Fig. 11 presents \( \text{eff} \)’s extension to the kind and type system. The effect annotations in \( \text{eff} \) are functions from finite signatures, assigning to each operation name its parameter type \( A \) and its return type \( B \). We add a new kind for handler types, which describes the kind and the returner type the handler can handle, and the kind and computation type of the handling clause.

In the kinding judgement for effects, the types in each operation’s arity assignment must be value types. The kinding judgement for handlers requires all the types and effects involved to be well-kindred.

Computation type judgements now include two additional rules for each new computation construct. An operation call is well-typed when the parameter and return type agree with the arity assignment in the effect annotation. An instance of the handling construct is well-typed when the type and effect of the handled computation and the type-and-effect of the construct agree with the types and effects in the handler type. The set of handled operations must strictly agree with the set of operations in the effect annotation. The variable bound to the return value has the returner type in the handler type. In each operation clause, the bound parameter variable has the parameter type from the arity assignment for this operation, and the continuation variable’s input type matches the return type in the operation’s arity assignment. The overall type of all operation clauses agrees with the computation type of the handler. The second effect annotation on the handler type matches the effect annotations on the continuation and the body of the operation and return clauses.

Example 3.3. The boolean state terms are assigned the types given in the right column of Fig. 10.
3.4 Operational metatheory
We follow mam’s development, formalising eff’s safety theorem in Abella:

**Theorem 3.4 (eff safety).** Well-typed programs don’t go wrong: for all closed eff returners \( \Theta; \tau_0 M : FA \), either \( M \leadsto N \) for some \( \Theta; \tau_0 N : FA \) or else \( M = \text{return} \ V \) for some \( \Theta; \vdash V : A \).

Using the monadic lifting from Kammar’s thesis [2014], we obtain termination for eff [Kammar et al. 2013]:

**Theorem 3.5 (eff termination).** There are no infinite reduction sequences: for all eff terms \( \vdash \tau_0 M : FA \), we have \( M \triangleright\triangleright^{\infty} \), and there exists a unique \( \vdash \tau_0 A \) such that \( M \leadsto^* \text{return} \ V \).

Eff shares mam’s ground types, and we define plugged contexts, \( \equiv \) and \( \equiv_{\text{cong}} \) as in mam.

3.5 Denotational semantics
We now give a set-theoretic denotational semantics for eff. First, recall the following concepts in universal and categorical algebra. A signature \( \Sigma \) is a pair consisting of a set \( |\Sigma| \) whose elements we call operation symbols, and a function \( \text{arity}_\Sigma \) from \( |\Sigma| \) assigning to each operation symbol \( \varphi \in |\Sigma| \) a (possibly infinite) set \( \text{arity}(\varphi) \). We write \( (\varphi : A) \in \Sigma \) when \( \varphi \in |\Sigma| \) and \( \text{arity}_\Sigma(\varphi) = A \). Given a signature \( \Sigma \) and a set \( X \), we inductively form the set \( T_\Sigma X \) of \( \Sigma \)-terms over \( X \):

\[
t ::= \varphi(t_a)_{a \in A} \quad (x \in X, (\varphi : A) \in \Sigma)
\]

The assignment \( T_\Sigma \) together with the following assignments form a monad

\[
\text{return} \ x := x \quad t := f : T(\Sigma Y)
\]

The \( T_\Sigma \)-algebras \( \langle C, \equiv^C \rangle \) are in bijective correspondence with \( \Sigma \)-algebras on the same carrier. These are pairs \( \langle C, [\text{-}] \rangle \) where \([\text{-}]\) assigns to each \( (\varphi : A) \in \Sigma \) a function \([\varphi] : C^A \to C \) from \( A \)-ary tuples of \( C \) elements to \( C \). The bijection is given by setting \( \equiv^C f \) to be the \( \Sigma \)-homomorphic extension of \( f : X \to |C| \) to \( T_\Sigma X \).

Eff’s denotational semantics is given by extending mam’s semantics as follows. Given a type variable assignment \( \Theta \), we assign to each handler type a pair \( [\Theta \triangleright_\Sigma R : \text{Handler}]_\Theta = \langle C, f \rangle \) consisting of an algebra \( C \) and a function \( f \) into the carrier \( |C| \) of this algebra.

Fig. 12 presents how eff extends mam’s denotational semantics. Each effect \( E \) gives rise to a signature whose operation symbols are the operation names in \( E \) tagged by an element of the denotation of the corresponding parameter type. This signature gives rise to the monad \( E \). When \( E = \emptyset \), the induced signature is empty, and gives rise to the identity monad, and so this semantic function extends mam’s semantics. Handlers of \( E \)-computations returning \( A \)-values using \( E' \)-computations of type \( C \) denote a pair. Its first component is an \( [E]_\Theta \)-algebra structure over the carrier \( |[C]_\Theta| \), which may have nothing to do with the \( [E']_\Theta \)-algebra structure \( |[C]_\Theta| \) already possesses. The second component is a function from \( [A]_\Theta \) to the carrier \( |[C]_\Theta| \).

The denotation of \( \text{op} \ V \) at effect type \( E \), where \( \text{op} : A \to B \in E \), is \( \text{op}([V]_\Theta(a))_{a \in [B]_\Theta} \). The denotation of the handling construct uses the Kleisli extension of the second component in the denotation of the handler. The denotation of a handler term defines the \( T_\Sigma \)-algebras by defining a \( \Sigma \)-algebra for the associated signature \( \Sigma \). The operation clause for \( \text{op} \) allows us to interpret each of the operation symbols associated to \( \text{op} \). The denotation of the return clause gives the second component of the handler.

3.6 Denotational metatheory
We repeat the recipe for proving adequacy.
On the Expressive Power of User-Defined Effects

Effects
\[ E \theta := T \{ \text{op}_p : [A] \theta \mid \text{op} : A \to B \in E, p \in [B] \theta \} \{ A^E \Rightarrow C \} := (\{ E \}-algebras with carrier \{ [C] \} \times [C][A] \]

Handler types

Computation terms
\[ \text{op} \{ \text{return} \ a \} \theta = \text{op} \{ \text{return} \ a \} \theta \]

Handler terms
\[ \{ \text{return} \ x \mapsto M \} \cup \{ \text{op} \ p k \mapsto N_i \} \theta (y) := \langle D, f \rangle \text{ where } \langle D, f \rangle (y) = \langle D, f : [A] \to [C] \rangle \]

Theorem 3.6 (eff compositionality). The meaning of a term depends only on the meaning of its sub-terms: for all pairs of well-typed plugged eff contexts \( M_P, M_Q \) in \( \Xi[\Theta; \Gamma \vdash_P P, Q : X] \), if \( [P] = [Q] \) then \([M_P] = [M_Q]\)

The proof is identical to mam, with two more cases for \( \rightsquigarrow_B \).

Theorem 3.7 (eff soundness). Reduction preserves the semantics: for every pair of well-typed eff terms \( \Theta; \Gamma \vdash_P P, Q : X \), if \( P \cong_{\text{cong}} Q \) then \([P] = [Q] \). In particular, for every well-typed closed term of ground type \( \vdash_0 P : FG \), if \( P \rightsquigarrow^* \text{return} V \) then \([P] = [V] \).

We combine the previous results, as with mam:

Theorem 3.8 (eff adequacy). Denotational equivalence implies contextual equivalence: for all well-typed eff terms \( \Theta; \Gamma \vdash_P P, Q : X \), if \([P] = [Q] \) then \( P \cong_Q \).

Therefore, eff also has a well-behaved operational semantics: for all well-typed computations \( \Theta; \Gamma \vdash_M M, M' : C \), if \( M \rightsquigarrow_{\text{cong}} M' \) then \( M \cong M' \).

4 MONADIC REFLECTION: mon

Moggi [1989] conceptualises computational effects as monads, which he uses to give a uniform denotational semantics for a wide range of different effects. Spivey [1990] and Wadler [1990] introduce programming abstractions based on monads, allowing new effects to be declared and used as if they are native. Examples include parsing [Hutton and Meijer 1998], backtracking and constraint solving [Schrijvers et al. 2013], and mechanised reasoning [Ziliani et al. 2015; Bulwahn et al. 2008]. Libraries now exist for monadic programming even in impure languages such as OCaml\(^2\), Scheme\(^3\), and C++ [Sinkovics and Porkolab 2013].

Languages that use monads as an abstraction for user-defined effects typically employ other mechanisms to support them—usually an overloading resolution mechanism, such as type-classes in Haskell and Coq, and functors/implicits in OCaml. As a consequence, such accounts do not study monads as an abstraction in their own right, and are intertwined with implementation details and concepts stemming from the added mechanism. Filinski’s work on monadic reflection [1994; 1996; 1999; 2010] provides a more canonical abstraction for incorporating monads into a programming language. In his calculi, user-defined monads stand independently.
4.1 Syntax

Fig. 13(a) presents MON’s syntax. The where \{return x = N; y ≡ f = N\} construct binds x in the term \(N_u\) and y and f in \(N_b\). The term \(N_u\) describes the unit and the term \(N_b\) describes the Kleisli extension/bind operation. We elaborate on the choice of the keyword where when we describe MON’s type system. Using monads, the programmer can write programs as if the new effect was native to the language. We call the mode of programming when the effect appears native the opaque view of the effect. In contrast, the transparent mode occurs when the code can access the implementation of the effect directly in terms of its defined monad. The reflect construct \(\mu(N)\) allows the programmer to graft code executing in transparent mode into a block of code executing in opaque mode. The reify construct \([N]^T\) turns a block of opaque code into the result obtained by the implementation of the effect.

Example 4.1. The left column of Fig. 14 expresses user-defined boolean state in MON using the standard State monad. To express get and put, we reflect the concrete definition of the corresponding operations of the state monad. To run a computation, we use reification to get the monadic representation of the computation as a state transformer, and apply it to the initial state.

4.2 Operational semantics

Fig. 13(b) describes the extension to the operational semantics. The ret transition uses the user-defined monadic return to reify a value. To explain the reflection transition, note that the pure context \(\mathcal{H}\) captures the continuation at the point of reflection delimited by an enclosing reification, with an opaque view of the effect T. The reflected computation N views this effect transparently. By reifying \(\mathcal{H}\), we can use the user-defined monadic bind to graft the two together.

\[M, N :: \ldots\]
\[| \mu(N)\] reflect
\[| [N]^T\] reify
\[T ::= \ldots\] monads
\[\text{where} \{\text{return } x = M; y \equiv f = N\}\] return clause
\[y \Rightarrow f = N\] bind clause

(a) Syntax (extending Fig. 2)

\[
\begin{align*}
\text{toggle} &= \{x \leftarrow \text{get!}; y \leftarrow \text{not! } x; \text{put! } y; x\} \quad \text{toggle} : U_{\text{State}} F_{\text{bit}} \\
\text{get} &= \{\mu(\lambda s . (s, s))\} \quad \text{get} : U_{\text{State}} F_{\text{bit}} \\
\text{put} &= \{\lambda s' . \mu(\lambda_1 . ((s), s'))\} \quad \text{put} : U_{\text{State}} (\text{bit} \to F_1) \\
\text{State} &= \text{where } \{\text{return } x = \lambda s . (x, s); f \Rightarrow k = \lambda s . (x, s') \leftarrow f s; k! x s'\} \\
\text{runState} &= \{\lambda c . [c!]_{\text{State}}\} \quad \text{runState} : U_0 ((U_{\text{State}} F_{\text{bit}}) \to \text{bit} \to F(\text{bit} \times \text{bit}))
\end{align*}
\]

(b) Operational semantics (extending Fig. 3)

\[\mathcal{F} ::= \mathcal{P} | [\_]^T\] computation frames

Beta reduction

for every \(T = \text{where } \{\lambda x . N_u ; \lambda y . \lambda f . N_b\}\):
\[
\begin{align*}
\text{(ret) } & [\text{return } V]^T \rightarrow_{\beta} N_u[V/x] \\
\text{(reflection) } & [\mathcal{H}[\mu(N)]]^T \rightarrow_{\beta} N_b[[N]/y, \{\lambda x . [\mathcal{H}[\text{return } x]^T]/f\}]
\end{align*}
\]
Example 4.2. With this semantics we have \texttt{runState! toggle True} \sim^* \texttt{return (True, False)}.

The example we have given here fits with the way in which monadic reflection is often used, but is not as flexible as the effect handler version because \texttt{get} and \texttt{put} are concrete functions rather than abstract operations, which means we cannot abstract over how to interpret them. To write a version of \texttt{toggle} that can be interpreted in different ways is possible using monadic reflection but requires more sophistication.

4.3 Type-and-effect system

Fig. 15 presents the natural extension to \texttt{mam}'s kind and type system for monadic reflection. Effects are a stack of monads. The empty effect is the identity monad. A monad \textit{T} can be \textit{layered} on top of an existing stack \textit{E} by \textit{E} \prec \textit{instance monad} \((\alpha.C)\) \textit{where} \{\textit{return} \texttt{x = M; y \gg f = N}\}. The intention is that the type constructor \textit{C}[−/\alpha] has an associated monad structure given by the bodies of the return \textit{M} and the bind \textit{N}, and can use effects from the rest of the stack \textit{E}. To be well-kinded, \textit{C} must be an \textit{E}-computation, and \textit{T} must be a well-typed monad: return should be typed \textit{C}[A/\alpha] when substituted for some value \textit{V} : \textit{A}, and \gg typed as a Kleisli extension operation.

Example 4.3. The boolean state terms are assigned the types given in the right column of Fig. 14.

The choice of keywords for monads and their types follows their syntax in Haskell. We stress that our calculus does not, however, include a type-class mechanism. The \textit{type} of a monad contains the return and bind \textit{terms}, which means that we must check for equality of terms during type-checking, for example, to ensure that we are sequencing two computations with compatible effect annotations (for our purposes \textit{α}-equivalence suffices). The need to check equality of terms arises from our choice of structural, anonymous, monads—in Haskell monads are given nominally, and two monads are compatible if they have exactly the same name. As our monads are structural, the bodies of the return and the bind operations must be closed, apart from their immediate arguments. If layered monad definitions were allowed to contain open terms, types in type contexts would contain these open terms through the effect annotations in thunks, requiring us to support dependently-typed contexts. The monad abstraction is parametric, so naturally requires the use of type variables, and for this reason we include type variables in the base calculus \texttt{mam}. We choose monads to be structural and closed primarily in order to keep them closer to the other abstractions.

Kinds and types

\[
E ::= \ldots | E \prec \text{instance monad} \!(\alpha.C)\! T \quad \text{layered monad}
\]

Effect kinding

\[
\Theta, \alpha \vdash_k C : \text{Comp}_E \vdash_m T : E \prec \text{instance monad} \!(\alpha.C)\! T \quad \Theta \vdash_k E \prec \text{instance monad} \!(\alpha.C)\! T : \text{Eff}
\]

Monad typing

\[
\Theta \vdash_m T : E \quad \Theta, \alpha ; x : \alpha \vdash_N u : C \quad \Theta, \alpha, \beta ; y : U_E C, f : U_E (\alpha \rightarrow C[\beta/\alpha]) \vdash_E N_b : C[\beta/\alpha] \quad \Theta \vdash_m \text{where} \{\text{return} \ x = N_u; \ y \gg f = N_b\} : E \prec \text{instance monad} \!(\alpha.C)\! \text{where} \{\text{return} \ x = N_u; \ y \gg f = N_b\}
\]

Computation typing

\[
\Theta ; \Gamma \vdash N : C[A/\alpha] \quad \Theta ; \Gamma \vdash E \prec \text{instance monad} \!(\alpha.C)\! T \quad \mu(N) : FA \quad \Theta ; \Gamma \vdash_E N : FA \quad \Theta ; \Gamma \vdash_E [N]^T : C[A/\alpha]
\]

Fig. 15. \texttt{mon}'s kinding and typing (extending Fig. 5 and 6)

We extend mam

\[ \exists' T \equiv \text{instance monad} (\alpha.C)N_uN_b \quad \equiv \quad \langle T, \text{return}, \gg \rangle \]

where \( TX := [[C]](\theta[\alpha \rightarrow X]) \) \n
\[ \gg \equiv X,Y := [[N_b]](\theta[\alpha \rightarrow X,\alpha_2 \rightarrow Y]) : TX \rightarrow (X \rightarrow TY) \rightarrow TY \]

(provided these form a monad)

Fig. 16. mon denotational semantics (extending Fig. 7 and 8)

Our calculus differs from Filinski’s [2010] in that our effect definitions are local and structural, whereas his allow nominal declarations of new effects only at the top level. Because we do not allow the bodies of the return and the bind to contain open terms, this distinction between the two calculi is minor. As a consequence, effect definitions in both calculi are static, and the monadic bindings can be resolved at compile time. Filinski’s calculus also includes a sophisticated effect-basing mechanism, that allows a computation to immediately use, via reflection, effects from any layer in the hierarchy below it, whereas our calculus only allows reflecting effects from the layer immediately below. However, effect-basing does not significantly change the expressiveness of the calculus: the monad stack is statically known, and, having access to the type information, we can insert multiple reflection operators and lift effects from lower levels into the current level.

### 4.4 Operational metatheory

We prove mon’s Felleisen-Wright safety in our Abella formalisation:

**Theorem 4.4 (mon safety).** Well-typed programs don’t go wrong: for all closed mon returners \( \Theta; \vdash_0 M : FA \), either \( M \leadsto N \) for some \( \Theta; \vdash_0 N : FA \) or else \( M = \text{return} V \) for some \( \Theta; \vdash V : A \).

As with eff, mon’s ground types are the same as mam’s. While we can define an observational equivalence relation in the same way as for mam and eff, we will not do so. Monads as a programming abstraction have a well-known conceptual complication — user-defined monads must obey the monad laws. These laws are a syntactic counterpart to the three equations in the definition of (set-theoretic/categorical) monads. The difficulty involves deciding what equality between such terms means. The natural candidate is observational equivalence, but as the contexts can themselves define additional monads, it is not straightforward to do so. Giving an acceptable operational interpretation to the monad laws is an open problem. We avoid the issue by giving a partial denotational semantics to mon.

### 4.5 Denotational semantics

We extend mam’s denotational semantics to mon as follows. Given a type variable assignment \( \theta \), we assign to each monad type and effect a monad \( [[\Theta \vdash_m T : E]] \equiv [[\Theta \vdash_k E : \text{Eff}]] \), if the sub-derivations have well-defined denotations, and this data does indeed form a set-theoretic monad. Consequently, the denotation of any derivation is undefined if at least one of its sub-derivations has undefined semantics. Moreover, the definition of kinding judgement denotations now depends on term denotations.

Fig. 16 extends the denotational semantics of mam to mon. The denotation of the layered monad construct is only well-defined if the user-defined type constructor, return, and bind, form a monad. For the denotation of computation terms, recall that \( \Theta \vdash_k FA : \text{Comp}_{E<\text{instance monad}(\alpha.C)T} \) as an E-computation of type \( C[A/\alpha] \).
Compare this semantics with Filinski’s original semantics [1994], in which \([\mu(N)] = [N] \gg \text{id}\), \([\langle N \rangle^T] = \text{return}^T [N]\). To explain the difference, bear in mind that our calculus is based on CBPV, whereas Filinski’s original calculus is based on a pure \(\lambda\)-calculus. Specifically, Filinski interprets the judgement \(M : A\) as \(M : TA\). The corresponding judgement for us is \(M : FA\). The semantics of the pure \(\lambda\)-calculus does not insert monadic returns and binds in the appropriate places, and so Filinski’s translation inserts them explicitly. In contrast, CBPV inserts returns and binds (if the term is pure, they cancel out), and so \text{mon}’s semantics need not add them.

### 4.6 Denotational metatheory

We define a \textit{proper derivation} to be a derivation whose semantics is well-defined for all type variable assignments, and a \textit{proper term or type} to be a term or type that has a proper derivation. Thus, a term is proper when all the syntactic monads it contains denote semantic set-theoretic monads. When dealing with the typed fragment of \text{mon}, we restrict our attention to such proper terms as they reflect the intended meaning of monads. Doing so allows us to mirror the metatheory of \text{mam} and \text{eff} for proper terms.

We define \textit{plugged proper contexts} as with \text{mam} and \text{eff} with the additional requirement that all terms are proper. The definitions of the equivalences \(\simeq\) and \(\simeq_{\text{cong}}\) are then identical to those of \text{mam} and \text{eff}.

**Theorem 4.5 (\text{mon} termination).** \textit{There are no infinite reduction sequences: for all proper \text{mon} terms; \(\vdash_{\emptyset} M : FA\), we have \(M \not\rightarrow^*\), and there exists some unique \(\vdash_{\emptyset} \text{such that} M \not\rightarrow^* \text{return} V\).}  

Our proof uses Lindley and Stark’s \(TT\)-lifting [2005].

**Theorem 4.6 (\text{mon} compositionality).** \textit{The semantics depends only on the semantics of sub-terms: for all pairs of well-typed plugged proper \text{mon} contexts \(M_P, M_Q\) in \(\Xi[\Theta; \Gamma \vdash E P, Q : X]\), if \([P] = [Q]\) then \([M_P] = [M_Q]\).}  

The proof is identical to \text{mam}, with two more cases for \(\rightarrow\_\beta\). Similarly, we have:

**Theorem 4.7 (\text{mon} soundness).** \textit{Reduction preserves the semantics: for every pair of well-typed proper \text{mon} terms \(\Theta; \Gamma \vdash E P, Q : X\), if \(P \simeq_{\text{cong}} Q\) then \([P] = [Q]\). In particular, for every well-typed proper closed term of ground type; \(\vdash_{\emptyset} P : FG\), if \(P \not\rightarrow^* \text{return} V\) then \([P] = [V]\).}  

We combine the previous results, as with \text{mam} and \text{eff}:

**Theorem 4.8 (\text{mon} adequacy).** \textit{Denotational equivalence implies contextual equivalence: for all well-typed proper \text{mon} terms \(\Theta; \Gamma \vdash E P, Q : X\), if \([P] = [Q]\) then \(P \simeq Q\).}  

Therefore, the \textit{proper} fragment of \text{mon} also has a well-behaved operational semantics: for all well-typed proper computations \(\Theta; \Gamma \vdash E M, M' : C\), if \(M \simeq_{\text{cong}} M'\) then \(M \simeq M'\).

In contrast to \text{eff} the semantics for \text{mon} is finite:

**Lemma 4.9 (finite denotation property).** \textit{For every type variable assignment \(\theta = \langle X_{\alpha}\rangle_{\alpha \in \Theta}\) of finite sets, every proper \text{mon} value type \(\Theta \vdash_k A : \text{and computation type} \Theta \vdash_k C : \text{denote finite sets [\(\setminus A\)]_{\emptyset} and [\(\setminus C\)]_{\emptyset}\).}
were to, instead, not re-wrap the continuation with a reset, we would obtain the control/prompt-zero
delimit any continuations captured by shift inside. The extension to the operational semantics in Fig. 17(b) reflects our informal description. The rule states that once the delimited computation returns a value, this value is substituted in the remainder of the reset computation. After the reduction takes place, the continuation is re-wrapped with the reset, while the body of the shift has access to the enclosing continuation. If we discard the state when we reach the final return value.

Frame and contexts
\[
M, N ::= \ldots \quad \text{computation frames}
\]

(a) Syntax (extending Fig. 2)
\[
| S_0 k.M \quad \text{shift-0} | (M|x.N) \quad \text{reset}
\]

(b) Operational semantics (extending Fig. 3)
\[
F ::= \ldots | \langle [ ] | x.N \rangle \quad \text{computation frames}
\]

Beta reduction
\[
\begin{align*}
\text{(ref)} & \quad \langle (\text{return } V)|x.M \rangle \rightarrow_{\beta} M[V/x] \\
\text{(capture)} & \quad \langle \mathcal{H}[S_0 k.M]|x.N \rangle \rightarrow_{\beta} M[\lambda y. \langle \mathcal{H}[\text{return } y]|x.N\rangle/k]
\end{align*}
\]

5.1 Syntax
Fig. 17(a) presents the extension \textsc{del}. The construct \textit{S}_0 k.M, which we abbreviate to “shift”, captures the current continuation and binds it to \textit{k}, and replaces it with \textit{M}. The construct \langle \textit{M}|x.N \rangle, which we will call “reset”, delimits any continuations captured by shift inside \textit{M}. Once \textit{M} runs its course and returns a value, this value is bound to \textit{x} and \textit{N} executes. For delimited control cognoscenti this construct is sometimes called “dollar”, and can macro express the entire CPS hierarchy [Materzok and Biernacki 2012; Kiselyov and Shan 2007].

Example 5.1. The left column of Fig. 18 expresses user-defined boolean state in \textsc{del} [Danvy 2006, Section 1.4]. The code assumes the environment outside the closest reset will apply it to the currently stored state. By shifting and abstracting over this state, \textit{get} and \textit{put} can access this state and return the appropriate result to the continuation. When running a stateful computation, we discard the state when we reach the final return value.

5.2 Operational semantics
The extension to the operational semantics in Fig. 17(b) reflects our informal description. The \textit{ref} rule states that once the delimited computation returns a value, this value is substituted in the remainder of the reset computation. For the \textit{capture} rule, the definition of pure contexts guarantees that in the reduct \langle \mathcal{H}[S_0 k.M]|x.N \rangle there are no intervening resets in \mathcal{H}, and as a consequence \mathcal{H} is the delimited continuation of the evaluated shift. After the reduction takes place, the continuation is re-wrapped with the reset, while the body of the shift has access to the enclosing continuation. If we were to, instead, not re-wrap the continuation with a reset, we would obtain the control/prompt-zero operators, (cf. Shan’s [2007] and Kiselyov et al.’s [2005] analyses of macro expressivity relationships between these two, and other, variations on untyped delimited control).

Example 5.2. We have: \textit{runState! toggle} True \rightarrow^* \langle \text{True}|x.\lambda s.s \rangle \textit{False} \rightarrow^* \textit{return} True.

5.3 Type-and-effect system
Fig. 19 presents the natural extension to \textsc{mam}’s kind and type system for delimited control. It is based on Danvy and Filinski’s description [Danvy and Filinski 1989]; they were the first to propose

\[
\begin{align*}
toggle & \quad = \{ \text{x \leftarrow \text{get}!}; \text{y \leftarrow not! x}; \text{put! y}; \text{x} \} \quad \text{toggle : UStateFbit} \\
get & \quad = \{S_0 k.\lambda s.k! s s\} \quad \text{get : UStateFbit} \\
put & \quad = \{\lambda s’.S_0 k.\lambda s.k! () s’\} \quad \text{put : UState(bit \rightarrow F1)} \\
runState & \quad = \{\lambda c. \langle c!|x.\lambda s.s \rangle \} \quad \text{runState : U}_0(\langle UStateFbit \rangle \rightarrow bit \rightarrow Fbit) \\
State & \quad = 0, bit \rightarrow Fbit : Eff
\end{align*}
\]

Fig. 18. User-defined boolean state in \textsc{del}.
**Kinds and types**

\[ E ::= \ldots \text{ effects} \]

\[ E, C \text{ enclosing continuation type} \]

**Effect kinding**

\[ \Theta \vdash_k E : \text{Eff} \]

\[ \Theta \vdash_k E, C : \text{Eff} \]

**Computation typing**

\[ \Theta; \Gamma, k : U_E(A \rightarrow C) \vdash_E M : C \]

\[ \Theta; \Gamma \vdash_E C : \text{Comp}_E \]

\[ \Theta; \Gamma \vdash E, C : \text{Eff} \]

\[ \Theta; \Gamma \vdash E, C : \text{Eff} \]

Fig. 19. DEL’s kinding and typing (extending Fig. 5 and 6)

a type system for delimited control. Effects are now a stack of computation types, with the empty effect standing for the empty stack. The top of this stack is the return type of the currently delimited continuation. Thus, as Fig. 19 presents, a shift pops the top-most type off this stack and uses it to type the current continuation, and a reset pushes the type of the delimited return typed onto it.

**Example 5.3.** The boolean state terms are assigned the types given in the right column of Fig. 18.

In this type system, the return type of the continuation remains fixed inside every reset. Existing work on type systems for delimited control (Kiselyov and Shan [2007] provide a substantial list of references) focuses on type systems that allow answer-type modification, as these can express typed printf and type-state computation (as in Asai’s analysis [2009]). We exclude answer-type modification to keep the fundamental account clearer and simpler: the type system with answer-type modification is further removed from the well-known abstractions for effect-handlers and monadic reflection. We conjecture that the relative expressiveness of delimited control does not change even with answer-type modification, once we add analogous capabilities to effect handlers [Brady 2013; Kiselyov 2016] and monadic reflection [Atkey 2009].

5.4 **Operational metatheory**

Our Abella formalisation establishes:

**Theorem 5.4 (DEL safety).** Well-typed programs don’t go wrong: for all closed DEL returners \( \Theta; \Gamma_0 M : FG \), either \( M \leadsto N \) for some \( \Theta; \Gamma_0 N : FG \) or else \( M = \text{return} V \) for some \( \Theta; \Gamma \vdash V : G \).

In the next section, we extend DEL’s metatheory using the translation from DEL to MON.

We define DEL’s ground types, plugged contexts and the equivalences \( \equiv \) and \( \equiv_{\text{cong}} \) as in MAM.

6 **MACRO TRANSLATIONS**

Felleisen [1991] argues that the usual notions of computability and complexity reduction do not capture the expressiveness of general-purpose programming languages. The Church-Turing thesis and its extensions assert that any reasonably expressive model of computation can be efficiently reduced to any other reasonably expressive model of computation. Thus the notion of a polynomial-time reduction with a Turing-machine is too crude to differentiate expressive power of two general-purpose programming languages. As an alternative, Felleisen introduces macro translation: a local reduction of a language extension, in the sense that it is homomorphic with respect to the syntactic constructs, and conservative, in the sense that it does not change the core language. We adapt this concept to local translations between conservative extensions of a shared core.

**Translation notation.** We define translations \( S \rightarrow T \) from each source calculus \( S \) to each target calculus \( T \). By default we assume untyped translations, writing \( \text{EFF}, \text{MON}, \text{and DEL} \) in translations.

that disregard typeability. In typeability preserving translations, which must also respect the monad laws where \( \text{MON} \) is concerned, we explicitly write \( \text{TYPED EFF}, \text{TYPED MON}, \) and \( \text{TYPED DEL} \). We allow translations to be \textit{hygienic} and introduce fresh binding occurrences. We write \( M \mapsto M \) for the translation at hand. We include only the non-core cases in the definition of each translation.

Out of the six possible untyped macro-translations, the ideas behind the following four already appear in the literature: \( \text{DEL} \mapsto \text{MON} \) [Wadler 1994], \( \text{MON} \mapsto \text{DEL} \) [Filinski 1994], \( \text{DEL} \mapsto \text{EFF} \) [Bauer and Pretnar 2015], and \( \text{EFF} \mapsto \text{MON} \) [Kammar et al. 2013]. The Abella formalisation contains the proofs of the simulation results for each of the six translations. Three translations formally simulate the source calculus by the target calculus: \( \text{MON} \mapsto \text{DEL} \), \( \text{DEL} \mapsto \text{EFF} \), and \( \text{MON} \mapsto \text{EFF} \). The other translations, \( \text{DEL} \mapsto \text{MON} \), \( \text{EFF} \mapsto \text{DEL} \), and \( \text{EFF} \mapsto \text{MON} \), introduce suspended redexes during reduction that invalidate simulation on the nose.

For the translations that introduce suspended redexes, we use a relaxed variant of simulation, namely the relations \( \sim_{\text{cong}} \), which are the smallest relations containing \( \rightarrow \) that are closed under the term formation constructs. We say that a translation \( M \mapsto M \) is a simulation up to congruence if for every reduction \( M \rightarrow N \) in the source calculus we have \( M \rightarrow^* N \) in the target calculus. In fact, the suspended redexes always \( \beta \)-reduce by substituting a variable, i.e., \( \{\lambda x.M\}! x \rightarrow^* \lambda x.M \), thus only performing simple rewiring.

### 6.1 Delimited continuations as monadic reflection (\( \text{DEL} \mapsto \text{MON} \))

We adapt Wadler’s analysis of delimited control [1994], using the continuation monad [Moggi 1989]:

**Lemma 6.1.** For all \( \Theta \vdash_k E : \text{Eff}, \Theta \vdash_k C : \text{Comp}_E \), we have the following proper monad \( \text{Cont} \):

\[
\Theta \vdash_k E \triangleleft \text{instance monad} (\alpha.U_E (\alpha \rightarrow C) \rightarrow C) \text{ where } \{\text{return } x = \lambda c.c! x; m \triangleright f = \lambda c.m! \{\lambda y.f! y c\}\} : \text{Eff}
\]

Using \( \text{Cont} \) we define the macro translation \( \text{DEL} \mapsto \text{MON} \) as follows:

\[
\begin{align*}
S_{nk.M} &:= \mu(\lambda k.M) \\
\langle M|x.N \rangle &:= [M]^{\text{Cont}} \{\lambda x.N\}
\end{align*}
\]

Shift is interpreted as reflection and reset as reification in the continuation monad.

**Theorem 6.2 (\( \text{DEL} \mapsto \text{MON} \) correctness).** \( \text{MON} \) simulates \( \text{DEL} \) up to congruence:

\[
M \rightarrow N \implies M \rightarrow^* N
\]

The only suspended redex arises in simulating the reflection rule, where we substitute a continuation into the bind of the continuation monad yielding a term of the form \( \{\lambda y.\{\lambda y.M\} y c\} \) which we must reduce to \( \{\lambda y.M c\} \).

\( \text{DEL} \mapsto \text{MON} \) extends to a macro translation at the type level:

\[
E, C := E \triangleleft \text{instance monad} (\alpha.U_E (\alpha \rightarrow C) \rightarrow C) \text{ Cont}
\]

**Theorem 6.3 (\( \text{DEL} \mapsto \text{MON} \) preserves typeability).** Every well-typed \( \text{DEL} \) phrase \( \Theta ; \Gamma \vdash_E P : X \) translates into a proper well-typed \( \text{MON} \) phrase: \( \Theta ; \Gamma \vdash_E P : X. \)

We use this result to extend the metatheory of \( \text{DEL} \):

**Corollary 6.4 (\( \text{DEL} \) termination).** All well-typed closed ground returners in \( \text{DEL} \) must reduce to a unique normal form: \( \vdash_0 M : FG \) then there exists \( V \) such that \( \vdash V : G \) and \( M \mapsto^* \text{return } V. \)
6.2 Monadic reflection as delimited continuations (MON→DEL)

We define the macro translation \( \text{MON} \rightarrow \text{DEL} \) as follows:

\[
\mu(M) := S_0k.\lambda b. b! (\{M\}, \{\lambda x.k! x b\})
\]

\[
[M] \text{ where } \{\text{return } x = N_0; y := f = N_0\} := \left( M \mid x.\lambda b. N_0 \right) \{\lambda (y, f). N_0\}
\]

Reflection is interpreted by capturing the current continuation and abstracting over the bind operator which is then invoked with the reflected computation and a function that wraps the continuation in order to ensure it uses the same bind operator. Reification is interpreted as an application of a reset. The continuation of the reset contains the unit of the monad. We apply this reset to the bind of the monad.

**Theorem 6.5 (MON→DEL correctness).** \( \text{DEL} \) simulates \( \text{MON} \) up to congruence:

\[
M \sim N \implies M \rightsquigarrow^+ N
\]

This translation does not preserve typeability because the bind operator can be used at different types. We conjecture that a) any other macro translation will suffer from the same issue and b) adding (predicative) polymorphism to the base calculus is sufficient to adapt this translation to one that does preserve typeability.

Filinski’s translation from monadic reflection to delimited continuations [1994] does preserve typeability, but it is a global translation. It is much like our translation except each instance of bind is inlined (hence it does not need to be polymorphic).

6.2.1 Alternative translation with nested delimited continuations. An alternative to \( \text{MON} \rightarrow \text{DEL} \) is to use two nested shifts for reflection and two nested resets for reification:

\[
\mu(M) := S_0k. S_0b. b! (\{M\}, \{\lambda x. (k! x | z! b)\})
\]

\[
[M] \text{ where } \{\text{return } x = N_0; y := f = N_0\} := \left( M \mid x.\lambda b. N_0 \right) \{\lambda (y, f). N_0\}
\]

In the translation of reflection, the reset inside the wrapped continuation ensures that any further reflections in the continuation are interpreted appropriately: the first shift, which binds \( k \), has popped one continuation off the stack so we need to add one back on. In the translation of reification, the shift guarding the unit garbage collects the bind once it is no longer needed.

6.3 Delimited continuations as effect handlers (DEL→EFF)

We define \( \text{DEL} \rightarrow \text{EFF} \) as follows:

\[
S_0k.N := \text{shift0 } \{\lambda k.N\} \quad \langle M|x.N \rangle := \text{handle } M \text{ with } \{\text{return } x \mapsto N\} \uplus \{\text{shift0 } y f \mapsto f! y\}
\]

Shift is interpreted as an operation and reset is interpreted as a straightforward handler.

**Theorem 6.6 (DEL→EFF correctness).** \( \text{EFF} \) simulates \( \text{DEL} \) on the nose: \( M \sim N \implies M \rightsquigarrow^+ N \).

This translation does not preserve typeability because inside a single reset shifts can be used at different types. We conjecture that a) any other macro translation will suffer from the same issue and b) adding polymorphic operations [Kammar et al. 2013] to \( \text{EFF} \) is sufficient to ensure this translation does preserve typeability.

One can adapt our translation to a global translation in which every static instance of a shift is interpreted as a separate operation, thus avoiding the need for polymorphic operations.
6.4 Effect handlers as delimited continuations (\(\text{EFF}\rightarrow\text{DEL}\))

We define \(\text{EFF}\rightarrow\text{DEL}\) as follows:

\[
\begin{align*}
\text{op} V & := S_0 k . \lambda h . h ! (\text{inj}_\text{op} (V, \lambda x. (\lambda y. k ! y) h )) \\
\text{handle } M \text{ with } H & := (M \dagger \text{eff}) \{ \text{Hops} \} \\
\text{handle } M \text{ with } \begin{cases}
\text{return } x \mapsto \text{Nret} \\
\cup \{ \text{op}_i p k \mapsto \text{N}_i \}_i
\end{cases} & \Rightarrow x \lambda h . \text{Nret} \begin{cases}
\text{return } x \mapsto \text{Nret} \\
\cup \{ \text{op}_i p k \mapsto \text{N}_i \}_i
\end{cases} \begin{cases}
y . \text{case } y \text{ of } \{ \\
\} \Rightarrow \text{inj}_{\text{op}_i} (p, k) \mapsto \text{N}_i \}
\end{align*}
\]

Operation invocation is interpreted by capturing the current continuation and abstracting over a dispatcher which is passed an encoding of the operation. The encoded operation is an injection whose label is the name of the operation containing a pair of the operation parameter and a wrapped \(\text{mon}\) 6.5 Monadic reflection as effect handlers (\(\text{MON}\rightarrow\text{EFF}\))

We simulate reflection with an operation and reification with a handler. Formally, for every anonymous monad \(T\) given by \(\text{where } \{ \text{return } x = \text{N}_0 ; y \Rightarrow f = \text{N}_b \} \) we define \(\text{MON}\rightarrow\text{EFF}\) as follows:

\[
\begin{align*}
\mu(N) & := \text{reflect } \{ N \} \\
[M]_T & := \text{handle } M \text{ with } T \\
T & := \{ \text{return } x \mapsto \text{N}_0 \} \cup \{ \text{reflect } y f \mapsto \text{N}_b \}
\end{align*}
\]

Reflection is interpreted as a reflect operation and reification as a handler with the unit of the monad as a handler and the bind of the handler as the implementation of the reflect operation.

**THEOREM 6.8 (\(\text{MON}\rightarrow\text{EFF} \text{ CORRECTNESS}.\))** \(\text{EFF} \text{ simulates } \text{MON on the nose}: M \leadsto N \implies M \leadsto^+ N.\)

\(\text{MON}\rightarrow\text{EFF}\) does not preserve typeability. For instance, consider the following computation of type \(\text{Fbit}\) using the environment monad Reader given on the right:

\[
\begin{align*}
\llbracket b \rrbracket & \leftarrow \mu (\lambda (b, f ). b) ; \\
f & \leftarrow \mu (\lambda (b, f ). f) ; \\
f f \rrbracket \text{Reader} (\text{inj}_{\text{true}} () , \lambda b. \text{return } b) \end{align*}
\]

\(\llbracket b \rrbracket \leftarrow \mu (\lambda (b, f ). b) ; \quad r_k \emptyset \prec \text{instance monad } \left( \alpha . \text{bit } \times \text{U}_0 \left( \text{bit } \rightarrow \text{Fbit} \right) \rightarrow \text{F} \alpha \right) \)

\[
\begin{align*}
f & \leftarrow \mu (\lambda (b, f ). f) ; \\
f f \rrbracket \text{Reader} (\text{inj}_{\text{true}} () , \lambda b. \text{return } b) \end{align*}
\]

\(\llbracket b \rrbracket \leftarrow \mu (\lambda (b, f ). b) ; \quad m \Rightarrow f = \lambda e . x \leftarrow m! e ; \quad f ! x e \} : \text{Eff}\)
We define eff which it abstracts. We conjecture that a) this observation can be used to prove that no macro translation \textsc{typed mon}→\textsc{typed eff} exists and that b) adding polymorphic operations [Kammar et al. 2013] to eff is sufficient for typing this translation.

### 6.6 Effect handlers as monadic reflection (\textsc{eff}→\textsc{mon})

We define \textsc{eff}→\textsc{mon} as follows:

\[
\begin{align*}
\text{op } V & := \mu (\lambda k. \lambda h. h!(\text{inj}_\text{op}(V, \lambda y. k! y h))) \\
\text{handle } M \text{ with } H & := [M]^{\text{Cont}} \{H^{\text{ret}}\} \{H^{\text{ops}}\}
\end{align*}
\]

The translation is much like \textsc{eff}→\textsc{del}, using the continuation monad in place of first class continuations.

Operation invocation is interpreted by using reflection to capture the current continuation and abstracting over a dispatcher which is passed an encoding of the operation. The encoded operation is an injection whose label is the name of the operation containing a pair of the operation parameter and a wrapped version of the captured continuation, which ensures the same dispatcher is threaded through the continuation.

Handling is interpreted as an application of a reified continuation monad computation to the return clause and a dispatcher function that encodes the operation clauses.

**Theorem 6.9 (\textsc{eff}→\textsc{mon} correctness).** \textsc{mon} simulates \textsc{eff} up to congruence:

\[ M \sim N \Rightarrow M \sim^{+}_{\text{cong}} N \]

This translation does not preserve typeability for the same reason as the \textsc{eff}→\textsc{del} translations: the interpretation of operations needs to be polymorphic in the return type of the dispatcher over which it abstracts. We conjecture that a) any other macro translation will suffer from the same issue and b) adding polymorphism to the base calculus is sufficient to adapt this translation to one that does preserve typeability.

#### 6.6.1 Alternative translation using a free monad

An alternative to interpreting effect handlers using a continuation monad is to use a free monad:

\[
\begin{align*}
\text{op } V & := \mu (\text{return } (\text{inj}_\text{op}(V, \lambda x. \text{return } x))) \\
\text{handle } M \text{ with } H & := H^{*} [M]^{H^{*}}
\end{align*}
\]

where

\[
\begin{align*}
\{ \text{return } x \mapsto N_{\text{ret}} \}^{+} & := \text{return } x = \text{return } (\text{inj}_{\text{ret}} x); \\
\cup \{ \text{op}_i p k \mapsto N_i \}_i & := \text{case } y \text{ of } \{ \text{inj}_{\text{ret}} x \mapsto k! x \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{return } x \mapsto N_{\text{ret}} \}^{*} & := h = \lambda y. \text{case } y \text{ of } \{ \text{inj}_{\text{ret}} x \mapsto N_{\text{ret}} \}
\end{align*}
\]

Both the bind operation for the free monad \( H^{+} \) and the function \( h \) that interprets the free monad \( H^{*} \) are recursive. Given that we are in an untyped setting we can straightforwardly implement the recursion using a suitable variation of the \( Y \) combinator. This translation does not extend to the typed calculi as they do not support recursion. Nevertheless, we conjecture that it can be adapted to a typed translation if we extend our base calculus to include inductive data types, as the recursive functions are structurally recursive.
6.7 Nonexistence results

Theorem 6.10. The following macro translations do not exist:

- Typed eff $\rightarrow$ Typed mon satisfying: $M \rightsquigarrow N \implies M \simeq N$.
- Typed eff $\rightarrow$ Typed del satisfying: $M \rightsquigarrow N \implies M \simeq N$.

Our proof of the first part hinges on the finite denotation property (Lemma 4.9). Briefly, assume to the contrary that there was such a translation. Consider a single effect operation symbol $\text{tick} : 1 \rightarrow 1$ and set $\text{tick}^0 := \text{return } ()$, and $\text{tick}^{n+1} := \text{tick}() \cdot \text{tick}^n$. All these terms have the same type, and by the homomorphic property of the hypothesised translation, their translations all have the same type. By the finite denotation property two of them are observationally equivalent and by virtue of a macro translation the two source terms are observationally equivalent in eff. But every distinct pair of $\text{tick}^n$ terms is observationally distinguishable using an appropriate handler. See Forster’s thesis [2016] for the full details. The second part follows from Theorem 6.3.

Regarding the remaining four possibilities, we have seen that there is a typeability-preserving macro translation Typed del $\rightarrow$ Typed mon (Theorem 6.3), but we conjecture that there are no typeability-preserving translations Typed mon $\rightarrow$ Typed del, Typed del $\rightarrow$ Typed eff, or Typed mon $\rightarrow$ Typed eff.

7 CONCLUSION AND FURTHER WORK

We have given a uniform family of formal calculi expressing the common abstractions for user-defined effects: effect handlers (eff), monadic reflection (mon), and delimited control (del), together with their natural type-and-effect systems. We have used these calculi to formally analyse the relative expressive power of these abstractions. Effect handlers, monadic reflection, and delimited control have equivalent expressivity when types are not taken into consideration. However, neither monadic reflection nor delimited control can macro-express effect handlers whilst preserving typeability. We have formalised the more syntactic aspects of our work in the Abella proof assistant, and have used set-theoretic denotational semantics to establish inexpressivity results.

Our work has already born unexpected if not entirely unsurprising fruit. By composing our translation from effect handlers to delimited continuations with a CPS translation for delimited continuations Hillerström et al. [2017] derived a CPS translation for effect handlers, which they then used as the basis for an implementation.

Further work abounds. We want to extend each type system until each translation preserves typeability. We conjecture that adding polymorphic operations to eff would allow it to macro express del and mon, and that adding polymorphism to mon and del would allow them to macro express eff. We conjecture polymorphism would also allow del to macro express mon, and inductive data types with primitive recursion would also allow mon to macro express eff.

We are also interested in analysing global translations between these abstractions. In particular, whereas mon and del allow reflection/shifts to appear anywhere inside a piece of code, in practice, library designers define a fixed set of primitives using reflection/shifts and only expose those primitives to users. This observation suggests calculi in which each reify/reset is accompanied by declarations of this fixed set of primitives. We conjecture that mon and del can be simulated on the nose via a global translation into the corresponding restricted calculus, and that the restricted calculi can be macro translated into eff whilst preserving typeability. Such two-stage translations would give a deeper reason why so many examples typically used for monadic reflection and delimited control can be directly recast using effect handlers. Other global pre-processing may also eliminate administrative reductions from our translations and establish simulation on the nose.

We hope the basic calculi we have analysed will form a foundation for systematic further investigation. Supporting answer-type modification [Asai 2009; Kobori et al. 2015] can inform
more expressive type system design for effect handlers and monadic reflection, and account for type-state [Atkey 2009] and session types [Kiselyov 2016]. In practice, effect systems are often extended with sub-effecting or effect polymorphism [Lucassen and Gifford 1988; Bauer and Pretnar 2014; Pretnar 2014; Leijen 2017; Hillerström and Lindley 2016; Lindley et al. 2017]. To these we add effect forwarding [Kammar et al. 2013] and rebasing [Filinski 2010].

We have taken the perspective of a programming language designer deciding which programming abstraction to select for expressing user-defined effects. In contrast, Schrijvers et al. [2016] take the perspective of a library designer for a specific programming language, Haskell, and compare the abstractions provided by libraries based on monads with those provided by effect handlers. They argue that both libraries converge on the same interface for user-defined effects via Haskell’s type-class mechanism.

Relative expressiveness results are subtle, and the potentially negative results that are hard to establish make them a risky line of research. We view denotational models as providing a fruitful method for establishing such inexpressivity results. It would be interesting to connect our work with that of Laird [2002, 2013, 2017], who analyses the macro-expressiveness of a hierarchy of combinations of control operators and exceptions using game semantics, and in particular uses such denotational techniques to show certain combinations cannot macro express other combinations.

We would like to apply similar techniques to compare the expressive power of local effects such as ML-style reference cells with effect handlers.

ACKNOWLEDGMENTS

Supported by the European Research Council grant 'events causality and symmetry — the next-generation semantics', and the Engineering and Physical Sciences Research Council grants EP/H005633/1 'Semantic Foundations for Real-World Systems', EP/K034413/1 'From Data Types to Session Types—A Basis for Concurrency and Distribution’, and EP/N007387/1 'Quantum computation as a programming language’. The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096. We thank Bob Atkey, Andrej Bauer, Paul Downen, Marcelo Fiore, Tamara von Glehn, Mathieu Huot, Daan Leijen, Craig McLaughlin, Kayvan Memarian, Sean Moss, Alan Mycroft, Ian Orton, Hugo Paquet, Jean Pichon-Pharabod, Matthew Pickering, Didier Remy, Reuben Rowe, Philip Saville, Ian Stark, Sam Staton, Philip Wadler, and Jeremy Yallop for useful suggestions and discussions.

REFERENCES


Olivier Danvy. 2006. An Analytical Approach to Programs as Data Objects. DSc dissertation. Department of Computer Science, University of Aarhus.


Andrzej Filinski. 1994. Representing Monads. In *POPL. ACM.*
Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP. ACM*, 15–27.
Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *POPL. ACM.*
Oleg Kiselyov. 2016. Parameterized extensible effects and session types (extended abstract). In *TyDe@ICFP. ACM*, 41–42.
On the Expressive Power of User-Defined Effects


Received February 2017; Accepted May 2017