Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 298 (2013) 51–69

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Normalization by Evaluation and Algebraic Effects

Danel Ahman[1]

Laboratory for Foundations of Computer Science
University of Edinburgh

Sam Staton[2]

Computer Laboratory
University of Cambridge

Abstract

We examine the interplay between computational effects and higher types. We do this by presenting a normalization by evaluation algorithm for a language with function types as well as computational effects. We use algebraic theories to treat the computational effects in the normalization algorithm in a modular way. Our algorithm is presented in terms of an interpretation in a category of presheaves equipped with partial equivalence relations. The normalization algorithm and its correctness proofs are formalized in dependent type theory (Agda).

Keywords: Algebraic effects, Type theory, Normalization by evaluation, Presheaves, Monads

1 Introduction

When studying computer programs it is often appropriate to consider them up to some equations. In this paper we consider an equational theory for impure functional programs. By finding a class of normal forms for this equational theory, we are able to understand and manipulate the notions under study directly. Moreover, it has been proposed that normalization algorithms are of use in partial evaluation: if a program fragment with free variables is normalized at compile-time then it will typically run faster.

[1] Email: d.ahman@ed.ac.uk

[2] Email: sam.staton@cl.cam.ac.uk

1571-0661/$ – see front matter
http://dx.doi.org/10.1016/j.entcs.2013.09.007

To be more precise, we introduce a small program in an ML-like language.

```
(fn (g:(unit -> unit) -> unit)
   => g (if recv()=0 then fn x => send 0 ; h x else fn y => send 1))
(fn (f:unit -> unit) => f () ; f ())                                    (∗)
```

Here `recv:unit->bit` and `send:bit->unit` are network communication primitives, as in Concurrent ML [41], and `h:unit->unit` is a free identifier of function type. Notice that we cannot naively compile and run this program to find out what it does, because it has a free identifier `h`, and because its execution will depend on what is received from the network.

Before we normalize the program, we translate it to an intermediate language which makes the evaluation order clear. We also remove the `bit` type from the program, since it complicates the normalization process and is orthogonal to what we are investigating. (We return to the issue of sum types in §6). We eliminate the need for a `bit` type by using algebraic operations, following [36]: we replace `(if recv()=0 then` M `else` N`)` by $\mathsf{inp}[M, N]$, replace `(send 0 ;` M`)` by $\mathsf{out}_0[M]$ and `(send 1 ;` M`)` by $\mathsf{out}_1[M]$. Thus the program (∗) becomes

$$\begin{array}{l}\big(\mathtt{fn}\, g{:}((\mathtt{unit} \rightharpoonup \mathtt{unit}) \rightharpoonup \mathtt{unit}) \Rightarrow \mathsf{inp}[\mathtt{return}\, \mathtt{fn}\, x \Rightarrow \mathsf{out}_0[h\, x], \qquad (\dagger)\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{return}\, \mathtt{fn}\, y \Rightarrow \mathsf{out}_1[\mathtt{return}\, \langle\rangle]]\ \ \mathtt{to}\, f.\, g\, f\big)\\ \big(\mathtt{fn}\, f{:}(\mathtt{unit} \rightharpoonup \mathtt{unit}) \Rightarrow f\, \langle\rangle\, \mathtt{to}\, y.\, f\, \langle\rangle\big)\end{array}$$

The intermediate language (§2) has a straightforward equational theory, including β- and η-equality. The program (†) is not in normal form for these equations, e.g. it has a β-redex. Our normalization algorithm yields the following program:

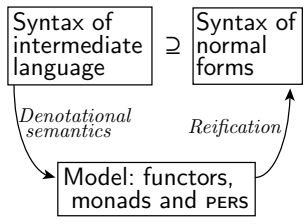
$$\mathsf{inp}[\mathsf{out}_0[h\, \langle\rangle\, \mathtt{to}\, x.\, \mathsf{out}_0[h\, \langle\rangle\, \mathtt{to}\, y.\, \mathtt{return}\, \langle\rangle]], \mathsf{out}_1[\mathsf{out}_1[\mathtt{return}\, \langle\rangle]]] \qquad (\ddagger)$$

So we discover what the program (∗) does: it inputs a bit from the network. If that bit is 0 then it outputs 0, calls h, outputs 0, and calls h again. If the bit from the network is 1 then it outputs 1 twice.

Notice how we are describing computational effects with an algebraic signature: inp is a binary operation, and out_0, out_1 are unary operations. A crucial observation is that the same normalization algorithm works if we begin with a different algebraic signature of computational effects. Many other effects have been described in an algebraic way, including non-determinism, probability, memory access [36,35,26] and logic programming [43]. Our framework is a general one for all these examples.

1.1 The essence of normalization by evaluation

We define our normalization algorithm in §3 using the paradigm of normalization by evaluation (NBE). The ideas of NBE were first discussed by Martin-Löf [24] and later developed by Berger and Schwichtenberg [8].



There are two key ingredients: (1) a denotational semantics of the programming language in an executable type theory (Agda[3]) in which terms are automatically normalized; (2) a "reification" function which takes inhabitants of the denotational semantics back to terms of the intermediate language in a sub-grammar of normal forms.

1.2 Components of our denotational semantics

There are three important components to our denotational semantics for NBE:

1. Semantics in a functor category: We follow the general paradigm of structuring denotational semantics by finding a category and interpreting types as objects and programs as morphisms between objects. Following [14,3,9], we base our denotational semantics on the category $\mathsf{Set}^{\mathsf{Ren}}$ of functors from a category Ren of contexts and renamings between them, to the category of sets. This category behaves very much like the category of sets, but has extra features that allow us to take care over interpreting terms with free identifiers. The key feature of $\mathsf{Set}^{\mathsf{Ren}}$ is that there is a distinguished object $\mathsf{Ren}(\tau, -)$ for each type τ of the intermediate language, and this object behaves like a special set of identifiers of type τ.

2. A residualizing monad: Our intermediate language is a variation on Moggi's monadic metalanguage, and we structure our denotational semantics using a monad. Following Plotkin and Power [35], we build the monad from operations in the algebraic signature describing the computational effects. However, for NBE we must add more into our monad: following Filinski's pioneering work [13] and subsequent developments [21,5], we also incorporate the effect of applying an identifier of function type to an argument. For instance, in the normal form (‡) above, although the result of the call to h is ignored, the function call may produce side effects, depending on what h stands for. We thus keep the 'residual' function call, which cannot be normalized any further.

3. Using PERs to account for equations on effect terms: In addition to operations in algebraic signatures, many computational effects are described with additional equations specifying their computational behaviour. Following [9,33], we accommodate such effects in our NBE algorithm by considering presheaves whose codomains are equipped with partial equivalence relations (PERs). This is a particularly elegant approach because from the perspective of the NBE algorithm, we can naively work with sets, and then refer to the PERs when justifying the correctness of the algorithm.

1.3 Contributions

Our main contribution is to build a normalization algorithm for our effectful functional language out of this semantic analysis. The three components of our denotational semantics (§1.2) have not been combined before. By combining (1) and (2) we

[3] Agda implementation of our NBE: https://github.com/danelahman/Normalization-By-Evaluation

achieve a clean and modular mathematical account of Filinski's ideas of residuation in monads. By combining (2) and (3) we are able to analyze equations and normalization at the level of effects (§5), separately from equations and normalization of the functional aspects of the language.

We also present a proof of correctness of the normalization algorithm. Our proof uses logical relations, and further exploits the tight connection between the residualizing monad and the syntax of normal forms.

Acknowledgments

We are especially grateful to James Chapman for suggestions on the Agda formalization. We also thank Andrej Bauer, Sam Lindley, Andy Pitts and Phil Scott for useful discussions. We also thank Pierre Clairambault who pointed us to Okada and Scott's undecidability result. The core of this work appeared in the first author's MPhil dissertation.

First author's participation at the conference was supported by Estonian national scholarship program Kristjan Jaak, which is funded and managed by Archimedes Foundation in collaboration with the Ministry of Education and Research.

2 A programming language with algebraic effects

We introduce the syntax and equational theory for a higher-order programming language which incorporates computational effects using algebraic theories, following [35]. Our language is based on the call-by-value paradigm. The evaluation order is totally explicit, so it is more of an intermediate language than a front-end. The language is based on Moggi's monadic metalanguage [29], following the analysis by Levy, Power and Thielecke [20] (see also [17,19,28,40]).

2.1 Algebraic effects

We describe simple effects involved in computation using algebraic signatures [36]. For example, we can describe the effects involved in input/output of bits over a fixed communication channel with a binary operation inp and unary operations out_0, out_1. The algebraic expression $\mathsf{inp}[M, N]$ describes a computation that first reads a bit from the channel and then proceeds as the computation M if it is 0, or as N if it is 1. The expression $\mathsf{out}_0[M]$ describes a computation that outputs a bit 0 to the channel and then proceeds as M.

For another example, we can describe the effects of non-determinism with a binary operation $\oplus$, with the understanding that $M \oplus N$ describes a computation that behaves either as M or as N.

Formally, an *algebraic signature* consists of a set Op of operations together with an assignment of arities $\mathrm{ar} : \mathsf{Op} \to \mathbb{N}$. For input/output, let $\mathsf{Op} \stackrel{\text{def}}{=} \{\mathsf{inp}, \mathsf{out}_0, \mathsf{out}_1\}$

and $\mathrm{ar}(\mathsf{inp}) \stackrel{\text{def}}{=} 2$, $\mathrm{ar}(\mathsf{out}_0) \stackrel{\text{def}}{=} 1$, $\mathrm{ar}(\mathsf{out}_1) \stackrel{\text{def}}{=} 1$. For non-determinism, let $\mathsf{Op} \stackrel{\text{def}}{=} \{\oplus\}$ and $\mathrm{ar}(\oplus) \stackrel{\text{def}}{=} 2$.

One would typically impose equations, such as idempotency, commutativity and associativity of $\oplus$. We postpone a discussion on this until §5. In §6 we discuss more general kinds of algebraic theories involving value parameters and variable binding.

2.2 *Extending algebraic effects to a call-by-value language with higher types*

The algebraic analysis of effects involves a class of computations of unspecified type. We now describe a typed language, for time being with product and function types:

$$\sigma, \tau \in \mathsf{Ty} \ ::= \ \mathtt{unit} \mid \sigma * \tau \mid \sigma \rightharpoonup \tau \ .$$

We use a harpoon symbol for the function type $\sigma \rightharpoonup \tau$ to emphasise that a function may have side effects. (Moggi's [29] notation for this is $\sigma \to T(\tau)$. Conversely in our language the thunking construction $(\mathtt{unit} \rightharpoonup (-))$ is a monad.)

We have not included other types, such as sums or recursive types, because our main aim in this paper is to present a clear underlying framework for NBE for effectful languages with algebraic effects. We return to this in §6.

A typing context is a list of types annotated with variable names x, y, z. We have no need to consider untyped terms, so we immediately provide a rule-based definition of typed terms in context. Following [20], there are two typing judgements: one for values $\Gamma \vdash_{\mathsf{v}} V : \tau$ and one for producers $\Gamma \vdash_{\mathsf{p}} M : \tau$. The idea is that a value is something that has no effects, whereas a producer may have side effects.

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash_{\mathsf{v}} x : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V_1 : \tau_1 \quad \Gamma \vdash_{\mathsf{v}} V_2 : \tau_2}{\Gamma \vdash_{\mathsf{v}} \langle V_1, V_2 \rangle : \tau_1 * \tau_2} \qquad \frac{\Gamma, x : \sigma \vdash_{\mathsf{p}} N : \tau}{\Gamma \vdash_{\mathsf{v}} \mathtt{fn}\, x{:}\sigma \Rightarrow N : \sigma \rightharpoonup \tau}$$

$$\frac{}{\Gamma \vdash_{\mathsf{v}} \langle\rangle : \mathtt{unit}} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V : \tau_1 * \tau_2}{\Gamma \vdash_{\mathsf{v}} \#_i\, V : \tau_i} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V : \sigma \rightharpoonup \tau \quad \Gamma \vdash_{\mathsf{v}} W : \sigma}{\Gamma \vdash_{\mathsf{p}} V\, W : \tau}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V : \tau}{\Gamma \vdash_{\mathsf{p}} \mathtt{return}\, V : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M : \sigma \quad \Gamma, x : \sigma \vdash_{\mathsf{p}} N : \tau}{\Gamma \vdash_{\mathsf{p}} M\, \mathtt{to}\, x.\, N : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M_1 : \tau \ \ldots \ \Gamma \vdash_{\mathsf{p}} M_n : \tau}{\Gamma \vdash_{\mathsf{p}} \mathsf{op}_\tau[M_1, \ldots, M_n] : \tau}$$

There is an instance of the bottom-right rule for each n-ary operation $\mathsf{op} \in \mathsf{Op}$ and each type τ. For instance, with the input/output signature we have this syntax:

$$\frac{\Gamma \vdash_{\mathsf{p}} M : \tau \quad \Gamma \vdash_{\mathsf{p}} M : \tau}{\Gamma \vdash_{\mathsf{p}} \mathsf{inp}[M, N] : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M : \tau}{\Gamma \vdash_{\mathsf{p}} \mathsf{out}_0[M] : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{p}} M : \tau}{\Gamma \vdash_{\mathsf{p}} \mathsf{out}_1[M] : \tau}$$

2.3 *Equational theory*

The equational theory of this language is built from the $\beta\eta$-equations of the λ-calculus, the laws of Kleisli composition (e.g. [20,29]), and algebraicity [40, §3.3]. We

have elided the usual laws of reflexivity, symmetry, transitivity, and congruence.

$$\frac{\Gamma \vDash_{\mathsf{v}} V_1 : \tau_1 \quad \Gamma \vDash_{\mathsf{v}} V_2 : \tau_2}{\Gamma \vDash_{\mathsf{v}} \mathtt{\#}_i \langle V_1, V_2 \rangle \equiv V_i : \tau_i} \qquad \frac{\Gamma \vDash_{\mathsf{v}} V : \tau_1 * \tau_2}{\Gamma \vDash_{\mathsf{v}} V \equiv \langle \mathtt{\#}_1 V, \mathtt{\#}_2 V \rangle : \tau_1 * \tau_2} \qquad \frac{\Gamma \vDash_{\mathsf{v}} V : \mathtt{unit}}{\Gamma \vDash_{\mathsf{v}} V \equiv \langle\rangle : \mathtt{unit}}$$

$$\frac{\Gamma, x : \sigma \vDash_{\mathsf{p}} M : \tau \quad \Gamma \vDash_{\mathsf{v}} V : \sigma}{\Gamma \vDash_{\mathsf{p}} (\mathtt{fn}\, x{:}\sigma \Rightarrow M)\, V \equiv M[V/x] : \tau} \qquad \frac{\Gamma \vDash_{\mathsf{v}} V : \sigma \rightharpoonup \tau}{\Gamma \vDash_{\mathsf{v}} V \equiv \mathtt{fn}\, x{:}\sigma \Rightarrow (V\, x) : \sigma \rightharpoonup \tau}$$

$$\frac{\Gamma \vDash_{\mathsf{v}} V : \sigma \quad \Gamma, x : \sigma \vDash_{\mathsf{p}} N : \tau}{\Gamma \vDash_{\mathsf{p}} \mathtt{return}\, V\, \mathtt{to}\, x.\, N \equiv N[V/x] : \tau} \qquad \frac{\Gamma \vDash_{\mathsf{p}} M : \tau}{\Gamma \vDash_{\mathsf{p}} M \equiv M\, \mathtt{to}\, x.\, \mathtt{return}\, x : \tau}$$

$$\frac{\Gamma \vDash_{\mathsf{p}} M : \sigma \quad \Gamma, x : \sigma \vDash_{\mathsf{p}} N : \tau \quad \Gamma, y : \tau \vDash_{\mathsf{p}} P : \rho}{\Gamma \vDash_{\mathsf{p}} (M\, \mathtt{to}\, x.\, N)\, \mathtt{to}\, y.\, P \equiv M\, \mathtt{to}\, x.\, (N\, \mathtt{to}\, y.\, P) : \rho}$$

$$\frac{\Gamma \vDash_{\mathsf{p}} M_1 : \sigma \ldots \Gamma \vDash_{\mathsf{p}} M_n : \sigma \quad \Gamma, x : \sigma \vDash_{\mathsf{p}} N : \tau}{\Gamma \vDash_{\mathsf{p}} \mathsf{op}_\sigma[M_1, \ldots, M_n]\, \mathtt{to}\, x.\, N \equiv \mathsf{op}_\tau[M_1\, \mathtt{to}\, x.\, N, \ldots, M_n\, \mathtt{to}\, x.\, N] : \tau}$$

2.4 Denotational semantics

We now recall the general programme of denotational semantics for the language in §2.2–2.3 in a category with sufficient structure [29,20,35]. Given an algebraic signature Op, a *monad model* is given by a category C with following data:

- a chosen cartesian closed structure, i.e. chosen finite products (including a terminal object 1), and for all objects A and B an object $[A \Rightarrow B]$ together with an evaluation morphism $\varepsilon : [A \Rightarrow B] \times A \to B$ such that for every $f : C \times A \to B$ there is a unique morphism $\lambda f : C \to [A \Rightarrow B]$ such that $f = \varepsilon \circ (\lambda f \times \mathsf{id}_A)$.
- a strong monad T on C, i.e. for each object A an object TA, and a morphism $\eta : A \to TA$, and for all objects A and B a morphism $\mathsf{str} : A \times TB \to T(A \times B)$, and for each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ (also called the Kleisli extension of f), satisfying appropriate conditions (e.g. [29]).
- for each operation $\mathsf{op} \in \mathsf{Op}$ with $\mathrm{ar}(\mathsf{op}) = n$, a natural transformation $\mathsf{T}\text{-}\mathsf{op} : T(-)^n \to T(-)$ between functors $\mathsf{C} \to \mathsf{C}$.

We interpret the intermediate language in a monad model by interpreting types as objects and terms as morphisms. The interpretation of types as objects proceeds as follows: $[\![\mathtt{unit}]\!] \stackrel{\text{def}}{=} 1$, $[\![\tau_1 * \tau_2]\!] \stackrel{\text{def}}{=} [\![\tau_1]\!] \times [\![\tau_2]\!]$, $[\![\sigma \rightharpoonup \tau]\!] \stackrel{\text{def}}{=} [[\![\sigma]\!] \Rightarrow T[\![\tau]\!]]$. We interpret a context $(x_1 : \tau_1, \ldots, x_n : \tau_n)$ as an object too, as the product of the interpretations of its consituent types: $[\![(x_1 : \tau_1, \ldots, x_n : \tau_n)]\!] \stackrel{\text{def}}{=} [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. That is, a context is interpreted as the object of environments for that context.

Value typing judgments $\Gamma \vDash_{\mathsf{v}} V : \tau$ are interpreted as morphisms $[\![V]\!]_{\mathsf{v}} : [\![\Gamma]\!] \to [\![\tau]\!]$, and producer typing judgments $\Gamma \vDash_{\mathsf{p}} M : \tau$ as morphisms $[\![M]\!]_{\mathsf{p}} : [\![\Gamma]\!] \longrightarrow T[\![\tau]\!]$.

These morphisms are defined by induction on the structure of derivations:

$$\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \pi_x & [\![\mathtt{fn}\, x{:}\sigma \Rightarrow N]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \lambda [\![N]\!]_{\mathsf{p}} \\
[\![\#_1\, V]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \pi_1 \circ [\![V]\!]_{\mathsf{v}} & [\![V\, W]\!]_{\mathsf{p}} & \stackrel{\text{def}}{=} & \varepsilon \circ \langle [\![V]\!]_{\mathsf{v}}, [\![W]\!]_{\mathsf{v}} \rangle \\
[\![\#_2\, V]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \pi_2 \circ [\![V]\!]_{\mathsf{v}} & [\![\mathtt{return}\, V]\!]_{\mathsf{p}} & \stackrel{\text{def}}{=} & \eta \circ [\![V]\!]_{\mathsf{v}} \\
[\![\langle V, W \rangle]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \langle [\![V]\!]_{\mathsf{v}}, [\![W]\!]_{\mathsf{v}} \rangle & [\![M\, \mathtt{to}\, x.\, N]\!]_{\mathsf{p}} & \stackrel{\text{def}}{=} & [\![N]\!]^*_{\mathsf{p}} \circ \mathsf{str} \circ \langle \mathsf{id}, [\![M]\!]_{\mathsf{p}} \rangle \\
[\![\langle\rangle]\!]_{\mathsf{v}} & \stackrel{\text{def}}{=} & \langle\rangle & [\![\mathsf{op}_\tau[M_1, \ldots, M_n]]\!]_{\mathsf{p}} & \stackrel{\text{def}}{=} & \mathsf{T}\text{-}\mathsf{op} \circ \langle [\![M_1]\!]_{\mathsf{p}}, \ldots, [\![M_n]\!]_{\mathsf{p}} \rangle
\end{array}$$

Proposition 2.1 (Soundness) *In any monad model:*
If $\Gamma \models_{\mathsf{v}} V \equiv W : \tau$ *then* $[\![V]\!]_{\mathsf{v}} = [\![W]\!]_{\mathsf{v}}$. *If* $\Gamma \models_{\mathsf{p}} M \equiv N : \tau$ *then* $[\![M]\!]_{\mathsf{p}} = [\![N]\!]_{\mathsf{p}}$.

For a simple example of a monad model, let C be the category Set of sets and functions between them. We can associate to any set A the least set $T(A)$ containing A and closed under the operations in Op. This yields a strong monad. The Eilenberg-Moore algebras for this monad can be understood as sets A that are equipped with a function $A^n \to A$ for each n-ary operation $\mathsf{op} \in \mathsf{Op}$. Unfortunately this set-theoretic model is not good enough for NBE, informally, because it does not support reification at higher types. We build a model suitable for NBE in §3.2.

3 Normalization by evaluation

The general programme of NBE proceeds in three steps, following Section 1.1: identifying normal forms (§3.1), building a model that supports a denotational semantics (§3.2), and defining a reification from the model to the normal forms (§3.3).

3.1 Normal forms

The normal forms for our language are based on the η-long β-normal forms of simply typed lambda calculus. We mutually define judgements of normal values ($\Vdash^{\mathsf{n}}_{\mathsf{v}}$), normal producers ($\Vdash^{\mathsf{n}}_{\mathsf{p}}$), atomic values ($\Vdash^{\mathsf{a}}_{\mathsf{v}}$) and atomic producers ($\Vdash^{\mathsf{a}}_{\mathsf{p}}$).

$$\frac{}{\Gamma, x : \tau, \Gamma' \Vdash^{\mathsf{a}}_{\mathsf{v}} x : \tau} \qquad \frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} V_1 : \tau_1 \quad \Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} V_2 : \tau_2}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} \langle V_1, V_2 \rangle : \tau_1 * \tau_2} \qquad \frac{\Gamma, x : \sigma \Vdash^{\mathsf{n}}_{\mathsf{p}} N : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} \mathtt{fn}\, x{:}\sigma \Rightarrow N : \sigma \rightharpoonup \tau}$$

$$\frac{}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} \langle\rangle : \mathtt{unit}} \qquad \frac{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{v}} V : \tau_1 * \tau_2}{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{v}} \#_i\, V : \tau_i} \qquad \frac{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{v}} V : \sigma \rightharpoonup \tau \quad \Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} W : \sigma}{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{p}} V\, W : \tau}$$

$$\frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} V : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} \mathtt{return}\, V : \tau} \qquad \frac{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{p}} M : \sigma \quad \Gamma, x : \sigma \Vdash^{\mathsf{n}}_{\mathsf{p}} N : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M\, \mathtt{to}\, x.\, N : \tau} \qquad \frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M_1 : \tau \ \ldots \ \Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M_n : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} \mathsf{op}_\tau[M_1, \ldots, M_n] : \tau}$$

The atomic judgements are an auxiliary notion that we use to define normal judgements. Informally, atomic judgements are built from destructors (projections, function application) and normal judgements are built from constructors (pairing, abstraction). The only thing that can be done with an atomic producer is to force

its execution and substitute the result, using `to`. Atomic values can be substituted for variables without denormalizing a term.

3.2 A model of set theory with identifiers

Our NBE algorithm works over programs with free variables, that is, open programs. To accommodate this, we build a model of set theory in which there is a 'set of identifiers' for each type. We build the model categorically, using the presheaf construction, following [3,9,14]. (Nominal sets [32] are also related from a semantic perspective.)

A category of contexts and renamings

Let Ren be the category whose objects are contexts of our language: lists of types, informally annotated with variables. A morphism $(\sigma_1, \ldots, \sigma_m) \longrightarrow (\tau_1, \ldots, \tau_n)$ is given by a function $f : m \to n$ such that $\sigma_i = \tau_{f(i)}$ for $1 \leq i \leq m$. Composition of morphisms is composition of functions.

A category of presheaves

We will consider the category $\mathsf{Set}^{\mathsf{Ren}}$ of (covariant) presheaves. The objects are functors $\mathsf{Ren} \to \mathsf{Set}$, and the morphisms are natural transformations. We understand a functor $F : \mathsf{Ren} \to \mathsf{Set}$ as assigning to each context a set which may depend on the free variables in that context. The functorial action on morphisms accounts for renamings of variables.

A helpful perspective is to think of this category as a model of intuitionistic set theory (e.g. [23]). For any type τ there is a representable presheaf $\mathsf{Ren}(\tau, -)$ which may be thought of as a 'set of identifiers' labelled with the type τ. These identifiers are pure: they cannot be manipulated or compared.

The category $\mathsf{Set}^{\mathsf{Ren}}$ has products, sums and function spaces (e.g. [23, §III.6]).

- *products:* for presheaves $F_1, \ldots, F_n$ we let $(F_1 \times \cdots \times F_n)(\Gamma) = F_1(\Gamma) \times \cdots \times F_n(\Gamma)$.
- *coproducts:* let $(F_1 + \cdots + F_n)(\Gamma) = F_1(\Gamma) \uplus \cdots \uplus F_n(\Gamma)$.
- *cartesian closure:* for $F, G \in \mathsf{Set}^{\mathsf{Ren}}$, let $[F \Rightarrow G](\Gamma) = \mathsf{Set}^{\mathsf{Ren}}(\mathsf{Ren}(\Gamma, -) \times F, G)$.

Syntactic presheaves

For any type τ we have six presheaves $\mathsf{Ren} \to \mathsf{Set}$ built from the syntactic constructions in §2.2 and §3.1: presheaves of values (VTerms_τ), producers (PTerms_τ), normal values ($\mathsf{NVTerms}_\tau$), atomic values ($\mathsf{AVTerms}_\tau$), normal producers ($\mathsf{NPTerms}_\tau$) and atomic producers ($\mathsf{APTerms}_\tau$). For example, $\mathsf{VTerms}_\tau(\Gamma) \stackrel{\text{def}}{=} \{V \mid \Gamma \vdash_{\mathsf{v}} V : \tau\}$. Presheaf actions are given by variable renaming: we let $\mathsf{VTerms}_\tau(f)(V) = V[f]$.

A residualizing monad

The crux of our semantic analysis is our residualizing monad T on the presheaf category $\mathsf{Set}^{\mathsf{Ren}}$. We begin with an abstract description of it, and follow with a concrete inductive definition.

We briefly define a *residualizing algebra* to be a presheaf $F : \mathsf{Ren} \to \mathsf{Set}$ together with a natural transformation $F^n \to F$ for each n-ary operation in the signature Op, and also a natural transformation $\mathsf{APTerms}_\tau \times ([\mathsf{Ren}(\tau, -) \Rightarrow F]) \to F$ for each type τ. The algebraic structure from the signature interprets the effects in the signature, and the additional structure describes sequencing of effects with atomic producers. Recall that atomic producers are function calls involving free identifiers; their effects are undetermined. With suitably defined algebra homomorphisms, we arrive at a category which is monadic over $\mathsf{Set}^{\mathsf{Ren}}$. That is, the category of residualizing algebras is the category of Eilenberg-Moore algebras for a strong monad T on the category $\mathsf{Set}^{\mathsf{Ren}}$. (This follows from the 'crude monadicity theorem'.)

The monad T has the following concrete inductive description. Let $F : \mathsf{Ren} \to \mathsf{Set}$ be a presheaf. We define a new presheaf $TF : \mathsf{Ren} \to \mathsf{Set}$ so that the sets $TF(\Gamma)$ are the least satisfying the following rules:

$$\frac{d \in F(\Gamma)}{(\mathsf{T\text{-}return}\, d) \in TF(\Gamma)} \qquad \frac{\Gamma \mid\!\!\stackrel{\mathsf{a}}{\mathsf{p}} M : \sigma \quad d \in TF(\Gamma, x{:}\sigma)}{(M\ \mathsf{T\text{-}to}\, x.\, d)\ \in TF(\Gamma)} \qquad \frac{d_1 \in TF(\Gamma)\ \ldots\ d_n \in TF(\Gamma)}{\mathsf{T\text{-}op}(d_1, \ldots, d_n)\ \in TF(\Gamma)}$$

The functorial action uses the action of F and the renaming of atomic producers. Note the tight correspondence between the residualizing monad and normal producers (§3.1): there is a natural isomorphism $\mathsf{NPTerms}_\tau \cong T(\mathsf{NVTerms}_\tau)$ (see also [21]). Another way to understand this monad is as the coproduct of the free monad generated by the algebraic signature Op and the free monad generated by $\mathsf{T\text{-}to}$ and $\mathsf{T\text{-}return}$, as described by Ghani, Uustalu, Adámek and others [1,15].

Proposition 3.1 *The category* $\mathsf{Set}^{\mathsf{Ren}}$ *together with the residualizing monad* T *forms a monad model in the sense of* §2.4.

3.3 Reification and reflection

Recall that a NBE algorithm has two components: denotational semantics into the model, and reification back to normal forms.

We define reification as two families of natural transformations: ${}^{\mathsf{v}}{\downarrow}^{\tau \in \mathsf{Ty}} : [\![\tau]\!] \to \mathsf{NVTerms}_\tau$ and ${}^{\mathsf{p}}{\downarrow}^{\tau \in \mathsf{Ty}} : T[\![\tau]\!] \to \mathsf{NPTerms}_\tau$. To account for the contravariance at function types, the reification functions must be defined mutually with reflection functions, ${}^{\mathsf{v}}{\uparrow}^{\tau \in \mathsf{Ty}} : \mathsf{AVTerms}_\tau \to [\![\tau]\!]$ and ${}^{\mathsf{p}}{\uparrow}^{\tau \in \mathsf{Ty}} : \mathsf{APTerms}_\tau \to T[\![\tau]\!]$.

- ${}^{\mathsf{v}}{\downarrow}^{\tau} : [\![\tau]\!] \to \mathsf{NVTerms}_\tau$ is defined by induction on the structure of types τ:

$$\begin{aligned}
{}^{\mathsf{v}}{\downarrow}^{\mathtt{unit}}_{\Gamma}\ d &\stackrel{\text{def}}{=} \langle\rangle \\
{}^{\mathsf{v}}{\downarrow}^{\tau_1 * \tau_2}_{\Gamma}\ d &\stackrel{\text{def}}{=} \langle {}^{\mathsf{v}}{\downarrow}^{\tau_1}_{\Gamma}\ (\pi_1\, d), {}^{\mathsf{v}}{\downarrow}^{\tau_2}_{\Gamma}\ (\pi_2\, d) \rangle \\
{}^{\mathsf{v}}{\downarrow}^{\sigma \to \tau}_{\Gamma}\ d &\stackrel{\text{def}}{=} \mathtt{fn}\, x{:}\sigma \Rightarrow ({}^{\mathsf{p}}{\downarrow}^{\tau}_{\Gamma, x:\sigma}\ (\varepsilon\, \langle d, ({}^{\mathsf{v}}{\uparrow}^{\sigma}_{\Gamma, x:\sigma}\ x) \rangle))
\end{aligned}$$

- $\mathsf{p}{\downarrow}^{\tau}\colon T[\![\tau]\!] \to \mathsf{NPTerms}_{\tau}$ is defined by induction on the structure of $T[\![\tau]\!]$:

$$\begin{aligned}
\mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ (\mathsf{T\text{-}return}\, d) &\stackrel{\text{def}}{=} \mathtt{return}\, (\mathsf{v}{\downarrow}^{\tau}_{\Gamma}\ d) \\
\mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ (M\ \mathsf{T\text{-}to}\, x.\, d) &\stackrel{\text{def}}{=} M\ \mathtt{to}\, x.\, (\mathsf{p}{\downarrow}^{\tau}_{\Gamma,x:\sigma}\ d) \\
\mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ (\mathsf{T\text{-}op}(d_1, \ldots, d_n)) &\stackrel{\text{def}}{=} \mathsf{op}_{\tau}[\mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ d_1, \ldots, \mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ d_n]
\end{aligned}$$

  (Notice, $(\mathsf{p}{\downarrow}^{\tau})$ is derived from the natural isomorphism $\mathsf{NPTerms}_{\tau} \cong T(\mathsf{NVTerms}_{\tau})$.)
- $\mathsf{v}{\uparrow}^{\tau}\colon \mathsf{AVTerms}_{\tau} \to [\![\tau]\!]$ is defined by induction on types τ:

$$\begin{aligned}
\mathsf{v}{\uparrow}^{\mathtt{unit}}_{\Gamma}\ V &\stackrel{\text{def}}{=} \langle\rangle \\
\mathsf{v}{\uparrow}^{\tau_1 * \tau_2}_{\Gamma}\ V &\stackrel{\text{def}}{=} \langle \mathsf{v}{\uparrow}^{\tau_1}_{\Gamma}\ (\pi_1\, V), \mathsf{v}{\uparrow}^{\tau_2}_{\Gamma}\ (\pi_2\, V)\rangle \\
\mathsf{v}{\uparrow}^{\sigma \to \tau}_{\Gamma}\ V &\stackrel{\text{def}}{=} \lambda d.\, \mathsf{p}{\uparrow}^{\tau}_{\Gamma,x:\sigma}\ (V\ (\mathsf{v}{\downarrow}^{\sigma}_{\Gamma,x:\sigma}\ d))
\end{aligned}$$

- $\mathsf{p}{\uparrow}^{\tau \in \mathsf{Ty}}\colon \mathsf{APTerms}_{\tau} \to T[\![\tau]\!]$ is defined by $\mathsf{p}{\uparrow}^{\tau}_{\Gamma}\ M \stackrel{\text{def}}{=} M\ \mathsf{T\text{-}to}\, x.\, (\mathsf{T\text{-}return}\ (\mathsf{v}{\uparrow}^{\tau}_{\Gamma,x:\tau}\ x))$.

Since variables are atomic values, the reflection morphisms allow us to map from the object of identifiers $\mathsf{Ren}(\tau, -)$ into the semantic domain $[\![\tau]\!]$, via the composite $\mathsf{Ren}(\tau, -) \longrightarrow \mathsf{AVTerms}_{\tau} \xrightarrow{\mathsf{v}{\uparrow}^{\tau}} [\![\tau]\!]$.

3.4 Summary of the normalization algorithm

We now combine the denotational semantics with reification to build a normalization algorithm.

Any context $\Gamma = (x_1 : \tau_1 \ldots x_n : \tau_n)$ has an environment $\mathsf{id\text{-}env}_{\Gamma}$ (in the set $[\![\Gamma]\!]_{\Gamma}$) in which variables are interpreted as identifiers: let $\mathsf{id\text{-}env}_{\Gamma} \stackrel{\text{def}}{=} \langle \mathsf{v}{\uparrow}^{\tau_1}_{\Gamma}\ x_1, \ldots, \mathsf{v}{\uparrow}^{\tau_n}_{\Gamma}\ x_n\rangle$.

The normal form of a value judgement $\Gamma \vDash_{\mathsf{v}} V : \tau$ is found by reifying the interpretation $[\![V]\!]_{\mathsf{v}} : [\![\Gamma]\!] \to [\![\tau]\!]$ in the environment $\mathsf{id\text{-}env}_{\Gamma}$. Similarly the normal form of a producer judgement $\Gamma \vDash_{\mathsf{p}} M : \tau$ is found by reifying the interpretation $[\![M]\!]_{\mathsf{p}} : [\![\Gamma]\!] \to T[\![\tau]\!]$ in the environment $\mathsf{id\text{-}env}_{\Gamma}$:

$$\mathsf{nf}(V) \stackrel{\text{def}}{=} \mathsf{v}{\downarrow}^{\tau}_{\Gamma}\ ([\![V]\!]_{\mathsf{v}\Gamma}(\mathsf{id\text{-}env}_{\Gamma})) \qquad \mathsf{nf}(M) \stackrel{\text{def}}{=} \mathsf{p}{\downarrow}^{\tau}_{\Gamma}\ ([\![M]\!]_{\mathsf{p}\Gamma}(\mathsf{id\text{-}env}_{\Gamma}))$$

We establish correctness of this normalization algorithm in Theorem 4.1.

Our normalization algorithm is based on a purely semantic analysis. Another common method for normalization is based on exhaustively rewriting syntactic program terms to compute their normal forms. To perform rewriting, one considers the equations $\Gamma \vDash_{\mathsf{v}} V \equiv W : \tau$ and $\Gamma \vDash_{\mathsf{p}} M \equiv N : \tau$ as rewrite rules. Lindley and Stark [22] have studied normalization for Moggi's monadic metalanguage in this setting. They developed a $\top\top$-lifting based proof method by building on the strong normalization results for simply-typed lambda calculus based on reducibility candidates (see also [11]).

3.5 A note on implementation

The algorithm in this section reduces normalization for the programming language to evaluation in set theory. For this to be an effective procedure, we need to understand the 'category of sets' in a constructive way. We do this using Agda [30], an implementation of Martin-Löf's type theory [25]. The structure of our implementation and its correctness proofs closely follow the presentation in this paper.

4 Correctness of the algorithm

We now show that the normalization algorithm we defined in §3 is correct. Our proof has been formalized in Agda. Similarly to [14], the proof of correctness is divided into three main theorems.

Theorem 4.1

(i) Normalization respects equivalence.
If $\Gamma \vDash_{\mathsf{v}} V \equiv W : \tau$ *then* $\mathsf{nf}(V) = \mathsf{nf}(W)$. *If* $\Gamma \vDash_{\mathsf{p}} M \equiv N : \tau$ *then* $\mathsf{nf}(M) = \mathsf{nf}(N)$.

(ii) Normalization preserves normal forms.
If $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} V : \tau$ *then* $\mathsf{nf}(V) = V$. *If* $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M : \tau$ *then* $\mathsf{nf}(M) = M$.

(iii) Terms are equivalent to their normal forms.
If $\Gamma \vDash_{\mathsf{v}} V : \tau$ *then* $\Gamma \vDash_{\mathsf{v}} V \equiv \mathsf{nf}(V) : \tau$. *If* $\Gamma \vDash_{\mathsf{p}} M : \tau$ *then* $\Gamma \vDash_{\mathsf{p}} M \equiv \mathsf{nf}(M) : \tau$.

Item (i) follows immediately from soundness of semantics (Prop. 2.1 and 3.1). Item (ii) is proved by induction on the derivations of normal values/producers. In the remainder of this section we outline a proof of item (iii) using logical relations.

4.1 Relating values and producers with their denotations

We begin by defining Kripke logical relations between values/producers and their denotations: $\lhd^{\tau}_{\mathsf{v}\,\Gamma} \subseteq [\![\tau]\!](\Gamma) \times \mathsf{VTerms}_{\tau}(\Gamma)$ and $\lhd^{\tau}_{\mathsf{p}\,\Gamma} \subseteq (T[\![\tau]\!])(\Gamma) \times \mathsf{PTerms}_{\tau}(\Gamma)$. We define them by induction: $\lhd^{\tau}_{\mathsf{v}}$ on the structure of τ, $\lhd^{\tau}_{\mathsf{p}}$ on the structure of T.

$$
\begin{aligned}
d \lhd^{\mathtt{unit}}_{\mathsf{v}\,\Gamma} V &\overset{\text{def}}{\iff} \mathsf{true} \\
d \lhd^{\tau_1 * \tau_2}_{\mathsf{v}\,\Gamma} V &\overset{\text{def}}{\iff} (\pi_1\, d \lhd^{\tau_1}_{\mathsf{v}\,\Gamma} \#_1\, V) \wedge (\pi_2\, d \lhd^{\tau_2}_{\mathsf{v}\,\Gamma} \#_2\, V) \\
d \lhd^{\sigma \rightharpoonup \tau}_{\mathsf{v}\,\Gamma} V &\overset{\text{def}}{\iff} \forall f \in \mathsf{Ren}(\Gamma, \Gamma').\forall d', V'. \\
&\qquad\qquad d' \lhd^{\sigma}_{\mathsf{v}\,\Gamma} V' \implies \varepsilon\,([\![\sigma \rightharpoonup \tau]\!]_f\, d, d') \lhd^{\tau}_{\mathsf{p}\,\Gamma'} (V[f])\, V' \\
(\mathsf{T\text{-}return}\, d) \lhd^{\tau}_{\mathsf{p}\,\Gamma} M &\overset{\text{def}}{\iff} \exists V.\ \Gamma \vDash_{\mathsf{p}} M \equiv \mathtt{return}\, V : \tau \wedge d \lhd^{\tau}_{\mathsf{v}\,\Gamma} V \\
(N\ \mathsf{T\text{-}to}\, x.\, d) \lhd^{\tau}_{\mathsf{p}\,\Gamma} M &\overset{\text{def}}{\iff} \exists P.\ \Gamma \vDash_{\mathsf{p}} M \equiv N\, \mathtt{to}\, x.\, P : \tau \wedge d \lhd^{\tau}_{\mathsf{p}\,\Gamma, x:\sigma} P \\
(\mathsf{T\text{-}op}(d_1\ \ldots\ d_n)) \lhd^{\tau}_{\mathsf{p}\,\Gamma} M &\overset{\text{def}}{\iff} \exists M_1 \ldots M_n \in \mathsf{PTerms}_{\tau}(\Gamma). \\
&\qquad \Gamma \vDash_{\mathsf{p}} M \equiv \mathsf{op}_{\tau}[M_1, \ldots, M_n] : \tau \wedge d_1 \lhd^{\tau}_{\mathsf{p}\,\Gamma} M_1 \wedge \ldots \wedge d_n \lhd^{\tau}_{\mathsf{p}\,\Gamma} M_n
\end{aligned}
$$

Proposition 4.2 The logical relations are invariant under equivalence: *If* $d \lhd^{\tau}_{\mathsf{v}\Gamma} V$ *and* $\Gamma \vDash_{\mathsf{v}} V \equiv W : \tau$ *then* $d \lhd^{\tau}_{\mathsf{v}\Gamma} W$. *If* $d \lhd^{\tau}_{\mathsf{p}\Gamma} M$ *and* $\Gamma \vDash_{\mathsf{p}} M \equiv N : \tau$ *then* $d \lhd^{\tau}_{\mathsf{p}\Gamma} N$.

Proposition 4.3 The logical relations are subobjects in $\mathsf{Set}^{\mathsf{Ren}}$. *For* $f \in \mathsf{Ren}(\Gamma, \Gamma')$*:* *If* $d \lhd^{\tau}_{\mathsf{v}\Gamma} V$ *then* $[\![\tau]\!]_f(d) \lhd^{\tau}_{\mathsf{v}\Gamma'} V[f]$. *If* $d \lhd^{\tau}_{\mathsf{p}\Gamma} M$ *then* $(T[\![\tau]\!])_f(d) \lhd^{\tau}_{\mathsf{p}\Gamma'} M[f]$.

Proposition 4.4 The logical relations interact well with reification and reflection.

(i) *If* $d \lhd^{\tau}_{\mathsf{v}\Gamma} V$ *then* $\Gamma \vDash_{\mathsf{v}} ({}^{\mathsf{v}}{\downarrow}^{\tau}_{\Gamma}\, d) \equiv V : \tau$. *If* $d \lhd^{\tau}_{\mathsf{p}\Gamma} M$ *then* $\Gamma \vDash_{\mathsf{p}} ({}^{\mathsf{p}}{\downarrow}^{\tau}_{\Gamma}\, d) \equiv M : \tau$.

(ii) *If* $\Gamma \vDash^{\mathsf{a}}_{\mathsf{v}} V : \tau$ *then* $({}^{\mathsf{v}}{\uparrow}^{\tau}_{\Gamma}\, V) \lhd^{\tau}_{\mathsf{v}\Gamma} V$. *If* $\Gamma \vDash^{\mathsf{a}}_{\mathsf{p}} M : \tau$ *then* $({}^{\mathsf{p}}{\uparrow}^{\tau}_{\Gamma}\, M) \lhd^{\tau}_{\mathsf{p}\Gamma} M$.

We extend logical relations to environments and simultaneous substitutions. For any context $\Gamma = (x_1 : \tau_1, \ldots, x_n : \tau_n)$, we let $\mathsf{Sub}_\Gamma \stackrel{\text{def}}{=} \mathsf{VTerms}_{\tau_1} \times \cdots \times \mathsf{VTerms}_{\tau_n}$. An element of Sub_Γ determines the substitution of a term for each variable in Γ. Given a judgement $\Gamma \vDash_{\mathsf{v}} V : \tau$, let $V[-] : \mathsf{Sub}_\Gamma \to \mathsf{VTerms}_\tau$ be defined by substitution. Similarly, given a producer $\Gamma \vDash_{\mathsf{p}} M : \tau$, we define $M[-] : \mathsf{Sub}_\Gamma \to \mathsf{PTerms}_\tau$ by substitution. We now define $\lhd^{\Gamma} \subseteq [\![\Gamma]\!] \times \mathsf{Sub}_\Gamma$ as $e \lhd^{\Gamma}_{\mathsf{v}\Gamma'} \rho \stackrel{\text{def}}{\iff} \forall (x : \tau) \in \Gamma.\, (e\, x) \lhd^{\tau}_{\mathsf{v}\Gamma'} (\rho\, x)$.

Proposition 4.5 (Fundamental lemma of logical relations) *If* $\Gamma \vDash_{\mathsf{v}} V : \tau$ *and* $e \lhd^{\Gamma}_{\mathsf{v}\Gamma'} \rho$ *then* $([\![V]\!]_{\mathsf{v}}\, e) \lhd^{\tau}_{\mathsf{v}\Gamma'} V[\rho]$. *If* $\Gamma \vDash_{\mathsf{p}} M : \tau$ *and* $e \lhd^{\Gamma}_{\mathsf{v}\Gamma'} \rho$ *then* $([\![M]\!]_{\mathsf{p}}\, e) \lhd^{\tau}_{\mathsf{p}\Gamma'} M[\rho]$.

4.2 Proof of Theorem 4.1(iii)

We use the logical relations to show that terms are equivalent to their normal forms.

Suppose $\Gamma \vDash_{\mathsf{v}} V : \tau$. We will show that $\Gamma \vDash_{\mathsf{v}} V \equiv \mathsf{nf}(V) : \tau$. (Recall that $\mathsf{nf}(V) \stackrel{\text{def}}{=} {}^{\mathsf{v}}{\downarrow}^{\tau}_{\Gamma}\, ([\![V]\!]_{\mathsf{v}\Gamma}(\mathsf{id\text{-}env}_\Gamma))$.) Using Prop. 4.4(ii), we deduce that identity environments and substitutions are related by $\lhd^{\Gamma}_{\mathsf{v}\Gamma'}$. By Prop. 4.5, $([\![V]\!]_{\mathsf{v}}\, \mathsf{id\text{-}env}_\Gamma) \lhd^{\tau}_{\mathsf{v}\Gamma} V$. From Prop. 4.4(i) we conclude $\Gamma \vDash_{\mathsf{v}} V \equiv \mathsf{nf}(V) : \tau$, as required. The case for producers is similar.

5 Equations and effects

The normalization process described in the previous sections is with respect to the equations in §2.3. We now discuss how to accommodate equations between effect terms.

5.1 Equations on effects

For a first example, the signature for non-determinism ($\oplus$) is usually considered together with the semilattice equations $x \oplus x = x$, $x \oplus y = y \oplus x$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$. To capture this in our language, we extend the equality for producers ($\Gamma \vDash_{\mathsf{p}} M \equiv N : \tau$, §2.3) by including these three equations at each type τ:

$$\frac{\Gamma \vDash_{\mathsf{p}} M : \tau}{\Gamma \vDash_{\mathsf{p}} M \oplus M \equiv M : \tau} \qquad \frac{\Gamma \vDash_{\mathsf{p}} M : \tau \quad \Gamma \vDash_{\mathsf{p}} N : \tau}{\Gamma \vDash_{\mathsf{p}} M \oplus N \equiv N \oplus M : \tau} \qquad \frac{\Gamma \vDash_{\mathsf{p}} M : \tau \quad \Gamma \vDash_{\mathsf{p}} N : \tau \quad \Gamma \vDash_{\mathsf{p}} P : \tau}{\Gamma \vDash_{\mathsf{p}} M \oplus (N \oplus P) \equiv (M \oplus N) \oplus P : \tau}$$

We also define equivalence relations on normal forms using the following three rules together with reflexivity, symmetry, transitivity and congruence.

$$\frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M \oplus M \equiv M : \tau} \qquad \frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M : \tau \quad \Gamma \vdash_{\mathsf{p}} N : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M \oplus N \equiv N \oplus M : \tau} \qquad \frac{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M : \tau \quad \Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} N : \tau \quad \Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} P : \tau}{\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M \oplus (N \oplus P) \equiv (M \oplus N) \oplus P : \tau}$$

Our NBE algorithm (§3) respects these equations:

Theorem 5.1

(i) *If* $\Gamma \vdash_{\mathsf{v}} V \equiv W : \tau$ *then* $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} \mathsf{nf}(V) \equiv \mathsf{nf}(W) : \tau$.
If $\Gamma \vdash_{\mathsf{p}} V \equiv W : \tau$ *then* $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} \mathsf{nf}(M) \equiv \mathsf{nf}(N) : \tau$.

(ii) *If* $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{v}} V : \tau$ *then* $\mathsf{nf}(V) = V$. *If* $\Gamma \Vdash^{\mathsf{n}}_{\mathsf{p}} M : \tau$ *then* $\mathsf{nf}(M) = M$.

(iii) *If* $\Gamma \vdash_{\mathsf{v}} V : \tau$ *then* $\Gamma \vdash_{\mathsf{v}} V \equiv \mathsf{nf}(V) : \tau$. *If* $\Gamma \vdash_{\mathsf{p}} M : \tau$ *then* $\Gamma \vdash_{\mathsf{p}} M \equiv \mathsf{nf}(M) : \tau$.

Although we do not have to change the NBE algorithm to respect the equivalence relations, we have to refine the residualizing model to establish correctness (Theorem 5.1). From a semantic perspective, we change the notion of residualizing algebra (§3.2), requiring that a residualizing algebra satisfies the semilattice equations. This gives us a different residualizing monad, which is a quotient of the monad in §3.2, so that we have an isomorphism $(\mathsf{NPTerms}_\tau/_{\equiv}) \cong T(\mathsf{NVTerms}_\tau)$.

From the perspective of implementation, however, the types of Agda are intensional and they do not permit quotients by equivalence relations. To remedy this we revisit the semantic framework. We understand a 'set' as an Agda type equipped with a partial equivalence relation $\approx$ (PER: symmetric, transitive relation), following Cubric, Dybjer, Scott [9] and Pitts [33, §C.1]. For example, the type of functions $[X \Rightarrow Y]$ is equipped with the following PER: $f \approx_{X \to Y} g$ iff $\forall x, x' : X.\ x \approx_X x' \implies f(x) \approx_Y g(x')$. We are led to redo category theory in this setting, so that a 'hom-set' is actually a type equipped with a PER. For more details, see [9] or our Agda implementation.

There is nothing specific about semilattices in our analysis. In general, we accommodate equations on effects using the PER on the residualizing monad. Also importantly, the PERs are not visible in the constructions of the normalization algorithm. They only play a role in the formalization of the correctness argument (Theorem 5.1).

We mention in passing an alternative way to arrive at a suitable model to accommodate equations on effect terms: the setoid construction [7]. A setoid is a type equipped with an equivalence relation (that is also reflexive: $\forall x.\, x \approx x$). The setoid model has a different cartesian closed structure: the setoid of functions between given setoids X and Y is $\{f : X \to Y \mid x \approx_X x' \implies f(x) \approx_Y f(x')\}$. (This is roughly the same as the domain of the PER.) In a proof-relevant system like Agda, a setoid-based implementation of the normalization algorithm would be littered with proof witnesses for all inhabitants of function types. Although the setoid model is well behaved in many ways, the PER construction is better for our purposes because it yields an algorithm that is not complicated by proof obligations.

5.2 Normalization of effects

In the previous section we only identified normal forms up-to the equations on effect terms. In specific situations we can do better. For example, consider the signature for a one-bit memory cell: $\mathsf{Op} \stackrel{\text{def}}{=} \{\mathsf{lookup}, \mathsf{update}_0, \mathsf{update}_1\}$, $\mathrm{ar}(\mathsf{lookup}) \stackrel{\text{def}}{=} 2$, $\mathrm{ar}(\mathsf{update}_0) \stackrel{\text{def}}{=} 1$, $\mathrm{ar}(\mathsf{update}_1) \stackrel{\text{def}}{=} 1$, with the following equations [26,35]:

$$\begin{array}{ll} x = \mathsf{lookup}[\mathsf{update}_0[x], \mathsf{update}_1[x]] & \mathsf{update}_i[\mathsf{update}_j[x]] = \mathsf{update}_j[x] \\ \mathsf{update}_0[\mathsf{lookup}[x, y]] = \mathsf{update}_0[x] & \mathsf{update}_1[\mathsf{lookup}[x, y]] = \mathsf{update}_1[y] \end{array} \tag{1}$$

The idea is that $\mathsf{lookup}[M, N]$ is the program that reads the memory, continuing as M or N depending on the result, and $\mathsf{update}_i[M]$ writes i to the memory before continuing as M.

Rather than equipping the normal producers with a PER generated by these equations, we can instead represent effect terms directly in normal form, following Melliès [26]. We use an auxiliary judgement ($\Vdash^{n'}_{p}$).

$$\frac{\Gamma \Vdash^{n'}_{p} M : \tau \quad \Gamma \Vdash^{n'}_{p} N : \tau}{\Gamma \Vdash^{n}_{p} \mathsf{lookup}[M, N] : \tau} \qquad \frac{\Gamma \Vdash^{n'}_{p} M : \tau \quad \Gamma \Vdash^{n'}_{p} N : \tau}{\Gamma \Vdash^{n}_{p} \mathsf{lookup}[\mathsf{update}_1[M], N] : \tau} \qquad \frac{\Gamma \Vdash^{n'}_{p} M : \tau \quad \Gamma \Vdash^{n'}_{p} N : \tau}{\Gamma \Vdash^{n}_{p} \mathsf{lookup}[M, \mathsf{update}_0[N]] : \tau}$$

$$\frac{\Gamma \Vdash^{n'}_{p} M : \tau \quad \Gamma \Vdash^{n'}_{p} N : \tau}{\Gamma \Vdash^{n}_{p} \mathsf{lookup}[\mathsf{update}_1[M], \mathsf{update}_0[N]] : \tau} \qquad \frac{\Gamma \Vdash^{n}_{v} V : \tau}{\Gamma \Vdash^{n'}_{p} \mathtt{return}\, V : \tau} \qquad \frac{\Gamma \Vdash^{a}_{p} M : \sigma \quad \Gamma, x{:}\sigma \Vdash^{n}_{p} N : \tau}{\Gamma \Vdash^{n'}_{p} M \,\mathtt{to}\, x.\, N : \tau}$$

Recall that the residualizing monad is a coproduct of two monads. In the present case we can understand it as a coproduct of the residualizing monad for no effects (§3.2), and the one-bit-state monad $[\{0,1\} \Rightarrow (\{0,1\} \times (-))]$. Concretely, this coproduct of monads is the following least fixed point (following the definition in [16]):

$$TF \;=\; \mu G.\, \Big[\{0,1\} \Rightarrow \big(\{0,1\} \times \big(F \;+\; \textstyle\sum_\tau (\mathsf{APTerms}_\tau \times [\mathsf{Ren}(\tau, -) \Rightarrow G])\big)\big)\Big]$$

In this monad the quotient by the equations (1) is made in the type, and a PER is not needed. Categorically speaking, this monad is isomorphic to the monad with a nontrivial PER. Concretely, however, this tailored monad provides a NBE algorithm that not only normalizes higher types, but also partially evaluates the imperative commands as much as possible. For illustration, consider the program (†) in the introduction, but with inp/out replaced by lookup/update. Rather than the normal form (‡), our algorithm also normalizes the effects, minimizing the number of writes:

$$\begin{array}{l} \mathsf{lookup}[h\,\langle\rangle\,\mathtt{to}\,x.\,\mathsf{lookup}[h\,\langle\rangle\,\mathtt{to}\,y.\,\mathsf{lookup}[\mathtt{return}\,\langle\rangle, \mathtt{return}\,\langle\rangle], \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{update}_0[h\,\langle\rangle\,\mathtt{to}\,y.\,\mathsf{lookup}[\mathtt{return}\,\langle\rangle, \mathtt{return}\,\langle\rangle]]], \\ \qquad \mathtt{return}\,\langle\rangle]. \end{array}$$

6 Remarks on extensions to the language

In this paper we have considered a restricted language with just enough features to demonstrate our contributions. While language features such as recursion and sum types are very important, they can be dealt with by using standard techniques from the literature. We briefly summarize the main ideas.

Recursion

Our NBE algorithm is guaranteed to terminate, because it is written in Agda. Nonetheless, realistic programming languages have the potential for non-termination. This leads us to the long-established connections between partial evaluation and NBE [10,12]. Roughly speaking, in a language with recursion, each sub-expression should be annotated with its 'binding time', to explain which parts of the program should be normalized at compile time (since they are somehow assumed to terminate) and which should not be touched until run time. Dybjer and Filinski [12,13] outline how to accommodate this in a monadic metalanguage.

Sum types

Most practical programming languages have sum types. For instance, we might have a type bit of bits with two constants $(0, 1)$ and following typing rule with equations:

$$\frac{\Gamma \vDash_{\mathsf{v}} V : \mathsf{bit} \quad \Gamma \vDash_{\mathsf{p}} M : \tau \quad \Gamma \vDash_{\mathsf{p}} N : \tau}{\Gamma \vDash_{\mathsf{p}} \mathsf{if}\ V\ \mathsf{then}\ M\ \mathsf{else}\ N : \tau} \qquad \begin{array}{c} \mathsf{if}\ i\ \mathsf{then}\ M_0\ \mathsf{else}\ M_1 \equiv M_i \quad (i = 0, 1) \\ M \equiv \mathsf{if}\ x\ \mathsf{then}\ M[0/x]\ \mathsf{else}\ M[1/x] \end{array} \tag{2}$$

The semantic analysis based on presheaf categories has been extended to explain NBE with sum types for pure languages without computational effects [2,6]. Filinski [13] and Lindley [21] have discussed NBE for effectful languages with sums from a more pragmatic perspective. The languages they consider type `case` expressions as computations rather than as values, which allows them to use the residualizing monad to treat pattern-matching on atomic values.

Base types and local effects

Our residualizing monad is a monad on a presheaf category. Various authors use monads on presheaf categories to describe local effects and name generation, including local store [27,35,37], π-calculus [42], and logic programming [43]. The second author has recently developed a syntactic framework for these analyses, based on a generalized kind of algebraic theory [28,44], which can be accommodated in our semantic analysis (see also [27,37]). This framework allows us to move closer to the original source program in our introduction, as follows.

We can add to our grammar for types two abstract base types: a type chan of channels and a type bit of communication data. We can then modify our algebraic signature for input/output effects so that the operations take parameters from chan,

specifying which channel to use for communication, and the input operation incorporates variable binding. This kind of signature is 'algebraic' in that it determines a monad on a presheaf category [43]. For input/output, we have this concrete syntax.

$$\frac{\Gamma \vDash_{\mathsf{v}} V : \mathsf{chan} \quad \Gamma, x : \mathsf{bit} \vDash_{\mathsf{p}} M : \tau}{\Gamma \vDash_{\mathsf{p}} \mathsf{inp}[V, x.\, M] : \tau} \qquad \frac{\Gamma \vDash_{\mathsf{v}} V : \mathsf{chan} \quad \Gamma \vDash_{\mathsf{v}} W : \mathsf{bit} \quad \Gamma \vDash_{\mathsf{p}} M : \tau}{\Gamma \vDash_{\mathsf{p}} \mathsf{out}[V, W, M] : \tau}$$

To allow manipulation of the data we add constants 0, 1 of type bit and also an operation if_then_else_ to our algebraic signature. In this way the typing rule in (2) arises from the algebraic signature of effects, not as an extra language construction. The equations for if_then_else_ (2) can be understood as part of the algebraic theory of the effects [44, §VC]. This suggests a new route to dealing with sum types in NBE, purely by using algebraic effects. We are currently experimenting with different implementations of the residualizing monad for this theory. We hope to recover a standard NBE algorithm for booleans [4] by implementing the monad carefully.

Categorically, this line of work amounts to investigating the free closed Freyd category on a Freyd category, in the terminology of [20,39]. Recall [20,39] that a *Freyd category* comprises $(\mathsf{C}, \mathsf{K}, J)$, where C is a category with finite products, K is a symmetric premonoidal category [38], and J is an identity-on-objects functor $\mathsf{C} \to \mathsf{K}$ that strictly preserves symmetric premonoidal structure and whose image lies in the centre of K. A Freyd category is *closed* if for all objects A the functor $J(A \times -) : \mathsf{C} \to \mathsf{K}$ has a right adjoint. In particular, to give a closed Freyd category is to give a category with finite products and a strong monad T on it for which Kleisli exponentials $[A \Rightarrow TB]$ exist (cf. [39]).

In this sense our investigations are analogous to the investigations by Cubric et al. [9, §7] into decidability for the free cartesian closed category on a category with finite products. However, whereas the $\beta\eta$-theory for free cartesian closed categories is not necessarily decidable, our equational theory is more fine-grained, leading us to make the following conjecture:

> *Let* $(\mathsf{C} \to \mathsf{K})$ *be a Freyd category where* C *is a free category with products on a set of objects and the word problem for* K *is decidable. Then the word problem for the free closed Freyd category on* $(\mathsf{C} \to \mathsf{K})$ *is decidable.*

Handlers of algebraic effects

While algebraic effects give a general way for constructing impure computations, recent developments suggest that it is also profitable to desconstruct computational effects. These 'effect handlers' generalize the idea of exception handlers to all algebraic effects. (See e.g. [34,40,18].)

To keep things simple, we consider the signature with one unary effect, op. We can add effect handlers for op to our language with the following term formation rule.

$$\frac{\Gamma \vDash_{\mathsf{p}} M : \sigma \quad \Gamma, x : \tau \vDash_{\mathsf{v}} H_{\mathsf{op}} : \tau \quad \Gamma, x : \sigma \vDash_{\mathsf{v}} H_{\mathtt{return}} : \tau}{\Gamma \vDash_{\mathsf{v}} \mathtt{handle}\, M\, \mathtt{with}\, \{\mathsf{op}(x) \Rightarrow H_{\mathsf{op}} \mid \mathtt{return}\, (x) \Rightarrow H_{\mathtt{return}}\} : \tau}$$

For an intuition, let $\mathsf{op}(M)$ be a computation that first 'beeps' and then continues

as M. The expression $\mathtt{handle}\, M\, \mathtt{with}\, \{\mathsf{op}(x) \Rightarrow H_{\mathsf{op}} \mid \mathtt{return}\, (x) \Rightarrow H_{\mathtt{return}}\}$ then captures each of the beeps in M and replaces them with H_{op}. For instance, the expression

$$(\mathtt{handle}\, M\, \mathtt{with}\, \{\mathsf{op}(x) \Rightarrow \lambda\langle\rangle.\, \mathsf{op}(\mathsf{op}(x\langle\rangle)) \mid \mathtt{return}\, (x) \Rightarrow \lambda\langle\rangle.\, x\})\, \langle\rangle$$

replaces each 'beep' in M with two beeps.

Mathematically, handler expressions reify the idea that the type $(\langle\rangle \rightharpoonup \tau)$ is the free algebra on τ generated by the unary operation op. This intuition suggests the following equations: firstly, that the handlers are homomorphisms between unary algebras:

$$\begin{aligned}\Gamma \vDash_{\mathrm{v}} \mathtt{handle}\, (\mathtt{return}\, V)\, \mathtt{with}\, \{\mathsf{op}(x) \Rightarrow H_{\mathsf{op}} \mid \mathtt{return}\, (x) \Rightarrow H_{\mathtt{return}}\} \\ \equiv H_{\mathtt{return}}[V/x] : \tau\end{aligned}$$

$$\begin{aligned}\Gamma \vDash_{\mathrm{v}} \mathtt{handle}\, (\mathsf{op}(M))\, \mathtt{with}\, \{\mathsf{op}(x) \Rightarrow H_{\mathsf{op}} \mid \mathtt{return}\, (x) \Rightarrow H_{\mathtt{return}}\} \\ \equiv H_{\mathsf{op}}[(\mathtt{handle}\, M\, \mathtt{with}\, H)/x] : \tau\end{aligned}$$

and secondly, that the handlers provide unique mediating morphisms:

$$\Gamma, x : \mathtt{unit} \rightharpoonup \sigma \vDash_{\mathrm{v}} V[(\lambda\langle\rangle.\, \mathsf{op}[x\langle\rangle])/x] \equiv H_{\mathsf{op}}[(\lambda\langle\rangle.\, V)/y] : \tau$$

$$\begin{aligned}&\Gamma, x : \mathtt{unit} \rightharpoonup \sigma \vDash_{\mathrm{v}} \\ &\quad V \equiv \mathtt{handle}\, (x\, \langle\rangle)\, \mathtt{with}\, \{\mathsf{op}(y) \Rightarrow H_{\mathsf{op}} \mid \mathtt{return}\, (z) \Rightarrow V[(\lambda\langle\rangle.\, \mathtt{return}\, z)/x]\} : \tau\end{aligned}$$

However, we conjecture that this equational theory is undecidable. This conjecture is based on the observation that computations of type $\mathtt{unit}$ are essentially natural numbers (thinking of $\mathtt{return}\, \langle\rangle$ as zero and $\mathsf{op}(M)$ as the successor of M). Thus our system is close to Gödel's System T, in which equality is undecidable (assuming 'uniqueness of recursors': see [31]).

7 Summary

We have investigated normalization by evaluation for a language with higher types and computational effects. The effects are specified by an algebraic signature, so our algorithm works for any notion of computation that can be expressed this way.

A key contribution of our work is our clear and modular semantic analysis of normalization by evaluation. At the heart of our analysis is the *residualizing monad*.

- It is a monad on a presheaf category. Following Altenkirch, Cubric, Fiore and others [3,9,14], we use a presheaf category as an alternative to classical set theory because we need to normalize open terms. The presheaf category provides us with well-behaved 'sets of free identifiers', while supporting the standard approach to denotational semantics using cartesian closed categories.
- The monad is built in a principled and modular way, using the operations and

equations in the algebraic theory that describes the computational effects, following the ideas of Plotkin, Power and others [35,26].

- In addition to algebraic operations, the monad also incorporates additional algebraic structure describing residualizing function calls, following Filinski [13].

Our normalization algorithm is implemented in the dependently typed language Agda, and also proved correct in Agda. To run our algorithm, we can naively think of sets as Agda types, but in the correctness proof we more properly understand sets as Agda types equipped with PERs, following [9].

References

[1] Adámek, J., S. Milius, N. Bowler and P. B. Levy, *Coproducts of monads on set*, in: *Proc. LICS* (2012), pp. 45–54.

[2] Altenkirch, T., P. Dybjer, M. Hofmann and P. Scott, *Normalization by evaluation for typed lambda calculus with coproducts*, in: *LICS'01*, Washington, DC, USA, 2001, pp. 303–310.

[3] Altenkirch, T., M. Hofmann and T. Streicher, *Categorical reconstruction of a reduction free normalization proof*, in: *CTCS'95*, 1995, pp. 182–199.

[4] Altenkirch, T. and T. Uustalu, *Normalization by evaluation for* $\lambda^{\rightarrow,2}$, in: *Proc. FLOPS'04*, 2004, pp. 260–275.

[5] Atkey, R., *A type checker that knows its monad from its elbow* (2011), http://bentnib.org/posts/2011-12-14-type-checker.html.

[6] Balat, V., R. Di Cosmo and M. Fiore, *Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums*, in: *POPL'04* (2004), pp. 64–76.

[7] Barthe, G., V. Capretta and O. Pons, *Setoids in type theory*, J. Funct. Program. **13** (2003), pp. 261–293.

[8] Berger, U. and H. Schwichtenberg, *An inverse of the evaluation functional for typed* λ*–calculus*, in: *Proc. LICS'91*, 1991, pp. 203–211.

[9] Cubric, D., P. Dybjer and P. J. Scott, *Normalization and the Yoneda embedding*, Math. Struct. Comput. Sci. **8** (1998), pp. 153–192.

[10] Danvy, O., *Type-directed partial evaluation*, in: *Proc. Partial Evaluation*, 1998, pp. 367–411.

[11] Doczkal, C. and J. Schwinghammer, *Formalizing a strong normalization proof for Moggi's computational metalanguage: a case study in Isabelle/Hol-nominal*, in: *Proc. LFMTP'09* (2009), pp. 57–63.

[12] Dybjer, P. and A. Filinski, *Normalization and partial evaluation*, in: *Proc. APPSEM 2000* (2002), pp. 137–192.

[13] Filinski, A., *Normalization by evaluation for the computational lambda-calculus*, in: *Proc. TLCA'01*, 2001, pp. 151–165.

[14] Fiore, M., *Semantic analysis of normalisation by evaluation for typed lambda calculus*, in: *Proc. PPDP'02*, 2002, pp. 26–37.

[15] Ghani, N. and T. Uustalu, *Coproducts of ideal monads*, ITA **38** (2004), pp. 321–342.

[16] Hyland, M., G. Plotkin and J. Power, *Combining effects: sum and tensor*, Theor. Comput. Sci. **357** (2006), pp. 70–99.

[17] Johann, P., A. Simpson and J. Voigtländer, *A generic operational metatheory for algebraic effects*, in: *Proc. LICS 2010*, 2010, pp. 209–218.

[18] Kammar, O., S. Lindley and N. Oury, *Handlers in action*, in: *To appear in Proc. ICFP 2013*, 2013.

[19] Kammar, O. and G. D. Plotkin, *Algebraic foundations for effect-dependent optimisations*, in: *Proc. POPL 2012*, 2013, pp. 349–360.

[20] Levy, P. B., J. Power and H. Thielecke, *Modelling environments in call-by-value programming languages*, Information and Computation **185** (2003), pp. 182–210.

[21] Lindley, S., *Accumulating bindings*, in: O. Danvy, editor, *Informal proceedings of the 2009 Workshop on Normalization by Evaluation*, 2009, pp. 49–56.

[22] Lindley, S. and I. Stark, *Reducibility and ⊤⊤-lifting for computation types*, in: *Proc. TLCA'05* (2005), pp. 262–277.

[23] Mac Lane, S. and I. Moerdijk, "Sheaves in geometry and logic: A First Introduction to Topos Theory," Springer-Verlag, 1992.

[24] Martin-Löf, P., *About models for intuitionistic type theories and the notion of definitional equality*, in: S. Kanger, editor, *3rd Scandinavian Logic Symp.*, North-Holland, 1975 pp. 81–109.

[25] Martin-Löf, P., *An intuitionistic theory of types, predicative part*, in: H. E. Rose and J. C. Sheperdson, editors, *Logic Colloquium 1973* (1975), pp. 73–118.

[26] Melliès, P.-A., *Segal condition meets computational effects*, in: *Proc. LICS 2010*, 2010, pp. 150–159.

[27] Melliès, P.-A., *Local stores in string diagrams* (2011), http://tinyurl.com/mellies-itu-2011.

[28] Møgelberg, R. E. and S. Staton, *Linearly-used state in models of call-by-value*, in: *Proc. CALCO'11*, 2011, pp. 298–313.

[29] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[30] Norell, U., "Towards a Practical Programming Language Based on Dependent Type Theory," Ph.D. thesis, Chalmers University of Technology (2007).

[31] Okada, M. and P. Scott, *A note on rewriting theory for uniqueness of iteration*, Theory and Applications of Categories **6** (1999), pp. 47–64.

[32] Pitts, A. M., *Alpha-structural recursion and induction*, J. ACM **53** (2006), pp. 459–506.

[33] Pitts, A. M., *Structural recursion with locally scoped names*, J. Funct. Program. **21** (2011), pp. 235–286.

[34] Plotkin, G. and M. Pretnar, *Handlers of algebraic effects*, in: *Proc. ESOP 2009* (2009), pp. 80–94.

[35] Plotkin, G. D. and J. Power, *Notions of computation determine monads*, in: *Proc. FOSSACS'02* (2002), pp. 342–356.

[36] Plotkin, G. D. and J. Power, *Algebraic operations and generic effects*, Applied Categorical Structures **11** (2003), pp. 69–94.

[37] Power, J., *Indexed Lawvere theories for local state*, in: *Models, Logics and Higher-Dimensional Categories*, AMS, 2011 pp. 268–282.

[38] Power, J. and E. Robinson, *Premonoidal categories and notions of computation*, Math. Struct. in Comput. Sci. **7** (1997), pp. 453–468.

[39] Power, J. and H. Thielecke, *Closed Freyd- and κ-categories*, in: *Proc. ICALP'99*, Lecture Notes in Comput. Sci. **1644**, 1999, pp. 625–634.

[40] Pretnar, M., "The logic and handling of algebraic effects," Ph.D. thesis, University of Edinburgh (2010).

[41] Reppy, J. H., *Concurrent ML*, in: *Encyclopedia of Parallel Computing*, Springer, 2011 pp. 371–377.

[42] Stark, I., *Free-algebra models for the pi-calculus*, Theor. Comput. Sci. **390** (2008), pp. 248–270.

[43] Staton, S., *An algebraic presentation of predicate logic*, in: *Proc. FOSSACS 2013*, 2013, pp. 401–417.

[44] Staton, S., *Instances of computational effects: An algebraic perspective*, in: *Proc. LICS 2013*, 2013, pp. 519–519.