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:
Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. **On the Expressive Power of User-Defined Effects.** ***Proc. ACM Program. Lang.*** **1, ICFP, Article 13 (September 2017), 29 pages.**
https://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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2017 Copyright held by the owner/author(s).
2475-1421/2017/9-ART13
https://doi.org/10.1145/3110257

calculus, like β-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 $\mathcal{L}$ is extended by some feature, we say that the extended language $\mathcal{L}_+$ is *macro expressible* when there is a local syntax-directed translation (a *macro translation*) from $\mathcal{L}_+$ to $\mathcal{L}$ that keeps the features in $\mathcal{L}$ 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 $\mathcal{L}_+^1$ of a base calculus $\mathcal{L}$ is macro expressible in another extension $\mathcal{L}_+^2$ of the same base calculus $\mathcal{L}$. 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

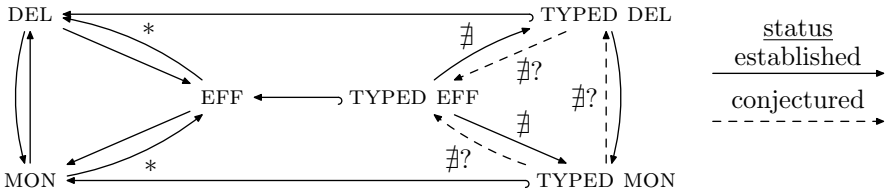

Fig. 1. Existing and conjectured macro translations

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 $\not\equiv$ 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;

$V, W ::=$	values	$M, N ::=$	computations	$\mid \mathbf{return}\ V$	returner
x	variable	$\mathbf{case}\ V\ \mathbf{of}$	product	$\mid x \leftarrow M;\ N$	sequencing
$\mid ()$	unit value	$\quad (x_1, x_2) \to M$	matching	$\mid \lambda x.M$	abstraction
$\mid (V_1, V_2)$	pairing	$\mid \mathbf{case}\ V\ \mathbf{of}\ \{$	variant	$\mid M\ V$	application
$\mid \mathbf{inj}_\ell\ V$	variant	$\quad (\mathbf{inj}_{\ell_i}\ x_i \to M_i)_i\}$	matching	$\mid \langle M_1, M_2 \rangle$	pairing
$\mid \{M\}$	thunk	$\mid V!$	force	$\mid \mathbf{prj}_i\ M$	projection

Fig. 2. MAM syntax

- formally mechanised metatheory in Abella[1] 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, ..., and a countable set of variant constructor literals ranged over by ℓ. 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.

[1] https://github.com/matijapretnar/user-defined-effects-formalization

Frames and contexts

$\mathcal{P}$	$::= x \leftarrow [\,];\ N \mid [\,]\ V \mid \mathbf{prj}_i\ [\,]$	pure frames
$\mathcal{F}$	$::= \mathcal{P}$	computation frames
$\mathcal{C}$	$::= [\,] \mid \mathcal{C}[\mathcal{F}[\,]]$	evaluation context
$\mathcal{H}$	$::= [\,] \mid \mathcal{H}[\mathcal{P}[\,]]$	pure context

Reduction $\boxed{M \rightsquigarrow M'}$

$$\frac{M \rightsquigarrow_\beta M'}{\mathcal{C}[M] \rightsquigarrow \mathcal{C}[M']}$$

Beta reduction $\boxed{M \rightsquigarrow_\beta M'}$

$$(\times)\quad \mathbf{case}\ (V_1, V_2)\ \mathbf{of}\ (x_1, x_2) \to M \rightsquigarrow_\beta M[V_1/x_1, V_2/x_2] \qquad (U)\quad \{M\}! \rightsquigarrow_\beta M$$
$$(+)\quad \mathbf{case}\ \mathbf{inj}_\ell\ V\ \mathbf{of}\ \{\ldots \mathbf{inj}_\ell\ x \to M \ldots\} \rightsquigarrow_\beta M[V/x] \qquad (\to)\quad (\lambda x.M)\ V \rightsquigarrow_\beta M[V/x]$$
$$(F)\quad x \leftarrow \mathbf{return}\ V;\ M \rightsquigarrow_\beta M[V/x] \qquad (\&)\quad \mathbf{prj}_i\ \langle M_1, M_2\rangle \rightsquigarrow_\beta M_i$$

Fig. 3. MAM operational semantics

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

$$\begin{aligned} \mathit{not} = \{\lambda b.\mathbf{case}\ \ b\ \mathbf{of}\ \{&\mathbf{inj}_{\mathtt{True}}\ \ x \to \mathbf{return}\ \mathbf{inj}_{\mathtt{False}}\ () \\ &\mathbf{inj}_{\mathtt{False}}\ x \to \mathbf{return}\ \mathbf{inj}_{\mathtt{True}}\ \ ()\}\} \end{aligned}$$

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 β-reduction rules and a single congruence rule. As usual, a β-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 ($\mathcal{C}[\mathcal{F}[\,]]$, $\mathcal{C}[M]$) 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 β-reducible term, making the semantics deterministic.

Example 2.2. With this semantics we have $\mathit{not}!\ (\mathbf{inj}_{\mathtt{True}}\ ()) \rightsquigarrow^+ \mathbf{return}\ \mathbf{inj}_{(\mathtt{False}())}$.

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 $\mathbf{inj}_{\mathtt{True}}\ ()$. We allow the application of functions and the elimination constructs to apply to arbitrary computations, and not just values, by setting for example $M\ N := x \leftarrow N;\ M\ x$ 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 $\mathit{runState}!\ \mathit{toggle}\ \mathtt{True} \rightsquigarrow^\star$ `(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 MAM's types and effects. As a core calculus for three calculi with different notions of effect, 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 **Monad** $m \implies m\,a$ 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 **bit** is given by $\{\mathbf{inj}_{\mathsf{False}}\ 1, \mathbf{inj}_{\mathsf{True}}\ 1\}$.

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

$$
\begin{array}{l}
toggle = \{\, x \leftarrow get!; \\
\qquad y \leftarrow not!\ x; \\
\qquad put!\ y; \\
\qquad x\}
\end{array}
$$

(a) Direct style

$$
\begin{array}{lll}
get & = \{ & \lambda s.(\ s, s\ )\} \\
put & = \{ & \lambda s'.\lambda\_.((), s')\} \\
runState & = & \lambda c.\lambda s.c!\ s
\end{array}
\qquad
\begin{array}{l}
toggle = \{\lambda s.\ (x, s) \leftarrow get!\ s; \\
\qquad\qquad y \leftarrow not!\ x; \\
\qquad\qquad (\_, s) \leftarrow put!\ y\ s; \\
\qquad\qquad (x, s)\}
\end{array}
$$

(b) State-passing style

Fig. 4. User-defined boolean state

$E ::=$	effects	$A, B ::=$	value types	$C, D ::=$	computation types
$\emptyset$	pure effect	α	type variable	FA	returners
$K ::=$	kinds	$\mid 1$	unit	$\mid A \to C$	functions
$\mid$ **Eff**	effects	$\mid A_1 \times A_2$	products	$\mid C_1\ \&\ C_2$	products
$\mid$ **Val**	values	$\mid \{\mathbf{inj}_{\ell_i}\ A_i\}_i$	variants	Environments:	
$\mid$ **Comp**$_E$	computations	$\mid U_E C$	thunks	$\Theta ::= \alpha_1, \ldots, \alpha_n$	
$\mid$ **Context**	environments			$\Gamma, \Delta ::= x_1 : A_1, \ldots, x_n : A_n$	

Fig. 5. MAM kinds and types

Effect kinding $\boxed{\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}}$

$$\frac{}{\Theta \vdash_{\mathrm{k}} \emptyset : \mathbf{Eff}}$$

Context kinding $\boxed{\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}}$

$$\frac{[\Theta \vdash_{\mathrm{k}} \Gamma(x) : \mathbf{Val}]_{x \in \mathrm{Dom}(\Gamma)}}{\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}}$$

Value kinding $\boxed{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}}$

$$\frac{\alpha \in \Theta}{\Theta \vdash_{\mathrm{k}} \alpha : \mathbf{Val}} \qquad \frac{}{\Theta \vdash_{\mathrm{k}} 1 : \mathbf{Val}} \qquad \frac{\Theta \vdash_{\mathrm{k}} A_1 : \mathbf{Val} \qquad \Theta \vdash_{\mathrm{k}} A_2 : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} A_1 \times A_2 : \mathbf{Val}}$$

$$\frac{[\Theta \vdash_{\mathrm{k}} A_i : \mathbf{Val}]_i}{\Theta \vdash_{\mathrm{k}} \{\mathbf{inj}_{\ell_i}\, A_i\}_i : \mathbf{Val}} \qquad \frac{\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} U_E C : \mathbf{Val}}$$

Computation kinding $\boxed{\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}$ $(\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff})$

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} FA : \mathbf{Comp}_E} \qquad \frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \qquad \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} A \to C : \mathbf{Comp}_E} \qquad \frac{\Theta \vdash_{\mathrm{k}} C_1 : \mathbf{Comp}_E \quad \Theta \vdash_{\mathrm{k}} C_2 : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} C_1 \,\&\, C_2 : \mathbf{Comp}_E}$$

Value typing $\boxed{\Theta; \Gamma \vdash V : A}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}, A : \mathbf{Val})$

$$\frac{(x : A) \in \Gamma}{\Theta; \Gamma \vdash x : A} \qquad \frac{}{\Theta; \Gamma \vdash () : 1} \qquad \frac{\Theta; \Gamma \vdash V_1 : A_1 \qquad \Theta; \Gamma \vdash V_2 : A_2}{\Theta; \Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Theta; \Gamma \vdash V : A_i}{\Theta; \Gamma \vdash \mathbf{inj}_{\ell_i}\, V : \{\mathbf{inj}_{\ell_i}\, A_i\}_i} \qquad \frac{\Theta; \Gamma \vdash_E M : C}{\Theta; \Gamma \vdash \{M\} : U_E C}$$

Computation typing $\boxed{\Theta; \Gamma \vdash_E M : C}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}, E : \mathbf{Eff}, C : \mathbf{Comp}_E)$

$$\frac{\Theta; \Gamma \vdash V : A_1 \times A_2 \qquad \Theta; \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Theta; \Gamma \vdash_E \mathbf{case}\ V\ \mathbf{of}\ (x_1, x_2) \to M : C} \qquad \frac{\Theta; \Gamma \vdash V : \{\mathbf{inj}_{\ell_i}\, A_i\}_i \qquad [\Theta; \Gamma, x_i : A_i \vdash_E M_i : C]_i}{\Theta; \Gamma \vdash_E \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inj}_{\ell_i}\, x_i \to M_i\}_i : C}$$

$$\frac{\Theta; \Gamma \vdash V : U_E C}{\Theta; \Gamma \vdash_E V! : C} \qquad \frac{\Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathbf{return}\ V : FA} \qquad \frac{\Theta; \Gamma \vdash_E M : FA \qquad \Theta; \Gamma, x : A \vdash_E N : C}{\Theta; \Gamma \vdash_E x \leftarrow M;\ N : C}$$

$$\frac{\Theta; \Gamma, x : A \vdash_E M : C}{\Theta; \Gamma \vdash_E \lambda x.M : A \to C} \qquad \frac{\Theta; \Gamma \vdash_E M : A \to C \qquad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E M\, V : C}$$

$$\frac{\Theta; \Gamma \vdash_E M_1 : C_1 \qquad \Theta; \Gamma \vdash_E M_2 : C_2}{\Theta; \Gamma \vdash_E \langle M_1, M_2 \rangle : C_1 \,\&\, C_2} \qquad \frac{\Theta; \Gamma \vdash_E M : C_1 \,\&\, C_2}{\Theta; \Gamma \vdash_E \mathbf{prj}_i\, M : C_i}$$

Fig. 6. MAM kind and type system

type judgements asserts that a computation term has a well-formed E-computation type under a well-formed environment for some well-formed effect E under some type variable environment. The rules for variables, value and computation products, variants, and functions are straightforward. The rules for thunking and forcing ensure that the computation's effect annotation agrees with the effect annotation of the thunk. The rule for **return** allows us to return a value at any effect annotation, reflecting the fact that this is a *may*-effect system: the effect annotations track which effects may be caused, without prescribing that any effect *must* occur. The rule for sequencing reflects our

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:

$$\mathit{not} : U_\emptyset(\mathbf{bit} \to F\mathbf{bit}) \qquad \mathit{get} : U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \qquad \mathit{put} : U_\emptyset(\mathbf{bit} \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit}))$$
$$\mathit{toggle} : U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \qquad \mathit{runState} : U_\emptyset(U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit}))$$

2.4 Operational Metatheory

We establish the basic properties of MAM.

Theorem 2.6 (MAM safety). *Well-typed programs don't go wrong: for all closed MAM returners* $\Theta; \vdash_\emptyset M : FA$*, either* $M \rightsquigarrow N$ *for some* $\Theta; \vdash_\emptyset N : FA$ *or else* $M = \mathbf{return}\ V$ *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 $M \rightsquigarrow^\infty$ if for every $n \in \mathbb{N}$ there exists some N such that $M \rightsquigarrow^n N$. We say that M *does not diverge* when $M \not\rightsquigarrow^\infty$.

Theorem 2.7 (MAM termination). *There are no infinite reduction sequences: for all MAM terms* $; \vdash_\emptyset M : FA$*, we have* $M \not\rightsquigarrow^\infty$*, and there exists some unique* $; \vdash V : A$ *such that* $M \rightsquigarrow^\star \mathbf{return}\ V$.

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 MAM terms. We define the subclass of *ground types*:

$$\text{(ground values)}\ G ::= 1 \mid G_1 \times G_2 \mid \{\mathbf{inj}_{\ell_i}\ G_i\}_i$$

The standard next step is to define well-typed *program contexts* $\mathcal{X}[\ ]$ — 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:

$$\Xi[M_1, M_2] \coloneqq \{\langle \mathcal{X}[M_1], \mathcal{X}[M_2]\rangle | \mathcal{X}[\ ] \text{ is a well-typed enclosing context}\}$$

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 $\langle \mathcal{X}[M_1], \mathcal{X}[M_2]\rangle$, their shared environments, Θ' and Γ', 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.

$$\textbf{Effects} \qquad \llbracket \emptyset \rrbracket_\theta \coloneqq \langle \mathrm{Id}, \mathrm{id}, \mathrm{id} \rangle$$

$$\textbf{Value types} \qquad \llbracket \alpha \rrbracket_\theta \coloneqq \theta(\alpha) \qquad \llbracket 1 \rrbracket_\theta \coloneqq \{\star\}$$

$$\llbracket A_1 \times A_2 \rrbracket_\theta \coloneqq \llbracket A_1 \rrbracket_\theta \times \llbracket A_2 \rrbracket_\theta \qquad \left\llbracket \{\mathbf{inj}_{\ell_i}\, A_i\}_i \right\rrbracket_\theta \coloneqq \bigcup_i \{\ell_i\} \times \llbracket A_i \rrbracket_\theta \qquad \llbracket U_E C \rrbracket_\theta \coloneqq \left| \llbracket C \rrbracket_\theta \right|$$

$$\textbf{Computation types}$$

$$\llbracket FA \rrbracket_\theta \coloneqq F \llbracket A \rrbracket_\theta \qquad \llbracket A \to C \rrbracket_\theta \coloneqq \left\langle \left| \llbracket C \rrbracket_\theta \right|^{\left| \llbracket A \rrbracket_\theta \right|}, \lambda f_s . \lambda x . c(\mathbf{fmap}\ (\lambda f . f(x))\ f_s) \right\rangle$$

$$\llbracket C_1 \,\&\, C_2 \rrbracket_\theta \coloneqq \left\langle \left| \llbracket C_1 \rrbracket_\theta \right| \times \left| \llbracket C_2 \rrbracket_\theta \right|, \lambda c_s . \langle c_1(\mathbf{fmap}\ \pi_1\ c_s), c_2(\mathbf{fmap}\ \pi_2\ c_s) \rangle \right\rangle$$

Fig. 7. MAM denotational semantics for types

The full definition is as follows. We say that an environment Γ' *extends* an environment Γ, and write $\Gamma' \geq \Gamma$ if Γ' extends Γ as a partial function from identifiers to value types. Consider any two computations with the same type C_0 under the same environments Θ_0, G_0, that is, $\Theta_0; \Gamma_0 \vdash_{E_0} M_1 : C_0$ and $\Theta_0; \Gamma_0 \vdash_{E_0} M_2 : C_0$. Define the set $\Xi[\Theta_0; \Gamma_0 \vdash_{E_0} M_1, M_2 : C_0]$ to be the smallest set of tuples $\langle \Theta', \Gamma', V_1, V_2, A \rangle$ and $\langle \Theta', \Gamma', E', N_1, N_2, C \rangle$ that is compatible with the typing rules and contains all the tuples $\langle \Theta, \Gamma, E_0, M_1, M_2, C_0 \rangle$, where $\Theta \supseteq \Theta_0$ and $\Gamma \geq \Gamma_0$. The compatibility with the rules means, for example, that if $\langle \Theta', \Gamma', V_1, V_2, A \rangle$ is in $\Xi[\Theta_0; \Gamma_0 \vdash_{E_0} M_1, M_2 : C_0]$, then so is $\langle \Theta', \Gamma', \emptyset, \mathbf{return}\ V_1, \mathbf{return}\ V_2, FA \rangle$. Define the set $\Xi[\Theta_0; \Gamma_0 \vdash V_1, V_2 : A]$ for contexts plugged with values analogously.

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 $\Theta; \Gamma \vdash_E P : X$ are meta judgements, ranging over value judgements $\Theta; \Gamma \vdash P : X$ and E-computation judgement $\Theta; \Gamma \vdash_E P : X$.

Let $\Theta; \Gamma \vdash_E P, Q : X$ be two MAM phrases. We say that P and Q are *contextually equivalent* and write $\Theta; E \vdash_\Gamma P \simeq Q : X$ when, for all pairs of plugged *closed ground-returner pure* contexts $\left\langle \emptyset, \emptyset, \emptyset, M_P, M_Q, FG \right\rangle$ in $\Xi[\Theta; \Gamma \vdash_E P, Q : X]$ and for all closed ground value terms $; \vdash V : G$, we have $M_P \rightsquigarrow^* \mathbf{return}\ V$ if and only if $M_Q \rightsquigarrow^* \mathbf{return}\ V$.

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 $\langle \mathrm{T}, \mathbf{return}, \gg\!= \rangle$ where T assigns to each set X a set $\mathrm{T}X$, **return** assigns to each set X a function $\mathbf{return}^X : X \to TX$ and $\gg\!=$ assigns to each function $f : X \to \mathrm{T}Y$ its *Kleisli extension*: a function $\gg\!= f : \mathrm{T}X \to TY$, and the three assignments satisfy the *monad laws*:

$$(\mathbf{return}\ x) \gg\!= f = f(x), \qquad a \gg\!= \mathbf{return} x = a, \qquad (a \gg\!= f) \gg\!= g = a \gg\!= (\lambda x . (f x \gg\!= g))$$

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

$$(\mathbf{return}\ x) \gg\!=^C f = f(x), \qquad (a \gg\!= g) \gg\!=^C f = a \gg\!=^C (\lambda y . (g y \gg\!=^C f))$$

for all $x \in X$, $f : X \to |C|$, $a \in TY$, and $g : Y \to TX$ of the appropriate types. For each set X, the pair $FX \coloneqq \langle TX, \gg\!= \rangle$ forms a T-algebra called the *free T-algebra over X*.

We parameterise MAM's semantics function $\llbracket \Theta \vdash_{\mathrm{k}} E : \mathbf{Eff} \rrbracket$ by an assignment θ of sets $\theta(\alpha)$ to each of the type variables α in Θ. Given such a type variable assignment θ, we assign to each

- effect: a monad $\llbracket \Theta \vdash_{\mathrm{k}} E : \mathbf{Eff} \rrbracket_\theta$, denoted by $\left\langle \mathrm{T}_{\llbracket E \rrbracket_\theta}, \mathbf{return}^{\llbracket E \rrbracket_\theta}, \gg\!=^{\llbracket E \rrbracket_\theta} \right\rangle$;

Value terms

$$\llbracket x \rrbracket_\theta\,(\gamma) \coloneqq \pi_x(\gamma) \qquad \llbracket () \rrbracket_\theta\,(\gamma) \coloneqq \star \qquad \llbracket (V_1, V_2) \rrbracket_\theta\,(\gamma) \coloneqq \left\langle \llbracket V_1 \rrbracket_\theta\,(\gamma), \llbracket V_2 \rrbracket_\theta\,(\gamma) \right\rangle$$
$$\llbracket \mathbf{inj}_\ell\, V \rrbracket_\theta\,(\gamma) \coloneqq \left\langle \ell, \llbracket V \rrbracket_\theta\,(\gamma) \right\rangle \qquad \llbracket \{M\} \rrbracket_\theta\,(\gamma) \coloneqq \llbracket M \rrbracket_\theta\,(\gamma)$$

Computation terms

$$\llbracket \mathbf{case}\ V\ \mathbf{of}\ \{(x_1, x_2) \to M\} \rrbracket_\theta\,(\gamma) \coloneqq \llbracket M \rrbracket_\theta\,(\gamma[x_1 \mapsto a_1, x_2 \mapsto a_2]) \quad \text{where } \llbracket V \rrbracket_\theta\,(\gamma) = \langle a_1, a_2 \rangle$$
$$\left\llbracket \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inj}_{\ell_i}\, x_i \to M_i\}_i \right\rrbracket_\theta \coloneqq \llbracket M_j \rrbracket_\theta\,(\gamma[x_j \mapsto a_j]) \quad \text{where } \llbracket V \rrbracket_\theta\,(\gamma) = \left\langle \ell_j, a_j \right\rangle$$
$$\llbracket V! \rrbracket_\theta\,(\gamma) \coloneqq \llbracket V \rrbracket_\theta\,(\gamma)$$
$$\llbracket \mathbf{return}\ V \rrbracket_\theta\,(\gamma) \coloneqq \mathbf{return}\ (\llbracket V \rrbracket_\theta\,(\gamma)) \qquad \llbracket x \leftarrow M;\ N \rrbracket_\theta\,(\gamma) \coloneqq \llbracket M \rrbracket_\theta\,(\gamma) \gg\!= \lambda a.\, \llbracket N \rrbracket_\theta\,(\gamma[x \mapsto a])$$
$$\llbracket \lambda x.M \rrbracket_\theta\,(\gamma) \coloneqq \lambda a.\, \llbracket M \rrbracket_\theta\,(\gamma[x \mapsto a]) \qquad \llbracket M\ V \rrbracket_\theta\,(\gamma) \coloneqq (\llbracket M \rrbracket_\theta\,(\gamma))(\llbracket V \rrbracket_\theta\,(\gamma))$$
$$\llbracket \langle M_1, M_2 \rangle \rrbracket_\theta\,(\gamma) \coloneqq \left\langle \llbracket M_1 \rrbracket_\theta\,(\gamma), \llbracket M_2 \rrbracket_\theta\,(\gamma) \right\rangle \qquad \llbracket \mathbf{prj}_i\, M \rrbracket_\theta\,(\gamma) \coloneqq \pi_i(\llbracket M \rrbracket_\theta\,(\gamma))$$

Fig. 8. MAM denotational semantics for terms

- value type: a set $\llbracket \Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \rrbracket_\theta$;
- E-computation type: a $\mathrm{T}_{\llbracket E \rrbracket_\theta}$-algebra $\llbracket \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E \rrbracket_\theta$; and
- context: the set $\llbracket \Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context} \rrbracket_\theta \coloneqq \prod_{x \in \mathrm{Dom}(\Gamma)} \llbracket \Gamma(x) \rrbracket_\theta$.

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 $0 \coloneqq \{\}$ denotes the empty set. The type of thunked E-computations of type C denotes the carrier of the $\mathrm{T}_{\llbracket E \rrbracket_\theta}$-algebra $\llbracket C \rrbracket_\theta$. The E-computation type of A returners denotes the free $\llbracket E \rrbracket_\theta$-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 $\lambda x.\mathbf{return}\ x$ has the types $1 \to F1$ and $0 \to F0$, 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 $\llbracket P \rrbracket$ and omit typing derivation for P.

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

- value term: a function $\llbracket \Theta; \Gamma \vdash V : A \rrbracket_\theta : \llbracket \Gamma \rrbracket_\theta \to \llbracket A \rrbracket_\theta$; and
- E-computation term: a function $\llbracket \Theta; \Gamma \vdash_E M : C \rrbracket_\theta : \llbracket \Gamma \rrbracket_\theta \to \left| \llbracket C \rrbracket_\theta \right|$.

Fig. 8 defines the standard set-theoretic semantics over the structure of derivations. The semantics of sequencing uses the Kleisli extended function $(\gg\!=^{\llbracket C \rrbracket} f) : TX \to |\llbracket C \rrbracket|$ for functions into non-free algebras $f : X \to |\llbracket C \rrbracket|$, 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 $\rightsquigarrow_{\text{cong}}$ be the smallest relation containing $\rightsquigarrow_\beta$ that is closed under the term formation constructs, and so contains $\rightsquigarrow$ as well, and let $\simeq_{\text{cong}}$ be the smallest congruence relation containing $\rightsquigarrow_\beta$.

Theorem 2.9 (MAM soundness). *Reduction preserves the semantics: for every pair of well-typed MAM terms $\Theta; \Gamma \vdash_E P, Q : X$, if $P \simeq_{\text{cong}} Q$ then $[\![P]\!] = [\![Q]\!]$. In particular, for every well-typed closed term of ground type $; \vdash_\emptyset P : FG$, if $P \rightsquigarrow^*$ **return** V then $[\![P]\!] = [\![V]\!]$.*

The proof is standard: check that $\rightsquigarrow_\beta$ 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 $\Theta; \Gamma \vdash_E P, Q : X$, if $[\![P]\!] = [\![Q]\!]$ then $P \simeq Q$.*

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

$$[\![\textbf{return } V]\!] = [\![\mathcal{X}[P]]\!] = [\![\mathcal{X}[Q]]\!] = [\![\textbf{return } V']\!]$$

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 $\Theta; \Gamma \vdash_E M, M' : C$, if $M \rightsquigarrow_{\text{cong}} M'$ then $M \simeq M'$.

3 EFFECT HANDLERS: 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 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 op. For each operation name op, EFF's operation call construct allows the programmer to

$M, N ::= \ldots$	computations	$H ::=$	handlers
$\mid$ op V	operation call	$\{\textbf{return}\ x \mapsto M\}$	return clause
$\mid$ **handle** M **with** H	handling construct	$\mid H \uplus \{\text{op}\ p\ k \mapsto N\}$	operation clause

(a) Syntax extensions to Fig. 2

Frames and contexts $\mathcal{F} ::= \ldots \mid \textbf{handle}\ [\ ]\ \textbf{with}\ H$ computation frames

Beta reduction

(*ret*) $\textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H \rightsquigarrow_\beta H^{\textbf{return}}[V/x]$

(*op*) $\textbf{handle}\ \mathcal{H}[\text{op}\ V]\ \textbf{with}\ H \rightsquigarrow_\beta H^{\text{op}}[V/p, \{\lambda x.\textbf{handle}\ \mathcal{H}[\textbf{return}\ x]\ \textbf{with}\ H\}/k]$

(b) Operational semantics extensions to Fig. 3

Fig. 9. EFF

$$
\begin{array}{lll}
\mathit{toggle} & = \{x \leftarrow \text{get}\ (); y \leftarrow \mathit{not}!\ x;\ \text{put}\ y;\ x\} & \mathit{toggle} : U_{State} F\textbf{bit} \\
H_{ST} & = \{\textbf{return}\ x \mapsto \lambda s.\textbf{return}\ x & H_{ST} : \textbf{bit}\ {}^{State}\!\Rightarrow^{\emptyset} \textbf{bit} \to F\textbf{bit} \\
 & \quad \text{get}\ \_\ \ k \mapsto \lambda s.k!\ s\ \ s & \\
 & \quad \text{put}\ s'\ k \mapsto \lambda \_.k!\ ()\ s'\} & State = \{\text{get} : 1 \to \textbf{bit}, \text{put} : \textbf{bit} \to 1\} : \textbf{Eff} \\
\mathit{runState} & = \{\lambda c.\textbf{handle}\ c!\ \textbf{with}\ H_{ST}\} & \mathit{runState} : U_\emptyset((U_{State} F\textbf{bit}) \to \textbf{bit} \to F\textbf{bit})
\end{array}
$$

Fig. 10. User-defined boolean state in EFF

invoke the effect associated with 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 EFF. The handler H_{ST} is parameterised by the current state. When the computation terminates, we discard this state. When the program calls get, the handler returns the current state and leaves it unchanged. When the program calls put, the handler returns the unit value, and instates the newly given state.

3.2 Operational Semantics

Fig. 9(b) presents EFF's extension to MAM's operational semantics. Computation frames $\mathcal{F}$ now include the handling construct, whereas the pure frames $\mathcal{P}$ do not, allowing a handled computation to β-reduce under the handler. We add two β-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

Kinds and types

$E ::= \ldots$ effects
$\mid \{\mathrm{op} : A \to B\} \uplus E$ arity assignment
$K ::= \ldots$ kinds
$\mid \mathbf{Handler}$ handlers
$R ::= A\ ^{E}\!\Rightarrow^{E'} C$ handler types

Effect kinding

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \quad \Theta \vdash_{\mathrm{k}} B : \mathbf{Val} \quad \mathrm{op} \notin E \quad \Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}}{\Theta \vdash_{\mathrm{k}} \{\mathrm{op} : A \to B\} \uplus E : \mathbf{Eff}}$$

Computation typing

$$\frac{(\mathrm{op} : A \to B) \in E \quad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathrm{op}\ V : FB}$$

$$\frac{\Theta; \Gamma \vdash_E M : FA \quad \Theta; \Gamma \vdash H : A\ ^{E}\!\Rightarrow^{E'} C}{\Theta; \Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C}$$

Handler kinding $\boxed{\Theta \vdash_{\mathrm{k}} R : \mathbf{Handler}}$

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \quad \Theta \vdash_{\mathrm{k}} E, E' : \mathbf{Eff} \quad \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_{E'}}{\Theta \vdash_{\mathrm{k}} A\ ^{E}\!\Rightarrow^{E'} C : \mathbf{Handler}}$$

Handler typing $\boxed{\Theta; \Gamma \vdash H : R}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}, R : \mathbf{Handler})$

$$\frac{\Theta; \Gamma, x : A \vdash_E M : C \quad [\Theta; \Gamma, p : A_i, k : U_E(B_i \to C) \vdash_E N_i : C]_i}{\Theta; \Gamma \vdash \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i : A\ ^{\{\mathrm{op}_i : A_i \to B_i\}_i}\!\Rightarrow^{E} C}$$

Fig. 11. EFF's kinding and typing (extending Fig. 5 and 6)

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.

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

$$\mathit{runState}!\ \mathit{toggle}\ \mathtt{True} \rightsquigarrow^{*} (\mathbf{handle}\ \mathtt{True}\ \mathbf{with}\ H_{ST})\ \mathtt{False} \rightsquigarrow^{*} \mathtt{True}$$

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 EFF's extension to the kind and type system. The effect annotations in 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-kinded.

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 follows 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* $\Theta; \vdash_\emptyset M : FA$*, either* $M \rightsquigarrow N$ *for some* $\Theta; \vdash_\emptyset N : FA$ *or else* $M = \mathbf{return}\ V$ *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* $;\ \vdash_\emptyset M : FA$*, we have* $M \not\rightsquigarrow^\infty$*, and there exists a unique* $;\ \vdash V : A$ *such that* $M \rightsquigarrow^\star \mathbf{return}\ V$.

EFF shares MAM's ground types, and we define plugged contexts, $\simeq$ and $\simeq_{\mathrm{cong}}$ 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* Σ is a pair consisting of a set $|\Sigma|$ whose elements we call *operation symbols*, and a function $arity_\Sigma$ from $|\Sigma|$ assigning to each operation symbol $\varphi \in |\Sigma|$ a (possibly infinite) set $arity(\varphi)$. We write $(\varphi : A) \in \Sigma$ when $\varphi \in |\Sigma|$ and $arity_\Sigma(\varphi) = A$. Given a signature Σ and a set X, we inductively form the set $\mathrm{T}_\Sigma X$ of Σ-*terms over* X:

$$t ::= x \mid \varphi \langle t_a \rangle_{a \in A} \qquad\qquad (x \in X, (\varphi : A) \in \Sigma)$$

The assignment T_Σ together with the following assignments form a monad

$$\mathbf{return}\ x \coloneqq x \qquad\qquad t \gg\!= f \coloneqq t[f(x)/x]_{x \in X} \quad (f : X \to \mathrm{T}_\Sigma Y)$$

The T_Σ-algebras $\left\langle C, \gg\!=^C \right\rangle$ are in bijective correspondence with Σ-*algebras* on the same carrier. These are pairs $\langle C, [\![-]\!] \rangle$ where $[\![-]\!]$ 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 $\gg\!=^C f$ to be the Σ-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 θ, we assign to each handler type a pair $[\![\Theta \vdash_{\mathrm{k}} R : \mathbf{Handler}]\!]_\theta = \langle C, f \rangle$ 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 denotes. 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 $\left|[\![C]\!]_\theta\right|$, 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 $\left|[\![C]\!]_\theta\right|$.

The denotation of op V at effect type E, where $\mathrm{op} : A \to B \in E$, is $\mathrm{op}_{[\![V]\!]_\theta(\gamma)} \langle a \rangle_{a \in [\![B]\!]_\theta}$. 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_Σ-algebras by defining a Σ-algebra for the associated signature Σ. The operation clause for op allows us to interpret each of the operation symbols associated to op. The denotation of the return clause gives the second component of the handler.

Effects **Handler types**

$[\![E]\!]_\theta \coloneqq \mathrm{T}_{\{\mathrm{op}_p : [\![A]\!]_\theta \mid (\mathrm{op}:A\to B)\in E, p\in[\![A]\!]_\theta\}}$ $[\![A\ ^{E}\!\Rightarrow^{E'}\ C]\!] \coloneqq \{[\![E]\!]\text{-algebras with carrier } |[\![C]\!]|\} \times |[\![C]\!]|^{[\![A]\!]}$

Computation terms

$$[\![\mathrm{op}\ V]\!]_\theta\,(\gamma) \coloneqq \mathrm{op}_{[\![V]\!]_\theta \gamma}\,\langle \textbf{return}\ a\rangle_{a\in[\![B]\!]_\theta}$$

$[\![\textbf{handle}\ M\ \textbf{with}\ H]\!]_\theta\,(\gamma) \coloneqq [\![M]\!]_\theta\,(\gamma) \gg\!= f$ where $[\![H]\!]\,(\gamma) = \langle D, f : [\![A]\!] \to |[\![C]\!]|\rangle$

Handler terms

$[\![\{\textbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i]\!]_\theta\,(\gamma) \coloneqq \langle D, f\rangle$ where D's algebra structure and f given by:

$$[\![\mathrm{op}_q]\!]_D\,\langle \xi_a\rangle_a \coloneqq [\![N_{\mathrm{op}}]\!]_\theta\,(\gamma[q/p, \langle \xi_a\rangle_a/k])\quad f(a) \coloneqq [\![M]\!]_\theta\,(\gamma[a/x])$$

Fig. 12. EFF denotational semantics (extending Fig. 7 and 8)

3.6 Denotational Metatheory

We repeat the recipe for proving adequacy.

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 $\Xi[\Theta; \Gamma \vdash_E P, Q : X]$, if $[\![P]\!] = [\![Q]\!]$ then $[\![M_P]\!] = [\![M_Q]\!]$.*

The proof is identical to MAM, with two more cases for $\rightsquigarrow_\beta$.

THEOREM 3.7 (EFF SOUNDNESS). *Reduction preserves the semantics: for every pair of well-typed EFF terms $\Theta; \Gamma \vdash_E P, Q : X$, if $P \simeq_{\mathrm{cong}} Q$ then $[\![P]\!] = [\![Q]\!]$. In particular, for every well-typed closed term of ground type $; \vdash_\emptyset P : FG$, if $P \rightsquigarrow^* \textbf{return}\ V$ 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 $\Theta; \Gamma \vdash_E P, Q : X$, if $[\![P]\!] = [\![Q]\!]$ then $P \simeq Q$.*

Therefore, EFF also has a well-behaved operational semantics: for all well-typed computations $\Theta; \Gamma \vdash_E M, M' : C$, if $M \rightsquigarrow_{\mathrm{cong}} M'$ then $M \simeq M'$.

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 Porkoláb 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.

[2] http://www.cas.mcmaster.ca/~carette/pa_monad/
[3] http://okmij.org/ftp/Scheme/monad-in-Scheme.html

$M, N ::= \ldots$ computations
$\mid \mu(N)$ reflect
$\mid [N]^T$ reify
$T ::=$ monads
where $\{$**return** $x = M$; return clause
$y \gg\!= f \quad = N\}$ bind clause

(a) Syntax (extending Fig. 2)

Frames and contexts
$\mathcal{F} ::= \mathcal{P} \mid [[\ ]]^T$ computation frames
Beta reduction
for every $T =$ **where** $\{\lambda x.N_u; \lambda y.\lambda f.N_b\}$:
(*ret*) $[\ \textbf{return}\ V\ ]^T \rightsquigarrow_\beta N_u[V/x]$
(*reflection*) $[\ \mathcal{H}[\mu(N)]\ ]^T \rightsquigarrow_\beta$
$N_b[\{N\}/y, \{(\lambda x.[\mathcal{H}[\textbf{return}\ x]]^T)\}/f]$

(b) Operational semantics (extending Fig. 3)

Fig. 13. MON

$$
\begin{array}{lll}
\mathit{toggle} & = \{x \leftarrow \mathit{get}!;\ y \leftarrow \mathit{not}!\ x;\ \mathit{put}!\ y;\ x\} & \mathit{toggle} : U_{\mathit{State}} F\mathbf{bit} \\
\mathit{get} & = \{\quad \mu(\lambda s.(s\ , s))\} & \mathit{get} : U_{\mathit{State}} F\mathbf{bit} \\
\mathit{put} & = \{\lambda s'.\mu(\lambda\_.((), s'))\} & \mathit{put} : U_{\mathit{State}}(\mathbf{bit} \to F1) \\
\mathit{State} & = \textbf{where}\,\{ & \\
 & \quad \textbf{return}\ x = \lambda s.(x, s); & \emptyset \prec \textbf{instance monad} \\
 & \quad f \gg\!= k \quad = \lambda s.(x, s') \leftarrow f\ s; & (\alpha.\mathbf{bit} \to F(\alpha \times \mathbf{bit}))\ \mathit{State} : \mathbf{Eff} \\
 & \qquad\qquad\qquad\quad k!\ x\ s'\} & \\
\mathit{runState} & = \{\lambda c.[c!]^{\mathit{State}}\} & \mathit{runState} : U_\emptyset((U_{\mathit{State}} F\mathbf{bit}) \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit}))
\end{array}
$$

Fig. 14. User-defined boolean state in MON

4.1 Syntax

Fig. 13(a) presents MON's syntax. The **where** $\{$**return** $x = N_u; y \gg\!= f = N_b\}$ 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.

Example 4.2. With this semantics we have *runState*! *toggle* `True` $\rightsquigarrow^\star$ **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 *get* and *put* are concrete functions rather than abstract operations, which means we cannot abstract over how to interpret them. To write a version of *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 MAM's kind and type system for monadic reflection. Effects are a stack of monads. The empty effect is the identity monad. A monad T can be *layered* on top of an existing stack E by $E \prec \textbf{instance monad}\,(\alpha.C)\,\textbf{where}\,\{\textbf{return}\ x = M; y \gg\!= f = N\}$. The intention is that the type constructor $C[-/\alpha]$ has an associated monad structure given by the bodies of the return M and the bind N, and can use effects from the rest of the stack E. To be well-kinded, C must be an E-computation, and T must be a well-typed monad: return should be typed $C[A/\alpha]$ when substituted for some value $V : A$, 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 *type* of a monad contains the return and bind *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 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 MAM. We choose monads to be structural and closed primarily in order to keep them closer to the other abstractions.

Kinds and types

$E ::= \ldots$ effects
$\mid E \prec \textbf{instance monad}\,(\alpha.C)\,T$ layered monad

Effect kinding

$$\frac{\Theta, \alpha \vdash_{\mathrm{k}} C : \textbf{Comp}_E \quad \vdash_{\mathrm{m}} T : E \prec \textbf{instance monad}\,(\alpha.C)\,T}{\Theta \vdash_{\mathrm{k}} E \prec \textbf{instance monad}\,(\alpha.C)\,T : \textbf{Eff}}$$

Monad typing $\boxed{\Theta \vdash_{\mathrm{m}} T : E}$

$$\frac{\Theta, \alpha; x : \alpha \vdash_E N_u : C \quad \Theta, \alpha, \beta; y : U_E C, f : U_E(\alpha \to C[\beta/\alpha]) \vdash_E N_b : C[\beta/\alpha]}{\Theta \vdash_{\mathrm{m}} \textbf{where}\,\{\textbf{return}\ x = N_u; y \gg\!= f = N_b\} : E \prec \textbf{instance monad}\,(\alpha.C)\,\textbf{where}\,\{\textbf{return}\ x = N_u; y \gg\!= f = N_b\}}$$

Computation typing

$$\frac{\Theta; \Gamma \vdash_E N : C[A/\alpha]}{\Theta; \Gamma \vdash_{E \prec \textbf{instance monad}(\alpha.C)T} \mu(N) : FA} \qquad \frac{\Theta \vdash_{\mathrm{m}} T : E \prec \textbf{instance monad}\,(\alpha.C)\,T \quad \Theta; \Gamma \vdash_{E \prec \textbf{instance monad}(\alpha.C)T} N : FA}{\Theta; \Gamma \vdash_E [N]^T : C[A/\alpha]}$$

Fig. 15. MON's kinding and typing (extending Fig. 5 and 6)

Effects

$[\![E \prec \textbf{instance monad}\ (\alpha.C)\ N_u N_b]\!]_\theta := \langle T, \textbf{return}, \gg= \rangle$
where $TX := \left|[\![C]\!]_{(\theta[\alpha \mapsto X])}\right|$ $\quad \textbf{return}^X := [\![N_u]\!]_{(\theta[\alpha \mapsto X])} : X \to TX$
$\gg=^{X,Y} := [\![N_b]\!]_{(\theta[\alpha_1 \mapsto X, \alpha_2 \mapsto Y])} : TX \to (X \to TY) \to TY$
(provided these form a monad)

Monads

$[\![\Theta \vdash_{\mathrm{m}} T : E]\!] := [\![E]\!]$

Computation terms

$[\![\lceil N \rceil^T]\!](\gamma) := [\![N]\!](\gamma)$
$[\![\mu(N)\ ]\!](\gamma) := [\![N]\!](\gamma)$

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* $\Theta; \vdash_\emptyset M : FA$*, either* $M \rightsquigarrow N$ *for some* $\Theta; \vdash_\emptyset N : FA$ *or else* $M = \textbf{return}\ V$ *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 θ, we assign to each monad type and effect a monad $[\![\Theta \vdash_{\mathrm{m}} T : E]\!]_\theta = [\![\Theta \vdash_{\mathrm{k}} E : \textbf{Eff}]\!]_\theta$, 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 $T_{[\![E \prec \textbf{instance monad}(\alpha.C)T]\!]}X = \left|[\![C]\!]_{(\theta[\alpha \mapsto X])}\right|$ and therefore, semantically, we can view any computation of type FA subject to the kinding judgement $\Theta \vdash_{\mathrm{k}} FA : \textbf{Comp}_{E \prec \textbf{instance monad}(\alpha.C)T}$ as an E-computation of type $C[A/\alpha]$.

Compare this semantics with Filinski's original semantics [1994], in which $[\![\mu(N)]\!] = [\![N]\!] \gg= \mathrm{id}$, $[\![\lceil N \rceil^T]\!] = \textbf{return}^T\ [\![N]\!]$. To explain the difference, bear in mind that our calculus is based on CBPV,

whereas Filinski's original calculus is based on a pure λ-calculus. Specifically, Filinski interprets the judgement $M : A$ as $M : TA$. The corresponding judgement for us is $M : FA$. The semantics of the pure λ-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 (and if the term is pure, they cancel out), and so MON's semantics need not add them.

4.6 Denotational Metatheory

We define a *proper derivation* to be a derivation whose semantics is well-defined for all type variable assignments, and a *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 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 MAM and EFF for proper terms.

We define *plugged proper contexts* as with MAM and 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 MAM and EFF.

THEOREM 4.5 (MON TERMINATION). *There are no infinite reduction sequences: for all* proper MON *terms* $; \vdash_\emptyset M : FA$*, we have* $M \not\rightsquigarrow^\infty$*, and there exists some unique* $; \vdash V : A$ *such that* $M \rightsquigarrow^\star \mathbf{return}\ V$.

Our proof uses Lindley and Stark's $\top\top$-lifting [2005].

THEOREM 4.6 (MON COMPOSITIONALITY). *The semantics depends only on the semantics of sub-terms: for all pairs of well-typed plugged* proper 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 MAM, with two more cases for $\rightsquigarrow_\beta$. Similarly, we have:

THEOREM 4.7 (MON SOUNDNESS). *Reduction preserves the semantics: for every pair of well-typed* proper 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* $P \rightsquigarrow^* \mathbf{return}\ V$ *then* $[\![P]\!] = [\![V]\!]$.

We combine the previous results, as with MAM and EFF:

THEOREM 4.8 (MON ADEQUACY). *Denotational equivalence implies contextual equivalence: for all well-typed* proper MON *terms* $\Theta; \Gamma \vdash_E P, Q : X$*, if* $[\![P]\!] = [\![Q]\!]$ *then* $P \simeq Q$.

Therefore, the *proper* fragment of MON also has a well-behaved operational semantics: for all well-typed proper computations $\Theta; \Gamma \vdash_E M, M' : C$, if $M \rightsquigarrow_{\text{cong}} M'$ then $M \simeq M'$.

In contrast to EFF the semantics for MON is finite:

LEMMA 4.9 (FINITE DENOTATION PROPERTY). *For every type variable assignment* $\theta = \langle X_\alpha \rangle_{\alpha \in \Theta}$ *of* finite *sets, every proper* MON *value type* $\Theta \vdash_{\text{k}} A :$ *and computation type* $\Theta \vdash_{\text{k}} C :$ *denote finite sets* $[\![A]\!]_\theta$ *and* $[\![C]\!]_\theta$.

5 DELIMITED CONTROL: DEL

Control operators have a long history of expressing both user-defined effects [Danvy 2006] and algorithms with sophisticated control flow [Felleisen et al. 1988] such as tree-fringe comparison, and other control mechanisms, such as coroutines. The delimited operators enjoy an improved metatheory in comparison with their undelimited counterparts [Felleisen et al. 1988]. The operator closest in spirit to handlers is S_0, pronounced "shift zero". It was introduced by Danvy and Filinski [1990] as part of a systematic study of continuation-passing-style conversion.

$M, N ::= \ldots$	computations	**Frames and contexts**
$\mid \mathrm{S}_0 k.M$	shift-0	$\mathcal{F} ::= \ldots \mid \langle [\ ] \mid x.N \rangle$ computation frames
$\mid \langle M \mid x.N \rangle$	reset	**Beta reduction**
		$(ret) \quad \langle (\mathbf{return}\ V) \mid x.M \rangle \rightsquigarrow_\beta M[V/x]$
		$(capture) \quad \langle \mathcal{H}[\mathrm{S}_0 k.M] \mid x.N \rangle \rightsquigarrow_\beta M[\lambda y. \langle \mathcal{H}[\mathbf{return}\ y] \mid x.N \rangle / k]$

(a) Syntax (extending Fig. 2) (b) Operational semantics (extending Fig. 3)

Fig. 17. DEL

5.1 Syntax

Fig. 17(a) presents the extension DEL. The construct $\mathrm{S}_0 k.M$, which we abbreviate to "shift", captures the current continuation and binds it to k, and replaces it with M. The construct $\langle M \mid x.N \rangle$, which we will call "reset", delimits any continuations captured by shift inside M. Once M runs its course and returns a value, this value is bound to x and 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 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, *get* and *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 *ret* rule states that once the delimited computation returns a value, this value is substituted in the remainder of the reset computation. For the *capture* rule, the definition of pure contexts guarantees that in the reduct $\langle \mathcal{H}[\mathrm{S}_0 k.M] \mid x.N \rangle$ 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: $runState!\ toggle\ \mathtt{True} \rightsquigarrow^* \langle \mathtt{True} \mid x.\lambda s.x \rangle\ \mathtt{False} \rightsquigarrow^* \mathbf{return}\ \mathtt{True}$.

5.3 Type-and-Effect System

Fig. 19 presents the natural extension to 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

$toggle = \{x \leftarrow get!;\ y \leftarrow not!\ x;\ put!\ y;\ x\}$	$toggle : U_{State} F\mathbf{bit}$
$get = \{\mathrm{S}_0 k.\lambda s.k!\ s\ s\}$	$get : U_{State} F\mathbf{bit}$
$put = \{\lambda s'.\mathrm{S}_0 k.\lambda\_.k!\ ()\ s'\}$	$put : U_{State}(\mathbf{bit} \to F1)$
$runState = \{\lambda c.\langle c! \mid x.\lambda s.x \rangle\}$	$runState : U_\emptyset((U_{State} F\mathbf{bit}) \to \mathbf{bit} \to F\mathbf{bit})$
	$State = \emptyset, \mathbf{bit} \to F\mathbf{bit} : \mathbf{Eff}$

Fig. 18. User-defined boolean state in DEL

Kinds and types

$E ::= \ldots$ effects
$\mid E, C$ enclosing continuation type

Effect kinding

$$\frac{\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff} \qquad \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} E, C : \mathbf{Eff}}$$

Computation typing

$$\frac{\Theta; \Gamma, k : U_E(A \to C) \vdash_E M : C}{\Theta; \Gamma \vdash_{E,C} \mathbf{S}_0 k.M : FA} \qquad \frac{\Theta; \Gamma \vdash_{E,C} M : FA \qquad \Theta; \Gamma, x : A \vdash_E N : C}{\Theta; \Gamma \vdash_E \langle M | x.N \rangle : C}$$

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* $\Theta; \vdash_\emptyset M : FG$, *either* $M \rightsquigarrow N$ *for some* $\Theta; \vdash_\emptyset N : FG$ *or else* $M = \mathbf{return}\ V$ *for some* $\Theta;\ \vdash V : G$.

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 $\simeq$ and $\simeq_{\mathrm{cong}}$ 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→T from each source calculus S to each target calculus T. By default we assume untyped translations, writing EFF, MON, and DEL in translations

that disregard typeability. In typeability preserving translations, which must also respect the monad laws where MON is concerned, we explicitly write TYPED EFF, TYPED MON, and TYPED DEL. We allow translations to be *hygienic* and introduce fresh binding occurrences. We write $M \mapsto \underline{M}$ 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: DEL→MON [Wadler 1994], MON→DEL [Filinski 1994], DEL→EFF [Bauer and Pretnar 2015], and EFF→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: MON→DEL, DEL→EFF, and MON→EFF. The other translations, DEL→MON, EFF→DEL, and EFF→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 $\rightsquigarrow_{\text{cong}}$, which are the smallest relations containing $\rightsquigarrow$ that are closed under the term formation constructs. We say that a translation $M \mapsto \underline{M}$ is a simulation *up to congruence* if for every reduction $M \rightsquigarrow N$ in the source calculus we have $\underline{M} \rightsquigarrow^{+}_{\text{cong}} \underline{N}$ in the target calculus. In fact, the suspended redexes always β-reduce by substituting a variable, i.e., $\{\lambda x.M\}!\ x \rightsquigarrow^{+}_{\text{cong}} \lambda x.M$, thus only performing simple rewiring.

6.1 Delimited Continuations as Monadic Reflection (DEL→MON)

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

LEMMA 6.1. *For all* $\Theta \vdash_{\text{k}} E : \mathbf{Eff}$, $\Theta \vdash_{\text{k}} C : \mathbf{Comp}_E$, *we have the following* proper *monad Cont:*

$$\Theta \vdash_{\text{k}} E \prec \mathbf{instance\ monad}\ (\alpha.U_E\ (\alpha \to C) \to C)\ \mathbf{where}\ \{\mathbf{return}\ x = \lambda c.c!\ x;\quad m \gg\!= f = \lambda c.m!\ \{\lambda y.f!\ y\ c\}\} : \mathbf{Eff}$$

Using Cont we define the macro translation DEL→MON as follows:

$$\underline{\mathsf{S}_0 k.M} := \mu(\lambda k.\underline{M}) \qquad\qquad \underline{\langle M | x.N \rangle} := [\![\underline{M}]\!]^{\text{Cont}}\ \{\lambda x.\underline{N}\}$$

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

THEOREM 6.2 (DEL→MON CORRECTNESS). *MON simulates DEL up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+}_{\text{cong}} \underline{N}$$

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 $\{\lambda y.M\ c\}$.

DEL→MON extends to a macro translation at the type level:

$$\underline{E, C} := \underline{E} \prec \mathbf{instance\ monad}\ \left(\alpha.U_{\underline{E}}\ \left(\alpha \to \underline{C}\right) \to \underline{C}\right)\ \text{Cont}$$

THEOREM 6.3 (DEL→MON PRESERVES TYPEABILITY). *Every well-typed DEL phrase* $\Theta; \Gamma \vdash_E P : X$ *translates into a proper well-typed MON phrase:* $\Theta; \underline{\Gamma} \vdash_{\underline{E}} \underline{P} : \underline{X}$.

We use this result to extend the metatheory of DEL:

COROLLARY 6.4 (DEL TERMINATION). *All well-typed closed ground returners in DEL must reduce to a unique normal form: if* $; \vdash_{\emptyset} M : FG$ *then there exists* V *such that* $; \vdash V : G$ *and* $M \rightsquigarrow^{\star} \mathbf{return}\ V$.

6.2 Monadic Reflection as Delimited Continuations (MON→DEL)

We define the macro translation MON→DEL as follows:

$$\underline{\mu(M)} := \mathrm{S}_0 k.\lambda b.b!\ (\{\underline{M}\}, \{\lambda x.k!\ x\ b\})$$

$$\underline{[M]^{\textbf{where}\ \{\textbf{return}\ x=N_\mathrm{u};y\gg\!=f=N_\mathrm{b}\}}} := \left\langle \underline{M} \middle| x.\lambda b.\underline{N_\mathrm{u}} \right\rangle\ \{\lambda(y,f).N_\mathrm{b}\}$$

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). *DEL simulates MON up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+}_{\mathrm{cong}} \underline{N}$$

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 MON→DEL is to use two nested shifts for reflection and two nested resets for reification:

$$\underline{\mu(M)} := \mathrm{S}_0 k.\mathrm{S}_0 b.b!\ (\{\underline{M}\}, \{\lambda x.\langle k!\ x | z.z!\ b\rangle\})$$

$$\underline{[M]^{\textbf{where}\ \{\textbf{return}\ x=N_\mathrm{u};y\gg\!=f=N_\mathrm{b}\}}} := \left\langle \left\langle \underline{M} \middle| x.\mathrm{S}_0 b.\underline{N_\mathrm{u}} \right\rangle \middle| (y,f).\underline{N_\mathrm{b}} \right\rangle$$

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 DEL→EFF as follows:

$$\underline{\mathrm{S}_0 k.N} := \mathsf{shift0}\ \{\lambda k.\underline{N}\} \qquad \underline{\langle M | x.N\rangle} := \textbf{handle}\ \underline{M}\ \textbf{with}\ \{\textbf{return}\ x \mapsto \underline{N}\} \uplus \{\mathsf{shift0}\ y\ f \mapsto f!\ y\}$$

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

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

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 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 (EFF→DEL)

We define EFF→DEL as follows:

$$\underline{\mathrm{op}\ V} \coloneqq \mathrm{S}_0 k.\lambda h.h!\,(\mathbf{inj}_{\mathrm{op}}\,(\underline{V}, \{\lambda y.k!\ y\ h\})) \qquad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq \left\langle \underline{M} \middle| H^{\mathrm{ret}} \right\rangle\ \{H^{\mathrm{ops}}\}$$

$$\begin{pmatrix}\mathbf{handle}\ M\ \mathbf{with}\\ \quad \{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \quad \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ret}} \coloneqq x.\lambda h.\underline{N_{\mathrm{ret}}} \qquad \begin{pmatrix}\mathbf{handle}\ M\ \mathbf{with}\\ \quad \{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \quad \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ops}} \coloneqq \begin{array}{l}\lambda y.\mathbf{case}\ y\ \mathbf{of}\ \{\\ \quad (\mathbf{inj}_{\mathrm{op}_i}\ (p,k) \to \underline{N_i})_i\\ \quad \}\end{array}$$

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 version of the captured continuation, which ensures the same dispatcher is threaded through the continuation.

Handling is interpreted as an application of a reset whose continuation contains the return clause. The reset is applied to a dispatcher function that encodes the operation clauses.

Theorem 6.7 (EFF→DEL correctness). *DEL simulates EFF up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+}_{\mathrm{cong}} \underline{N}$$

The EFF→DEL translation is simpler than Kammar et al.'s which uses a global higher-order memory cell storing the handler stack [2013].

This translation does not preserve typeability because 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 preserves typeability.

6.4.1 Alternative Translation with Nested Delimited Continuations. Similarly to the MON→DEL translation there is an alternative to EFF→DEL which uses two nested shifts for operations and two nested resets for handlers:

$$\underline{\mathrm{op}\ V} \coloneqq \mathrm{S}_0 k.\mathrm{S}_0 h.h!\,(\mathbf{inj}_{\mathrm{op}}\,(\underline{V}, \{\lambda x.\,\langle k!\ x | y.y!\ h\rangle\})) \qquad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq \left\langle \left\langle \underline{M} \middle| H^{\mathrm{ret}} \right\rangle \middle| H^{\mathrm{ops}} \right\rangle$$

$$\begin{pmatrix}\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ret}} \coloneqq x.\mathrm{S}_0 h.\underline{N_{\mathrm{ret}}} \qquad \begin{pmatrix}\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ops}} \coloneqq \begin{array}{l} y.\mathbf{case}\ y\ \mathbf{of}\ \{\\ \quad (\mathbf{inj}_{\mathrm{op}_1}\ (p,k) \to \underline{N_i})\}\end{array}$$

6.5 Monadic Reflection as Effect Handlers (MON→EFF)

We simulate reflection with an operation and reification with a handler. Formally, for every anonymous monad T given by $\mathbf{where}\,\{\mathbf{return}\ x = N_{\mathrm{u}}; y \gg\!= f = N_{\mathrm{b}}\}$ we define MON→EFF as follows:

$$\underline{\mu(N)} \coloneqq \mathrm{reflect}\ \{\underline{N}\} \qquad \underline{[\![M]\!]^T} \coloneqq \mathbf{handle}\ \underline{M}\ \mathbf{with}\ \underline{T}$$

$$\underline{T} \coloneqq \{\mathbf{return}\ x \mapsto \underline{N_{\mathrm{u}}}\} \uplus \{\mathrm{reflect}\ y\ f \mapsto \underline{N_{\mathrm{b}}}\}$$

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 (MON→EFF correctness). *EFF simulates MON on the nose:* $M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+} \underline{N}$.

MON→EFF does not preserve typeability. For instance, consider the following computation of type $F\mathbf{bit}$ using the environment monad Reader given on the right:

$$\begin{array}{l}[\![\, b \leftarrow \mu(\{\lambda(b,f).b\});\\ \quad f \leftarrow \mu(\{\lambda(b,f).f\});\\ \quad f!\ b\,]\!]^{\mathrm{Reader}}\ (\mathbf{inj}_{\mathsf{true}}\ (), \{\lambda b.\mathbf{return}\ b\})\end{array} \qquad \begin{array}{l}\vdash_{\mathrm{k}} \emptyset \prec \mathbf{instance\ monad}\ (\alpha.\mathbf{bit} \times U_{\emptyset}\ (\mathbf{bit} \to F\,\mathbf{bit}) \to F\alpha)\\ \quad \mathbf{where}\,\{\mathbf{return}\ x = \lambda e.\mathbf{return}\ x;\\ \qquad m \gg\!= f = \lambda e.x \leftarrow m!\ e;\ f!\ x\ e\} : \mathbf{Eff}\end{array}$$

Its translation into EFF is not typeable: reflection can appear at any type, whereas a single operation is monomorphic. We conjecture that a) this observation can be used to prove that *no* macro translation TYPED MON→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 (EFF→MON)

We define EFF→MON as follows:

$$\underline{\mathrm{op}\ V} \coloneqq \mu(\lambda k.\lambda h.h!\,(\mathbf{inj}_{\mathrm{op}}\,(\underline{V}, \{\lambda y.k!\ y\ h\}))) \qquad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq [\![\underline{M}]\!]^{\mathrm{Cont}}\,\{H^{\mathrm{ret}}\}\,\{H^{\mathrm{ops}}\}$$

$$\begin{pmatrix}\mathbf{handle}\ M\ \mathbf{with}\\ \quad\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus\ \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ret}} \coloneqq \lambda x.\lambda h.\underline{N_{\mathrm{ret}}} \qquad \begin{pmatrix}\mathbf{handle}\ M\ \mathbf{with}\\ \quad\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus\ \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ops}} \coloneqq \begin{array}{l}\lambda y.\mathbf{case}\ y\ \mathbf{of}\ \{\\ \quad(\mathbf{inj}_{\mathrm{op}_i}\,(p,k) \to \underline{N_i})_i\\ \}\end{array}$$

The translation is much like EFF→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 (EFF→MON CORRECTNESS). *MON simulates EFF up to congruence:*

$$M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+}_{\mathrm{cong}} \underline{N}$$

This translation does not preserve typeability for the same reason as the EFF→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:

$$\underline{\mathrm{op}\ V} \coloneqq \mu(\mathbf{return}\ (\mathbf{inj}_{\mathrm{op}}\,(\underline{V}, \lambda x.\mathbf{return}\ x))) \qquad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq H^{\star}\ [\![\underline{M}]\!]^{H^{\dagger}}$$

$$\begin{pmatrix}\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus\ \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\dagger} \coloneqq \begin{array}{l}\mathbf{where}\ \{\\ \quad \mathbf{return}\ x = \mathbf{return}\ (\mathbf{inj}_{\mathrm{ret}}\ x);\\ \quad y \gg\!= f = \mathbf{case}\ y\ \mathbf{of}\ \{\mathbf{inj}_{\mathrm{ret}}\ x \to k!\ x\\ \qquad\qquad\qquad (\mathbf{inj}_{\mathrm{op}_i}\,(p,k) \to \mathbf{return}\ (\mathbf{inj}_{\mathrm{op}_i}\,(p, \lambda x.k!\ x \gg\!= f)))_i\}\\ \}\end{array}$$

$$\begin{pmatrix}\{\mathbf{return}\ x \mapsto N_{\mathrm{ret}}\}\\ \uplus\ \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i\end{pmatrix}^{\star} \coloneqq \begin{array}{l}h = \lambda y.\mathbf{case}\ y\ \mathbf{of}\ \{\mathbf{inj}_{\mathrm{ret}}\ x \to \underline{N_{\mathrm{ret}}}\\ \qquad\qquad (\mathbf{inj}_{\mathrm{op}_i}\,(p,k') \to k \leftarrow \mathbf{return}\ \{\lambda x.y \leftarrow k'!\ x;\ h!\ y\};\ \underline{N_i})_i\}\end{array}$$

Both the bind operation for the free monad $H^{\dagger}$ and the function h that interprets the free monad $H^{\star}$ 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⟶typed mon satisfying:* $M \rightsquigarrow N \implies \underline{M} \simeq \underline{N}$.
- *typed eff⟶typed del satisfying:* $M \rightsquigarrow N \implies \underline{M} \simeq \underline{N}$.

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 $\mathsf{tick} : 1 \longrightarrow 1$ and set $\mathsf{tick}^0 := \mathbf{return}\ ()$, and $\mathsf{tick}^{n+1} := \mathsf{tick}(); \mathsf{tick}^n$. 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 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⟶typed mon (Theorem 6.3), but we conjecture that there are no typeability-preserving translations typed mon⟶typed del, typed del⟶typed eff, or typed mon⟶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

Kenichi Asai. 2009. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22, 3 (2009), 275–291.

Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376.

M. Barr and C. Wells. 1985. *Toposes, triples, and theories.* Springer-Verlag.

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014).

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 133–144.

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs (Lecture Notes in Computer Science)*, Vol. 5170. Springer, 134–149.

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

Olivier Danvy and Andrzej Filinski. 1989. *A Functional Abstraction of Typed Contexts.* Technical Report 89/12. DIKU.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160.

Christian Doczkal. 2007. *Strong Normalization of CBPV.* Technical Report. Saarland University.

Christian Doczkal and Jan Schwinghammer. 2009. Formalizing a Strong Normalization Proof for Moggi's Computational Metalanguage. In *LFMTP*. ACM, 57–63.

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.
Matthias Felleisen and Daniel P. Friedman. 1987. A Reduction Semantics for Imperative Higher-Order Languages. In *PARLE (2) (Lecture Notes in Computer Science)*, Vol. 259. Springer, 206–223.
Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract Continuations: A Mathematical Semantics for Handling Full Jumps. In *LISP and Functional Programming*. 52–62.
Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM.
Andrzej Filinski. 1996. *Controlling effects*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.
Andrzej Filinski. 1999. Representing Layered Monads. In *POPL*. ACM.
Andrzej Filinski. 2010. Monads in Action. *SIGPLAN Not.* 45, 1 (Jan. 2010), 483–494.
Yannick Forster. 2016. *On the expressive power of effect handlers and monadic reflection*. Technical Report. University of Cambridge.
Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *IJCAR*, Vol. 5195. Springer, 154–161.
Andrew Gacek. 2009. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. Dissertation. University of Minnesota.
Claudio Hermida. 1993. *Fibrations, logical predicates and related topics*. Ph.D. Dissertation. University of Edinburgh.
Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27.
Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs)*, Vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Article 18.
Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *J. Funct. Program.* 8, 4 (1998), 437–444.
Ohad Kammar. 2014. *An Algebraic Theory of Type-and-Effect Systems*. Ph.D. Dissertation. University of Edinburgh.
Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. ACM, 145–158.
Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *POPL*. ACM.
Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *J. Funct. Program.* 27 (2017), e7.
Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. *SIGPLAN Not.* 49, 1 (Jan. 2014), 633–645.
Oleg Kiselyov. 2016. Parameterized extensible effects and session types (extended abstract). In *TyDe@ICFP*. ACM, 41–42.
Oleg Kiselyov, Daniel P. Friedman, and Amr A. Sabry. 2005. *How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible*. Technical Report. 16 pages. Technical Report TR611.
Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Haskell*. ACM, 59–70.
Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *TLCA*. 223–239.
Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP*. ACM, 26–37.
Ikuo Kobori, Yukiyoshi Kameyama, and Oleg Kiselyov. 2015. Answer-Type Modification without Tears: Prompt-Passing Style Translation for Typed Delimited-Control Operators. In *WoC 2015 (EPTCS)*, Vol. 212. 36–52.
James Laird. 2002. Exceptions, Continuations and Macro-expressiveness. In *ESOP*. 133–146.
James Laird. 2013. Combining and Relating Control Effects and their Semantics. In *COS*. 113–129.
J. Laird. 2017. Combining control effects and their models: Game semantics for a hierarchy of static, dynamic and delimited control effects. *Annals of Pure and Applied Logic* 168, 2 (2017), 470–500. Eighth Games for Logic and Programming Languages Workshop (GaLoP).
Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499.
Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.
Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.
Sam Lindley and Ian Stark. 2005. Reducibility and ⊤⊤-Lifting for Computation Types. In *TLCA (Lecture Notes in Computer Science)*, Vol. 3461. Springer, 262–277.
John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM, 47–57.
Francisco Marmolejo and Richard J. Wood. 2010. Monads as extension systems — no iteration is necessary. *Theory and Applications of Categories* 24, 4 (2010), 84–113.
Marek Materzok and Dariusz Biernacki. 2012. A Dynamic Interpretation of the CPS Hierarchy. In *APLAS (LNCS)*, Vol. 7705. Springer, 296–311.
Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. IEEE Computer Society, 14–23.
Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FoSSaCS*. Springer-Verlag.
Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categ. Structures* 11, 1 (2003), 69–94.
Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *LICS*. IEEE Computer Society, 118–129.

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP*. Springer-Verlag.
Matija Pretnar. 2014. Inferring Algebraic Effects. *Logical Methods in Computer Science* 10, 3 (2014).
Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electr. Notes Theor. Comput. Sci.* 319 (2015), 19–35.
John C. Reynolds. 2009. *Theories of Programming Languages.* Cambridge University Press.
Tom Schrijvers and others. 2016. *Monad transformers and modular algebraic effects.* Technical Report. University of Leuven.
Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. 2013. Search combinators. *Constraints* 18, 2 (2013), 269–305.
Chung-chieh Shan. 2007. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation* 20, 4 (2007), 371–401.
Ábel Sinkovics and Zoltán Porkoláb. 2013. Implementing monads for C++ template metaprograms. *Sci. Comput. Program.* 78, 9 (2013), 1600–1621.
J. Michael Spivey. 1990. A Functional Theory of Exceptions. *Sci. Comput. Program.* 14, 1 (1990), 25–42.
Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436.
William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 02 (1967), 198–212.
Philip Wadler. 1990. Comprehending Monads. In *LISP and Functional Programming*. 61–78.
Philip Wadler. 1994. Monads and Composable Continuations. *Lisp and Symbolic Computation* 7, 1 (1994), 39–56.
Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.
Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *J. Funct. Program.* 25 (2015).

Received February 2017; Accepted May 2017