On the expressive power of user-defined effects

Under consideration for publication in J. Functional Programming

On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control

Yannick Forster
Saarland University Department of Computer Science, Germany
and University of Cambridge Computer Laboratory, England
and
Ohad Kammar
The University of Edinburgh School of Informatics, Scotland
and University of Oxford Department of Computer Science and Balliol College
and University of Cambridge Computer Laboratory, England
and
Sam Lindley
The University of Edinburgh School of Informatics, Scotland
and Imperial College London Department of Computing, England
and
Matija Pretnar
University of Ljubljana Faculty of Mathematics and Physics, Slovenia
(*e-mail:* forster@ps.uni-saarland.de and ohad.kammar@ed.ac.uk and sam.lindley@ed.ac.uk and matija.pretnar@fmf.uni-lj.si)

Abstract

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. 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.

1 Introduction

How should we compare abstractions for user-defined effects?

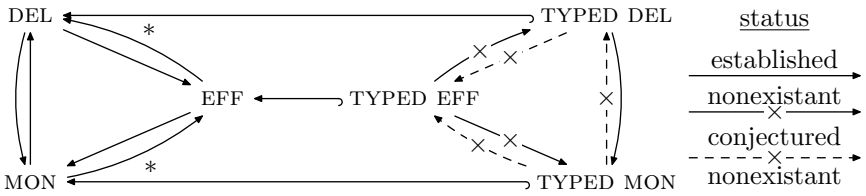
Approaches to handling computational effects, such as file, terminal, and network I/O, random-number generation, and memory allocation and mutation, vary between different functional programming languages. Whereas strict languages like Scheme and ML allow these effects to occur everywhere, languages like Haskell restrict the use of effects. One reason to be wary of incorporating computational effects into a language is that doing so can mean giving up some of the most basic properties of the lambda calculus, like referential transparency, and confluence. The loss of these properties leads to unpredictable behaviour in lazy languages like Haskell, 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 lazy languages. Recently, Bauer & Pretnar (2015) proposed the use of *algebraic effects and handlers* (Plotkin & 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 control transfers immediately to the enclosing handler without resumption following an exception, 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 & Biernacki, 2012) typed with simple types[1] 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, 2003), 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 (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

[1] but neither answer-type-modification (Asai, 2009; Kobori *et al.*, 2016) nor answer-type-polymorphism (Asai & Kameyama, 2007)



Arrows labelled by $*$ are new untyped direct translations

Fig. 1: Existing and conjectured macro translations.

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, and the contrary: whether 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 & Pretnar (2017), who give a straightforward typeability preserving macro-translation of *delimited dynamic state* into a calculus of effect handlers, whereas existing translations using monads and delimited control require more sophistication (Kiselyov *et al.*, 2006). We show that there exists no macro translation from effect handlers to monadic reflection that preserves typeability. The proof relies on the denotational semantics for the monadic calculus. This set-theoretic denotational semantics and its adequacy for Filinski's multi-monadic metalanguage (2010) is another piece of folklore which we formalise here. We conjecture that a similar proof, though with more mathematical sophistication, can be used to prove the non-existence of a typeability-preserving macro translation from the monadic calculus to effect handlers. To this end, we give adequate set-theoretic semantics to the effect handler calculus with its type-and-effect system, and highlight the critical semantic invariant a monadic calculus will invalidate.

Fig. 1 summarises our contributions and conjectured results. Untyped calculi appear on the left and their typed fragments on the right. Unlabelled arrows between the typed calculi signify that the corresponding macro translation between the untyped calculi preserves typeability. Arrows labelled by $*$ are new untyped direct translations. Arrows labelled with non-existence 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. Indeed Piróg *et al.* (2019), building on our work, have recently proved that typed macro translations do exist between a polymorphic *call-by-value* lambda calculus extended variously with effect handlers and delimited control. As well as standard data type polymorphism, they rely on polymorphic operations (Kammar *et al.*, 2013) and a novel form of answer-type polymorphism (Asai & Kameyama, 2007). Kiselyov & Sivaramakrishnan (2018) also observe the need for some form of answer-type polymorphism in their typed embedding of effect handlers into a general delimited continuations library. To avoid implementing answer-type polymorphism in practice, they rely on an encoding in terms of a universal type.

Adding features to each calculus blurs the distinction between each abstraction. This sensitivity means that in a full-blown language, such as Haskell, OCaml, or Scheme, the different abstractions are often practically equivalent (Schrijvers *et al.*, 2019). 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 & Felleisen style progress-and-preservation safety theorem (1994), and correctness theorems for our translations.

This article is a revised and extended version of a previous paper (Forster *et al.*, 2017). The core contributions are as follows:

- syntax and semantics of formal calculi for effect handlers, monadic reflection, and delimited control, where each calculus extends a shared call-by-push-value core, and their metatheory:
  - — set-theoretic denotational semantics for effect handlers and monadic reflection;
  - — denotational soundness and adequacy proofs for effect handlers and monadic reflection;
  - — a termination proof for monadic reflection (termination proofs for the other calculi appear in existing work);
- six macro-translations between the three untyped calculi, and variations on three of those translations;
- formally mechanised metatheory in Abella[2] 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

[2] https://github.com/matijapretnar/proofs

Core metatheory

	MAM	EFF	MON	DEL
Safety	✓Theorem 1	✓Theorem 6	✓Theorem 11	✓Theorem 17
Termination	Theorem 2	Theorem 7	Theorem 12	Corollary 21
Compositionality	Theorem 3	Theorem 8	Theorem 13	
Soundness	Theorem 4	Theorem 9	Theorem 14	
Adequacy	Theorem 5	Theorem 10	Theorem 15	
Finite denotation property			Lemma 16	
Continuation monad			Lemma 18	

Simulation results

source \ target	EFF	MON	DEL
EFF		✓Theorem 26	✓Theorem 24
MON	✓Theorem 25		✓Theorem 22
DEL	✓Theorem 23	✓Theorem 19	

Other translation results

TYPED DEL→TYPED MON	
Typeability preservation	✓Theorem 20
TYPED EFF→TYPED DEL & TYPED EFF→TYPED MON	
Inexpressivity	Theorem 27

✓ means result is mechanised in Abella[2]

Table 1: Metatheory summary

- a proof that there exists no typeability-preserving macro translation from effect handlers to either monadic reflection or delimited control.

Moreover, this article extends these contributions with the following new contributions:

- extensions to the mechanised metatheory in Abella[2] with:
  - — a formalisation of the kind system;
  - — the variations on the translations into delimited continuations; and
  - — the typeability preservation proof;
- an experience report on using Abella,

as well as updating related work, and providing additional explanations to the metatheoretic development and the technical details involved in the termination, adequacy, and non-existence proofs. Table 1 summarises our metatheoretic results and the coverage of their Abella formalisation.

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 (Table 1). Section 6 presents the macro translations between these calculi, their correctness, and typeability-preservation. Section 7

$V,W ::=$	values	$M,N ::=$	computations	$\mid$ **return** V	returner
x	variable	**case** V **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	$\mid$ **case** V **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

reports on our experience of using Abella for mechanising proofs. Section 8 concludes and outlines further work.

We compare the expressive power of the various abstractions in Section 6. Our positive translation results appear in Sections 6.1–6.6, which only depend on Parts 1–4 of Sections 2–5. The negative translation results of Section 6.7 depend on the more technical denotational metatheory (Parts 5–6 of Sections 2–5) which may be skipped on first reading.

2 The Core-Calculus: MAM

We seek a functional-imperative calculus where effects and higher-order features interact well. Levy's call-by-push-value (CBPV) calculus fits the bill (2003): 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 (Kammar & Plotkin, 2012; Plotkin & Pretnar, 2008). We extend CBPV with an effect type system, obtaining a variant of Kammar & Plotkin's multi-adjunctive intermediate language (2012) without effect operations or coercions. We call the resulting calculus the *multi-adjunctive metalanguage* (MAM). Later, each of Sec. 3–5 introduces a different extension of MAM for each notion of user-defined effect.

2.1 Syntax

Fig. 2 presents MAM's raw term syntax, which distinguishes between value terms V (data) and computation terms M (programs). We let P,Q range over the union of value and computation terms. 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. When eliminating variants, we will use a tuple notation when working abstractly, as in the figure, and a comma-separated list when working concretely, as in the example below. 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.

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 1. Representing booleans as variants, we may define negation as follows.

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

□

2.2 Operational Semantics

Fig. 3 presents MAM's standard structural operational semantics, in the style of Felleisen & Friedman (1987). 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. Any term can be decomposed into at most one pair of evaluation context and β-reducible term, making the semantics deterministic.

Example 2. With this semantics we have the following three-step reduction:

$$\begin{aligned} \mathit{not}!\ (\mathbf{inj}_{\mathtt{True}}\ ()) &\rightsquigarrow (\lambda b.\mathbf{case}\quad b\quad \mathbf{of}\ \{\mathbf{inj}_{\mathtt{True}}\ \ x \to \mathbf{return}\ (\mathbf{inj}_{\mathtt{False}}\ ())\\ &\quad\ \ (U) \qquad\qquad\qquad\qquad\quad ,\mathbf{inj}_{\mathtt{False}}\,x \to \mathbf{return}\ (\mathbf{inj}_{\mathtt{True}}\ \ ())\})\\ &\qquad (\mathbf{inj}_{\mathtt{True}}\ ())\\ &\rightsquigarrow \quad \mathbf{case}\ \mathbf{inj}_{\mathtt{True}}\ ()\ \mathbf{of}\ \{\mathbf{inj}_{\mathtt{True}}\ \ x \to \mathbf{return}\ (\mathbf{inj}_{\mathtt{False}}\ ())\\ &\quad\ \ (\to) \qquad\qquad\qquad\qquad\quad ,\mathbf{inj}_{\mathtt{False}}\,x \to \mathbf{return}\ (\mathbf{inj}_{\mathtt{True}}\ \ ())\}\\ &\rightsquigarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{return}\ (\mathbf{inj}_{\mathtt{False}}\ ())\\ &\quad\ \ (+) \end{aligned}$$

□

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

(a) Direct style

$$\begin{array}{l} \mathit{get} \quad\ = \{\ \ \lambda s.(\ s,s\ )\} \\ \mathit{put} \quad\ = \{\lambda s'.\lambda\_.((),s')\} \\ \mathit{runState} = \lambda c.\lambda s.c\,!\,s \end{array}$$

$$\begin{array}{l} \mathit{toggle} = \{\lambda s.(x,s) \leftarrow \mathit{get}\,!\,s; \\ \qquad\quad y \leftarrow \mathit{not}\,!\,x; \\ \qquad\quad (\_,s) \leftarrow \mathit{put}\,!\,y\,s; \\ \qquad\quad \mathbf{return}\ (x,s)\} \end{array}$$

(b) State-passing style

Fig. 4: User-defined boolean state

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., $\mathtt{True} \coloneqq \mathtt{True}\ () \coloneqq \mathbf{inj}_{\mathtt{True}}\ ()$. We allow elimination constructs to apply to arbitrary computations, and not just values, by setting for example $M\ N \coloneqq x \leftarrow N;\ M\ x$ for some fresh x, giving a more readable, albeit call-by-value, appearance.

Example 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 then run *toggle* with the initial value `True` to get the expected result:

$$\mathit{runState}\,!\ \mathit{toggle}\ \mathtt{True} \rightsquigarrow^{\star} \mathbf{return}\ (\mathtt{True},\mathtt{False})$$

This transformation is *not* a macro translation from the extension of MAM with global state to MAM. In addition to the definition of *put* and *get*, it globally threads the state through *toggle*'s structure, changing core MAM constructs. For example, $x \leftarrow M;\ N$ changes to $(x,s) \leftarrow M';\ N'$. 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 ensures types are 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 type variables ranging over value-types, i.e., types of kind **Val** (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. For completeness, 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 finite sets of type variables. So the type

$E ::=$	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
$\mid \mathbf{Eff}$	effects	$\mid A_1 \times A_2$	products	$\mid C_1 \,\&\, C_2$	products
$\mid \mathbf{Val}$	values	$\mid \{\ell_i\, A_i\}_i$	variants	Environments:	
$\mid \mathbf{Comp}_E$	computations	$\mid U_E C$	thunks	$\Theta ::= \alpha_1, \ldots, \alpha_n$	
$\mid \mathbf{Context}$	environments			$\Gamma,\Delta ::= x_1 : A_1, \ldots, x_n : A_n$	

Fig. 5: MAM kinds and types

environment $\Theta := \alpha_1, \beta, \alpha_2$ is in fact the set $\Theta = \{\alpha_1, \alpha_2, \beta\}$. 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. So the domain of definition of the value environment $\Gamma := x : \alpha, f : \alpha \to F\alpha$ is $\mathrm{Dom}\,(\Gamma) = \{x, f\}$.

Example 4. The value-type of booleans **bit** is given by $\{\mathsf{False}\ 1, \mathsf{True}\ 1\}$. □

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 a 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 type judgements asserts that a computation term has a well-formed E-computation type under a well-formed environment for a well-formed effect E under a 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, yielding a *may*-effect system: the effect annotations specify which effects may occur, without prescribing that any particular effect *must* occur. The rule for sequencing comes from 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 5. The values from Fig. 4(b) have the following types:

$$\begin{array}{ll} \mathit{not} : U_\emptyset(\mathbf{bit} \to F\mathbf{bit}) & \mathit{toggle} \;\; : U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \\ \mathit{get} : U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) & \mathit{runState} : U_\emptyset(U_\emptyset(\mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \\ \mathit{put} : U_\emptyset(\mathbf{bit} \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) & \qquad\qquad \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit})) \end{array}$$

as expected. □

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6: MAM kind and type system

2.4 Operational Metatheory

We now establish the basic properties of MAM.

Theorem 1 (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* $\Theta; \vdash V : A$*.*

We have mechanised the standard inductive progress-and-preservation proof using Abella.

We now extend existing termination results for CBPV (Doczkal, 2007; Doczkal & 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 $M \rightsquigarrow^n N$. Because the operational semantics is deterministic,

N is unique, and if $M \rightsquigarrow^i N_i$ for all $1 \leq i \leq n$, then $N_i \rightsquigarrow N_{i+1}$ for all $1 \leq i < n$. We say that M *does not diverge* when $M \not\rightsquigarrow^\infty$.

Theorem 2 (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* $M \rightsquigarrow^\star$ **return** V.

Proof
We use Tait's method (1967), i.e., a unary logical relation, to establish totality. In detail, we define a (unary) relational interpretation of types and establish a basic lemma. The full definition follows the logical structure, except for the following case. In order to define the relation on returners FA, we need a monadic lifting. We use the lifting from Hermida's thesis (1993), defined to contain the returners that reduce to a return value for all closed substitutions. Forster *et al.* (2019) have since formalised in Coq a similar termination proof for call-by-push-value; see their manuscript (Fig. 12 on page 6) for more details. ■

We define contextual equivalence of MAM terms, which we will use to state our inexpressivity result (Theorem 27). First, we define the subclass of *ground types*.

$$\text{(ground types)}\quad G \ ::= 1 \mid G_1 \times G_2 \mid \{\ell_i\ G_i\}_i$$

We also introduce some convenient notational conventions. For uniformity's sake, we let types X range over both value and E-computation types, and recall that phrases P, Q range over both value and computation terms. Judgements of the form $\Theta;\Gamma \vdash_E P : X$ are meta judgements, ranging over value judgements $\Theta;\Gamma \vdash V : X$ when $P = V$ (in which case $E = \emptyset$), and E-computation judgements $\Theta;\Gamma \vdash_E M : X$ when $P = M$.

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 $\mathcal{C}[\ ]$, which always contain exactly one hole. Defining program contexts and their type judgements directly is straightforward but tedious and lengthy. Such a definition would have four kinds of judgements, depending on whether the hole takes a value or a computation, and whether the whole context is of value or computation type. For our purposes, we can exploit a folklore simplification, that relies on the following observations.

We only need program contexts $\mathcal{X}[\ ]$ indirectly, to plug two terms P, Q of the appropriate type, obtaining pairs $\langle \mathcal{X}[P], \mathcal{X}[Q] \rangle$ in which both components have the same type. Consider the set of all such pairs:

$$\Xi[P,Q] \coloneqq \{\langle \mathcal{X}[P], \mathcal{X}[Q] \rangle \mid \mathcal{X}[\ ] \text{ is a well-typed enclosing context}\}$$

Example 6. Taking $P \coloneqq (not\,!\,\texttt{True})$ and $Q \coloneqq (\textbf{return}\ \texttt{False})$ we want to include:

$$\left\langle \begin{array}{l} x \leftarrow not\,!\,\texttt{True}; \\ not\,!\,x \end{array} \ , \ \begin{array}{l} x \leftarrow \textbf{return}\ \texttt{False}; \\ not\,!\,x \end{array} \right\rangle \qquad \langle \{not\,!\,\texttt{True}\}, \{\textbf{return}\ \texttt{False}\} \rangle$$

in the set $\Xi[(not\,!\,\texttt{True}), (\textbf{return}\ \texttt{False})]$. □

To define the contextual equivalence of P and Q, we quantify over all pairs $\langle M, N \rangle$ in $\Xi[P,Q]$ of closed ground returners FG, and require that M and N reduce to the same value. So, besides pairing M and N, we also need to know their shared type X (and effect E when they are computations), their free variables and their type, i.e., an environment Γ, and the

type environment Θ containing the free variables in X and Γ. However, nowhere do we need to refer to the context $\mathcal{X}[\ ]$ we plugged them into. As we need to know Θ, Γ, E, and X in addition to P and Q, instead of defining $\Xi[P,Q]$, we will define $\Xi[\Theta;\Gamma \vdash_E P,Q : X]$.

Moreover, P and Q may be open terms, cause effects, and the enclosing context may capture some of their free variables. Hence we consider, not only pairs of plugged contexts, but tuples $\langle \Theta,\Gamma,E,M,N,C\rangle$ in which $\Theta;\Gamma \vdash_E M,N : C$ are computation terms arising by plugging a context with P and Q respectively, and similarly tuples $\langle \Theta,\Gamma,V,W,A\rangle$ where $\Theta;\Gamma \vdash V,W : A$ are value terms. We can then quantify over the tuples in which Θ and Γ are empty, and $C = FG$ is a ground returner type, as those represent closed, ground contexts.

To present the full definition, we require some additional standard auxiliary definitions. We say that environment Γ' *extends* environment Γ, and write $\Gamma' \geq \Gamma$ when Γ' extends Γ as a partial function from identifiers to value types, i.e., when $\mathrm{Dom}\,(\Gamma) \subseteq \mathrm{Dom}\,(\Gamma')$ and for all $(x : A) \in \Gamma$, we have $(x : A) \in \Gamma'$. Consider any two computations of the same type C_0 under the same environments Θ_0,Γ_0, that is, $\Theta_0;\Gamma_0 \vdash_{E_0} M_0 : C_0$ and $\Theta_0;\Gamma_0 \vdash_{E_0} N_0 : C_0$. Define $\Xi[\Theta_0;\Gamma_0 \vdash_{E_0} M_0,N_0 : C_0]$ to be the smallest set with the following closure properties:

- for all $\Theta' \supseteq \Theta_0$, $\Theta' \vdash_{\mathrm{k}} \Gamma' : \mathbf{Context}$ with $\Gamma' \geq \Gamma_0$, we have $\langle \Theta',\Gamma',E_0,M_0,N_0,C_0\rangle \in \Xi$;
- Ξ is closed under the typing rules, for example:
  - for all $\langle \Theta,\Gamma,E,M,N,C\rangle \in \Xi$, we also have: $\langle \Theta,\Gamma,\{M\},\{N\},U_E C\rangle \in \Xi$[3];
  - for all $\langle \Theta,\Gamma,V_1,W_1,A_1\rangle, \langle \Theta,\Gamma,V_2,W_2,A_2\rangle \in \Xi$, we also have:

$$\langle \Theta,\Gamma,\langle V_1,V_2\rangle,\langle W_1,W_2\rangle,A_1 \times A_2\rangle \in \Xi$$

  and so on.

Example 7. Going back to Example 6, we have that:

$$\left\langle \cdot,\cdot,\emptyset,\begin{pmatrix} x \leftarrow \mathit{not}\,!\,\mathtt{True}; \\ \mathit{not}\,!\,x \end{pmatrix},\begin{pmatrix} x \leftarrow \mathbf{return}\ \mathtt{False}; \\ \mathit{not}\,!\,x \end{pmatrix}, F\mathbf{bit}\right\rangle$$
$$\langle \cdot,\cdot,\{\mathit{not}\,!\,\mathtt{True}\},\{\mathbf{return}\ \mathtt{False}\},U_\emptyset \mathbf{bit}\rangle$$

are in the set $\Xi[\cdot;\cdot \vdash_\emptyset (\mathit{not}\,!\,\mathtt{True}),(\mathbf{return}\ \mathtt{False}) : F\mathbf{bit}]$. For an example involving open terms and type-variables, take the tuple:

$$\left\langle \alpha,\cdot,\emptyset,\begin{pmatrix} u \leftarrow \mathtt{True}; \\ s \leftarrow \mathit{not}\,!\,u; \\ \mathit{runState}\,!\,\mathit{toggle}\ s \end{pmatrix},\begin{pmatrix} u \leftarrow \mathtt{True}; \\ s \leftarrow \mathit{not}\,!\,u; \\ y \leftarrow \mathit{not}\,!\,s; \\ \mathbf{return}\ (s,y) \end{pmatrix}, F(\mathbf{bit} \times \mathbf{bit})\right\rangle$$

from the set:

$$\Xi[\alpha;s : \mathbf{bit} \vdash_\emptyset (\mathit{runState}\,!\,\mathit{toggle}\ s),\begin{pmatrix} y \leftarrow \mathit{not}\,!\,s; \\ \mathbf{return}\ (s,y) \end{pmatrix} : F(\mathbf{bit} \times \mathbf{bit})]$$

□

[3]Closure under typing rules necessitates including tuples representing value judgements in Ξ as well as those representing computation judgements.

In line with the motivation for this definition, we call the components $\langle M, N\rangle$ of each tuple in $\Xi[\Theta_0;\Gamma_0 \vdash_{E_0} M_0, N_0 : C_0]$ *context plugged with* M_0 *and* N_0. We will use the notation $\langle \mathcal{X}[M_0], \mathcal{X}[N_0]\rangle$ for such contexts plugged with M_0 and N_0, but emphasise that we have not defined contexts $\mathcal{X}[-]$ on their own, only plugged contexts.

We now say that $\Theta_0;\Gamma_0 \vdash_{E_0} M_0, N_0 : C_0$ are *contextually equivalent* when, for all:

$$\langle \cdot, \cdot, \emptyset, \mathcal{X}[M_0], \mathcal{X}[N_0], FG\rangle \in \Xi[\Theta_0;\Gamma_0 \vdash_{E_0} M_0, N_0 : C_0]$$

and V, we have:

$$\mathcal{X}[M_0] \rightsquigarrow^* \mathbf{return}\ V \qquad \Longleftrightarrow \qquad \mathcal{X}[N_0] \rightsquigarrow^* \mathbf{return}\ V$$

We similarly define contextual equivalence for values. We write $\Theta;\Gamma \vdash_E P \simeq Q : X$ when P and Q are contextually equivalent.

2.5 Denotational Semantics

MAM has a straightforward set-theoretic denotational semantics. Presenting the semantics for the core calculus simplifies 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 $\gg\!= f : \mathrm{T}X \to \mathrm{T}Y$, and the three assignments satisfy the *monad laws*, with the convention that $\gg\!= f$ is a post-fix operator:

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

for all $f : X \to TY$, $x \in X$, $a \in TX$, and $g : Y \to TZ$.

Example 8 (see Moggi (1989))**.** Let R be any set. The *R-continuation monad* is given by:

$$K_R X \coloneqq R^{(R^X)} \qquad \mathbf{return}\ x \coloneqq \lambda k : R^X.k(x) \qquad (m \gg\!= f) \coloneqq \lambda k : R^Y.m\,(\lambda x : X.k\ ((f\ x)\ k))$$

As is well-known, these definitions satisfy the monad laws. □

A *T-algebra* for a monad $\langle \mathrm{T}, \mathbf{return}, \gg\!=\rangle$, following Marmolejo & Wood (2010), is a pair $C = \langle |C|, \gg\!=^C\rangle$ 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:

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

for all $x \in X$, $f : X \to |C|$, $a \in TY$, and $g : Y \to TX$ of the appropriate types.

Example 9. For each set X, the pair $FX \coloneqq \langle TX, \gg\!=\rangle$ forms a T-algebra called the *free T-algebra over X*. □

Example 10 (Paré's theorem (1974))**.** Let $\mathbf{2} \coloneqq \{\mathtt{True}, \mathtt{False}\}$ be the set of boolean values. For each set Y, the powerset $\mathcal{P}Y$ is the carrier of an algebra for the $\mathbf{2}$-continuation monad. For every function $f : X \to \mathcal{P}Y$, and element $a \in \mathbf{2}^{(\mathbf{2}^X)}$, we assign the subset:

$$(a \gg\!= f) \coloneqq \left\{ y \in Y \,\middle|\, a\left(\lambda x : X. \begin{cases} \mathtt{True} & y \in f(x) \\ \mathtt{False} & \text{otherwise} \end{cases}\right) = \mathtt{True} \right\}$$

Then $\langle \mathcal{P}Y, \gg\!=\rangle$ is a $K_{\mathbf{2}}$-algebra. □

Effects

$$[\![\emptyset]\!]_\theta \coloneqq \langle \mathrm{Id}, \mathrm{id}, \mathrm{id} \rangle$$

Value types

$$[\![\alpha]\!]_\theta \coloneqq \theta(\alpha) \qquad [\![1]\!]_\theta \coloneqq \{\star\}$$

$$[\![A_1 \times A_2]\!]_\theta \coloneqq [\![A_1]\!]_\theta \times [\![A_2]\!]_\theta \qquad [\![\{\ell_i\, A_i\}_i]\!]_\theta \coloneqq \bigcup_i \{\ell_i\} \times [\![A_i]\!]_\theta \qquad [\![U_E C]\!]_\theta \coloneqq \left|[\![C]\!]_\theta\right|$$

Computation types

$$[\![FA]\!]_\theta \coloneqq F[\![A]\!]_\theta \qquad [\![A \to C]\!]_\theta \coloneqq \left\langle \left|[\![C]\!]_\theta\right|^{[\![A]\!]_\theta}, \lambda f\, c\, a.\, c \gg\!=^{C} \lambda x. f\, x\, a \right\rangle$$

$$[\![C_1 \,\&\, C_2]\!]_\theta \coloneqq \left\langle \left|[\![C_1]\!]_\theta\right| \times \left|[\![C_2]\!]_\theta\right|, \lambda f\, c.\, \left\langle c \gg\!=^{C_1} (\pi_1 \circ f), c \gg\!=^{C_2} (\pi_2 \circ f) \right\rangle \right\rangle$$

Fig. 7: MAM denotational semantics for types

We parameterise MAM's semantics 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 $[\![\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}]\!]_\theta$, denoted by $\left\langle \mathrm{T}_{[\![E]\!]_\theta}, \mathbf{return}^{[\![E]\!]_\theta}, \gg\!=^{[\![E]\!]_\theta} \right\rangle$;
- value type: a set $[\![\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}]\!]_\theta$;
- E-computation type: a $\mathrm{T}_{[\![E]\!]_\theta}$-algebra $[\![\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E]\!]_\theta$; and
- context: the set $[\![\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Context}]\!]_\theta \coloneqq \prod_{x \in \mathrm{Dom}(\Gamma)} [\![\Gamma(x)]\!]_\theta$.

Fig. 7 defines the standard set-theoretic semantics 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 $\mathrm{T}_{[\![E]\!]_\theta}$-algebra $[\![C]\!]_\theta$. The E-computation type of A returners denotes the free $[\![E]\!]_\theta$-algebra. Function and product types denote well-known algebra structures over the sets of functions and pairs, respectively (Barr & 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, 1998) by defining the semantic function for type judgement derivations rather than for terms. For readability, we often write $[\![P]\!]$ and omit the 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 $[\![\Theta; \Gamma \vdash V : A]\!]_\theta : [\![\Gamma]\!]_\theta \to [\![A]\!]_\theta$; and
- E-computation term: a function $[\![\Theta; \Gamma \vdash_E M : C]\!]_\theta : [\![\Gamma]\!]_\theta \to \left|[\![C]\!]_\theta\right|$.

Fig. 8 defines the standard set-theoretic semantics over the structure of derivations. Each definition takes an *environment* $\gamma \in [\![\Gamma]\!]_\theta$.

We begin with values. The semantics of variables looks the appropriate value up in this environment. The unit value denotes the unique element of the singleton $[\![1]\!]$. A pair of values denote the pair of their denotations. A variant constructor denotes the injection of a value into a disjoint union by pairing the value with the constructor label. Thunking denotes the element of the carrier the computation denotes.

Value terms

$$[\![x]\!]_\theta(\gamma) \coloneqq \pi_x(\gamma) \qquad [\![()]\!]_\theta(\gamma) \coloneqq \star \qquad [\![(V_1,V_2)]\!]_\theta(\gamma) \coloneqq \langle [\![V_1]\!]_\theta(\gamma), [\![V_2]\!]_\theta(\gamma)\rangle$$
$$[\![\mathbf{inj}_\ell V]\!]_\theta(\gamma) \coloneqq \langle \ell, [\![V]\!]_\theta(\gamma)\rangle \qquad [\![\{M\}]\!]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma)$$

Computation terms

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

Fig. 8: MAM denotational semantics for terms

Moving to computations, each pattern match denotes the function defined by case splitting on the denotation of the matched value. Forcing a value denotes treating its denotation from the algebra carrier as a computation. Returning a value denotes applying the unit of the monad to the denotation of said value. The semantics of sequencing uses the Kleisli extended function $(\gg\!=^{[\![C]\!]} f) : TX \to |[\![C]\!]|$ given by the algebra structure. Function abstraction denotes the set-theoretic function definition, and application denotes set-theoretic evaluation. Pairing and projection terms denote the set-theoretic pairing and projections.

2.6 Denotational Metatheory

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

Theorem 3 (MAM compositionality)**.** *The meaning of a term depends only on the meaning of its sub-terms: for all* MAM *contexts* $\langle \mathcal{X}[P], \mathcal{X}[Q]\rangle$ *plugged with* P *and* Q *in* $\Xi[\Theta;\Gamma \vdash_E P,Q : X]$*, if* $[\![P]\!] = [\![Q]\!]$ *then* $[\![\mathcal{X}[P]]\!] = [\![\mathcal{X}[Q]]\!]$*.*

Proof

Straightforward induction on the set Ξ of plugged contexts. ■

In order to be able to express our simulation results in Section 6, 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 4 (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* $[\![P]\!] = [\![V]\!]$*.*

Proof

First check that $\rightsquigarrow_\beta$ preserves the semantics by calculating the denotations of both sides of each rule. Next, take any β-reduction $M \rightsquigarrow_\beta N$, and consider a pair of plugged contexts $\langle \mathcal{X}[M], \mathcal{X}[N]\rangle$. Because $\rightsquigarrow_\beta$ preserves the semantics, by appeal to compositionality

$[\![\mathcal{X}[M]]\!] = [\![\mathcal{X}[N]]\!]$. Therefore $\rightsquigarrow_{\text{cong}}$ is contained in denotational equivalence, which is also a congruence, hence $\simeq_{\text{cong}}$ implies denotational equivalence. ■

It now follows that the semantics is adequate:

Theorem 5 (MAM adequacy)**.** *Denotational equivalence implies contextual equivalence: for all well-typed* MAM *terms* $\Theta;\Gamma \vdash_E P,Q : X$*, if* $[\![P]\!] = [\![Q]\!]$ *then* $P \simeq Q$*.*

Recall the convention from page 11 for uniformly treating values and computation. With this convention, the adequacy theorem is stated for both values and computation terms.

Proof

Consider any two denotationally equivalent terms P and Q, a closed ground context plugged with them $\langle \mathcal{X}[P], \mathcal{X}[Q] \rangle$, and assume $\mathcal{X}[P] \rightsquigarrow^* \textbf{return}\ V$. By the Compositionality Theorem 3, $\mathcal{X}[P]$ and $\mathcal{X}[Q]$ have equal denotations. By the Safety Theorem 1 and Termination Theorem 2, $\mathcal{X}[Q] \rightsquigarrow^* \textbf{return}\ V'$ for some value $\textbf{return}\ V'$. And so by the Soundness Theorem 4:

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

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

As a consequence, we deduce that our operational semantics is 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 (Bauer & Pretnar, 2015; Hillerström & Lindley, 2016; Kammar *et al.*, 2013; Kiselyov *et al.*, 2013; Leijen, 2017; Lindley *et al.*, 2017). Programmable effect handlers arose as part of Plotkin & 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 & 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 & 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 & 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

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

(a) Syntax extensions to Fig. 2

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

Beta reduction

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

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

(b) Operational semantics extensions to Fig. 3

Fig. 9: EFF

$toggle$	$= \{x \leftarrow \mathsf{get}\ \mathsf{()};\ y \leftarrow not!\, x;\ \mathsf{put}\ y;$ $\mathbf{return}\ x\}$	$toggle : U_{State} F\mathbf{bit}$
H_{ST}	$= \{\mathbf{return}\ \ x \mapsto \lambda s.\mathbf{return}\ x$ $\mathsf{get}\ \_\ k \mapsto \lambda s.k!\, s\ \ s$ $\mathsf{put}\ s'\, k \mapsto \lambda \_.k!\, ()\ s'\}$	$H_{ST} : \mathbf{bit}^{State} \Rightarrow^{\emptyset} \mathbf{bit} \to F\mathbf{bit}$ $State = \{\mathsf{get} : 1 \to \mathbf{bit}, \mathsf{put} : \mathbf{bit} \to 1\} : \mathbf{Eff}$
$runState$	$= \{\lambda c.\mathbf{handle}\ c!\ \mathbf{with}\ H_{ST}\}$	$runState : U_\emptyset((U_{State} F\mathbf{bit}) \to \mathbf{bit} \to F\mathbf{bit})$

(a) terms (b) types

Fig. 10: User-defined boolean state in EFF

the programmer to 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. Handlers are specified by two kinds of clauses. A *return clause* describes how to handle a final return value. An *operation clause* describes how to invoke an operation op. The variable p binds the value from the operation call in the body of the operation clause and is entirely analogous to an exception variable in an exception handler. However, unlike exceptions, more general effects, like reading from or writing to memory, may resume. Therefore the body of an operation clause can also access the continuation k at the operation's calling point.

Example 11. Fig. 10(a) 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 (s) and leaves it unchanged. When the program calls put, the handler returns the unit value, and updates the state to the newly supplied value (s'). □

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

Kinds and types

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

Effect kinding

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

Computation typing

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

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

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

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

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

clause proceeds with the return value. When the returner computation inside a handler needs to evaluate an operation call, the definition of pure contexts $\mathcal{H}$ ensures $\mathcal{H}$ is precisely the continuation of the operation call delimited by the handler. Put differently, it ensures that the handler in the root of the reduct is the closest handler to the operation call in the call stack. The operation clause corresponding to the operation called then proceeds with the supplied parameter and current continuation. Rewrapping the handler around this continuation ensures that all operation calls invoked in the continuation are handled in the same way.

Example 12. With this semantics, the user-defined state from Fig. 10 behaves as expected:

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

More generally, the handler H_{ST} expresses *dynamically scoped* state (Kammar & 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 13. Fig. 10(b) types the boolean state terms. □

3.4 Operational Metatheory

We follow MAM's development.

Theorem 6 (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* $\Theta;\vdash V : A$*.*

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

Theorem 7 (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* $M \rightsquigarrow^\star \mathbf{return}\ V$*.*

The proof follows that of Theorem 2, replacing the monadic lifting with a folklore monadic lifting for algebraic effects (Kammar, 2014; Kammar & McDermott, 2018).

We define ground types, plugged contexts, $\simeq$, and $\simeq_{\text{cong}}$ for EFF 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 $\langle C, \gg\!=^C\rangle$ 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 $T_\Sigma X$:

$$\left(x \gg\!=^C f\right) \coloneqq f(x) \qquad \left(\varphi\,\langle t_a\rangle_{a\in A} \gg\!=^C f\right) \coloneqq [\![\varphi]\!]\,\langle t_a \gg\!= f\rangle_{a\in A}$$

Effects **Handler types**

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

Computation terms

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

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

Handler terms

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

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

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

EFF's denotational semantics is given by the following extension to MAM's. Given a type variable assignment θ, we assign to each handler type a pair $[\![\Theta \vdash_{\mathrm{k}} R : \textbf{Handler}]\!]_\theta = \langle C, f \rangle$ 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 $|[\![C]\!]_\theta|$, which may have nothing to do with the $[\![E']\!]_\theta$-algebra structure $[\![C]\!]_\theta$ already possesses. The second component is a function from $[\![A]\!]_\theta$ to the carrier $|[\![C]\!]_\theta|$.

The denotation of $\mathsf{op}\ V$ at effect type E, where $\mathsf{op} : A \to B \in E$, is the algebraic term $\mathsf{op}_{[\![V]\!]_\theta(\gamma)} \langle b \rangle_{b \in [\![B]\!]_\theta}$. 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.

3.6 Denotational Metatheory

We repeat the recipe for proving adequacy.

Theorem 8 (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 follows the same line as MAM's Theorem 3: by induction on plugged contexts.

Theorem 9 (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 $[\![P]\!] = [\![V]\!]$.*

Our proof is identical to MAM's soundness Theorem 4, with two more cases for $\rightsquigarrow_\beta$. We combine the previous results, as with MAM:

Theorem 10 (EFF adequacy)**.** *Denotational equivalence implies contextual equivalence: for all well-typed* EFF *terms $\Theta; \Gamma \vdash_E P, Q : X$, if $[\![P]\!] = [\![Q]\!]$ then $P \simeq Q$.*

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

(b) Operational semantics (extending Fig. 3)

Fig. 13: MON

Therefore, EFF also has a well-behaved operational semantics: for all well-typed computations $\Theta;\Gamma \vdash_E M,M' : C$, if $M \rightsquigarrow_{\text{cong}} M'$ 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 & Meijer, 1998), backtracking and constraint solving (Schrijvers *et al.*, 2013), and mechanised reasoning (Bulwahn *et al.*, 2008; Ziliani *et al.*, 2015). Libraries now exist for monadic programming even in impure languages such as OCaml[4], Scheme[5], and C++ (Sinkovics & 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; and 2010) provides a more canonical abstraction for incorporating monads into a programming language. In his calculi, user-defined monads stand independently.

4.1 Syntax

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

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

$$
\begin{array}{lll}
\mathit{toggle} & = \{x \leftarrow \mathit{get}!;\ y \leftarrow \mathit{not}!\ x;\ \mathit{put}!\ y; & \mathit{toggle} : U_{\mathit{State}} F\,\mathbf{bit} \\
 & \quad \mathbf{return}\ x\} & \\
\mathit{get} & = \{\quad \mu(\lambda s.(s\ ,s))\} & \mathit{get} : U_{\mathit{State}} F\,\mathbf{bit} \\
\mathit{put} & = \{\lambda s'.\mu(\lambda\_.((),s'))\} & \mathit{put} : U_{\mathit{State}}(\mathbf{bit} \to F\,1) \\
\mathit{State} & = \mathbf{where}\,\{ & \\
 & \quad \mathbf{return}\ x = \lambda s.(x,s); & \emptyset \prec \mathbf{instance\ monad} \\
 & \quad f \gg\!= k \quad = \lambda s.(x,s') \leftarrow f\ s; & \quad (\alpha.\mathbf{bit} \to F(\alpha \times \mathbf{bit}))\,\mathit{State} : \mathbf{Eff} \\
 & \qquad\qquad\qquad\quad k!\ x\ s'\} & \\
\mathit{runState} & = \{\lambda c.[c!]^{\mathit{State}}\} & \mathit{runState} : U_{\emptyset}((U_{\mathit{State}} F\,\mathbf{bit}) \to \mathbf{bit} \to F(\mathbf{bit} \times \mathbf{bit}))
\end{array}
$$

(a) terms (b) types

Fig. 14: User-defined boolean state in MON

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 14. Fig. 14(a) 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 15. With this semantics we have

$$\mathit{runState}!\ \mathit{toggle}\ \texttt{True} \rightsquigarrow^{\star} \mathbf{return}\ \ (\texttt{True},\texttt{False})$$

as expected. □

This example 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

$$E \prec \mathbf{instance\ monad}\,(\alpha.C)\,\mathbf{where}\,\{\mathbf{return}\ x = M; y \gg\!= f = N\}$$

Kinds and types

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

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)

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 16. Fig. 14(b) types the boolean state terms. □

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.

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 can be simulated in our calculus: the monad stack is statically known, and,

Effects

$[\![E \prec \textbf{instance monad}\,(\alpha.C)\,N_u N_b]\!]_\theta \coloneqq \langle T, \textbf{return}, \gg\!= \rangle$

where $TX \coloneqq \left|[\![C]\!]_{(\theta[\alpha \mapsto X])}\right|$ $\quad \textbf{return}^X \coloneqq [\![N_u]\!]_{(\theta[\alpha\mapsto X])} : X \to TX$

$\gg\!=^{X,Y} \coloneqq [\![N_b]\!]_{(\theta[\alpha_1\mapsto X, \alpha_2 \mapsto Y])} : TX \to (X \to TY) \to TY$

(provided these form a monad)

Monads

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

Computation terms

$[\![\, \lceil N \rceil^T \,]\!](\gamma) \coloneqq [\![N]\!](\gamma)$

$[\![\, \mu(N) \,]\!](\gamma) \coloneqq [\![N]\!](\gamma)$

Fig. 16: MON denotational semantics (extending Fig. 7 and 8)

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 11 (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

$$[\![\mu(N)]\!] = [\![N]\!] \gg\!= \mathrm{id} \qquad [\![\, \lceil N \rceil^T \,]\!] = \textbf{return}^T\, [\![N]\!]$$

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 12 (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* $M \rightsquigarrow^\star$ **return** V.

Our proof uses Lindley & Stark's $\top\top$-lifting (2005).

Theorem 13 (MON compositionality)**.** *The semantics depends only on the semantics of subterms: 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 14 (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* $P \rightsquigarrow^*$ **return** V *then* $[\![P]\!] = [\![V]\!]$.

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

Theorem 15 (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 16 (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* $[\![C]\!]_\theta$.

$M, N ::= \ldots$ computations
$\mid \mathbf{S_0} k.M$ shift-0
$\mid \langle M | x.N \rangle$ reset

(a) Syntax (extending Fig. 2)

Frames and contexts

$\mathcal{F} ::= \ldots \mid \langle [\,] | x.N \rangle$ computation frames

Beta reduction

$(ret) \quad \langle (\mathbf{return}\ V) | x.M \rangle \rightsquigarrow_\beta M[V/x]$

$(capture) \quad \langle \mathcal{H}[\mathbf{S_0} k.M] | x.N \rangle \rightsquigarrow_\beta M[\lambda y.\, \langle \mathcal{H}[\mathbf{return}\ y] | x.N \rangle / k]$

(b) Operational semantics (extending Fig. 3)

Fig. 17: DEL

EFF does not possess the finite denotation property. For example, for the effect $E := \{\mathsf{tick} : 1 \to 1\}$, we have infinitely many different E-returner denotations:

$$|[\![\vdash_{\mathrm{k}} F\,1 : \mathbf{Comp}_E]\!]_\theta| = \{\mathsf{tick}^n\, [\![()]\!]_\theta | n \in \mathbb{N}\}$$

Our inexpressivity proof (Theorem 27) will use the facts that: (a) all of these returners are definable in EFF, and (b) they are observationally distinguishable.

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 $\mathbf{S_0}$, pronounced "shift zero". It was introduced by Danvy & Filinski (1990) as part of a systematic study of continuation-passing-style conversion.

5.1 Syntax

Fig. 17(a) presents the extension DEL. The construct $\mathbf{S_0} k.M$, which we will call "shift", captures the current continuation and binds it to k, and replaces it with M. The construct $\langle M | x.N \rangle$, 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 (Kiselyov & Shan, 2007; Materzok & Biernacki, 2012).

Example 17. Fig. 18(a) 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

$$
\begin{array}{lll}
\mathit{toggle} & = \{x \leftarrow \mathit{get}!;\ y \leftarrow \mathit{not}!\ x;\ \mathit{put}!\ y; & \mathit{toggle} : U_{\mathit{State}} F\,\mathbf{bit} \\
 & \quad \mathbf{return}\ x\} & \\
\mathit{get} & = \{\mathbf{S_0} k.\lambda s.k!\ s\ s\} & \mathit{get} : U_{\mathit{State}} F\,\mathbf{bit} \\
\mathit{put} & = \{\lambda s'.\mathbf{S_0} k.\lambda\_.k!\ ()\ s'\} & \mathit{put} : U_{\mathit{State}}(\mathbf{bit} \to F1) \\
\mathit{runState} & = \{\lambda c.\langle c!|x.\lambda s.x\rangle\} & \mathit{runState} : U_{\emptyset}((U_{\mathit{State}} F\,\mathbf{bit}) \to \mathbf{bit} \to F\,\mathbf{bit}) \\
 & & \mathit{State} = \emptyset, \mathbf{bit} \to F\,\mathbf{bit} : \mathbf{Eff}
\end{array}
$$

(a) terms (b) types

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)

substituted in the remainder of the reset computation. For the *capture* rule, the definition of pure contexts guarantees that in the reduct $\langle \mathcal{H}[\mathbf{S_0} k.M]|x.N\rangle$ there are no intervening resets in $\mathcal{H}$, and as a consequence $\mathcal{H}$ *is* the delimited continuation of the evaluated shift. After the reduction takes place, the continuation is re-wrapped with the reset, while the body of the shift has access to the enclosing continuation. If we were to, instead, not re-wrap the continuation with a reset, we would obtain the control/prompt-zero operators (cf. Shan's (2007) and Kiselyov *et al.*'s (2005) analyses of macro expressivity relationships between these two, and other, variations on untyped delimited control).

Example 18. We have:

$$\mathit{runState}!\ \mathit{toggle}\ \texttt{True} \rightsquigarrow^* \langle \texttt{True}|x.\lambda s.x\rangle\ \texttt{False} \rightsquigarrow^* \mathbf{return}\ \ \texttt{True}.$$ □

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 & Filinski, 1989); they were the first to propose 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 19. Fig. 18(b) types the boolean state terms. □

In this type system, the return type of the continuation remains fixed inside every reset. Existing work on type systems for delimited control (Kiselyov & 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 17 (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, $\simeq$, and $\simeq_{\text{cong}}$ as in MAM.

6 Macro Translations

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

Translation Notation We define translations S→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 & 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 18. *For all* $\Theta \vdash_{\text{k}} E : \mathbf{Eff}$, $\Theta \vdash_{\text{k}} C : \mathbf{Comp}_E$, *we have the following* proper *monad Cont:*

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

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

$$\underline{\mathbf{S_0}k.M} := \mu(\lambda k.\underline{M}) \qquad\qquad \underline{\langle M|x.N\rangle} := [\underline{M}]^{\,\mathsf{Cont}}\ \{\lambda x.\underline{N}\}$$

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

Theorem 19 (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:

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

Theorem 20 (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 21 (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:

$$\underline{\mu(M)} := \mathbf{S_0}k.\lambda b.b!\ (\{\underline{M}\},\{\lambda x.k!\ x\ b\})$$

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

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 22 (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 a form of answer-type polymorphism along the lines of Piróg *et al.* (2019) 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 bind need not 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:

$$\underline{\mu(M)} := \mathbf{S_0} k.\mathbf{S_0} b.b!\ (\{\underline{M}\}, \{\lambda x.\, \langle k!\ x | (y,f)\, .b!\, (y,f)\rangle\})$$

$$\underline{[M]^{\,\mathbf{where}\,\{\mathbf{return}\ x=N_{\mathrm{u}};\, y \gg= f=N_{\mathrm{b}}\}}} := \left\langle \left\langle \underline{M} \middle| x.\mathbf{S_0} b.\underline{N_{\mathrm{u}}} \right\rangle \middle| (y,f)\, .\underline{N_{\mathrm{b}}} \right\rangle$$

In the translation of reflection, the reset inside the wrapped continuation ensures that any further reflections in the continuation are interpreted appropriately: the two shifts have popped unit and bind off the stack; the reset first pushes bind back on the stack and then invoking k implicitly pushes unit back on the stack. In the translation of reification, the shift guarding the unit garbage collects bind once it is no longer needed.

(There is an error in our earlier paper (Forster *et al.*, 2017): $\langle k!\ x | (y,f)\, .b!\, (y,f)\rangle$ was $\langle k!\ x | z.z!\ b\rangle$.)

*6.3 Delimited Continuations as Effect Handlers (*DEL→EFF*)*

We define DEL→EFF as follows:

$$\underline{\mathbf{S_0} k.N} := \mathsf{shift0}\ \{\lambda k.\underline{N}\}$$

$$\underline{\langle M | x.N\rangle} := \mathbf{handle}\ \underline{M}\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto \underline{N}\} \uplus \{\mathsf{shift0}\, p\, k \mapsto p!\ k\}$$

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

Theorem 23 (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:

$$\underline{\mathsf{op}\ V} \coloneqq \mathbf{S_0}k.\lambda h.h!\,(\mathbf{inj}_{\mathsf{op}}\ (\underline{V}, \{\lambda y.k!\ y\ h\}))\qquad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq \left\langle \underline{M} \middle| H^{\mathrm{ret}} \right\rangle\ \{H^{\mathrm{ops}}\}$$

$$\begin{pmatrix} \{\mathbf{return}\ x \mapsto N_{\mathsf{ret}}\} \\ \uplus\ \{\mathsf{op}_i\, p\, k \mapsto N_i\}_i \end{pmatrix}^{\mathrm{ret}} \coloneqq x.\lambda h.\underline{N_{\mathsf{ret}}}$$

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

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 24 (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 a form of answer-type polymorphism along the lines of Piróg *et al.* (2019) is sufficient to adapt this translation to one that does preserve 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:

$$\underline{\mathsf{op}\ V} \coloneqq \mathbf{S_0}k.\mathbf{S_0}h.h!\,(\mathbf{inj}_{\mathsf{op}}\ (\underline{V}, \{\lambda x.\langle k!\ x | y.h!\ y\rangle\}))\quad \underline{\mathbf{handle}\ M\ \mathbf{with}\ H} \coloneqq \langle\langle \underline{M} | H^{\mathrm{ret}}\rangle | H^{\mathrm{ops}}\rangle$$

$$\begin{pmatrix} \{\mathbf{return}\ x \mapsto N_{\mathsf{ret}}\} \\ \uplus\ \{\mathsf{op}_i\, p\, k \mapsto N_i\}_i \end{pmatrix}^{\mathrm{ret}} \coloneqq x.\mathbf{S_0}h.\underline{N_{\mathsf{ret}}}$$

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

(There is an error in our earlier paper (Forster *et al.*, 2017): $\langle k!\ x | y.h!\ y\rangle$ was $\langle k!\ x | y.y!\ h\rangle$.)

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:

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

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

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 25 (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 below it:

$$\begin{array}{l}[b \leftarrow \mu(\{\lambda\,(b,f).b\});\\ \ f \leftarrow \mu(\{\lambda\,(b,f).f\});\\ \ f!\,b]^{\mathsf{Reader}}\ (\mathbf{inj}_{\mathsf{true}}\,(), \{\lambda b.\mathbf{return}\ b\})\end{array}$$

$$\begin{array}{l}\vdash_{\mathrm{k}} \emptyset \prec \mathbf{instance\ monad}\,(\alpha.\mathbf{bit} \times U_{\emptyset}\,(\mathbf{bit} \to F\,\mathbf{bit}) \to F\alpha)\\ \qquad \mathbf{where}\,\{\mathbf{return}\ x = \lambda e.\mathbf{return}\ x;\\ \qquad\qquad m \gg\!= f = \lambda e.x \leftarrow m!\ e;\ f!\ x\ e\} : \mathbf{Eff}\end{array}$$

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:

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

$$\begin{pmatrix}\{\mathbf{return}\ x \mapsto N_{\mathsf{ret}}\}\\ \uplus\ \{\mathsf{op}_i\, p\, k \mapsto N_i\}_i\end{pmatrix}^{\mathrm{ret}} \coloneqq \lambda x.\lambda h.\underline{N_{\mathsf{ret}}}$$

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

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 26 (EFF→MON correctness)**.** MON *simulates* EFF *up to congruence:*

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

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:

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

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

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

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 27. *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:* $M \rightsquigarrow N \implies \underline{M} \simeq \underline{N}$.

Proof

Our proof of the first part hinges on the finite denotation property (Lemma 16). Assume to the contrary that there was such a translation. Consider a single effect operation symbol $\mathsf{tick} : 1 \to 1$ and set $\mathsf{tick}^0 := \textbf{return}\ ()$, 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 20. ■

Regarding the remaining four possibilities, we have seen that there is a typeability-preserving macro translation TYPED DEL→TYPED MON (Theorem 20), but we conjecture that there are no typeability-preserving translations TYPED MON→TYPED DEL, TYPED DEL→TYPED EFF, or TYPED MON→TYPED EFF.

Returning to the untyped translations, we emphasise that though macro expressivity captures some of the intuitive differences in expressiveness of programming language features, it leaves something to be desired, as not all macro translations have equal status.

A concrete feature that distinguishes the translations into EFF is that they satisfy simulation on the nose, whereas all of the other translations only satisfy simulation up to congruence. In principle, this could have practical consequences as administrative reductions may be deferred and slow down computation. That said, we do not have concrete evidence that this is a problem in practice.

Inspecting the translations between EFF and the other calculi, there is a clear sense in which the translations into EFF are "simpler" than those from EFF. This intuition extends to the polymorphically typed translations of Piróg *et al.* (2019). Their translation from delimited continuations into effect handlers relies only on a natural notion of polymorphic operations, whilst the converse translation relies on a bespoke variant of answer-type-polymorphism. In our setting, whilst the translations from DEL and MON into EFF are direct, those from EFF into MON and DEL have the flavour of a double-negation translation using the continuation monad, or a deep-embedding using a free monad.

7 Abella Experience Report

We have mechanised the proofs of safety (Theorems 1, 6, 11, 17) for each calculus and the correctness theorems for all translations (Theorems 19, 20, 22, 23, 24, 25, 26) in the Abella proof assistant (Gacek, 2008). Additionally, we have mechanised the proofs of correctness for the two alternative translations described in Sections 6.2.1 and 6.4.1.

We already had positive prior experience (Bauer & Pretnar, 2014; Kammar & Pretnar, 2017) with the concise higher-order abstract syntax (HOAS) encoding in Twelf, and it made sense to follow the same approach in this development, especially with the large number of bindings in the programming abstractions we considered. Fig. 20 confirms this, as even with all the repetition, both the specification and the proofs are reasonable in size. We chose Abella because in addition to HOAS, it provides a simple tactic language for interactively building proofs and allows one to write propositions in terms of a first-order logic with

Specification		
automatically generated (template LoC)		
calculi	4×223	892
translations	6×74	444
manual		
calculi		224
translations		292
	sub-total	1,852
Proofs		
lemmas		465
safety		351
correctness of untyped translations		1,392
type preservation for DEL→MON		544
	sub-total	2,752
	total	**4,604**

Fig. 20: Abella formalisation size in lines-of-code (LoC)

equality. This is in contrast to Twelf where proof terms are constructed manually, while theorems amount to the existence of a total relation between input and output types of $\forall\exists$-statements.

In general, the user experience with Abella was pleasant. We managed to discover a bug in the logic programming engine[6], but that was quickly resolved. On the top of our wish list for a future release of Abella is support for tactic automation, as a significant portion of our proofs amounts to routine proofs by induction. In fact, a lot of proofs would already be much shorter with a construct that attempts to use the same tactic for all subgoals (as in Coq). A smaller improvement would be an abbreviation mechanism similar to `%abbrev` in Twelf, which would allow us to transparently annotate the considerably large translations of effect constructs with a single term.

Avoiding boilerplate when formalising multiple calculi. As we are comparing different calculi, our specification uses separate syntactic sorts for each calculus. Each calculus has a significant number of distinct sorts: effects, value types, computation types, values, computations, and contexts. In addition, variants require auxiliary sorts of computation and value type lists. Finally, we need type kinding judgements for each type sort, typing judgements for each term sort, and translation relations for each sort and each pair of calculi. The calculi share most constructs and judgements, while macro translations are mostly trivial, so a lot of the listed specification is boilerplate. In order to reduce it, we used two mechanisms. First, a simple Python script that instantiates a template for a calculus and a translation definition. Next, instead of modifying the generated files, we used the specification accumulation mechanism of Abella, which allowed us to provide the additional parts of the specification in a separate file that imports the automatically generated one. In this way, the generated files can be replaced without a problem if the base calculus changes.

[6] https://github.com/abella-prover/abella/issues/107

The extension mechanism works well. It allows one to import multiple such signatures, while Abella keeps track that they all agree on the common definitions.

Encoding of translations. Abella has no function definitions, and we formalise type and term translations as relations. Thus, we formalise statements like $M \rightsquigarrow N \implies \underline{M} \rightsquigarrow^{+}_{\text{cong}} \underline{N}$ as:

$$\forall M, N, \underline{M}. \quad (M \rightsquigarrow N \wedge M \rightarrow_{\text{trans}} \underline{M}) \implies \exists \underline{N}.(N \rightarrow_{\text{trans}} \underline{N} \wedge \underline{M} \rightsquigarrow^{+}_{\text{cong}} \underline{N})$$

In a proof assistant that incorporates inductive types, such as Coq or Agda, the existence of translations could be proven by induction on the structure of M. In Abella, which provides induction only over relations, one needs to define an define auxiliary predicate on terms that traces their structure (Baelde *et al.*, 2014, page 21). In our case, the existence of $\underline{N}$ could also be proven by induction on the relation $M \rightsquigarrow N$, so we did not have to modify the theorem statement.

Well-kindedness of types. The most significant extension to our previous mechanisation (Forster *et al.*, 2017) is the proof of Theorem 20, i.e. a well-typed phrase $\Theta; \Gamma \vdash_E P : X$ in DEL implies $\Theta; \underline{\Gamma} \vdash_{\underline{E}} \underline{P} : \underline{X}$ in MON. Again, one can show that $\underline{P}$ exists by induction on the typing derivation, while for types, effects and environments, we again needed an inductive relation, amounting to the well-kindedness relation, which we now include. To minimise the overhead, this was done by splitting each typing judgement into two: one stating the usual conditions of the typing rule, and the second one requiring the first and well-kindedness of all types involved. This change did break all of our previous safety proofs, though with the aid of a good text editor, the rewriting of proofs was straightforward. In a proof assistant with inductively defined types, we could get away with induction on their structure, but this would swiftly break when moving to a polymorphic type-system, where well-kindedness is not trivial.

Environment translations. Following the HOAS approach, the value environments are encoded in Abella by assuming typing judgements for fresh value terms. For example, $\Theta; x : A \vdash_E M : C$ is given by stating that for any fresh value x, the computation $M\ x$ (recall that in HOAS, a term with a free variable is represented by a term abstraction) has the type C under the assumption that x has type A. The type environment Θ is encoded implicitly by assuming appropriate fresh types. Theorems about such judgements are given through an auxiliary predicate, which limits the form of possible assumptions (Baelde *et al.*, 2014, page 40).

For translation of environments, we require a fresh value $\underline{x}$ and three judgements with synchronised assumptions (Baelde *et al.*, 2014, page 75): first, we require that $M\ x$ has type C under the assumption $x : A$ as before; next, we similarly require that $\underline{M}\ \underline{x}$ has type $\underline{C}$ under the assumption $\underline{x} : \underline{A}$; finally, we require that $M\ x \rightarrow_{\text{trans}} \underline{M}\ \underline{x}$ under the assumption $x \rightarrow_{\text{trans}} \underline{x}$. (Through subordination, Abella can infer that $\underline{M}$ cannot depend on x as they belong to distinct syntactic sorts.) Since fresh values in Abella are represented with nominal constants, there are infinitely many fresh values $\underline{x}$ that x can be translated to. This breaks the

obvious inductive proof of the Theorem 20, which in terms of relations is written as:

$$\forall \Gamma, E, P, X.\ (\Gamma \vdash_E P : X) \implies$$
$$\exists \underline{\Gamma}, \underline{E}, \underline{P}, \underline{X}.\ (\Gamma \to_{\text{trans}} \underline{\Gamma}) \wedge (E \to_{\text{trans}} \underline{E}) \wedge (P \to_{\text{trans}} \underline{P}) \wedge (X \to_{\text{trans}} \underline{X}) \wedge (\underline{\Gamma} \vdash_{\underline{E}} \underline{P} : \underline{X})$$

The reason is that each inductive hypothesis $\Theta; \Gamma_i \vdash_{E_i} P_i : X_i$ gives us a different translation $\underline{\Gamma}_i$. A workaround is to first prove

$$\forall \Gamma, E, P, X, \underline{\Gamma}.\ (\Gamma \vdash_E P : X) \wedge (\Gamma \to_{\text{trans}} \underline{\Gamma}) \implies$$
$$\exists \underline{E}, \underline{P}, \underline{X}.\ (E \to_{\text{trans}} \underline{E}) \wedge (P \to_{\text{trans}} \underline{P}) \wedge (X \to_{\text{trans}} \underline{X}) \wedge (\underline{\Gamma} \vdash_{\underline{E}} \underline{P} : \underline{X})$$

and then show separately that each well-kinded environment Γ has a suitable translation $\underline{\Gamma}$, and appealing to well-kindedness.

8 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 would like to extend each type system until each translation preserves typeability. We conjecture that adding polymorphic operations to EFF, data type polymorphism to MON, and a suitable form of answer-type polymorphism to DEL would enable typed macro-transformations between each pair of calculi. We have reason to believe this should pan out as Piróg *et al.* (2019) have recently shown such a correspondence for call-by-value analogues of EFF and DEL extended respectively with polymorphic operations and a novel form of answer-type polymorphism. Their calculi also include other features including a row-polymorphic effect type system in the style of Leijen (2017), supporting duplicate effect labels and effect subtyping.

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.*, 2016) 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 (Bauer & Pretnar, 2014; Hillerström & Lindley, 2016; Leijen, 2017; Lindley *et al.*, 2017; Lucassen & Gifford, 1988; Pretnar, 2014; Saleh *et al.*, 2018). To these we add effect forwarding (Kammar *et al.*, 2013) and rebasing (Filinski, 2010).

Recent work (Brachthäuser *et al.*, 2019; Inostroza & van der Storm, 2018; Zhang & Myers, 2019) explores alternative formulations of effect handlers with close connections to object-oriented programming models. We would like to apply our methodology to study the relative expressiveness of these alternative formulations. Inspired by the suggestion of Bračevac *et al.* (2018) to associate effect handlers with implicits, Brachthauser & Leijen (2019) have recently proposed another basis for user-defined effects. They extend a calculus of dynamic binding with implicit functions and implicit control. Following our lead, they give macro translations back and forth between effect handlers and their calculus of dynamic binding.

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.* (2019) 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 (2017, 2002, 2013), 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.

Acknowledgements. Supported by the European Research Council grant 'events causality and symmetry — the next-generation semantics', a Balliol College Oxford Career Development Fellowship, a Royal Society University Research Fellowship, 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'. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326. We thank Bob Atkey, Andrej Bauer, Paul Downen, Marcelo Fiore, Tamara von Glehn, Mathieu Huot, Oleg Kiselyov, 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, Filip Sieczkowsi, Ian Stark, Sam Staton, Philip Wadler, Jeremy Yallop, and the anonymous referees for useful suggestions and discussions.

Bibliography

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

Asai, Kenichi, & Kameyama, Yukiyoshi. (2007). Polymorphic delimited continuations. *Pages 239–254 of:* Shao, Zhong (ed), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Atkey, Robert. (2009). Parameterised notions of computation. *Journal of functional programming*, **19**(3-4), 335–376.

Baelde, David, Chaudhuri, Kaustuv, Gacek, Andrew, Miller, Dale, Nadathur, Gopalan, Tiu, Alwen, & Wang, Yuting. (2014). Abella: A system for reasoning about relational specifications. *Journal of formalized reasoning*, **7**(2), 1–89.

Barr, Michael, & Wells, Charles. (1985). *Toposes, triples, and theories*. Grundlehren der mathematischen Wissenschaften. Springer-Verlag.

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

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, **84**(1), 108 – 123. Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

Brachthauser, Jonathan, & Leijen, Daan. 2019 (March). *Programming with implicit values, functions, and control (or, implicit functions: dynamic binding with lexical scoping)*. Tech. rept. MSR-TR-2019-7. Microsoft.

Brachthäuser, Jonathan Immanuel, Schuster, Philipp, & Ostermann, Klaus. (2019). *Effekt: Type- and effect-safe, extensible effect handlers in scala*. Unpublished draft[7]

Brady, Edwin. (2013). Programming and reasoning with algebraic effects and dependent types. *Pages 133–144 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Bračevac, Oliver, Amin, Nada, Salvaneschi, Guido, Erdweg, Sebastian, Eugster, Patrick, & Mezini, Mira. (2018). Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, **2**(ICFP), 67:1–67:31.

Bulwahn, Lukas, Krauss, Alexander, Haftmann, Florian, Erkök, Levent, & Matthews, John. (2008). Imperative functional programming with Isabelle/HOL. *Pages 134–149 of:* Mohamed, Otmane Ait, Muñoz, César, & Tahar, Sofiène (eds), *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Danvy, Olivier. (2006). *An analytical approach to programs as data objects*. DSc dissertation, Department of Computer Science, University of Aarhus.

Danvy, Olivier, & Filinski, Andrzej. (1989). *A functional abstraction of typed contexts*. Tech. rept. 89/12. DIKU.

[7] Available from:
http://ps.informatik.uni-tuebingen.de/publications/brachthaeuser19effekt/ .

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. New York, NY, USA: ACM.

Doczkal, Christian. (2007). *Strong normalization of CBPV*. Tech. rept. Saarland University.

Doczkal, Christian, & Schwinghammer, Jan. (2009). Formalizing a strong normalization proof for moggi's computational metalanguage: A case study in Isabelle/HOL-nominal. *Pages 57–63 of: Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP '09. New York, NY, USA: ACM.

Felleisen, Matthias. (1991). On the expressive power of programming languages. *Science of computer programming*, **17**(1), 35 – 75.

Felleisen, Matthias, & Friedman, Daniel P. (1987). A reduction semantics for imperative higher-order languages. *Pages 206–223 of:* de Bakker, J. W., Nijman, A. J., & Treleaven, P. C. (eds), *PARLE Parallel Architectures and Languages Europe*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Felleisen, Matthias, Wand, Mitch, Friedman, Daniel, & Duba, Bruce. (1988). Abstract continuations: A mathematical semantics for handling full jumps. *Pages 52–62 of: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. LFP '88. New York, NY, USA: ACM.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. New York, NY, USA: ACM.

Filinski, Andrzej. (1996). *Controlling effects*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Filinski, Andrzej. (1999). Representing layered monads. *Pages 175–188 of: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. New York, NY, USA: ACM.

Filinski, Andrzej. (2010). Monads in action. *Pages 483–494 of: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. New York, NY, USA: ACM.

Forster, Yannick. (2016). *On the expressive power of effect handlers and monadic reflection*. Tech. rept. University of Cambridge.

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2017). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, **1**(ICFP), 13:1–13:29.

Forster, Yannick, Schäfer, Steven, Spies, Simon, & Stark, Kathrin. (2019). Call-by-push-value in Coq: Operational, equational, and denotational theory. *Pages 118–131 of: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. New York, NY, USA: ACM.

Gacek, Andrew. (2008). The Abella interactive theorem prover (system description). *Pages 154–161 of:* Armando, Alessandro, Baumgartner, Peter, & Dowek, Gilles (eds), *Automated Reasoning*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Gacek, Andrew. 2009 (September). *A framework for specifying, prototyping, and reasoning about computational systems*. Ph.D. thesis, University of Minnesota.

Hermida, Claudio. (1993). *Fibrations, logical predicates and related topics*. Ph.D. thesis, University of Edinburgh.

Hillerström, Daniel, & Lindley, Sam. (2016). Liberating effects with rows and handlers. *Pages 15–27 of: Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. New York, NY, USA: ACM.

Hillerström, Daniel, Lindley, Sam, Atkey, Robert, & Sivaramakrishnan, K. C. (2017). Continuation Passing Style for Effect Handlers. *Pages 18:1–18:19 of:* Miller, Dale (ed), *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 84. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Hutton, Graham, & Meijer, Erik. (1998). Monadic parsing in Haskell. *Journal of functional programming*, **8**(4), 437–444.

Inostroza, Pablo, & van der Storm, Tijs. (2018). JEff: Objects for effect. *Pages 111–124 of: Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. New York, NY, USA: ACM.

Kammar, Ohad. (2014). *An algebraic theory of type-and-effect systems*. Ph.D. thesis, University of Edinburgh.

Kammar, Ohad, & McDermott, Dylan. (2018). Factorisation systems for logical relations and monadic lifting in type-and-effect system semantics. *Electronic notes in theoretical computer science*, **341**, 239 – 260. Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV).

Kammar, Ohad, & Plotkin, Gordon D. (2012). Algebraic foundations for effect-dependent optimisations. *Pages 349–360 of: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: ACM.

Kammar, Ohad, & Pretnar, Matija. (2017). No value restriction is needed for algebraic effects and handlers. *Journal of functional programming*, **27**, e7.

Kammar, Ohad, Lindley, Sam, & Oury, Nicolas. (2013). Handlers in action. *Pages 145–158 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Katsumata, Shin-ya. (2014). Parametric effect monads and semantics of effect systems. *Pages 633–645 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Kiselyov, Oleg. (2016). Parameterized extensible effects and session types (extended abstract). *Pages 41–42 of: Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. New York, NY, USA: ACM.

Kiselyov, Oleg, & Shan, Chung-chieh. (2007). A substructural type system for delimited continuations. *Pages 223–239 of:* Della Rocca, Simona Ronchi (ed), *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Kiselyov, Oleg, & Sivaramakrishnan, KC. (2018). Eff directly in OCaml. *Pages 23–58 of:* Asai, Kenichi, & Shinwell, Mark (eds), Proceedings *ML Family Workshop / OCaml Users and Developers workshops,* Nara, Japan, September 22-23, 2016. Electronic Proceedings in Theoretical Computer Science, vol. 285. Open Publishing Association.

Kiselyov, Oleg, Friedman, Daniel P., & Sabry, Amr A. (2005). *How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible*. Tech. rept. Indiana University School of Informatics and Computing. Technical Report TR611.

Kiselyov, Oleg, Shan, Chung-chieh, & Sabry, Amr. (2006). Delimited dynamic binding. *Pages 26–37 of: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. New York, NY, USA: ACM.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. (2013). Extensible effects: An alternative to monad transformers. *Pages 59–70 of: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. New York, NY, USA: ACM.

Kobori, Ikuo, Kameyama, Yukiyoshi, & Kiselyov, Oleg. (2016). Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. *Pages 36–52 of:* Danvy, Olivier, & de'Liguoro, Ugo (eds), Proceedings of the *Workshop on Continuations,* London, UK, April 12th 2015. Electronic Proceedings in Theoretical Computer Science, vol. 212. Open Publishing Association.

Laird, J. (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), 470 – 500. Eighth Games for Logic and Programming Languages Workshop (GaLoP).

Laird, James. (2002). Exceptions, continuations and macro-expressiveness. *Pages 133–146 of:* Le Métayer, Daniel (ed), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Laird, James. (2013). Combining and relating control effects and their semantics. *Pages 113–129 of:* de'Liguoro, Ugo, & Saurin, Alexis (eds), Proceedings First Workshop on *Control Operators and their Semantics, Eindhoven, The Netherlands, June 24-25, 2013*. Electronic Proceedings in Theoretical Computer Science, vol. 127. Open Publishing Association.

Leijen, Daan. (2017). Type directed compilation of row-typed algebraic effects. *Pages 486–499 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Levy, Paul Blain. (2003). *Call-by-push-value: A functional/imperative synthesis*. Semantics Structures in Computation, vol. 2. Dordrecht: Springer Netherlands.

Lindley, Sam, & Stark, Ian. (2005). Reducibility and $\top\top$-lifting for computation types. *Pages 262–277 of:* Urzyczyn, Paweł (ed), *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Lindley, Sam, McBride, Conor, & McLaughlin, Craig. (2017). Do be do be do. *Pages 500–514 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Lucassen, J. M., & Gifford, D. K. (1988). Polymorphic effect systems. *Pages 47–57 of: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. New York, NY, USA: ACM.

Marmolejo, Francisco, & Wood, Richard J. (2010). Monads as extension systems — no iteration is necessary. *Theory and applications of categories*, **24**(4), 84–113.

Materzok, Marek, & Biernacki, Dariusz. (2012). A dynamic interpretation of the cps hierarchy. *Pages 296–311 of:* Jhala, Ranjit, & Igarashi, Atsushi (eds), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Moggi, E. (1989). Computational lambda-calculus and monads. *Pages 14–23 of: Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Piscataway, NJ, USA: IEEE Press.

Paré, Robert. (1974). Colimits in topoi. *Bulletin of the american mathematical society*, **80**(3), 556–561.

Piróg, Maciej, Polesiuk, Piotr, & Sieczkowski, Filip. (2019). Typed equivalence of effect handlers and delimited control. *Pages 30:1–30:16 of:* Geuvers, Herman (ed), *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 131. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Plotkin, Gordon, & Power, John. (2002). Notions of computation determine monads. *Pages 342–356 of:* Nielsen, Mogens, & Engberg, Uffe (eds), *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Plotkin, Gordon, & Power, John. (2003). Algebraic operations and generic effects. *Applied categorical structures*, **11**(1), 69–94.

Plotkin, Gordon, & Pretnar, Matija. (2008). A logic for algebraic effects. *Pages 118–129 of: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*. LICS '08. Washington, DC, USA: IEEE Computer Society.

Plotkin, Gordon, & Pretnar, Matija. (2009). Handlers of algebraic effects. *Pages 80–94 of:* Castagna, Giuseppe (ed), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Pretnar, Matija. (2014). Inferring Algebraic Effects. *Logical Methods in Computer Science*, **Volume 10, Issue 3**(Sept.).

Pretnar, Matija. (2015). An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science*, **319**, 19 – 35. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

Reynolds, John C. (1998). *Theories of programming languages*. Cambridge University Press.

Saleh, Amr Hany, Karachalias, Georgios, Pretnar, Matija, & Schrijvers, Tom. (2018). Explicit effect subtyping. *Pages 327–354 of:* Ahmed, Amal (ed), *Programming Languages and Systems*. Cham: Springer International Publishing.

Schrijvers, Tom, Tack, Guido, Wuille, Pieter, Samulowitz, Horst, & Stuckey, Peter J. (2013). Search combinators. *Constraints*, **18**(2), 269–305.

Schrijvers, Tom, Piróg, Maciej, Wu, Nicolas, & Jaskelioff, Mauro. (2019). Monad transformers and modular algebraic effects: what binds them together. *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. New York, NY, USA: ACM. To appear.

Shan, Chung-chieh. (2007). A static simulation of dynamic delimited control. *Higher-order and symbolic computation*, **20**(4), 371–401.

Sinkovics, Ábel, & Porkoláb, Zoltán. (2013). Implementing monads for C++ template metaprograms. *Science of computer programming*, **78**(9), 1600 – 1621.

Spivey, Mike. (1990). A functional theory of exceptions. *Science of computer programming*, **14**(1), 25 – 42.

Swierstra, Wouter. (2008). Data types à la carte. *Journal of functional programming*, **18**(4), 423–436.

Tait, William W. (1967). Intensional interpretations of functionals of finite type I. *Journal of symbolic logic*, **32**(2), 198–212.

Wadler, Philip. (1990). Comprehending monads. *Pages 61–78 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. New York, NY, USA: ACM.

Wadler, Philip. (1994). Monads and composable continuations. *Lisp and symbolic computation*, **7**(1), 39–55.

Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Information and computation*, **115**(1), 38 – 94.

Zhang, Yizhou, & Myers, Andrew C. (2019). Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.*, **3**(POPL), 5:1–5:29.

Ziliani, Beta, Dreyer, Derek, Krishnaswami, Neelakantan R., Nanevski, Aleksandar, & Vafeiadis, Viktor. (2015). Mtac: A monad for typed tactic programming in Coq. *Journal of functional programming*, **25**, e12.