On the expressive power of effect handlers and monadic reflection

Yannick Forster
Robinson College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: yf272@cam.ac.uk

February 9, 2017

Abstract

Effect handlers and monadic reflection are both programming paradigms for implementing computational effects such as exceptions or I/O. In this thesis we compare the expressive power of effect handlers and monadic reflection. This comparison is based on two core calculi λ_{eff}, introduced by Kammar, Lindley and Oury and λ_{mon}, introduced by Filinski, both being extensions of Levy's call-by-push-value calculus.

We prove adequacy of the set-theoretic dentotational semantics of λ_{eff}. We give a finite, adequate set-theoretic semantics for λ_{mon}, define the notion of typed and untyped macro expressability following Felleisen and show that there is a macrox translation from λ_{mon} to untyped λ_{eff}, but no translation from λ_{eff} to typed λ_{mon}.

Contents

Chapter 1

Introduction

Computational effects such as exceptions, global state, or I/O are used in many functional programming languages to implement backtracking, multithreading, or nondeterminism. They are built in as features in general purpose languages like OCaml or Standard ML and can be used as monads for instance in Haskell. However, depending on the language used, it may be hard or impossible to declare new or change the implementation of old effects for the programmer. For instance, the change of a global memory cell usually persists the raising of an exception. To implement new concepts like software transactional memory, one would want to change this behaviour, such that the initial value of a memory cell is restored if an exception is raised.

This thesis compares the expressiveness of two calculi that model different approaches to user-defined computational effects.

The first one is the λ_{eff}-calculus by Kammar, Lindley, and Oury [9], a higher-order calculus of effect handlers [20, 1] based on Levy's CBPV [11]. Computational effects in λ_{eff} arise from the use of effect operations, like `raise` for exceptions, `set` and `get` for state, or `read` and `write` for I/O. Programmers can then define effect handlers to define the implementation of this effects. The clear separation between the effect and its handling allows for both abstraction and modularity. Pretnar and Bauer implement a programming language called *eff* whose treatment of effects uses handlers.

Monads are a widespread concept of abstraction in functional programming. General purpose languages like Haskell or OCaml allow user-defined effects using monads. In Haskell, every effect has to be encapsulated by a monad whereas in OCaml there are some built-in effects, like exceptions and reference-cells. We would like to study monads as a programming abstraction, without introducing auxiliary notions like type classes or modules, but as independent abstractions.

Thus, the second calculus we consider is the λ_{mon}-calculus, a variation of the calculus with monadic reflections introduced by Filinski [7]. In λ_{mon}, monadic effects can be defined by the programmer. For example, a computation of type A that may raise an error can be described by the well-known exception monad $A+1$ (`α option` in OCaml/`Maybe a` in Haskell). Monadic reflection comes with two syntactic constructs

$$\frac{M : 1 \xrightarrow{\text{err}} A}{\texttt{reify}_{\text{err}}(M) : A+1} \qquad\qquad \frac{N : A+1}{\texttt{reflect}_{\text{err}}(N) : 1 \xrightarrow{\text{err}} A}$$

witnessing the bijection between a pure implementation based on the $A+1$ monad and an effectful computation, here written as $1 \xrightarrow{\text{err}} A$.

The main contribution of the thesis is to compare the expressability of effect handlers and monadic reflection with each other. We adapt Felleisen's [4] notion and define the concept of typed and untyped macro expressability of λ_{mon} in λ_{eff} and vice versa.

One can compare two programming features by defining them on top of the same base calculus, which is what we do for λ_{eff} and λ_{mon}, both being based on CBPV. One extension can then macro express another if there is a structurally recursive translation function that is homomorphic on the common base fragment and preserves normal forms in the sense that if a term evaluates to a CBPV-value, then the translated term has to evaluate to the same value. Crucially, the translation has to replace the syntactical constructs which are not part of the base calculus by constructs from the target calculus — i.e. has to behave the same every time the construct is encountered. Typed macro expressability additionally adds the requirement that the translated term can be typed in the target calculus.

$$\begin{tikzcd}
 & \text{CBPV} \arrow[dl, "\text{typed}"'] \arrow[dr, "\text{typed}"] & \\
\lambda_{\text{eff}} \arrow[rr, dashed, shift left=0] & & \lambda_{\text{mon}} \arrow[ll, shift right=2, "\text{untyped}"'] \arrow[ll, dotted, shift left=2, "\text{typed}"]
\end{tikzcd}$$

$X \longrightarrow Y$: *There is a translation from* X *to* Y
$X \dashrightarrow Y$: *There is no translation*
$X \overset{\cdots}{\rightarrow} Y$: *We conjecture that there is no translation*

The diagram gives an overview of our results with respect to the existence of translations. There is an untyped translation from λ_{mon} to λ_{eff}. This is based on the idea that `reify` behaves like an effect handler for the effect operation `reflect`.

We show that there is no typed translation from λ_{eff} to λ_{mon}. The proof is based on the observation that the set-theoretic model for λ_{mon} is finite, whereas λ_{eff} has infinitely many oberservationally different terms of the same type.

Thus, we first recall a set-theoretic denotational semantics for λ_{eff} and prove its adequacy using Hermida's lifting [8]. We then introduce a finite set-theoretic denotational semantics for λ_{mon} and give an adequacy proof using $\top\top$-lifting [16, 15, 13, 10, 3]

Contribution

This thesis makes the following contributions:

- Adequacy proof for the set theoretic model of λ_{eff}
- Adequate denotational semantics for λ_{mon}
- Definition of macro (typed) expressability
- Proof that λ_{mon} is macro expressible in λ_{eff}
- Proof that λ_{eff} is not macro typed expressible in λ_{mon}

Structure

Chapter 2 introduces some preliminary concepts: monads, algebras for a monad, monad morphisms and Levy's CBPV Chapter 3 presents the syntax and operational semantics of λ_{eff} and an adequate denotational semantics. Chapter 4 presents syntax, operational and denotational semantics of λ_{mon} Chapter 5 introduces typed and untyped macro expressability and shows that λ_{eff} can untyped macro express λ_{mon} but λ_{mon} cannot typed macro express λ_{eff}. Chapter 6 concludes the thesis.

Chapter 2

Preliminaries

We begin by reviewing some basic category theoretic concepts specified to the category of sets and functions. We use the to give an adequate semantics to CBPV and use the definitions and proofs throughout the thesis.

2.1 Monads and algebras

Monads and algebras can be defined in a very general sense for arbitrary categories. However, for the sake of simplicity, it will suffice to introduce them for the category **Set**.

2.1.1 Monads

Monads are ubiquitous in functional programming languages. They are wired into the operational semantics of some programming languages (as the I/O monad in Haskell) or are at least definable as a structure (as in OCaml). We define what it means to be a monad in a more abstract sense:

Definition 2.1. *A monad is a triple* $\langle \mathsf{T}, \mathbf{return}, \gg\!= \rangle$ *where* T *is a class function* $\mathbf{Set} \to \mathbf{Set}$ *and* $\mathbf{return}^{\mathsf{X}} : \mathsf{X} \to \mathsf{TX}$ *and* $\gg\!=^{\mathsf{X},\mathsf{Y}} : \mathsf{TX} \times (\mathsf{X} \to \mathsf{TY}) \to \mathsf{TY}$ *are families of functions such that for every* $\mathsf{f} : \mathsf{X} \to \mathsf{TY}$, $\mathsf{g} : \mathsf{Y} \to \mathsf{TZ}$ *the following diagrams commute:*

$$\begin{tikzcd}
\mathsf{X} \arrow[r, "\mathbf{return}"] \arrow[dr, "\mathsf{f}"'] & \mathsf{TX} \arrow[d, "\gg= \mathsf{f}"] \\
& \mathsf{TY}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{TX} \arrow[r, bend left, "id"] \arrow[r, bend right, "\gg=\mathbf{return}"'] & \mathsf{TX}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{TX} \arrow[r, "\gg=\mathsf{f}"] \arrow[dr, "\gg=(\lambda x.\mathsf{f}x \gg= \mathsf{g})"'] & \mathsf{TY} \arrow[d, "\gg=\mathsf{g}"] \\
& \mathsf{TZ}
\end{tikzcd}$$

We say that T *satisfies the* ***mono requirement*** *[14] if* $\mathbf{return} : \mathsf{X} \to \mathsf{TX}$ *is an injection for all* X*.*

Sometimes, monads are defined as triples $\langle \mathsf{T}, \mathbf{return}, \mu \rangle$, where $\mu : \mathsf{T}(\mathsf{TX}) \to \mathsf{TX}$ is called the “multiplication” for the monad. The definitions are equivalent, because we can set $\mu x := x \gg\!= \mathrm{id}$ and, vice versa, $x \gg\!= f := \mu(\mathsf{T} f\, x)$. In a programming language that has monad support built-in one defines for $f : X \to Y$ the function $\mathbf{fmap}\, f : \mathsf{T}X \to \mathsf{T}Y$ as $\mathbf{fmap}\, f\, xs := xs \gg\!= (\mathbf{return} \circ f)$. This definition yields $\mathbf{fmap}\, f = \mathsf{T}\, f$ in the abstract setting if T is a functor.

We will use the following equation for **fmap** and μ:

$$\mu(\mathbf{fmap}\ f\ xs) = \mathbf{fmap}\ f\ xs \gg\!= \mathrm{id} = xs \gg\!= f$$

One can define the category of monads for a given category $\mathbb{C}$ by defining monad morphisms between monads:

Definition 2.2. *Let* $(\mathsf{T}_1, \mathbf{return}_1, \gg\!=_1)$ *and* $(\mathsf{T}_2, \mathbf{return}_2, \gg\!=_2)$ *be two monads over* **Set**. *A **monad morphism*** $\mathsf{T}_1 \to \mathsf{T}_2$ *is a natural transformation* $m_X : \mathsf{T}_1 X \to \mathsf{T}_2 X$ *with* $m_X(\mathbf{return}_1\ x) = \mathbf{return}_2\ x$ *and* $m_X(t \gg\!=_1 f) = m_X t \gg\!=_2 (m_Y \circ f)$.

In order for m to be a natural transformation we need $m_Y \circ \mathbf{fmap}_1\ f\ = \mathbf{fmap}_2\ f \circ m_X$.

Example 1. There is always a unique monad morphism from the identity monad, i.e. the identity monad is initial in this category

$$\mathrm{I} := \langle \mathrm{I}X = X, \mathbf{return}_{\mathrm{I}}\ x = x, m \gg\!=_{\mathrm{I}} f = f m\rangle$$

to any monad T given by $m_X(x) = \mathbf{return}_{\mathsf{T}}\ x$. We then have $m_X(\mathbf{return}_{\mathrm{I}}\ x) = F_X(x) = \mathbf{return}_{\mathsf{T}}\ x$ and $m_X(t \gg\!=_{\mathrm{I}} f) = m_X(f\, t) = \mathbf{return}_{\mathsf{T}}\ (f\, t) = \mathbf{return}_{\mathsf{T}}\ t \gg\!=_{\mathsf{T}} \lambda x.\mathbf{return}_{\mathsf{T}}\ (f\, x)$. ■

2.1.2 Algebras

Universal algebra can be used as a tool to describe effectful programs [18]. An algebra in the universal algebraic sense is a generalisation of an algebraic structure. It consists of a carrier set and a set of n-ary operations. The set of possible operations is described by a parameterised signature:

Definition 2.3. *A **parameterised signature** is a pair* $\mathcal{O} = \langle |\mathcal{O}|, \mathrm{arity}_{\mathcal{O}} \rangle$ *where* $|\mathcal{O}|$ *is a set and* $\mathrm{arity}_{\mathcal{O}}$ *assigns to each* f *in* $|\mathcal{O}|$ *two sets* $\mathrm{arity}_{\mathcal{O}}(f) = \langle P_f, A_f \rangle$.

We call the elements f of $|\mathcal{O}|$ **operation symbols**, the set P_f the **parameter type** of f, and the set A_f the **arity** of f. When $\mathrm{arity}_{\mathcal{O}}(f) = \langle P_f, A_f \rangle$, we write $f : P_f \to A_f$. The notation already gives an idea on how we intend to use those operation symbols.

An $\mathcal{O}$-algebra now interprets these symbols:

Definition 2.4 ($\mathcal{O}$-algebra)**.** *Given a (parameterised) signature* $\mathcal{O}$ *an* $\mathcal{O}$*-algebra is a pair* $\langle |C|, [\![-]\!] \rangle$ *where* $|C|$ *is a set and* $[\![-]\!]$ *assigns, for each* $f : P \to A$ *in* Σ, *a function* $[\![-]\!] : |C|^A \to |C|^P$.

The idea is that given a parameter $p \in P$ and a (possibly infinite) sequence $\langle c_a \in |C|\rangle_{a\in A}$ the interpretation of f produces something in $|C|$. Thus, for a finite set A with n elements $[\![f : P \to A]\!]$ is simply an n-ary operation on $|C|$ parameterised by P.

In addition to defining an interpretation for those symbols, we can also build terms over those symbols:

Definition 2.5 (The $T_{\mathcal{O}}$ monad)**.** *Given a (parameterised) signature $\mathcal{O}$, and a set X, we can inductively define the set $T_{\mathcal{O}}X$ of $\mathcal{O}$-terms with variables in X:*

$$t ::= x \mid f_p(\lambda a : A.t_a)$$

for all $x \in X$, $(f : P \to A) \in \mathcal{O}$, $p \in P$, and A-indexed sequence of $\mathcal{O}$-terms $\langle t_a\rangle_{a\in A}$ in $T_{\mathcal{O}}X$.

$T_{\mathcal{O}}$ has a monad structure given by:

$$\textbf{return}\ x \coloneqq x \qquad\qquad x \gg\!= f \coloneqq f x$$
$$f_p(\lambda a.t_a) \gg\!= f \coloneqq f_p(\lambda a. \gg\!= t_a f)$$

Note that the $T_{\mathcal{O}}$ monad fulfils the mono-requirement (as **return** is the identity).

Monads also have a notion of an algebra. This notion generalises our initial definition in the sense that $\mathcal{O}$-algebras and algebras for the monad $T_{\mathcal{O}}$ are in bijection.

We first define the concept:

Definition 2.6 (Algebra over a monad)**.** *A T-**algebra** for a monad T is a pair $C = \langle |C|, c\rangle$ where $|C|$ is a set and $c : T\,|C| \to |C|$ is a function satisfying:*

$$c(\textbf{return}\ x) = x \qquad\qquad c(\textbf{fmap}\ c\ xs) = c(xs \gg\!= \mathrm{id})$$

*for all $x \in |C|$ and $xs \in T^2\,|C|$. $|C|$ is called the **carrier** and we call c the **algebra structure**.*

We say that a set A forms a T-algebra if there is a monad operation a s.t. $\langle A, a\rangle$ is a T-algebra.

An algebra over the monad $T_{\mathcal{O}}$ now is a set $|C|$ and a function $c : T_{\mathcal{O}}|C| \to |C|$. We can use this structure to interpret symbols in $\mathcal{O}$ by setting:

$$[\![f]\!]\,(xs : |C|^A)(p : P) \coloneqq c(f_p(xs))$$

Vice versa, given an $\mathcal{O}$-algebra $\langle |C|, [\![-]\!]\rangle$ we can build the $T_{\mathcal{O}}$-algebra with carrier

$|C|$:

$$c(x \in |C|) \coloneqq x$$
$$c(f_p(\lambda a.t_a)) \coloneqq [\![f]\!]\,(\lambda a.c(t_a))(p)$$

Free Algebras

Defining multiplication for the $T_{\mathcal{O}}$ monad is simple: We need $\mu : T_{\mathcal{O}}(T_{\mathcal{O}}X) \to T_{\mathcal{O}}X$, i.e. a way to create a term with variables in X from a term where the variables are in $T_{\mathcal{O}}X$. Given such a term, multiplication can just reinterpret the very same term as a term in $T_{\mathcal{O}}X$ by treating the former variables in $T_{\mathcal{O}}X$ as extension, i.e. $\mu = {\gg\!=}\mathrm{id}$.

This idea can be used to define the notion of a free algebra:

Definition 2.7 (Free algebra)**.** *For each set* X*, the pair* $FX \coloneqq \langle TX, {\gg\!=}\mathrm{id}\rangle$ *forms a* T*-algebra called the* ***free*** T***-algebra over*** X*.*

The bijection between $T_{\mathcal{O}}$ and $\mathcal{O}$-algebras assigns to the free algebra $F_{T_{\mathcal{O}}}X = \langle T_{\mathcal{O}}X, {\gg\!=}\mathrm{id}\rangle$ the term algebra $F_{\mathcal{O}}$ over $T_{\mathcal{O}}X$ given by:

$$[\![f]\!]\,(ts)(p) \coloneqq f_p(\lambda a.ts(a))$$

The operation F can be seen as a function **Set** $\to$ **Alg** (i.e. mapping a set to an algebra). We can define the inverse function U : **Alg** $\to$ **Set** as $UC \coloneqq |C|$ and obtain a bijection $U(FX) \cong TX$ (If F and U are functors in the category-theoretical sense we get an adjunction $U \dashv F$). This bijection of building a free algebra structure and forgetting it again will play a major role in the denotational semantics for our calculi CBPV, λ_{eff}, and λ_{mon}.

We can extend the Kleisli extension operator ${\gg\!=}$, which maps a function $f : X \to TY$ to a function $({\gg\!=}f) : TX \to TY$ to algebras, in the following way. Given a T-algebra C and a function $X \to |C|$, define $({\gg\!=}f) : TX \to |C|$ by

$$(xs{\gg\!=}f) \coloneqq c(\mathbf{fmap}\ f\ xs)$$

When $C = FY = \langle TY, {\gg\!=}\mathrm{id}\rangle$, this operator coincides with the given Kleisli extension operator, as shown in lemma 2.1.1.

Some special algebras

Apart from the free algebra we introduce three more standard algebras over a given monad T: singleton algebras, exponential algebras, and product algebras. Understanding those constructions is both useful in understanding the concept of algebras and in the denotational semantics given in the next chapters.

Lemma 2.8.

- *Singleton algebra:* $\langle\{\star\}, \lambda xs.\star\rangle$ *is a* T*-algebra for any monad* T*.*
- *Exponential algebra: Given a* T*-algebra* $\langle|C|, c\rangle$ *and a set* A*,*

  $$\langle|C|^A, \lambda f_s.\lambda x.c(\mathbf{fmap}\ (\lambda f.f(x))\ f_s)\rangle$$

  forms a T*-algebra.*
- *Product algebra: Given two* T*-algebras* $\langle|C_i|, c_i\rangle$*,* $i \in \{1,2\}$ *we have a* T*-algebra as follows:* $\langle|C_1| \times |C_2|, \lambda c_s.\ \langle c_1(\mathbf{fmap}\ \pi_1\ c_s), c_2(\mathbf{fmap}\ \pi_2\ c_s)\rangle\rangle$

We leave out the proofs here, but they are well-known.

2.2 CBPV: Call-by-push-value

Most commonly, when doing programming language semantics, one imposes either a call-by-value or a call-by-name semantics to a λ-calculus with possibly some additional structure. However, if one adds effects to the language, the evaluation order matters and call-by-value and call-by-name yield different observational behaviour of terms. Instead of comitting to to one and defining the other independently, we use CBPV which includes both behaviours as subcalculi.

Call-by-push-value (CBPV) is a computational metalanguage that subsumes both call-by-value and call-by-name. It was introduced by Levy [11, 12] and will serve as the base language for the thesis. In the remainder of the chapter we introduce CBPV, its operational semantics and an adequate (set-theoretic) denotational semantics.

2.2.1 Syntax and operational semantics of CBPV

Figures 2.1–2.4 show the definition of CBPV, its type system, and its reduction rules.

$$\begin{array}{rrl}
\text{(values)} & V, W & ::= x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\} \\
\text{(computations)} & & \\
 & M, N & ::= \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case_0}(V) \\
 & & \mid\ \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V! \\
 & & \mid\ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
 & & \mid\ \lambda x.M \mid M\ V \\
 & & \mid\ \langle M_1, M_2\rangle \mid \mathbf{prj}_i\, M
\end{array}$$

Figure 2.1: CBPV syntax

$$
\begin{array}{lrl}
\text{(value types)} & A, B & ::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid UC \\
\text{(computation types)} & C, D & ::= FA \mid A \to C \mid \top \mid C_1 \mathbin{\&} C_2 \\
\text{(environments)} & \Gamma & ::= x_1 : A_1, \ldots, x_n : A_n
\end{array}
$$

Figure 2.2: CBPV types

Value typing $\boxed{\Gamma \vdash V : A}$

$$
\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}
\qquad
\frac{}{\Gamma \vdash () : 1}
\qquad
\frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}
\qquad
\frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2}
$$

$$
\frac{\Gamma \vdash M : C}{\Gamma \vdash \{M\} : UC}
$$

Computation typing $\boxed{\Gamma \vdash M : C}$

$$
\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : C}{\Gamma \vdash \mathbf{split}(V, x_1.x_2.M) : C}
\qquad
\frac{\Gamma \vdash V : 0}{\Gamma \vdash \mathbf{case_0}(V) : C}
$$

$$
\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash M_1 : C \qquad \Gamma, x_2 : A_2 \vdash M_2 : C}{\Gamma \vdash \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}
\qquad
\frac{\Gamma \vdash V : UC}{\Gamma \vdash V! : C}
$$

$$
\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\ V : FA}
\qquad
\frac{\Gamma \vdash M : FA \qquad \Gamma, x : A \vdash N : C}{\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C}
\qquad
\frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.M : A \to C}
$$

$$
\frac{\Gamma \vdash M : A \to C \qquad \Gamma \vdash V : A}{\Gamma \vdash M\, V : C}
\qquad
\frac{}{\Gamma \vdash \langle\rangle : \top}
\qquad
\frac{\Gamma \vdash M_1 : C_1 \qquad \Gamma \vdash M_2 : C_2}{\Gamma \vdash \langle M_1, M_2 \rangle : C_1 \mathbin{\&} C_2}
$$

$$
\frac{\Gamma \vdash M : C_1 \mathbin{\&} C_2}{\Gamma \vdash \mathbf{prj}_i\, M : C_i}
$$

Figure 2.3: CBPV type system

CBPV distinguishes between value types and computation types. Terms of the first type are non-computing objects, that can for instance be given as an argument to a function. Terms of the latter type are computing objects, and terms of computation type are the only terms that reduce. A computation returning a value is captured by the construct **return** V, yielding an object of type FA if V is of type A. The value can be extracted with the let-construct. To pass a computation as an argument (i.e. to write a higher-order function) one has to **thunk** it first as $\{M\}$, yielding an object of type UC if M was of type C. The inverse operation $M!$ is called **force**.

Reduction frames

$$\text{(computation frames)}\ \mathcal{C} ::= \textbf{let}\ x \leftarrow [\,]\ \textbf{in}\ N \mid [\,]\ V \mid \textbf{prj}_i\ [\,]$$

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

$$\begin{array}{lr}
(\beta.\times) & \textbf{split}((V_1, V_2), x_1.x_2.M) \longrightarrow M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \textbf{case}(\textbf{inj}_i\ V, x_1.M_1, x_2.M_2) \longrightarrow M_i[V/x_i] \\
(\beta.U) & \{M\}! \longrightarrow M \\
(\beta.F) & \textbf{let}\ x \leftarrow \textbf{return}\ V\ \textbf{in}\ M \longrightarrow M[V/x] \\
(\beta.\to) & (\lambda x.M)\ V \longrightarrow M[V/x] \\
(\beta.\&) & \textbf{prj}_i\ \langle M_1, M_2 \rangle \longrightarrow M_i \\
(\textit{frame}) & \dfrac{M \longrightarrow M'}{\mathcal{C}[M] \longrightarrow \mathcal{C}[M']}
\end{array}$$

Figure 2.4: CBPV operational semantics

2.2.2 Finite types in CBPV

We define **ground types** as exactly those value types that do not contain thunked computations:

$$\text{(ground values)}\ G ::= 1 \mid G_1 \times G_2 \mid 0 \mid G_1 + G_2$$

Ground types will play an important role. Obviously, equality of ground values is computationally decidable, even if we add additional computation constructs to the language, which is also why we will use this definition for all presented languages. Thus, our soundness theorems of the form "If $\vdash M : FA$, then $M \longrightarrow^* \textbf{return}\ V$" will always restrict A to ground types.

We will need ground types with exactly n elements. In CBPV we can simply define them as the n-ary sum of the 1-type:

$$\mathbb{F}_0 := 0$$
$$\mathbb{F}_{n+1} := 1 + \mathbb{F}_n$$

We can define elements of the type $\mathbb{F}_n$ as $0_n := ()$, $(m+1)_n := \textbf{inj}_1\ m_n$. The elements of $\mathbb{F}_n$ then are exactly $0_n, \dots (n-1)_n$.

We can define the function $\text{succ} : \mathbb{F}_n \to \mathbb{F}_n := \lambda x.\textbf{inj}_1\ x$ such that for all $m < n - 1$, $\text{succ}\ m_n \longrightarrow^* (m+1)_n$.

Lemma 2.9. *For different natural numbers* $m \neq m'$ *with* $m, m' \leqslant n$ *we have* $m_n \neq m'_n$ *syntactically.*

Proof

Immediately from the definition. ■

2.2.3 Syntactic sugar

We define several constructs that make it easier two write down terms in CBPV. In CBPV one cannot apply two computations to each other, so we set:

$$M\ N := \textbf{let}\ x \leftarrow N\ \textbf{in}\ M\ x$$

We can implement the type of booleans with $\mathbb{F}_2$. We can then implement the usual if/then/else construct as:

$$\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2 := \textbf{case}(b, \_.C_1, \_.C_2).$$

Where it seems convenient we might write V for **return** V, as the ambiguity in addition to the notation $M\ N$ does not change the operational behaviour of the term.

2.2.4 Denotational Semantics

We are not going to directly need denotational semantics for CBPV. However, we can reuse the semantic definitions and most of the proofs for λ_{eff} and λ_{mon}.

We define the denotational semantics for CBPV for any monad T over **Set** fulfilling the mono-requirement. The denotational semantics for types is shown in figure 2.5. For every value type A we associate a set $[\![A]\!]$ that contains the denotation of the closed terms of type A. To every computation type C we associate a T-algebra $[\![C]\!]$.

The denotations of value types are the evident set-theoretic constructions, i.e. disjoint union for $+$, product for $\times$, a singleton set for 1 and the empty set for 0. The type of thunks UC denotes the carrier of the algebra $[\![C]\!]$.

The denotation of computation types has more structure. For the $\top$, & and $\to$ we use the singleton, exponential and product constructions for algebras from section 2.1.2.

The most interesting part is the denotational semantics for FA. Here we use the free algebra for the monad T over the set $[\![A]\!]$.

Recall that a context Γ is just a partial function from variable names to value types with a finite domain of definition Dom (Γ). A denotation for an environment then

The semantics assigns to each well kinded:

- value type A a set $[\![A]\!]$;
- computation type C a T-algebra $[\![C]\!]$;
- context $\vdash_k \Gamma : \mathbf{Ctxt}$ a set $[\![\Gamma]\!]$; and

Value types

$$[\![1]\!] := \{\star\} \qquad [\![A_1 \times A_2]\!] := [\![A_1]\!] \times [\![A_2]\!] \qquad [\![0]\!] := \emptyset$$
$$[\![A_1 + A_2]\!] := \{1\} \times [\![A_1]\!] \cup \{2\} \times [\![A_2]\!] \qquad [\![UC]\!] := |[\![C]\!]|$$

Computation types

$$[\![FA]\!] := F[\![A]\!] \qquad [\![A \to C]\!] := \langle |[\![C]\!]|^{[\![A]\!]}, \lambda f_s.\lambda x.c(\mathbf{fmap}\ (\lambda f.f(x))\ f_s)\rangle$$
$$[\![\top]\!] := \langle \{\star\}, \lambda xs.\star\rangle$$
$$[\![C_1 \mathbin{\&} C_2]\!] := \langle |[\![C_1]\!]| \times |[\![C_2]\!]|, \lambda c_s.\, \langle c_1(\mathbf{fmap}\ \pi_1\ c_s), c_2(\mathbf{fmap}\ \pi_2\ c_s)\rangle\rangle$$

Contexts $[\![\Gamma]\!] := \prod_{x \in \mathrm{Dom}(\Gamma)} [\![\Gamma(x)]\!]$

Figure 2.5: CBPV denotational semantics for types

simply maps variable names to the denotation of value types.

We can give semantics to CBPV by setting T to the identity monad. However, for λ_{eff} and λ_{mon} we will need more involved monads and reuse the generalised semantics.

In our setting, a term can have multiple types, for example the function $\lambda x.x$ can have type $1 \to 1$ or $0 \to 0$. Moreover, a given type judgement might have multiple type derivations. We thus in fact give a Church-style semantics [21] to our calculi, i.e. define the denotations for type judgements rather then for terms. However, we will sometimes pretend to assign it to terms directly for brevity and better readability.

Figure 2.6 defines the denotational semantics for terms. Value derivations $\Gamma \vdash V : A$ denote functions $[\![V]\!] : [\![\Gamma]\!] \to [\![A]\!]$, and computation derivations $\Gamma \vdash M : C$ denote functions $[\![M]\!] : [\![\Gamma]\!] \to |[\![C]\!]|$. This does not reflect any use of the algebra structure on $[\![C]\!]$ explicitly. However, as these definitions all give algebra structures over their carrier sets, we obtain, for each function $f : X \to |[\![C]\!]|$, a Kleisli extension $(\gg\!\!= f) : TX \to |[\![C]\!]|$. Our semantics makes use of the Kleisli extension in the semantics of **let** $x \leftarrow M$ **in** N.

As a sanity check we prove the following lemma about the denotation of ground types:

Lemma 2.10. *For all ground types G and for all $a \in [\![G]\!]$ there exists a closed value term*

Value terms

$$[\![x]\!](\gamma) := \pi_x(\gamma) \qquad [\![()]\!](\gamma) := \star \qquad [\![(V_1, V_2)]\!](\gamma) := \langle [\![V_1]\!](\gamma), [\![V_2]\!](\gamma)\rangle$$
$$[\![\mathbf{inj}_i\, V]\!](\gamma) := \langle i, [\![V]\!](\gamma)\rangle \qquad [\![\{M\}]\!](\gamma) := [\![M]\!](\gamma)$$

Computation terms

$$[\![\mathbf{split}(V, x_1.x_2.M)]\!](\gamma) := [\![M]\!](\gamma[x_1 \mapsto a_1, x_2 \mapsto a_2]), \text{ where } [\![V]\!](\gamma) = \langle a_1, a_2\rangle$$
$$[\![\mathbf{case_0}(V)]\!] \text{ is the empty map, as } [\![V]\!] : [\![\Gamma]\!] \to \emptyset \text{ necessitates } [\![\Gamma]\!] = \emptyset$$
$$[\![\mathbf{case}(V, x_1.M_1, x_2.M_2)]\!] := \begin{cases} [\![M_1]\!](\gamma[x_1 \mapsto a_1]) & [\![V]\!](\gamma) = \langle 1, a_1\rangle \\ [\![M_2]\!](\gamma[x_2 \mapsto a_2]) & [\![V]\!](\gamma) = \langle 2, a_2\rangle \end{cases}$$
$$[\![V!]\!](\gamma) := [\![V]\!](\gamma) \qquad [\![\mathbf{return}\ V]\!](\gamma) := \mathbf{return}\ ([\![V]\!](\gamma))$$
$$[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!](\gamma) := [\![M]\!](\gamma) \gg\!= \lambda a. [\![N]\!](\gamma[x \mapsto a])$$
$$[\![\lambda x.M]\!](\gamma) := \lambda a. [\![M]\!](\gamma[x \mapsto a]) \qquad [\![M\ V]\!](\gamma) := ([\![M]\!](\gamma))([\![V]\!](\gamma)) \qquad [\![\langle\rangle]\!](\gamma) := \star$$
$$[\![\langle M_1, M_2\rangle]\!](\gamma) := \langle [\![M_1]\!](\gamma), [\![M_2]\!](\gamma)\rangle \qquad [\![\mathbf{prj}_i\, M]\!](\gamma) := \pi_i([\![M]\!](\gamma))$$

Figure 2.6: CBPV denotational semantics for terms

$\vdash V^G_a : G$ *such that* $[\![V^G_a]\!] = a$.

Proof

Define V^G_a by induction on G:

$$V^1_\star := () \qquad V^{G_1 \times G_2}_{\langle a_1, a_2\rangle} := (V^{G_1}_{a_1}, V^{G_2}_{a_2}) \qquad V^{G_1+G_2}_{\langle i, a\rangle} := \mathbf{inj}_i\, V^{G_i}_a$$

■

2.2.5 Contextual equivalence for CBPV

We define contexts and their type system for CBPV as in Figure 2.7–2.8. Note that it is straightforward to define contexts with two holes, but we omit this here for conciseness. We say that a type environment Γ' **extends** a type environment Γ, and write $\Gamma' \geqslant \Gamma$ if Γ' extends Γ as a partial function from identifiers to value types.

Definition 2.11 (contextual equivalence). *Let* $\Gamma \vdash P, Q : X$ *be two* CBPV *phrases. We say that* P *and* Q *are* ***contextually equivalent*** *and write* $P \simeq Q$ *when for all* ***closed*** *well-typed* ***ground-returners*** *contexts* $\emptyset[\Gamma'] \vdash \mathcal{X}[\ ] : FG[X]$ *with* $\Gamma' \geqslant \Gamma$ *and for all closed ground value terms* $\vdash V : G$*, we have:*

$$\mathcal{X}[P] \longrightarrow^\star \mathbf{return}\ V \qquad \Longleftrightarrow \qquad \mathcal{X}[Q] \longrightarrow^\star \mathbf{return}\ V$$

Lemma 2.12 (substitution). *For all* $\Gamma \vdash P : X$ *and* $\langle V_x\rangle_{x \in \mathrm{Dom}(\Gamma)}$ *such that, for all* $x \in$

$$
\begin{array}{lll}
\text{(value contexts)} & \mathcal{X}_{\text{val}} & ::= [\ ] \mid (\mathcal{X}_{\text{val}}, V_2) \mid (V_1, \mathcal{X}_{\text{val}}) \mid \mathbf{inj}_i\, \mathcal{X}_{\text{val}} \mid \{\mathcal{X}_{\text{comp}}\} \\
\text{(computation contexts)} & & \\
 & \mathcal{X}_{\text{comp}} & ::= [\ ] \mid \mathbf{split}(\mathcal{X}_{\text{val}}, x_1.x_2.M) \\
 & & \mid\ \mathbf{split}(V, x_1.x_2.\mathcal{X}_{\text{comp}}) \mid \mathbf{case_0}(\mathcal{X}_{\text{val}}) \\
 & & \mid\ \mathbf{case}(\mathcal{X}_{\text{val}}, x_1.M_1, x_2.M_2) \\
 & & \mid\ \mathbf{case}(V, x_1.\mathcal{X}_{\text{comp}}, x_2.M_2) \\
 & & \mid\ \mathbf{case}(V, x_1.M_1, x_2.\mathcal{X}_{\text{comp}}) \mid \mathcal{X}_{\text{val}}! \\
 & & \mid\ \mathbf{return}\ \mathcal{X}_{\text{val}} \\
 & & \mid\ \mathbf{let}\ x \leftarrow \mathcal{X}_{\text{comp}}\ \mathbf{in}\ N \\
 & & \mid\ \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ \mathcal{X}_{\text{comp}} \\
 & & \mid\ \lambda x.\mathcal{X}_{\text{comp}} \mid \mathcal{X}_{\text{comp}}\ V \mid M\ \mathcal{X}_{\text{val}} \\
 & & \mid\ \langle \mathcal{X}_{\text{comp}}, M_2 \rangle \mid \langle M_1, \mathcal{X}_{\text{comp}} \rangle \mid \mathbf{prj}_i\, \mathcal{X}_{\text{comp}}
\end{array}
$$

Figure 2.7: CBPV contexts

Dom (Γ), $\Delta \vdash V_x : \Gamma(x)$, *we have* $\Delta \vdash P[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} : X$.

Proof

Straightforward induction over $\Gamma \vdash P : X$. ■

Lemma 2.13 (context substitution). *For all* $\Delta[\Gamma'] \vdash \mathcal{X}[\ ] : Y[X]$*:*

1. *For all* $\Gamma \vdash P : X$ *where* $\Gamma' \geqslant \Gamma$, *we have* $\Delta \vdash \mathcal{X}[P] : Y$.
2. *For all* $\Gamma \vdash P, Q : X$ *where* $\Gamma' \geqslant \Gamma$, *we have* $[\![P]\!] = [\![Q]\!]$ *implies* $[\![\mathcal{X}[P]]\!] = [\![\mathcal{X}[Q]]\!]$.

Proof

Both parts follow from a straightforward induction over the derivation of $\Delta[\Gamma'] \vdash \mathcal{X}[\ ] : Y[X]$. ■

Lemma 2.14 (basic contextual equivalence properties). *The relation $\simeq$ is an equivalence relation that is congruent w.r.t.* CBPV *terms, and contains the reduction relation $\longrightarrow$.*

2.2.6 Adequacy

Our denotational semantics is adequate, i.e. denotationally equivalent terms are observationally equivalent. We prove this using a standard logical relations argument [17]. The lifting for the U type first appeared in [8].

For every (value/computation) type X, we define $\text{CBPV}_X = \{P \mid \vdash P : X\}$. We define the logical relations for a CBPV type X by induction on X as in Figure 2.9.

We first show some technical properties of the logical relations:

Context typing $\boxed{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : C[D]}$

$$\frac{}{\Gamma[\Gamma] \vdash [\ ] : X[X]}$$

Value context typing

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A_1[X] \qquad \Gamma \vdash V_2 : A_2}{\Gamma[\Delta] \vdash (\mathcal{X}[\ ], V_2) : A_1 \times A_2[X]} \qquad \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma[\Delta] \vdash \mathcal{X}[\ ] : A_2[X]}{\Gamma[\Delta] \vdash (V_1, \mathcal{X}[\ ]) : A_1 \times A_2[X]}$$

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A_i[X]}{\Gamma[\Delta] \vdash \mathbf{inj}_i\, \mathcal{X}[\ ] : A_1 + A_2[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : C[X]}{\Gamma[\Delta] \vdash \{\mathcal{X}[\ ]\} : UC[X]}$$

Computation context typing

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A_1 \times A_2[X] \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : C}{\Gamma[\Delta] \vdash \mathbf{split}(\mathcal{X}[\ ], x_1.x_2.M) : C[X]}$$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2[\Delta] \vdash M : C[X]}{\Gamma[\Delta] \vdash \mathbf{split}(V, x_1.x_2.\mathcal{X}[\ ]) : C[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : 0[X]}{\Gamma[\Delta] \vdash \mathbf{case_0}(\mathcal{X}[\ ]) : C[X]}$$

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A_1 + A_2[X] \qquad \Gamma, x_1 : A_1 \vdash M_1 : C \qquad \Gamma, x_2 : A_2 \vdash M_2 : C}{\Gamma[\Delta] \vdash \mathbf{case}(\mathcal{X}[\ ], x_1.M_1, x_2.M_2) : C[X]}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1[\Delta] \vdash \mathcal{X}[\ ] : C[X] \qquad \Gamma, x_2 : A_2 \vdash M_2 : C}{\Gamma[\Delta] \vdash \mathbf{case}(V, x_1.\mathcal{X}[\ ], x_2.M_2) : C[X]}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash M_1 : C \qquad \Gamma, x_2 : A_2[\Delta] \vdash \mathcal{X}[\ ] : C[X]}{\Gamma \vdash \mathbf{case}(V, x_1.M_1, x_2.\mathcal{X}[\ ]) : C}$$

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : UC[X]}{\Gamma[\Delta] \vdash \mathcal{X}[\ ]! : C[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A[X]}{\Gamma[\Delta] \vdash \mathbf{return}\ \mathcal{X}[\ ] : FA[X]}$$

$$\frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : FA[X] \qquad \Gamma, x : A \vdash N : C}{\Gamma[\Delta] \vdash \mathbf{let}\ x \leftarrow \mathcal{X}[\ ]\ \mathbf{in}\ N : C[X]} \qquad \frac{\Gamma \vdash M : FA \qquad \Gamma, x : A[\Delta] \vdash \mathcal{X}[\ ] : C[X]}{\Gamma[\Delta] \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ \mathcal{X}[\ ] : C[X]}$$

$$\frac{\Gamma, x : A[\Delta] \vdash \mathcal{X}[\ ] : C[X]}{\Gamma[\Delta] \vdash \lambda x.\mathcal{X}[\ ] : A \to C[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : A \to C[X] \qquad \Gamma \vdash V : A}{\Gamma[\Delta] \vdash \mathcal{X}[\ ]\ V : C[X]}$$

$$\frac{\Gamma \vdash M : A \to C \qquad \Gamma[\Delta] \vdash \mathcal{X}[\ ] : A[X]}{\Gamma[\Delta] \vdash M\ \mathcal{X}[\ ] : C[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : C_1[X] \qquad \Gamma \vdash M_2 : C_2}{\Gamma[\Delta] \vdash \langle \mathcal{X}[\ ], M_2 \rangle : C_1 \mathbin{\&} C_2[X]}$$

$$\frac{\Gamma \vdash M_1 : C_1 \qquad \Gamma[\Delta] \vdash \mathcal{X}[\ ] : C_2[X]}{\Gamma[\Delta] \vdash \langle M_1, \mathcal{X}[\ ] \rangle : C_1 \mathbin{\&} C_2[X]} \qquad \frac{\Gamma[\Delta] \vdash \mathcal{X}[\ ] : C_1 \mathbin{\&} C_2[X]}{\Gamma[\Delta] \vdash \mathbf{prj}_i\, \mathcal{X}[\ ] : C_i[X]}$$

Figure 2.8: CBPV context types

Value relations $\boxed{R_A \subseteq [\![A]\!] \times \textbf{CBPV}_A}$

$$R_1 := \{\langle \star, V\rangle | V \simeq ()\}$$

$$R_{A_1 \times A_2} := \{\langle\langle a_1, a_2\rangle, V\rangle | \exists V_1, V_2 \forall i. V_i : A_i. \langle a_i, V_i\rangle \in R_{A_i}, V \simeq (V_1, V_2)\} \qquad R_0 := \emptyset$$

$$R_{A_1 + A_2} = \bigcup_{i=1,2} \left\{\langle \iota_i a, V\rangle | \exists V' : A_i. \langle a, V'\rangle \in R_{A_i}, V \simeq \textbf{inj}_i\, V'\right\}$$

$$R_{U\,C} := \{\langle a, V\rangle | \exists M : C. \langle a, M\rangle \in R_C, V \simeq \{M\}\} \qquad \text{note the thunk in } \{M\}$$

Computation relations $\boxed{R_C \subseteq |[\![C]\!]| \times \textbf{CBPV}_C}$

$$R_{FA} := \{\langle \textbf{return } a, M\rangle | \exists V : A. \langle a, V\rangle \in R_A, M \simeq \textbf{return } V\}$$

$$R_{A \to C} := \{\langle f, M\rangle | \forall \langle a, V\rangle \in R_A, \langle f(a), M\ V\rangle \in R_C\} \qquad R_\top := \{\langle \star, M\rangle | M \simeq \langle\rangle\}$$

$$R_{C_1 \& C_2} := \{\langle\langle c_1, c_2\rangle, M\rangle | \exists M_1 : C_1, M_2 : C_2. M \simeq \langle M_1, M_2\rangle, \forall i. \langle c_i, M_i\rangle \in R_{C_i}\}$$

Figure 2.9: CBPV logical relations

Lemma 2.15. *The logical relations* R_X *are closed under contextual equivalence. Explicitly: For all* $\langle a, P\rangle \in R_X$ *and* $\vdash Q : X$*,* $P \simeq Q$ *implies* $\langle a, Q\rangle \in R_X$*.*

Proof

By induction on X. ■

Lemma 2.16. *For all ground types* G*,* $a \in [\![G]\!]$*, and closed value terms* $\vdash V : , V' : G$*:*

$$\langle a, V\rangle, \langle a, V'\rangle \in R_G \implies V \simeq V'$$

Proof

By induction on G. ■

Lemma 2.17 (basic lemma). *For all* $\Gamma \vdash P : X$*,* $\gamma \in [\![\Gamma]\!]$ *and* $\langle V_x\rangle_{x \in \mathrm{Dom}(\Gamma)}$*:*

$$(\textit{for all } x \in \mathrm{Dom}(\Gamma)\textit{: } \langle \pi_x \gamma, V_x\rangle \in R_{\Gamma(x)}) \implies \left\langle [\![P]\!]\gamma, P[V_x/x]_{x \in \mathrm{Dom}(\Gamma)}\right\rangle \in R_X$$

Proof

We prove the lemma by induction over type derivations. The cases are all straightforward. As an example, we show the value type derivation rule for (×-I) and the computation type derivation for (×-E):

(×-I):

Assume the inductive hypothesis for $\Gamma \vdash v_i : A_i$, $i = 1, 2$, i.e.: $\left\langle [\![V_i]\!]\gamma, V_i[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{A_i}$.

But $\left\langle [\![(V_1, V_2)]\!]\gamma, (V_1, V_2)[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle$ is equal to

$$\left\langle \langle [\![V_1]\!]\gamma, [\![V_2]\!]\gamma\rangle, (V_1[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}, V_2[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle$$

and in $R_{A_1\times A_2}$ by the definition of $R_{A_1\times A_2}$.

($\times$-E): Assume the inductive hypothesis for $\Gamma \vdash V : A_1 \times A_2, \Gamma, x_1 : A_1, x_2 : A_2 \vdash M : C$. Consider any γ, $[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}$ as in the IH. Then by the the first IH we have that $\left\langle [\![V]\!]\gamma, V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle$ is in $R_{A_1\times A_2}$, so by definition there are $\vdash V_{x_1} : A_1, \vdash V_{x_2} : A_2, a_1 \in [\![A_1]\!], a_2 \in [\![A_2]\!]$ such that $[\![V]\!]\gamma = \langle a_1, a_2\rangle$, $V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)} \simeq (V_{x_1}, V_{x_2})$ and$\langle a_i, V_{x_i}\rangle \in R_{A_i}$ for $i = 1, 2$.

But then $\gamma' := \gamma[x_1 \mapsto a_1, x_2 \mapsto a_2]$, $\langle V_x\rangle_{x\in\mathrm{Dom}(\Gamma')}$ where $\Gamma' := \Gamma, x_1 : A_1, x_2 : A_2$ satisfy the premise of the second IH, so we have $\left\langle [\![M]\!]\gamma', M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_C$.

But: $\left\langle [\![\mathbf{split}(V, x_1.x_2.M)]\!]\gamma, \mathbf{split}(V, x_1.x_2.M)[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_C$ and this is equivalent to
$\left\langle [\![M]\!]\gamma', \mathbf{split}(V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}, x_1.x_2.M[V_x/x]_{x\in\mathrm{Dom}(\Gamma\setminus\{x_1,x_2\})})\right\rangle \in R_C$ We then have

$$\begin{aligned}
&\left\langle [\![\mathbf{split}(V, x_1.x_2.M)]\!]\gamma, \mathbf{split}(V, x_1.x_2.M)[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_C\\
= &\left\langle [\![M]\!]\gamma', \mathbf{split}(V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}, x_1.x_2.M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle \in R_C\\
\Leftarrow &\left\langle [\![M]\!]\gamma', \mathbf{split}((V_{x_1}, V_{x_2}), x_1.x_2.M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle \in R_C\\
\Leftarrow &\left\langle [\![M]\!]\gamma', M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_C
\end{aligned}$$

where both implications follow from Lemma 2.14 and Lemma 2.15. ■

Theorem 2.18 (adequacy). *Denotational equivalence implies contextural equivalence. Explicitly: Given a monad satisfying the mono requirement, then for all* $\Gamma \vdash P, Q : X$, *if* $[\![P]\!] = [\![Q]\!]$ *then* $P \simeq Q$.

Proof

Let $\Gamma \vdash P_1, P_2 : X$ be any well-typed phrases satisfying $[\![P_1]\!] = [\![P_2]\!]$. Consider any closed well-typed ground-returner context $\emptyset[\Gamma'] \vdash \mathcal{X}[\ ] : FG[X]$ with $\Gamma' \geqslant \Gamma$. By Lemma 2.13, $[\![\mathcal{X}[P_1]]\!] = [\![\mathcal{X}[P_2]]\!]$. Let c be this common denotation.

Consider any $i \in \{1, 2\}$. By the basic lemma:

$$\langle c, \mathcal{X}[P_i]\rangle \in R_{FG}$$

By R_{FG}'s definition, there exist $\langle a_i, V_i\rangle \in R_G$ such that $c = \mathbf{return}\ a_i$ and $\mathcal{X}[P_i] \simeq \mathbf{return}\ V_i$.

Thus:

$$\mathbf{return}\ a_1 = c = \mathbf{return}\ a_2$$

The mono requirement implies that $a_1 = a_2$. Therefore, by Lemma 2.16, $V_1 \simeq V_2$. Therefore:

$$\mathcal{X}[P_1] \simeq \mathbf{return}\ V_1 \simeq \mathbf{return}\ V_2 \simeq \mathcal{X}[P_2]$$

Therefore $\mathcal{X}[P_1] \longrightarrow^\star \mathbf{return}\ V$ iff $\mathcal{X}[P_2] \longrightarrow^\star \mathbf{return}\ V$, hence $P_1 \simeq P_2$. ■

Corollary 2.19 (soundness)**.** *All well-typed closed ground returners reduce to a normal form. Explicitly: for all* $\vdash M : FG$ *there exists some* $\vdash V : G$ *such that:*

$$M \longrightarrow^\star \mathbf{return}\ V$$

Proof

Pick the identity monad, which satisfies the mono requirement, and consider the induced denotational semantics.

By the basic lemma, $\langle [\![M]\!]\gamma, M\rangle \in R_{FG}$, so by definition $[\![M]\!] = \mathbf{return}\ a$ and $M \simeq \mathbf{return}\ V$ for some $\langle a, V\rangle \in R_G$. As M is a ground-returner, we can instantiate $M \simeq \mathbf{return}\ V$ at the empty context and at V, and deduce that $M \longrightarrow^\star \mathbf{return}\ V$ iff $\mathbf{return}\ V \longrightarrow^\star \mathbf{return}\ V$, and the latter holds by reflexivity.

Note that by Lemma 2.16 and Lemma 2.10, we can choose $V := V^G_a$ ■

Chapter 3

Effect handlers: λ_{eff}

In this chapter we recall λ_{eff} [9], an extension of CBPV with effects and effect handlers. This calculus can be seen as the core calculus of actual programming languages featuring effects and handlers like *eff* [1]. Effects arise from the use of operations like `raise` for exceptions, `set` and `get` for global store, or `read` and `write` for I/O. In λ_{eff}, one can declare such effect operations, use them to construct effectful code and specify how they are implemented by providing effect handlers. This leads to a clear separation between an effect and its implementation.

Section 3.1 introduces the syntax of λ_{eff}, a small-step operational semantics and a sound type system. We define contextual equivalence for λ_{eff} in section 3.2. We then use this definition to give an equational specification every implementation of exceptions and global store has to fulfil in section 3.3, before we present the implementations and prove them correct. Finally, we introduce a denotational semantics for λ_{eff} in section 3.4, prove its adequacy and the soundness of the type system.

3.1 Syntax and operational semantics of λ_{eff}

Figure 3.1 introduces the syntax of λ_{eff} and figure 3.2 the operational semantics. Figures 3.3–3.5 define the typing relation. As in the whole thesis we highlight parts that are different from CBPV via shading.

The relevant syntactic additions to CBPV are effect operations $\mathrm{op}\ V\ (\lambda x.M)$, handling constructs **handle** M **with** H and effect handlers H which describe how these effects should be executed. Handlers are a set of clauses and have to contain a return clause $\textbf{return}\ x \mapsto M$. Furthermore, they can have finitely many more handler clauses $\mathrm{op}\, p\, k \mapsto N$, where the operation symbols have to be distinct, which we emphasise by using the symbol for disjoint union $\uplus$ in the definition. Although the $\lambda x.M$ in operations is technically part of the syntax we will often treat it as an ordinary CBPV function and write $\mathrm{op}\ V\ M$ where M is of function type instead.

$$
\begin{array}{lrl}
\text{(values)} & V, W & ::= x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\} \\
\text{(computations)} & & \\
 & M, N & ::= \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case_0}(V) \\
 & & \mid\ \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V! \\
 & & \mid\ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
 & & \mid\ \lambda x.M \mid M\ V \\
 & & \mid\ \langle M_1, M_2 \rangle \mid \mathbf{prj}_i\, M \\
 & & \mid\ \mathrm{op}\ V(\lambda x.M) \mid \mathbf{handle}\ M\ \mathbf{with}\ H \\
\text{(handlers)} & H & ::= \{\mathbf{return}\ x \mapsto M\} \\
 & & \mid\ H \uplus \{\mathrm{op}\, p\, k \mapsto N\} \quad \text{where op does not occur in } H
\end{array}
$$

Figure 3.1: λ_{eff}-calculus syntax

Reduction frames

$$
\begin{array}{lrl}
\text{(hoisting frames)} & \mathcal{H} & ::= \mathbf{let}\ x \leftarrow [\,]\ \mathbf{in}\ N \mid [\,]\ V \mid \mathbf{prj}_i\, [\,] \\
\text{(computation frames)} & \mathcal{C} & ::= \mathcal{H} \mid \mathbf{handle}\ [\,]\ \mathbf{with}\ H
\end{array}
$$

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

$$
\begin{array}{lrl}
(\beta.\times) & \mathbf{split}((V_1, V_2), x_1.x_2.M) & \longrightarrow M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \mathbf{case}(\mathbf{inj}_i\, V, x_1.M_1, x_2.M_2) & \longrightarrow M_i[V/x_i] \\
(\beta.U) & \{M\}! & \longrightarrow M \\
(\beta.F) & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ M & \longrightarrow M[V/x] \\
(\beta.\to) & (\lambda x.M)\ V & \longrightarrow M[V/x] \\
(\beta.\&) & \mathbf{prj}_i\, \langle M_1, M_2 \rangle & \longrightarrow M_i
\end{array}
$$

$$
(\mathit{frame}) \quad \frac{M \longrightarrow M'}{\mathcal{C}[M] \longrightarrow \mathcal{C}[M']}
$$

$$
(\mathit{hoist}.\mathrm{op}) \quad \frac{x \notin FV(\mathcal{H})}{\mathcal{H}[\mathrm{op}\ V(\lambda x.M)] \longrightarrow \mathrm{op}\ V(\lambda x.\mathcal{H}[M])}
$$

$$
(\mathit{handle}.F) \quad \frac{H^{\mathbf{return}} = \lambda x.M}{\mathbf{handle}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \longrightarrow M[V/x]}
$$

$$
(\mathit{handle}.\mathrm{op}) \quad \frac{H^{\mathrm{op}} = \lambda p\, k.N \qquad x \notin FV(H)}{\begin{array}{l}\mathbf{handle}\ \mathrm{op}\ V(\lambda x.M)\ \mathbf{with}\ H \\ \qquad \longrightarrow N[V/p, \{\lambda x.\mathbf{handle}\ M\ \mathbf{with}\ H\}/k]\end{array}}
$$

Figure 3.2: λ_{eff}-calculus operational semantics

(kinds)	K	$::= \textbf{Val} \mid \textbf{Eff} \mid \textbf{Comp}_E \mid \textbf{Ctxt} \mid \textbf{Hndlr}$
(value types)	A, B	$::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid U_E C$
(computation types)	C, D	$::= FA \mid A \to C \mid \top \mid C_1 \mathbin{\&} C_2$
(effect signatures)	E	$::= \{op : A \to B\} \uplus E \mid \emptyset$
(handler types)	R	$::= A \;{}^{E}\!\!\Rightarrow^{E'}\, C$
(environments)	Γ	$::= x_1 : A_1, \ldots, x_n : A_n$

Figure 3.3: λ_{eff}-calculus kinds and types

Value kinding $\boxed{\vdash_k A : \textbf{Val}}$

$$\frac{}{\vdash_k 1 : \textbf{Val}} \qquad \frac{\vdash_k A_1 : \textbf{Val} \qquad \vdash_k A_1 : \textbf{Val}}{\vdash_k A_1 \times A_2 : \textbf{Val}} \qquad \frac{}{\vdash_k 0 : \textbf{Val}}$$

$$\frac{\vdash_k A_1 : \textbf{Val} \qquad \vdash_k A_1 : \textbf{Val}}{\vdash_k A_1 + A_2 : \textbf{Val}} \qquad \frac{\vdash_k E : \textbf{Eff} \qquad \vdash_k C : \textbf{Comp}_E}{\vdash_k U_E C : \textbf{Val}}$$

Effect kinding $\boxed{\vdash_k E : \textbf{Eff}}$

$$\frac{\vdash_k A : \textbf{Val} \qquad \vdash_k B : \textbf{Val} \qquad op \notin E \qquad \vdash_k E : \textbf{Eff}}{\vdash_k \{op : A \to B\} \uplus E : \textbf{Eff}} \qquad \frac{}{\vdash_k \emptyset : \textbf{Eff}}$$

Computation kinding $\boxed{\vdash_k C : \textbf{Comp}_E}$

$$\frac{\vdash_k A : \textbf{Val} \qquad \vdash_k E : \textbf{Eff}}{\vdash_k FA : \textbf{Comp}_E} \qquad \frac{\vdash_k A : \textbf{Val} \qquad \vdash_k C : \textbf{Comp}_E}{\vdash_k A \to C : \textbf{Comp}_E} \qquad \frac{}{\vdash_k \top : \textbf{Comp}_E}$$

$$\frac{\vdash_k C_1 : \textbf{Comp}_E \qquad \vdash_k C_2 : \textbf{Comp}_E}{\vdash_k C_1 \mathbin{\&} C_2 : \textbf{Comp}_E}$$

Context kinding $\boxed{\vdash_k \Gamma : \textbf{Ctxt}}$

$$\frac{\text{for all } x \in \text{Dom }(\Gamma)\text{:} \;\; \vdash_k \Gamma(x) : \textbf{Val}}{\vdash_k \Gamma : \textbf{Ctxt}}$$

Handler kinding $\boxed{\vdash_k X : \textbf{Hndlr}}$

$$\frac{\vdash_k \Gamma : \textbf{Ctxt} \qquad \vdash_k A : \textbf{Val} \qquad \vdash_k E, E' : \textbf{Eff} \qquad \vdash_k C : \textbf{Comp}_{E'}}{\vdash_k A \;{}^{E}\!\!\Rightarrow^{E'}\, C : \textbf{Hndlr}}$$

Figure 3.4: λ_{eff}-calculus kinding rules

Value typing $\boxed{\Gamma \vdash V : A}$ where $\vdash_k \Gamma : \mathbf{Ctxt}$ and $\vdash_k A : \mathbf{Val}$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2}$$

$$\frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : U_E C}$$

Computation typing $\boxed{\Gamma \vdash_E M : C}$ where $\vdash_k \Gamma : \mathbf{Ctxt}$ and $\vdash_k C : \mathbf{Comp}_E$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C} \qquad \frac{\Gamma \vdash V : 0}{\Gamma \vdash_E \mathbf{case_0}(V) : C}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_E M_1 : C \qquad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C} \qquad \frac{\Gamma \vdash V : U_E C}{\Gamma \vdash_E V! : C}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash_E \mathbf{return}\ V : FA} \qquad \frac{\Gamma \vdash_E M : FA \qquad \Gamma, x : A \vdash_E N : C}{\Gamma \vdash_E \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C} \qquad \frac{\Gamma, x : A \vdash_E M : C}{\Gamma \vdash_E \lambda x.M : A \to C}$$

$$\frac{\Gamma \vdash_E M : A \to C \qquad \Gamma \vdash V : A}{\Gamma \vdash_E M\,V : C} \qquad \frac{}{\Gamma \vdash_E \langle\rangle : \top} \qquad \frac{\Gamma \vdash_E M_1 : C_1 \qquad \Gamma \vdash_E M_2 : C_2}{\Gamma \vdash_E \langle M_1, M_2 \rangle : C_1 \,\&\, C_2}$$

$$\frac{\Gamma \vdash_E M : C_1 \,\&\, C_2}{\Gamma \vdash_E \mathbf{prj}_i\, M : C_i} \qquad \frac{(\mathrm{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathrm{op}\ V(\lambda x.M) : C}$$

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

Handler typing $\boxed{\Gamma \vdash H : A \ {}^{E}\!\!\Rightarrow^{E'}\ C}$ where

$$\frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathrm{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \qquad \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \ {}^{E}\!\!\Rightarrow^{E'}\ C}$$

Figure 3.5: λ_{eff}-calculus type system

Handlers H define how to proceed when an operation is encountered during the evaluation of a program. They have clauses $\mathsf{H}^{\mathsf{op}_i} = \lambda p_i\, k_i.\mathsf{N}_i$ for every possible operation op_i, defining how to deal with operations that occur with parameter p_i and continuation k_i (rule *handle*.op). The idea is that (if we ignore the second argument of operations for a moment) when an operation $\mathsf{op}_i\ \mathsf{V}$ is encountered inside a context $\mathcal{H}$ the evaluation proceeds with the matching handling term N_i, where the parameter p_i is bound to V and the continuation k_i is set to the continuation where $\mathsf{op}_i\ \mathsf{V}$ in $\mathcal{H}$ is replaced by the passed value:

$$\textbf{handle}\ \mathcal{H}[\mathsf{op}_i \mathsf{V}]\ \textbf{with}\ \mathsf{H} \longrightarrow^* \mathsf{N}_i[\mathsf{V}\ \mathsf{p}_i, \{\lambda x.\textbf{handle}\ \mathcal{H}[\textbf{return}\ x]\ \textbf{with}\ \mathsf{H}\}/\mathsf{k}_i].$$

If N_i then invokes k_i with an argument, the evaluation gets resumed at the point where the operation was called. Our handlers are deep, meaning the resumed computation gets executed under the presence of the same handler. If N_i does not use k_i, the previous computation is just discarded.

The actual formalisation generalises this idea a bit, to ease implementations of actual effects and the definition of the small step semantics. Operations have a second argument $\lambda x.\mathsf{N} : \mathsf{B} \to \mathsf{C}$, carrying the current continuation. This allows us to define $\mathcal{H}[\mathsf{op}\mathsf{V}(\lambda \mathsf{x}.\mathsf{N})] \longrightarrow \mathsf{op}\mathsf{V}(\lambda \mathsf{x}.\mathcal{H}[\mathsf{N}])$. The new construct generalises the previous construct by defining $\widehat{\mathsf{op}}\ \mathsf{V} := \mathsf{op}\ \mathsf{V}\ (\lambda x.\textbf{return}\ x)$. Plotkin and Power call the simplified form of operations the **generic effect** of op [19]. They are equivalent to our operations, because we can set $\mathsf{op}\ \mathsf{V}(\lambda x.\mathsf{N}) := \textbf{let}\ x \leftarrow \widehat{\mathsf{op}}\ \mathsf{V}\ \textbf{in}\ \mathsf{N}$.

Furthermore, following [2] we enforce every handler H to have a **return clause** $\mathsf{H}^{\textbf{return}} \equiv \lambda \mathsf{x}.\mathsf{M}$ that defines how the term **handle return** V **with** H proceeds (rule *handle*.F). This will be useful for implementing certain effects like store.

Finally, the typing judgement for λ_{eff} is parameterised by a set E of effect operations that may occur during evaluation and so is the type U of thunked computations. Effect operations op have an **arity** $\mathsf{A} \to \mathsf{B}$. The idea is that they can be invoked with a **parameter** of type A and the computation might resume later with a value of type B.

3.2 Contextual equivalence

Similar to CBPV we define contextual equivalence for λ_{eff}. We first extend the contexts of CBPV to contexts of λ_{eff} by adding the contexts in figure 3.6. Figure 3.7 shows how to type them. The typing judgment for λ_{eff}-contexts is, similar to the judgment for terms, parameterised by a set E of effects and a set E' of allowed effects for the term to be inserted.

The definition of contextual equivalence is then a generalisation that also accounts

For the omitted parts see CBPV contexts in figure 2.7.

$$
\begin{array}{lll}
\text{(value contexts)} & \mathcal{X}_{\text{val}} & ::= \ldots \\
\\
\text{(computation contexts)} & & \\
 & \mathcal{X}_{\text{comp}} & ::= \ldots \\
 & & \mid \ \mathrm{op}\, \mathcal{X}_{\text{val}}\, (\lambda x.M) \mid \mathrm{op}\, V\, (\lambda x.\mathcal{X}_{\text{comp}}) \\
 & & \mid \ \mathbf{handle}\ \mathcal{X}_{\text{comp}}\ \mathbf{with}\ H \\
 & & \mid \ \mathbf{handle}\ M\ \mathbf{with}\ \mathcal{X}_{\text{han}} \\
\text{(handler contexts)} & & \\
 & \mathcal{X}_{\text{han}} & ::= [\ ] \\
 & & \mid \ \left\{\mathbf{return}\ x \mapsto \mathcal{X}_{\text{comp}}\right\} \mid H \uplus \{\mathrm{op}\, p\, k \mapsto N\} \\
 & & \mid \ \{\mathbf{return}\ x \mapsto M\} \mid H \uplus \left\{\mathrm{op}\, p\, k \mapsto \mathcal{X}_{\text{comp}}\right\}
\end{array}
$$

Figure 3.6: λ_{eff}-calculus contexts

for effect signatures:

Definition 3.1 (contextual equivalence)**.** *Let* $\Gamma \vdash_{E'} P, Q : X$ *be two* λ_{eff} *phrases. We say that* P *and* Q *are* ***contextually equivalent****, and write* $P \simeq^{E} Q$ *when, for all* ***closed*** *well-typed* ***ground-returner*** *contexts* $\emptyset[\Gamma'] \vdash_{E[E']} \mathcal{X}[\ ] : FG[X]$ *with* $\Gamma' \geqslant \Gamma$ *and for all closed ground value terms* $\vdash V : G$*, we have:*

$$\mathcal{X}[P] \longrightarrow^{\star} \mathbf{return}\ V \qquad \iff \qquad \mathcal{X}[Q] \longrightarrow^{\star} \mathbf{return}\ V$$

We write $P \simeq Q$ *for* $P \simeq^{\emptyset} Q$ *(i.e. for* ***closed, effect-free*** *ground-returners in the definition).*

We prove the following lemma as sanity check for our definition:

Lemma 3.2. *For all* $\Gamma \vdash_E P, Q : X$, $P \simeq Q$ *iff* $\forall E, P \simeq^{E} Q$.

Proof

The $\Leftarrow$-direction is immediate. For the other direction, let $E = \{\mathrm{op}_i : A_i \to B_i\}_{1 \leqslant i \leqslant n}$ be an effect signature and consider the context $\mathcal{Y} := \emptyset[\Gamma] \vdash_{\emptyset[E]} \mathbf{handle}\ [\ ]\ \mathbf{with}\ H : F(G + \mathbb{F}_n)[X]$ where $H^{\mathbf{return}} = \lambda x.\mathbf{return}\ (\mathbf{inj}_1\, x)$ and $H^{\mathrm{op}_i} = \lambda p, k.\mathbf{return}\ (\mathbf{inj}_2\, i_n)$. Recall that $\mathbb{F}_n$ is the finite type with exactly n elements $0_n, \ldots, (n-1)_n$.

Now assume $\mathcal{X}[P] \longrightarrow^{*} \mathbf{return}\ V$. This means that $\mathcal{Y}[\mathcal{X}[P]] \longrightarrow^{*} \mathbf{return}\ (\mathbf{inj}_1\, V)$. Thus, $\mathcal{Y}[\mathcal{X}[Q]] \longrightarrow^{*} \mathbf{return}\ (\mathbf{inj}_1\, V)$, because $\mathcal{Y}[\mathcal{X}[\ ]]$ is an effect free ground returner context. But this can only be the case if $\mathcal{X}[Q] \longrightarrow^{*} \mathbf{return}\ V$. ■

Lemma 3.3 (basic contextual equivalence properties)**.** *The relation* $\simeq$ *is an equivalence relation that is congruent w.r.t.* λ_{eff} *contexts, and contains the reduction relation* $\longrightarrow$.

Context typing $\boxed{\Gamma[\Delta] \vdash_{E[E']} \mathcal{X}[\ ] : C[X]}$

$$\frac{}{\Gamma[\Gamma] \vdash_{E[E]} [\ ] : X[X]}$$

Value context typing see figure 2.8, where just the rule for thunks is replaced by:

$$\frac{\Gamma[\Delta] \vdash_{E[E']} \mathcal{X}[\ ] : C[X]}{\Gamma[\Delta] \vdash \{\mathcal{X}[\ ]\} : U_E C[X]}$$

Computation context typing see figure 2.8 and:

$$\frac{(\mathsf{op} : A \to B) \in E \qquad \Gamma[\Delta] \vdash_{E[E']} \mathcal{X}[\ ] : A[X] \qquad \Gamma, x : B \vdash_E M : C}{\Gamma[\Delta] \vdash_{E[E']} \mathsf{op}\,\mathcal{X}[\ ]\,(\lambda x.M) : C[X]}$$

$$\frac{(\mathsf{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B[\Delta] \vdash_E \mathcal{X}[\ ] : C[X]}{\Gamma[\Delta] \vdash_{E[E']} \mathsf{op}\,V\,(\lambda x.\mathcal{X}[\ ]) : C[X]}$$

$$\frac{\Gamma \vdash_E M : FA \qquad \Gamma[\Delta] \vdash_{[E'']} H : A \ {}^{E}\!\!\Rightarrow^{E'} C[X]}{\Gamma[\Delta] \vdash_{E'[E'']} \textbf{handle}\ M\ \textbf{with}\ \mathcal{X}_{\mathrm{han}}[\ ] : C[X]}$$

$$\frac{\Gamma[\Delta] \vdash_{E[E'']} \mathcal{X}_{\mathrm{comp}}[\ ] : FA[X] \qquad \Gamma \vdash H : A \ {}^{E}\!\!\Rightarrow^{E'} C}{\Gamma[\Delta] \vdash_{E'[E'']} \textbf{handle}\ \mathcal{X}_{\mathrm{comp}}[\ ]\ \textbf{with}\ H : C[X]}$$

Handler context typing $\boxed{\Gamma[\Delta] \vdash_{[E'']} H : A \ {}^{E}\!\!\Rightarrow^{E'} C[X]}$

$$\frac{\begin{array}{c} E = \{\mathsf{op}_i : A_i \to B_i\}_i \\ H = \left\{\textbf{return}\ x \mapsto \mathcal{X}_{\mathrm{comp}}[\ ]\right\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \qquad \Gamma, x : A[\Delta] \vdash_{E'} \mathcal{X}_{\mathrm{comp}}[\ ] : C[X] \end{array}}{\Gamma[\Delta] \vdash_{[E'']} H : A \ {}^{E}\!\!\Rightarrow^{E'} C[X]}$$

$$\frac{\begin{array}{c} E = \{\mathsf{op}_i : A_i \to B_i\}_i \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i\}_i \uplus \left\{\mathsf{op}\ p\ k \mapsto \mathcal{X}_{\mathrm{comp}}[\ ]\right\} \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \\ \Gamma, p : A, k : U_{E'}(B \to C)[\Delta] \vdash_{E'} \mathcal{X}_{\mathrm{comp}}[\ ] : C[X] \qquad \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma[\Delta] \vdash_{[E'']} H : A \ {}^{E}\!\!\Rightarrow^{E'} C[X]}$$

Figure 3.7: λ_{eff}-calculus context types

3.3 Programming in λ_{eff}

We demonstrate how to program with effect handlers by implementing exception handlers and a global store handler.

3.3.1 Exceptions in $\lambda_{\textit{eff}}$

One way to implement exceptions is to give terms `raise` and `try` that can be used in the following style:

$$\texttt{try}\{1 + \texttt{raise}\ \text{"number"}\}\{\lambda s.\textbf{if}\ s = \text{"number"}\ \textbf{then}\ 0\ \textbf{else}\ 1\}$$

That is, we want to have terms

$$\begin{aligned} \texttt{raise} &: \text{string} \to FA \\ \texttt{try} &: U_{\{\text{exc : string} \to 0\}} FA \to U_E(\text{string} \to FA) \to FA \end{aligned}$$

for every type A and effect signature E with exc $\notin E$, where exc : string $\to 0$.

The general idea is that $\texttt{try}\{M\}N \simeq \texttt{try}\{M'\}N$ if $M \longrightarrow M'$ and $\texttt{try}\{\textbf{return}\ V\}\ M \simeq \textbf{return}\ V$, but for any stacked hoisting frame $\mathcal{H}^*$ we have $\texttt{try}\{\mathcal{H}^*[\texttt{raise}\ s]\}\ M \simeq M!\ s$. With stacked hoisting frame we mean $\mathcal{H}^* ::= [\ ] \mid \mathcal{H}[\mathcal{H}^*]$.

We will implement `raise` using the operation exc :

$$\texttt{raise} := \lambda s.\textbf{let}\ x \leftarrow \text{exc}\ s\ (\lambda x.\textbf{return}\ x)\ \textbf{in}\ \textbf{case}_\mathbf{0}(x)$$

Consequently, `try` handles the effect:

$$\texttt{try} := \lambda c\ h.\textbf{handle}\ c!\ \textbf{with}\ H$$

where $H^{\textbf{return}} := \lambda x.\textbf{return}\ x$ and $H^{\text{exc}} := \lambda s\ k.h!\ s$. That means, whenever no operation is encountered, the computation just proceeds, but when an operation is encountered, its argument is passed to the second argument, the exception handler h.

We have

$$\begin{aligned} \texttt{try}\{\textbf{return}\ V\}\ M &= \textbf{handle}\ \{\textbf{return}\ V\}!\ \textbf{with}\ H \\ &\longrightarrow \textbf{handle return}\ V\ \textbf{with}\ H \\ &\longrightarrow \textbf{return}\ V \end{aligned}$$

and

$$\begin{aligned}
\texttt{try}\{\mathcal{H}^*[\texttt{raise s}]\}\, M &= \textbf{handle } \{\mathcal{H}^*[\texttt{raise s}]\}! \textbf{ with } H \\
&\longrightarrow \textbf{handle } \mathcal{H}^*[\texttt{raise s}] \textbf{ with } H \\
&= \textbf{handle } \mathcal{H}^*[\textbf{let } x \leftarrow \text{exc} \quad s(\lambda x.\textbf{return } x) \textbf{ in case}_0(x)] \textbf{ with } H \\
&\longrightarrow \textbf{handle } \text{exc } s\ (\lambda x.\mathcal{H}^*[\textbf{let } y \leftarrow \textbf{return } x \textbf{ in case}_0(x)]) \textbf{ with } H \\
&\longrightarrow M!s[\lambda x.\textbf{handle } \mathcal{H}^*[\textbf{let } y \leftarrow \textbf{return } x \textbf{ in case}_0(x)]) \textbf{ with } H/k] \\
&= M!s
\end{aligned}$$

as wanted. Note that because $M!s$ does not mention k, the continuation is just ignored.

3.3.2 Global store in λ_{eff}

We implement a single cell of global storage by giving terms

$$\begin{aligned}
\texttt{withst} &: A \to U_{\{\texttt{get},\texttt{set}\}}C \to C \\
\texttt{get} &: 1 \to FA \\
\texttt{set} &: A \to F1
\end{aligned}$$

for an arbitrary value type A. `get` and `set` are just generic effect for the operations `get` and `set`. We wrap the entire program with the `withst` construct to simulate this global state. The handler now simply translates the effectful computation to a pure computation:

$$\begin{aligned}
\texttt{withst}\ V\ N &:= (\textbf{handle } N! \textbf{ with } H)V \\
H^{\textbf{return}} &:= \lambda x.\lambda\_.\textbf{return } x \\
H^{\texttt{get}} &:= \lambda\_\ k.\lambda s.(k s)s \\
H^{\texttt{set}} &:= \lambda s\ k.\lambda\_.(k())s \\
\texttt{get ()} &:= \texttt{get ()}\ (\lambda x.\textbf{return } x) \\
\texttt{set } s &:= \texttt{set } s\ (\lambda x.\textbf{return } x)
\end{aligned}$$

The correctness properties are:

$$\begin{aligned}
\texttt{withst}\ V\,\{\textbf{return } V'\} &\simeq \textbf{return } V' \\
\texttt{withst}\ V\,\{M\} &\simeq \texttt{withst}\ V\,\{M\}' \quad \text{if } M \longrightarrow M' \\
\texttt{withst}\ V\,\{\mathcal{H}^*[\texttt{get }])\} &\simeq \texttt{withst}\ V\,\{\mathcal{H}^*[\textbf{return } V]\} \\
\texttt{withst}\ V\ (\mathcal{H}^*[\texttt{set } V']) &\simeq \texttt{withst}\ V'\ (\mathcal{H}^*[\textbf{return }()])
\end{aligned}$$

The first two properties follow immediately from the definition. The third and fourth property are also straightforward to prove:

$$\begin{aligned}
\texttt{withst}\ V\,\{\mathcal{H}^*[\texttt{get}\ ])\} &= (\textbf{handle}\ \{\mathcal{H}^*[\texttt{get}\ ]\}!\ \textbf{with}\ H)V \\
&\longrightarrow (\textbf{handle}\ \mathcal{H}^*[\texttt{get}\ ]\ \textbf{with}\ H)V \\
&\longrightarrow^* (\lambda s.(\lambda x.\textbf{handle}\ \mathcal{H}^*[\textbf{return}\ x]\ \textbf{with}\ H)ss)V \\
&\longrightarrow^* (\textbf{handle}\ \mathcal{H}^*[\textbf{return}\ V]\ \textbf{with}\ H)V \\
&\longleftarrow \texttt{withst}\ V\,\{\mathcal{H}^*[\textbf{return}\ V]\}
\end{aligned}$$

$$\begin{aligned}
\texttt{withst}\ V\ (\mathcal{H}^*[\texttt{set}\ V']) &= (\textbf{handle}\ \{\mathcal{H}^*[\texttt{set}\ V']\}!\ \textbf{with}\ H)V \\
&\longrightarrow (\textbf{handle}\ \mathcal{H}^*[\texttt{set}\ V']\ \textbf{with}\ H)V \\
&\longrightarrow^* (\lambda\_.(\lambda x.\textbf{handle}\ \mathcal{H}^*[\textbf{return}\ x]\ \textbf{with}\ H)()V')V \\
&\longrightarrow^* (\textbf{handle}\ \mathcal{H}^*[\textbf{return}\ ()]\ \textbf{with}\ H)V' \\
&\longleftarrow \texttt{withst}\ V'\ (\mathcal{H}^*[\textbf{return}\ ()])
\end{aligned}$$

3.4 Denotational semantics for λ_{eff}

We give a set-theoretic denotational semantics for λ_{eff}.

The semantics assigns to each well kinded:

- value type $\vdash_k A : \textbf{Val}$ a set $[\![A]\!]$;
- effect $\vdash_k E : \textbf{Eff}$ a (parameterised) signature $[\![E]\!]$;
- computation type $\vdash_k C : \textbf{Comp}_E$ a $[\![E]\!]$-algebra $[\![C]\!]$;
- context $\vdash_k \Gamma : \textbf{Ctxt}$ a set $[\![\Gamma]\!]$; and
- handler $\vdash_k X : \textbf{Hndlr}$ an algebra and a function into the carrier of that algebra.

We use the CBPV denotations (see figure 2.5) for value types and contexts, where $[\![U_E C]\!]$ is the carrier for the $[\![E]\!]$-algebra $[\![C]\!]$ (which is the same carrier as for the corresponding $T_{[\![E]\!]}$-algebra).

Effect signatures $\vdash_k E : \textbf{Eff}$ are assigned a (parameterised) signature $[\![E]\!]$ in the sense of 2.3:

$$[\![\{\mathrm{op} : A \to B\} \uplus E]\!] := \big\langle \{\mathrm{op}\} \cup |[\![E]\!]|, \mathit{arity}_{[\![E]\!]}[\mathrm{op} \mapsto \langle [\![A]\!], [\![B]\!] \rangle] \big\rangle \qquad [\![\emptyset]\!] := \langle \emptyset, ¡ \rangle$$

Recall that our CBPV semantics was parameterised by a monad T. This comes in handy now. For a computation with possible effects E (i.e. $\vdash_k C : \textbf{Comp}_E$) we use the CBPV denotation with $T = T_{[\![E]\!]}$, associating to $\vdash_k FA : \textbf{Comp}_E$ the free algebra

$F_{T_{[\![E]\!]}}[\![A]\!]$. Formally, we will use an $[\![E]\!]$ algebra as denotation, which is sound due to the observation following def. 2.6 that $T_{[\![E]\!]}$ and E algebras are isomorphic, as shown in Figure 2.5.

We postpone the denotational semantics for handler types for now and explain it after the introduction of denotational semantics for terms.

We again give denotations to **derivations** of types. Value derivations again denote functions, and E-computation derivations denote functions into the carrier of an $[\![E]\!]$-algebra.

For the CBPV fragment of the language we use the same definitions as Figure 2.6. As the bijection between $[\![E]\!]$-algebras and $T_{[\![E]\!]}$-algebras acts as the identity on the carrier set, this semantics is well-defined.

The semantics of an effect operation (in an effect signature E) is simply this operation considered as an element of the $T_{[\![E]\!]}$-algebra:

$$[\![\mathrm{op}\ V(\lambda x.M)]\!]\,(\gamma) := \mathrm{op}_{[\![V]\!]\gamma}(\lambda b : [\![B]\!]\,.\,[\![M]\!]\,(\gamma[x \mapsto b]))$$

where $\mathrm{op} : A \to B \in E$.

The denotation of a handling construct is the Kleisli-extended denotation of the return-clause applied to the denotation of the handled term:

$$[\![\textbf{handle}\ M\ \textbf{with}\ H]\!]\,(\gamma) := [\![M]\!]\,(\gamma) \gg\!= f$$

where $[\![H]\!]\,(\gamma) = \langle D, f : [\![A]\!] \to |[\![C]\!]|\rangle$ and the Kleisli extension is with respect to the $[\![E]\!]$-algebra structure D given on the carrier of the $[\![E']\!]$-algebra denoted by C.

Finally, we define the semantics of handlers. The semantics for handlers needs to contain the semantics of the return clause, i.e. a function $[\![A]\!] \to [\![C]\!]$. Furthermore, in order to define the needed Kleisli-extension we need an algebra over E. The interpretation of each symbol in this algebra is given by all the handling clauses.

Thus, we define the semantics of handler types to be pairs of algebras and return-clause denotations:

$$\left[\!\!\left[A\ {}^{E}\!\!\Rightarrow^{E'}\ C\right]\!\!\right] := \sum_{[\![E]\!]\text{-algebras with carrier } |[\![C]\!]|} |[\![C]\!]|^{[\![A]\!]}$$

The algebra for a given handler consists of the set $|[\![C]\!]|$ and an interpretation according to the clauses for operation handling.

Each handler term $\Gamma \vdash H : X$ thus denotes a function from $[\![\Gamma]\!]$ into $[\![X]\!]$. Given a well-typed handler term $\Gamma \vdash H : A\ {}^{E}\!\!\Rightarrow^{E'}\ C$, matching the single rule for deriving this judgement, and any $\gamma \in [\![\Gamma]\!]$, define an $[\![E]\!]$-algebra structure on $[\![C]\!]$ by setting:

$$[\![\mathrm{op}_i]\!]\,(\gamma)(k \in |[\![C]\!]|^{[\![B_i]\!]})(p \in [\![A_i]\!]) := [\![N_i]\!]\,(\gamma[p \mapsto p, k \mapsto k]) \in |[\![C]\!]|$$

We therefore have a $[\![E]\!]$-algebra structure $D_\gamma = \langle |[\![C]\!]|, [\![-]\!]\,(\gamma)\rangle$. Define a function $f_\gamma : [\![A]\!] \to |[\![C]\!]|$ by:

$$f_\gamma(a \in [\![A]\!]) := [\![M]\!]\,(\gamma[x \mapsto a])$$

Thus we define the semantics of the handler term as follows:

$$[\![H]\!]\,(\gamma) := \langle D_\gamma, f_\gamma\rangle \in \sum_{[\![E]\!]\text{-algebras with carrier } |[\![C]\!]|} |[\![C]\!]|^{[\![A]\!]}$$

3.4.1 Adequacy proof

We only changed the denotational semantics for the CBPV part of λ_{eff} up to isomorphism. We can therefore reuse all of the proofs for CBPV and only have to prove the additional cases.

Lemma 3.4 (substitution). *For all* $\Gamma \vdash P : X$ *and* $\langle V_x\rangle_{x\in\mathrm{Dom}(\Gamma)}$ *such that, for all* $x \in \mathrm{Dom}\,(\Gamma)$, $\Delta \vdash V_x : \Gamma(x)$, *we have* $\Delta \vdash P[V_x/x]_{x\in\mathrm{Dom}(\Gamma)} : X$.

Lemma 3.5 (context substitution). *For all* $\Delta[\Gamma'] \vdash_{E[E']} \mathcal{X}[\ ] : Y[X]$*:*

1. *For all* $\Gamma \vdash_{E'} P : X$ *where* $\Gamma' \geqslant \Gamma$, *we have* $\Delta \vdash_E \mathcal{X}[P] : Y$.
2. *For all* $\Gamma \vdash_{E'} P, Q : X$ *where* $\Gamma' \geqslant \Gamma$, *we have* $[\![P]\!] = [\![Q]\!]$ *implies* $[\![\mathcal{X}[P]]\!] = [\![\mathcal{X}[Q]]\!]$.

For every value type A, define $\boldsymbol{\lambda_{\text{eff}}}_A := \{V \mid \vdash V : A\}$ and for every computation type C and effect signature E: $\boldsymbol{\lambda_{\text{eff}}}_{C,E} = \{M \mid \vdash_E M : C\}$. Define **handlers**$(A\ {}^{E}\!\!\Rightarrow^{E'}\ C) := \{H \mid \vdash H : A\ {}^{E}\!\!\Rightarrow^{E'}\ C\}$.

We define logical relations indexed by λ_{eff} types and effect signatures as in Figure 3.8. The value relations are exactly the same as for CBPV. The computation relations are supersets of those for CBPV, as we now add effect operations to every relation. Handler relations merely state that the operation clauses and the return clause preserve the relation.

Lemma 3.6. *The logical relations* R_A *and* $R_{E,C}$ *are closed under contextual equivalence. Explicitly: For all* $\langle a, P\rangle \in R_{E,X}$ *and* $\vdash Q : X$, $P \simeq Q$ *implies* $\langle a, Q\rangle \in R_{E,X}$ *and similarly for value relations.*

Proof

By induction over X. ■

Value relations $\boxed{R_A \subseteq [\![A]\!] \times \boldsymbol{\lambda}_{\textbf{eff}\,A}}$

$$R_1 := \{\langle \star, V\rangle | V \simeq ()\}$$

$$R_{A_1 \times A_2} := \left\{ \langle \langle a_1, a_2\rangle, V\rangle \middle| \exists V_1, V_2 \forall i. V_i : A_i. \langle a_i, V_i\rangle \in R_{A_i}, V \simeq (V_1, V_2) \right\} \qquad R_0 := \emptyset$$

$$R_{A_1 + A_2} = \bigcup_{i \in \{1,2\}} \left\{ \langle \iota_i a, V\rangle \middle| \exists V' : A_i. \langle a, V'\rangle \in R_{A_i}, V \simeq \mathbf{inj}_i\, V' \right\}$$

$$R_{U_E C} := \{\langle a, V\rangle | \exists M : C. \langle a, M\rangle \in R_{E,C}, V \simeq \{M\}\}$$

Computation relations $\boxed{R_{E,C} \subseteq |[\![C]\!]| \times \boldsymbol{\lambda}_{\textbf{eff}\,E,C}}$

$$\begin{aligned}
R_{E,FA} &:= R'_{E,FA} \cup \{\langle \mathbf{return}\ a, M\rangle | \exists V : A. \langle a, V\rangle \in R_A, M \simeq \mathbf{return}\ V\} \\
R_{E,A\to C} &:= R'_{E,A\to C} \cup \{\langle f, M\rangle | \forall \langle a, V\rangle \in R_{E,A}, \langle f(a), M\ V\rangle \in R_{E,C}\} \\
R_{E,\top} &:= R'_{E,\top} \cup \{\langle \star, M\rangle | M \simeq \langle\rangle\} \\
R_{E,C_1 \& C_2} &:= R'_{E,C_1\&C_2} \cup \left\{ \langle \langle c_1, c_2\rangle, M\rangle \middle| \exists M_1 : C_1, M_2 : C_2.M \simeq \langle M_1, M_2\rangle, \forall i. \langle c_i, M_i\rangle \in R_{E,C_i} \right\}
\end{aligned}$$

where

$$\begin{aligned}
R'_{E,C} := \{ & \\
& \langle op_a(\lambda b.c), N\rangle \mid \exists VM. \\
& \langle a, V\rangle \in R_{E,A} \quad (op : A \to B) \in E \\
& \langle \lambda b.c, \lambda x.M\rangle \in R_{E,B\to C} \\
& N \simeq op\ V\ (\lambda x.M) \\
\} &
\end{aligned}$$

Handler relations $\boxed{R_{A\ ^E\!\Rightarrow^{E'} C} \subseteq [\![A\ ^E\!\Rightarrow^{E'} C]\!] \times \mathbf{handlers}(A\ ^E\!\Rightarrow^{E'} C)}$

$$\begin{aligned}
R_{A\ ^E\!\Rightarrow^{E'} C} := \{ & \\
& \langle \langle D_\gamma, f_\gamma\rangle, H\rangle \mid \\
& (op_i : A \to B) \in E, \quad D_\gamma = \langle |[\![C]\!]|, [\![-]\!]\,(\gamma)\rangle, \\
& \forall i. \exists M N_i. \langle f_\gamma, \lambda x.M\rangle \in R_{E',A\to C} \wedge, \\
& \forall \langle a, V\rangle \in R_{E,A}, \langle k, \lambda x.M'\rangle \in RB \to C. \\
& \left\langle [\![op_i]\!]\,(\gamma)\,(\lambda x.kx \gg\!= f)\ a, N_i[V/p, (\lambda x.\mathbf{handle}\ M'\ \mathbf{with}\ H)/k] \right\rangle \\
\} &
\end{aligned}$$

Figure 3.8: λ_{eff}-calculus logical relations

We restate Lemma 2.16 here, where the proof is unchanged because the ground types and relations for ground types are unchanged:

Lemma 3.7. *For all ground types* G, $a \in [\![G]\!]$, *and closed value terms* $\vdash V :, V' : G$*:*

$$\langle a, V\rangle, \langle a, V'\rangle \in R_G \implies V \simeq V'$$

Lemma 3.8 (basic lemma)**.** *For all* $\Gamma \vdash_E P : X$, $\gamma \in [\![\Gamma]\!]$ *and* $\langle V_x\rangle_{x \in \mathrm{Dom}(\Gamma)}$*:*

$$(\textit{for all } x \in \mathrm{Dom}\ (\Gamma)\textit{: } \langle \pi_x\gamma, V_x\rangle \in R_{E,\Gamma(x)}) \implies \left\langle [\![P]\!]\,\gamma, P[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,X}$$

Proof

We only prove the new additional computation cases for λ_{eff}:

(op) Assume $(\mathrm{op} : A \to B) \in E$, $\Gamma \vdash V : A$, $\left\langle [\![V]\!]\,\gamma, V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,A}$ and $\Gamma \vdash \lambda x.M : B \to C$.

We have to show that $\left\langle [\![\mathrm{op}V(\lambda x.M)]\!]\,\gamma, \mathrm{op}V(\lambda x.M)[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,C}$. This is the case iff R_C contains

$$\left\langle \mathrm{op}([\![V]\!]\,\gamma)\,(\lambda b : [\![B]\!].M(\gamma[x \mapsto b]))\,\gamma, \mathrm{op}(V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})(\lambda x.M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle.$$

By the definition of the logical relations, this boils down to showing that $\left\langle [\![V]\!]\,\gamma, V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,A}$, $(\mathrm{op} : A \to B) \in E$ and

$\left\langle \lambda b : [\![B]\!].M(\gamma[x \mapsto b]), \lambda x.M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,B\to C}$. Everything follows immediately from the assumptions (with the observation that $\lambda b : [\![B]\!].M(\gamma[x \mapsto b]) = [\![\lambda x.M]\!]\,\gamma$).

(handle) Assume $\Gamma \vdash M : FA$, $\left\langle [\![M]\!]\,\gamma, M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E,FA}$, $\Gamma \vdash H : A\ {}^{E}\!\!\Rightarrow^{E'} C$ and $\left\langle [\![H]\!]\,\gamma, H[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{A^{E}\Rightarrow^{E'}C}$. Let $[\![H]\!]\,(\gamma) = \langle D, f\rangle$. We have to show that $\left\langle [\![\textbf{handle}\ M\ \textbf{with}\ H]\!]\,\gamma, \textbf{handle}\ M\ \textbf{with}\ H[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E',C}$.

By definition of $R_{E,FA}$, either $M \simeq \textbf{return}\ V$ or $M \simeq \mathrm{op}V(\lambda x.M')$. The first case is similar to the let-case of CBPV. We only show the second case here, where we also know that $[\![M]\!]\,\gamma = \mathrm{op}_a(\lambda b.c)$, $\langle a, V\rangle \in R_{E,A}$ and $\langle \lambda b.c, \lambda x.M'\rangle \in R_{E,B\to C}$.

We have

$$\left\langle [\![\textbf{handle}\ M\ \textbf{with}\ H]\!]\,\gamma, \textbf{handle}\ M\ \textbf{with}\ H[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E',C}$$

$$\iff \left\langle \mathrm{op}_a(\lambda b.c) \gg\!=_D f, \textbf{handle}\ \mathrm{op}\ (V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\ (\lambda x.M'[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\ \textbf{with}\ H[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E',C}$$

$$\Longleftarrow \left\langle \mathrm{op}_a(\lambda b.c) \gg\!=_D f, N[V/p, \{\lambda x.\textbf{handle}\ M'\ \textbf{with}\ H\}/k][V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E',C} \quad (\simeq \text{closed})$$

Def. bijection
$\downarrow$

$$\Longleftarrow \left\langle [\![\mathrm{op}]\!]\,(\gamma)\ (\lambda b.cb \gg\!= f)\ a, N[V/p, \{\lambda x.\textbf{handle}\ M'\ \textbf{with}\ H\}/k][V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{E',C}$$

This now holds by the definition of $R_{A^E \Rightarrow^{E'} C}$ and the induction hypothesis for H.

(let) The let-case is very similar to the handle-case, therefore we omit it here.

The case for handler typing follows immediately from the inductive hypotheses. ■

Theorem 3.9 (adequacy). *Denotational equivalence implies contextual equivalence. Explicitly: For all* $\Gamma \vdash_E P, Q : X$, *if* $[\![P]\!] = [\![Q]\!]$ *then* $P \simeq Q$.

Proof

Let $\Gamma \vdash P_1, P_2 : X$ be any well-typed phrases satisfying $[\![P_1]\!] = [\![P_2]\!]$. Consider any closed well-typed ground-returner context $\emptyset[\Gamma'] \vdash \mathcal{X}[\;] : FG[X]$ with $\Gamma' \geqslant \Gamma$. By Lemma 3.5, $[\![\mathcal{X}[P_1]]\!] = [\![\mathcal{X}[P_2]]\!]$. Let c be this common denotation.

Consider any $i \in \{1, 2\}$. By the basic lemma:

$$\langle c, \mathcal{X}[P_i] \rangle \in R_{FG}$$

By R_{FG}'s definition, there exist $\langle a_i, V_i \rangle \in R_G$ such that one of the two cases applies:

- $c = \textbf{return } a_i$ and $\mathcal{X}[P_i] \simeq \textbf{return } V_i$.

  Thus:

  $$a_1 = \textbf{return } a_1 = c = \textbf{return } a_2 = a_2$$

  Therefore, by Lemma 3.7, $V_1 \simeq V_2$ and:

  $$\mathcal{X}[P_1] \simeq \textbf{return } V_1 \simeq \textbf{return } V_2 \simeq \mathcal{X}[P_2]$$

  Thus $\mathcal{X}[P_1] \longrightarrow^\star \textbf{return } V$ iff $\mathcal{X}[P_2] \longrightarrow^\star \textbf{return } V$, hence $P_1 \simeq P_2$.

- or $c = \text{op}_a f$ and $\mathcal{X}[P_i] \simeq \text{op } A\ (\lambda x.M)$ for some $(\text{op} : A \to B) \in \emptyset$, which clearly is a contradiction.

■

Corollary 3.10 (soundness and strong normalisation). *All well-typed, effect-free closed ground returners reduce to a normal form. Explicitly: for all* $\vdash_\emptyset M : FG$ *there exists some* $\vdash V : G$ *such that:*

$$M \longrightarrow^\star \textbf{return } V$$

Proof

By the basic lemma, $\langle [\![M]\!] \gamma), M \rangle \in R_{FG}$, so by definition either $[\![M]\!] = \textbf{return } a$, $M \simeq \textbf{return } V$ for some $\langle a, V \rangle \in R_G$ or $[\![M]\!] = \text{op}_a f$ and $\text{op} \in \emptyset$, which is a contradiction.

As M is a ground-returner, we can instantiate $M \simeq \textbf{return}\ V$ at the empty context $M \longrightarrow^{\star} \textbf{return}\ V$ iff $\textbf{return}\ V \longrightarrow^{\star} \textbf{return}\ V$, and the latter holds by reflexivity.

Note that by Lemma 3.7 and Lemma 2.10, we can choose $V := V_a^G$ ■

Chapter 4

Monadic reflection: λ_{mon}

Filinski [7, 6] introduces **monadic reflection**, a way of incorporating monads into the syntax of a language. With monadic reflection, the user can introduce new effects using a construct called leteffect. This construct lets the user define a monad, based on already existing effects of the language. The language has a **reflect** operation that gives the user the ability to use the definition of the underlying monad to implement an effect. The **reify** operation allows the user to get a representation of a computation in terms of the underlying monad.

Filinski introduces a calculus for monadic reflection featuring recursion and recursive types that is loosely based on CBPV [7]. We make the relation to CBPV more explicit, and simplify it by removing all recursive constructs to obtain the calculus λ_{mon}. This way we ensure that our analysis focuses on monadic reflection only.

4.1 Syntax and operational semantics of λ_{mon}

Figure 4.1 introduces the syntax of the λ_{mon}-calculus. We add a reification construct $[N]^{\varepsilon}$ and a reflection construct $\widehat{\mu}^{\varepsilon}(N)$ to the CBPV base calculus. The construct **leteffect** $\varepsilon \succ e$ **be** $(\alpha.C, N_u, N_b)$ **in** N allows us to define new effects.

Note that following Filinski [7] the leteffect-construct contains a type C, so we have types in the syntax of terms.

We almost exactly take the CBPV types as shown in figure 4.2. The only difference is the type of thunked computations, that is parameterised with a single effect e. We use a single effect here instead of a set of effect operations, as a single monad can implement many effects. We additionally include type variables α in the value types.

Effects can be either user-defined effects ε or the base effect (in our case, no effect at all) $\bot$. Effect hierarchies contain the name of the introduced effect together with the action of types, the return term N_u and the bind term N_b. Filinski calls them

$$
\begin{array}{lrl}
\text{(values)} & V, W & ::= x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\} \\
\text{(computations)} & & \\
 & M, N & ::= \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case_0}(V) \\
 & & \mid\ \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V! \\
 & & \mid\ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
 & & \mid\ \lambda x.M \mid M\ V \\
 & & \mid\ \langle M_1, M_2 \rangle \mid \mathbf{prj}_i\, M \\
 & & \mid\ [N]^{\varepsilon} \mid \widehat{\mu}^{\varepsilon}(N) \\
 & & \mid\ \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ N
\end{array}
$$

Figure 4.1: λ_{mon}-calculus syntax

(effects)	e	$::= \bot \mid \varepsilon$
(value types)	A, B	$::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid U_e C \mid \alpha$
(computation types)	C, D	$::= FA \mid A \to C \mid \top \mid C_1\ \&\ C_2$
(effect hierarchies)	Σ	$::= \cdot \mid \Sigma, \varepsilon \succ e \sim (\alpha.C, N_u, N_b)$
(type environments)	Θ	$::= \alpha_1, \ldots, \alpha_n$
(environments)	Γ	$::= x_1 : A_1, \ldots, x_n : A_n$

Figure 4.2: λ_{mon}-calculus kinds and types

effect signatures, but because they are inherently different from effect signatures in λ_{eff} and the notion from universal algebra, we call them effect hierarchies. As user-defined effects always have to be based on a previously existing effect, we obtain a semi-lattice of effects. The effect hierarchy contains this ordering as $e \prec \varepsilon$.

We will oftentimes only need effect hierarchies with less information. For convenience, we will still write Σ to denote an effect hierarchy that only contains the ordering, or only the mapping of effects to N_u and N_b. We write $\Sigma' \supseteq \Sigma$ if Σ' extends Σ, i.e. if Σ' is a suffix-extension of Σ as a list.

We also add type environments Θ which are finite sets of type variables originating from the leteffect construct.

Figure 4.3 introduces a kind system for λ_{mon}. The judgment for value types is straightforward. We now have multiple computation kinds $\mathbf{Comp}_e$ for each $e \in \Sigma$. Intuitively, e-computations may only invoke the effect e.

Vale type judgments $\Theta \mid \Gamma \vdash_{\Sigma} V : A$ differ from their CBPV counterpart in two ways: They include type variable contexts Θ, and an effect hierarchy Σ, where we require that A is well-kinded under Θ and Σ.

For computation judgments $\Theta \mid \Gamma \vdash_{\Sigma,e} M : C$ we also add an effect $e \in \Sigma$ and we require that $\Theta \mid \Sigma \vdash_k C : \mathbf{Comp}_e$ (as well as the well-kinding of e, Σ and Γ). If we

Effect hierarchy kinding $\boxed{\Theta \vdash_k \Sigma : \mathbf{Eff}}$

$$\frac{}{\Theta \vdash_k \cdot : \mathbf{Eff}} \qquad \frac{\Theta, \alpha \vdash_k C : \mathbf{Comp}_e \qquad \varepsilon \notin \Sigma \qquad \Theta \vdash_k \Sigma : \mathbf{Eff}}{\Theta \vdash_k \Sigma, \varepsilon \succ e \sim (\alpha.C, N_u, N_b) : \mathbf{Eff}}$$

Value kinding $\boxed{\Theta \mid \Sigma \vdash_k A : \mathbf{Val}}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$

$$\frac{}{\Theta \mid \Sigma \vdash_k 1 : \mathbf{Val}} \qquad \frac{\Theta \mid \Sigma \vdash_k A_1 : \mathbf{Val} \qquad \Theta \mid \Sigma \vdash_k A_1 : \mathbf{Val}}{\Theta \mid \Sigma \vdash_k A_1 \times A_2 : \mathbf{Val}} \qquad \frac{}{\Theta \mid \Sigma \vdash_k 0 : \mathbf{Val}}$$

$$\frac{\Theta \mid \Sigma \vdash_k A_1 : \mathbf{Val} \qquad \Theta \mid \Sigma \vdash_k A_1 : \mathbf{Val}}{\Theta \mid \Sigma \vdash_k A_1 + A_2 : \mathbf{Val}} \qquad \frac{\Theta \mid \Sigma \vdash_k e : \mathbf{Eff} \qquad \Theta \mid \Sigma \vdash_k C : \mathbf{Comp}_e}{\Theta \mid \Sigma \vdash_k U_e C : \mathbf{Val}}$$

$$\frac{\alpha \in \Theta}{\Theta \mid \Sigma \vdash_k \alpha : \mathbf{Val}}$$

Effect kinding $\boxed{\Sigma \vdash_k e : \mathbf{Eff}}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$

$$\frac{}{\Sigma \vdash_k \bot : \mathbf{Eff}} \qquad \frac{\varepsilon \in \Sigma}{\Sigma \vdash_k \varepsilon : \mathbf{Eff}}$$

Computation kinding $\boxed{\Theta \mid \Sigma \vdash_k C : \mathbf{Comp}_e}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$ and $\Sigma \vdash_k e : \mathbf{Eff}$

$$\frac{}{\Theta \mid \Sigma \vdash_k FA : \mathbf{Comp}_e} \qquad \frac{\Theta \mid \Sigma \vdash_k A : \mathbf{Val} \qquad \Theta \mid \Sigma \vdash_k C : \mathbf{Comp}_e}{\Theta \mid \Sigma \vdash_k A \to C : \mathbf{Comp}_e}$$

$$\frac{}{\Theta \mid \Sigma \vdash_k \top : \mathbf{Comp}_e} \qquad \frac{\Theta \mid \Sigma \vdash_k C_1 : \mathbf{Comp}_e \qquad \Theta \mid \Sigma \vdash_k C_2 : \mathbf{Comp}_e}{\Theta \mid \Sigma \vdash_k C_1 \& C_2 : \mathbf{Comp}_e}$$

Context kinding $\boxed{\Theta \mid \Sigma \vdash_k \Gamma : \mathbf{Ctxt}}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$

$$\frac{\text{for all } x \in \mathrm{Dom}\ (\Gamma)\text{: } \Theta \mid \Sigma \vdash_k \Gamma(x) : \mathbf{Val}}{\Theta \mid \Sigma \vdash_k \Gamma : \mathbf{Ctxt}}$$

Figure 4.3: λ_{mon}-calculus kind system

Value typing $\boxed{\Theta \mid \Gamma \vdash_{\Sigma} V : A}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$, $\Theta \mid \Sigma \vdash_k \Gamma : \mathbf{Ctxt}$, and $\Theta \mid \Sigma \vdash_k A : \mathbf{Val}$

$$\frac{(x : A) \in \Gamma}{\Theta \mid \Gamma \vdash_{\Sigma} x : A} \qquad \frac{}{\Theta \mid \Gamma \vdash_{\Sigma} () : 1} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma} V_1 : A_1 \quad \Theta \mid \Gamma \vdash_{\Sigma} V_2 : A_2}{\Theta \mid \Gamma \vdash_{\Sigma} (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma} V : A_i}{\Theta \mid \Gamma \vdash_{\Sigma} \mathbf{inj}_i\, V : A_1 + A_2} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma, e} M : C}{\Theta \mid \Gamma \vdash_{\Sigma} \{M\} : U_e C}$$

Computation typing $\boxed{\Theta \mid \Gamma \vdash_{\Sigma, e} M : C}$ for $\Theta \vdash_k \Sigma : \mathbf{Eff}$, $\Sigma \vdash_k e : \mathbf{Eff}$, $\Theta \mid \Sigma \vdash_k \Gamma : \mathbf{Ctxt}$, and $\Theta \mid \Sigma \vdash_k C : \mathbf{Comp}_e$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma} V : A_1 \times A_2 \quad \Theta \mid \Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\Sigma, e} M : C}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{split}(V, x_1.x_2.M) : C} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma} V : 0}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{case}_0(V) : C}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma} V : A_1 + A_2 \quad \Theta \mid \Gamma, x_1 : A_1 \vdash_{\Sigma, e} M_1 : C \quad \Theta \mid \Gamma, x_2 : A_2 \vdash_{\Sigma, e} M_2 : C}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma} V : U_e C}{\Theta \mid \Gamma \vdash_{\Sigma, e} V! : C} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma} V : A}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{return}\ V : FA}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma, e} M : FA \quad \Theta \mid \Gamma, x : A \vdash_{\Sigma, e} N : C}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C} \qquad \frac{\Theta \mid \Gamma, x : A \vdash_{\Sigma, e} M : C}{\Theta \mid \Gamma \vdash_{\Sigma, e} \lambda x.M : A \to C}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma, e} M : A \to C \quad \Theta \mid \Gamma \vdash_{\Sigma} V : A}{\Theta \mid \Gamma \vdash_{\Sigma, e} M\, V : C} \qquad \frac{}{\Theta \mid \Gamma \vdash_{\Sigma, e} \langle\rangle : \top}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma, e} M_1 : C_1 \quad \Theta \mid \Gamma \vdash_{\Sigma, e} M_2 : C_2}{\Theta \mid \Gamma \vdash_{\Sigma, e} \langle M_1, M_2 \rangle : C_1 \,\&\, C_2} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma, e} M : C_1 \,\&\, C_2}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{prj}_i\, M : C_i}$$

$$\frac{\Theta \mid \Gamma \vdash_{\Sigma,\varepsilon} N : FA \quad (\varepsilon \sim \alpha.C, e \prec \varepsilon) \in \Sigma}{\Theta \mid \Gamma \vdash_{\Sigma,e} [N]^{\varepsilon} : C[A/\alpha]} \qquad \frac{\Theta \mid \Gamma \vdash_{\Sigma,e} N : C[A/\alpha] \quad (\varepsilon \sim \alpha.C, e \prec \varepsilon) \in \Sigma}{\Theta \mid \Gamma \vdash_{\Sigma,\varepsilon} \widehat{\mu}^{\varepsilon}(N) : FA}$$

$$\frac{\begin{array}{c}\Theta, \alpha_1 \mid \emptyset \vdash_{\Sigma, e} N_u : \alpha_1 \to C[\alpha_1/\alpha] \\ \Theta, \alpha_1, \alpha_2 \mid \emptyset \vdash_{\Sigma, e} N_b : U_e(C[\alpha_1/\alpha]) \to U_e(\alpha_1 \to C[\alpha_2/\alpha]) \to C[\alpha_2/\alpha] \\ \Theta \mid \Gamma \vdash_{(\Sigma, \varepsilon \sim (\alpha.C, N_u, N_b), e \prec \varepsilon), e} M : D\end{array}}{\Theta \mid \Gamma \vdash_{\Sigma, e} \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M : D}$$

Figure 4.4: λ_{mon}-calculus type system

have for a term M that $\Theta \mid \Gamma \vdash_{\Sigma,e} M : C$ this means that the type M has type C and can invoke effect e if supplied with a type A_α for every $\alpha \in \Theta$ and a well-typed term t_x of type A for every $(x : A) \in \Gamma[A_\alpha/\alpha]_{\alpha\in\Theta}$ (we prove this in Lemma 4.2). We will sometimes call e the **level** of the computation.

To refer to both computations and values we will sometimes write $\Theta \mid \Gamma \vdash_{\Sigma,e} P : X$. If P is a value V, this simply means $\Theta \mid \Gamma \vdash_{\Sigma} V : X$. For $\Theta \mid \Gamma \vdash_{\Sigma,e} P : X$ we always implicitly assume that e,Σ and X are well-kinded:

The interesting typing rules are those for $\widehat{\mu}^{\varepsilon}(-)$ and $[-]^{\varepsilon}$. Reflection $\widehat{\mu}^{\varepsilon}(N)$ types as a value returning computation of type FA at level ε i.e. as a value returning computation with effect ε, if N is of type $C[A/\alpha]$ at level e, i.e. a computation that has the type matching the definition of ε.

Dually, if N is a computation at level ε, $[N]^{\varepsilon}$ turns this into a computation of type $C[A/\alpha]$ at level e, i.e. replaces the effect ε with its implementation as a monad which uses effect e.

Finally, for the *(leteffect)*-rule, we first have to check that N_u and N_b are closed terms which have the matching types to serve as return and bind.

The operational semantics for λ_{mon} is shown in figure 4.5.

To reify a value-returning computation we use the return of the monad (rule *(reify)*). Reflection uses the supplied bind operation to sequence its argument before the remainder of the computation. It captures the remaining computation by matching with the nearest enclosing reify-construct.

To match the reify with the closest reflect, we require that the reified effects in the frame $\mathcal{H}$ do not contain ε.

The various *(leteff-)*rules ensure that the normal forms of effect free computations are of canonical shape, for instance **return** V for FA : $\mathbf{Comp}_{\perp}$.

4.2 Programming in λ_{mon}

We only briefly sketch how to implement exceptions in λ_{mon} and omit store. Filinski [7] describes how to implement exceptions and store in detail.

Define:

$$\begin{aligned}
C^{\mathrm{ex}} &:= \alpha + \mathrm{string} \\
N^{\mathrm{ex}}_u &:= \lambda x.\mathbf{return}\ \mathbf{inj}_1\ x \\
N^{\mathrm{ex}}_b &:= \lambda x\, f.\mathbf{let}\ y \leftarrow x\ \mathbf{in}\ \mathbf{case}(y, x.fx, s.\mathbf{return}\ (\mathbf{inj}_2\ s))) \\
\mathrm{ex} \succ \perp &\sim (\alpha.C^{\mathrm{ex}}, N^{\mathrm{ex}}_u, N^{\mathrm{ex}}_b)
\end{aligned}$$

Reduction frames

$$\begin{array}{llcl}
\text{(computation frames)} & \mathcal{C} & ::= & \textbf{let } x \leftarrow [\,] \textbf{ in } N \mid [\,]\ V \mid \textbf{prj}_i\ [\,] \mid [\![\,[\,]\,]\!]^{\varepsilon} \\
\text{(hoisting contexts)} & \mathcal{H} & ::= & [\,] \mid \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } \mathcal{H} \mid \mathcal{C}[\mathcal{H}]
\end{array}$$

Reduction $\boxed{\vdash_\Sigma M \longrightarrow M'}$

$$\begin{array}{lr}
(\beta.\times) & \vdash_\Sigma \textbf{split}((V_1, V_2), x_1.x_2.M) \longrightarrow M[V_1/x_1, V_2/x_2] \\
(\beta.+) & \vdash_\Sigma \textbf{case}(\textbf{inj}_i\ V, x_1.M_1, x_2.M_2) \longrightarrow M_i[V/x_i] \\
(\beta.U) & \vdash_\Sigma \{M\}! \longrightarrow M \\
(\beta.F) & \vdash_\Sigma \textbf{let } x \leftarrow \textbf{return } V \textbf{ in } M \longrightarrow M[V/x] \\
(\beta.\to) & \vdash_\Sigma (\lambda x.M)\ V \longrightarrow M[V/x] \\
(\beta.\&) & \vdash_\Sigma \textbf{prj}_i\ \langle M_1, M_2 \rangle \longrightarrow M_i \\
(\textit{frame}) & \dfrac{\vdash_\Sigma M \longrightarrow M'}{\vdash_\Sigma \mathcal{C}[M] \longrightarrow \mathcal{C}[M']}
\end{array}$$

$$(\textit{reify}) \quad \frac{(\varepsilon = (N_u, N_b)) \in \Sigma}{\vdash_\Sigma [\![\textbf{return } V]\!]^{\varepsilon} \longrightarrow N_u V}$$

$$(\textit{reflect}) \quad \frac{(\varepsilon \sim (\alpha.C, N_u, N_b)) \in \Sigma \quad \varepsilon \notin \text{effects}(\mathcal{H})}{\vdash_\Sigma [\![\mathcal{H}[\widehat{\mu}^{\varepsilon}(N)]]\!]^{\varepsilon} \longrightarrow N_b\ \{N\}\ \{(\lambda x. [\![\mathcal{H}[\textbf{return } x]]\!]^{\varepsilon})\}}$$

$$(\textit{leteff}) \quad \frac{\mathcal{H} = \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } [\ ]_{\Sigma, \varepsilon \sim (\alpha.C, N_u, N_b), e \prec \varepsilon} M \longrightarrow M'}{\vdash_\Sigma \mathcal{H}[M] \longrightarrow \mathcal{H}[M']}$$

$$(\textit{leteff-return}) \quad \frac{}{\vdash_\Sigma \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in return } V \longrightarrow \textbf{return } V}$$

$$(\textit{leteff-lambda}) \quad \frac{\mathcal{H} = \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } [\ ]}{\vdash_\Sigma \mathcal{H}[\lambda x.N] \longrightarrow \lambda x.\mathcal{H}[N]}$$

$$(\textit{leteff-pair}) \quad \frac{\mathcal{H} = \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } [\ ]}{\vdash_\Sigma \mathcal{H}[\langle N_1, N_2 \rangle] \longrightarrow \langle \mathcal{H}[N_1], \mathcal{H}[N_2] \rangle}$$

$$(\textit{leteff-unit}) \quad \frac{}{\vdash_\Sigma \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } \langle\rangle} \longrightarrow \langle\rangle$$

where

$$\begin{aligned}
\text{effects}([]) &= \emptyset \\
\text{effects}(\textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } \mathcal{H}) &= \text{effects}(\mathcal{H}) \\
\text{effects}([\![\mathcal{H}]\!]^{\varepsilon}) &= \{\varepsilon\} \cup \text{effects}(\mathcal{H}) \\
\text{effects}(\mathcal{C}[\mathcal{H}]) &= \text{effects}(\mathcal{H}) \qquad (\text{for } \mathcal{C} \neq [\![[]]\!]^{\varepsilon})
\end{aligned}$$

Figure 4.5: λ_{mon}-calculus operational semantics

We can then define

$$\begin{aligned}\text{raise} &\coloneqq \lambda s.\widehat{\mu}^{\text{ex}}(\textbf{return}\ (\textbf{inj}_2\ s))\\ \text{try} &\coloneqq \lambda c\ h.\textbf{let}\ s \leftarrow [c]^{\text{ex}}\ \textbf{in case}(s, a.\textbf{return}\ a, x.hx)\end{aligned}$$

We do not recall and check the correctness properties here, as this is done extensively by Filinski [7].

4.3 Denotational semantics for λ_{mon}

We define the denotational semantics for λ_{mon} using mutual recursion over types, effects and terms for every level. This is because in λ_{mon} types contain effects, effects contain terms and terms have types.

For a type variable environment Θ, we use $\theta \in [\![\Theta]\!]$ to denote the fact that θ is a tuple of sets induced by Θ.

The semantics assigns to each $\theta \in [\![\Theta]\!]$ and well kinded

- value type $\Theta \mid \Sigma \vdash_k A : \textbf{Val}$ a set $[\![A]\!](\theta)$;
- effect $\Sigma \vdash_k e : \textbf{Eff}$ a monad $[\![e]\!]$;
- e-based computation type $\Theta \mid \Sigma \vdash_k C : \textbf{Comp}_e$ a $[\![e]\!]$-algebra $[\![C]\!](\theta)$;
- context $\Theta \mid \Sigma \vdash_k \Gamma : \textbf{Ctxt}$ a set of tuples $[\![\Gamma]\!](\theta)$ induced by Γ

We will write γ for a denotation in $[\![\Gamma]\!](\theta)$.

We will only assign a denotation to terms that fulfil the following conditions, this our semantics is partial. We require that for **leteffect** $\varepsilon \succ e$ **be** $(\alpha.C, N_u, N_b)$ **in** M the two functions $\textbf{return}_\varepsilon$ and $\gg\!=_\varepsilon$ induced by N_u and N_b form a monad. If they do not, the semantics is undefined. Following Felleisen we require the monads to be layered, i.e. a monad morphism $T_e \to T_\varepsilon$. We call a term P that has a denotation a **proper term**.

We define $[\![\Theta]\!] \coloneqq \prod_{\alpha\in\Theta} \text{Set}$.

For types we reuse the CBPV semantics (figure 2.5) again, with the following differences:

We use the same denotations for value types as we did for CBPV, i.e. for instance $[\![A_1 \times A_2]\!](\theta) = [\![A_1]\!](\theta) \times [\![A_2]\!](\theta)$. We additionally set $[\![\alpha]\!](\theta) = \theta(\alpha)$.

For effects introduced as **leteffect** $\varepsilon \succ e$ **be** $(\alpha.C, N_u, N_b)$ **in** we define

$$[\![\varepsilon]\!] \coloneqq \langle T_e, \textbf{return}_e, \gg\!=_e\rangle$$

where

$$\begin{aligned}
T_\varepsilon X &:= |[\![C]\!](\theta[\alpha \mapsto X])| \\
\mathbf{return}_\varepsilon{}^X &:= [\![N_u]\!](\theta[\alpha \mapsto X]) : X \to T_\varepsilon X \\
\gg\!=_\varepsilon{}^{X,Y} &:= [\![N_b]\!](\theta[\alpha_1 \mapsto X, \alpha_2 \mapsto Y]) : T_\varepsilon X \to (X \to T_\varepsilon Y) \to T_\varepsilon Y.
\end{aligned}$$

and define $[\![\bot]\!]$ to be the identity monad.

The denotations for computations are unchanged from CBPV, apart from the F-type.

All the algebras are indeed T_e algebras for types at level e by Lemma 2.8. We set

$$[\![\Theta \mid \Sigma \vdash_k FA : \mathbf{Comp}_e]\!](\theta) := F_{[\![e]\!]}([\![A]\!](\theta))$$

where $F_{[\![e]\!]}$ is the free algebra for the monad $[\![e]\!]$. Note that this means at level ε that $|[\![FA]\!](\theta)| = T_\varepsilon([\![A]\!](\theta)) = |[\![C[A/\alpha]]\!](\theta)|$.

Finally, as before we define $[\![\Gamma]\!](\theta) := \prod_{x \in \text{Dom}(\Gamma)} [\![\Gamma(x)]\!](\theta)$.

Note that in contrast to λ_{eff} the semantics for λ_{mon} is finite:

Lemma 4.1. $[\![A]\!](\theta)$ *and* $|[\![C]\!](\theta)|$ *are always finite if* $\theta = \langle A_\alpha \rangle_{\alpha \in \Theta}$ *and all* A_α *are finite.*

4.3.1 Denotations of terms

Giving denotations to derivation of types instead of terms is crucial for λ_{mon}. A derivation $\Theta \mid \Gamma \vdash_{\Sigma,e} M : C$ denotes for every Θ-denotation θ a function of type $\prod_{\gamma \in [\![\Gamma]\!](\theta)} |[\![C]\!](\theta)|$ for every $\theta \in [\![\Theta]\!]$. A value derivation $\Theta \mid \Gamma \vdash_\Sigma V : A$ denotes for every $\theta \in [\![\Theta]\!]$ a function $\prod_{\gamma \in [\![\Gamma]\!](\theta)} [\![A]\!](\theta)$.

The denotation for $\widehat{\mu}^\varepsilon(N)$ now has to be an element in $|[\![FA]\!](\theta)| = |[\![C[A/\alpha]]\!](\theta)|$ for the T_ε-algebra $[\![FA]\!](\theta)$. To define this, we have access to $[\![N]\!](\theta) \in |[\![C[A/\alpha]]\!](\theta)|$ at level e, so we can set

$$[\![\widehat{\mu}^\varepsilon(N)]\!](\theta)(\gamma) := [\![N]\!](\theta)(\gamma).$$

Note that because terms denote functions to the *carrier set* of the algebra we do not have to worry about the level here.

The same idea works for the semantics of reify for the same reason, and, obviously, as it does not have any influence on the computation, for the leteffect construct:

$$\begin{aligned}
&[\![\Theta \mid \Gamma \vdash_{\Sigma,e} [N]^\varepsilon : C[A/\alpha]]\!](\theta)(\gamma) : [\![\Gamma]\!](\theta) \to \left|[\![FA : \mathbf{Comp}_e]\!](\theta)\right| = |[\![C[A/\alpha]]\!](\theta)| \\
&[\![\,[N]^\varepsilon]\!](\theta)(\gamma) := [\![N]\!](\theta)(\gamma)
\end{aligned}$$

$$[\![\Theta \mid \Gamma \vdash_{\Sigma,e} \textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, N_b) \textbf{ in } N : D]\!](\theta)(\gamma) := \begin{cases} [\![N]\!](\theta)(\gamma) & \text{if } N_u \text{ and } N_b \text{ form a monad} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The other denotations are as for CBPV (see figure 2.5), with the return and bind at level e taken from the monad T_e, i.e. for instance

$$[\![\Theta \mid \Gamma \vdash_{\Sigma,e} \textbf{return } V : FA]\!](\theta)(\gamma) := \textbf{return}_e\ ([\![V]\!](\theta)(\gamma)).$$

We prove this semantics to be adequate and sound in Lemma 4.11 and Lemma 4.12. As a sanity check, we can look at the term $[\textbf{return } V]^{\varepsilon}$ which reduces to $N_u\ V$.

$$[\![[\textbf{return } V]^{\varepsilon}]\!](\theta)(\gamma) = \textbf{return}_{\varepsilon}\ ([\![V]\!](\theta)(\gamma)) = [\![N_u]\!](\theta)(\star)([\![V]\!](\theta)(\gamma)) = [\![N_u\ V]\!](\theta)(\gamma)$$

A similar observation shows that the semantics is correct for the term $[\textbf{let } x \leftarrow \widehat{\mu}^{\varepsilon}(N) \textbf{ in } M]^{\varepsilon}$ which steps to $N_b\ N(\lambda x.M)$, because the bind that is used in the denotation for the let-construct is at level ε, i.e. induced by N_b in the definition of ε.

4.4 Differences to Filinski's calculus

Our elimination of recursive constructs and the explicit basing on CBPV introduces several differences to Filinski's calculus [7].

Filinski's value returning computation type is parameterised by an effect ($F_e A$) and he introduces an effect basing judgment $\vdash_\Sigma e \preceq C$ expressing that C is a computation that might invoke the effect e. For $e \prec \varepsilon$ and $\vdash_\Sigma \varepsilon \preceq C$ one can always rebase C to $\vdash_\Sigma e \preceq C$. Because we make the level of every computation explicit, we can remove the annotation on F. Rebasing is not built-in to λ_{mon}, but the user has to make this points where terms change their level explicit using coercions (see section 4.4.1).

This change enables us to use the exact same rule for let-constructs as in CBPV, where for Filinski the return-type C hat to be based on e if $M : F_e A$.

When introducing an effect $\varepsilon \succ e$ in Filinski, the underlying monads have to be layered (i.e. we need a monad morphism $T_e \to T_\varepsilon$. Because we eliminate rebasing, we can relax this condition.

We also do not introduce an explicit subeffecting judgement. Our ordering is defined in an effect hierarchy Σ and we write $(e \precnsim e') \in \Sigma$ for the transitive closure of the relation induced by the $(e \prec e') \in \Sigma$ and $(e \preceq \Sigma') \in \Sigma$ for the reflexive, transitive closure.

Because we use CBPV explicitly, we have to make a design choice how to incorporate the term N_b. The alternative to our presentation would be to let N_b be synctactically a context $\Theta[\Theta] \mid \emptyset[\emptyset] \vdash_{\Sigma[\Sigma],e[e]} N_b : (U_\Sigma(\alpha_1 \to C[\alpha_2/\alpha]) \to C[\alpha_2/\alpha])[C[\alpha_1/\alpha]]$. We chose this presentation using explicit thunking, because it seems to be closer to Filinsis presentation.

Filinski only allows leteffect at the toplevel and distinguished programs and terms. We allow leteffect anywhere and merge those two notions. Every program of Filinski can still be typechecked in our system, and by lifting all leteffects to the toplevel and introducing coercions at the right points, this is also the case in the other direction.

4.4.1 Coercions

Oftentimes one wants to use computations at level e at a higher level $\varepsilon \succ e$. In Filinski's calculus, this is implicitly always possible. We make the level of computation explicit as in Haskell, which makes some terms that are well-typed in Filinski's calculus now not typeable.

We can fix this problem and obtain that every program of Filinski's calculus can be translated to our calculus by inserting so called **coercions** ("liftings" in Haskell).

We define $\text{coerce}_{e \prec \varepsilon}(M)$ for a $\Theta \mid \Gamma \vdash_{\Sigma,e} M : FA$ as

$$\text{coerce}_{e \prec \varepsilon}(M) := \widehat{\mu}^{\varepsilon}(\textbf{let } x \leftarrow M \textbf{ in } [\textbf{return } x]^{\varepsilon}).$$

We then have $\Theta \mid \Gamma \vdash_{\Sigma,\varepsilon} \text{coerce}_{e \prec \varepsilon}(M) : FA$.

The inverse of this operation is reification, as for $[\text{coerce}_{e \prec \varepsilon}(M)]^{\varepsilon} \simeq N_b\ (\textbf{let } x \leftarrow M \textbf{ in } [\textbf{return } x]^{\varepsilon})\ (\lambda x.\textbf{return } x) \simeq N_b\ (\textbf{let } x \leftarrow M \textbf{ in } N_u x)\ (\lambda x.\textbf{return } x) \simeq \textbf{let } x \leftarrow M \textbf{ in } N_b\ (N_u x)\ (\lambda x.\textbf{return } x) \simeq \textbf{let } x \leftarrow M \textbf{ in return } x \simeq M$.

4.5 Contextual equivalence

We extend CBPV contexts for λ_{mon} as shown in figure 4.6 and show their typing in figure 4.7. To simplify the presentation, we exclude the contexts $\textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, \mathcal{X}_{\text{comp}}[\ ], N_b) \textbf{ in } N$ and $\textbf{leteffect } \varepsilon \succ e \textbf{ be } (\alpha.C, N_u, \mathcal{X}_{\text{comp}}[\ ]) \textbf{ in } N$, because this does not change the notion of contextual equivalence.

The following two lemmas are slightly more involved for λ_{mon} than they were before, because we have to incorporate the Θ:

Lemma 4.2 (substitution). *For all* $\Theta \mid \Gamma \vdash_{\Sigma,e} P : X$, $\langle V_x \rangle_{x \in \text{Dom}(\Gamma)}$ *and* $\langle A_\alpha \rangle_{\alpha \in \Theta}$*: such that, for all* $x \in \text{Dom}\ (\Gamma)$, $\widehat{\Theta} \mid \Delta \vdash_{\Sigma,e} V_x : \Gamma[A_\alpha/\alpha]_{\alpha \in \Theta}(x)$, *we have* $\widehat{\Theta} \mid \Delta \vdash_{\Sigma,e} P[V_x/x]_{x \in \text{Dom}(\Gamma)} : X[A_\alpha/\alpha]_{\alpha \in \Theta}$.

Lemma 4.3 (context substitution). *For all* $\Theta[\Theta'] \mid \Delta[\Gamma'] \vdash_{\Sigma[\Sigma'],e[e']} \mathcal{X}[\ ] : Y[X]$*:*

1. *For all* $\Theta \mid \Gamma \vdash_{\Sigma',e'} P : X$ *where* $\Gamma' \geqslant \Gamma$, *we have* $\Theta \mid \Delta \vdash_{\Sigma,e} \mathcal{X}[P] : Y$.

For the omitted parts see CBPV contexts in figure 2.7.

$$
\begin{array}{ll}
\text{(value contexts)} & \mathcal{X}_{\text{val}} ::= \dots \\
\text{(computation contexts)} & \\
& \mathcal{X}_{\text{comp}} ::= \dots \\
& \quad \mid\ [\mathcal{X}_{\text{comp}}]^{\varepsilon} \mid \widehat{\mu}^{\varepsilon}(\mathcal{X}_{\text{comp}}) \\
& \quad \mid\ \textbf{leteffect}\ \varepsilon \succ e\ \textbf{be}\ (\alpha.C, N_u, N_b)\ \textbf{in}\ \mathcal{X}_{\text{comp}}
\end{array}
$$

Figure 4.6: λ_{mon}-calculus contexts

2. *For all* $\Theta \mid \Gamma \vdash_{\Sigma',e'} P, Q : X$ *where* $\Gamma' \geqslant \Gamma$ *and* $\mathcal{X}[P], \mathcal{X}[Q]$ *are proper terms, we have* $[\![P]\!] = [\![Q]\!]$ *implies* $[\![\mathcal{X}[P]]\!] = [\![\mathcal{X}[Q]]\!]$.

We call a context $\mathcal{X}$ a **proper context** if for all pluggable proper terms P, $\mathcal{X}[P]$ is proper.

Contextual equivalence for λ_{mon} is straightforward. We do not use the the detour via defining contextual equivalence parameterised by an effect hierarchy Σ as we did for λ_{eff}.

Definition 4.4 (contextual equivalence). *Let* $\Theta \mid \Gamma \vdash_{\Sigma,e} P, Q : X$ *be two proper* λ_{mon} *terms. We say that* P *and* Q *are **contextually equivalent**, and write* $P \simeq Q$ *when, for all **closed, effect-free** well-typed proper **ground-returner** contexts* $\emptyset[\Theta] \mid \emptyset[\Gamma'] \vdash_{\emptyset[E],\bot[e]} \mathcal{X}[\ ] : FG[X]$ *with* $\Gamma' \geqslant \Gamma$ *are defined and for all closed ground value terms* $\Theta \mid \ \vdash_{\emptyset} V : G[]$, *we have:*

$$\mathcal{X}[P] \longrightarrow^{\star} \textbf{return}\ V \qquad \Longleftrightarrow \qquad \mathcal{X}[Q] \longrightarrow^{\star} \textbf{return}\ V$$

Lemma 4.5 (basic contextual equivalence properties). *The relation* $\simeq$ *is an equivalence relation that is congruent w.r.t.* λ_{mon} *terms, and contains the reduction relation* $\longrightarrow$ *in the following sense: If* $[\![M]\!]\,(\theta)$ *is defined and* $M \longrightarrow M'$ *then* $M \simeq M'$.

Corollary 4.6. *If for proper terms* M, M' *we have* $M \longrightarrow M'$, *then* $\mathcal{X}[M] \simeq \mathcal{X}[M']$.

Proof

Follows immediately from the last lemma, as $M \longrightarrow M' \implies M \simeq M'$ and $\simeq$ is a congruence. ∎

4.6 Adequacy

Proving adequacy for the denotational semantics of λ_{mon} is the most delicate proof in this thesis. We use logical relations again. But this time the simple lifting from section 3.4 is insufficient. Instead, we employ a technique called **$\top\top$-lifting** (read: "top-top-lifting") [16, 15, 13, 10] and use two intermediate relations.

Context typing $\boxed{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} \mathcal{X}[\ ] : C[X]}$

$$\frac{}{\Theta[\Theta] \mid \Gamma[\Gamma] \vdash_{\Sigma[\Sigma]} [\ ] : X[X]}$$

Value context typing as in figure 2.8, suitably modified with type variable environments.

Computation context typing as in figure 2.8 and:

$$\frac{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} \mathcal{X}[\ ] : FA[X] \qquad (\varepsilon \sim \alpha.C) \in \Sigma}{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} [\mathcal{X}[\ ]]^{\varepsilon} : C[A/\alpha][X]}$$

$$\frac{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} \mathcal{X}[\ ] : C[A/\alpha][X] \qquad (\varepsilon \sim \alpha.C) \in \Sigma}{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} \widehat{\mu}^{\varepsilon}(\mathcal{X}[\ ]) : FA[X]}$$

$$\frac{\begin{array}{c}\Theta, \alpha_1 \mid \emptyset \vdash_{\Sigma,e} N_u : \alpha_1 \to C[\alpha_1/\alpha] \\ \Theta, \alpha_1, \alpha_2 \mid \emptyset \vdash_{\Sigma,e} N_b : U_e(C[\alpha_1/\alpha]) \to U_e(\alpha_1 \to C[\alpha_2/\alpha]) \to C[\alpha_2/\alpha] \\ \Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{(\Sigma,\varepsilon\sim(\alpha.C,N_u,N_b),e\prec\varepsilon)[\Sigma'],e[e']} \mathcal{X}[\ ] : D[X]\end{array}}{\Theta[\Theta'] \mid \Gamma[\Delta] \vdash_{\Sigma[\Sigma'],e[e']} \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \mathcal{X}[\ ] : D[X]}$$

Figure 4.7: λ_{mon}-calculus context types

Doczkal and Schwinghammer [3] use $\top\top$-lifting to prove strong normalisation of CBPV. We follow their approach and extend it to λ_{mon}.

We define

$$\begin{aligned}
\boldsymbol{\lambda}^{\Theta,\Sigma,e}_{\textbf{mon}}(C) &:= \{P \mid \Theta \mid \emptyset \vdash_{\Sigma,e} P : C\} \\
\boldsymbol{\lambda}^{\Theta,\Sigma}_{\textbf{mon}}(A) &:= \{P \mid \Theta \mid \emptyset \vdash_{\Sigma} P : A\} \\
\mathcal{X}^{\Theta,\Sigma,e}_{\text{mon}}(FG[C]) &:= \left\{\mathcal{X} \mid \emptyset[\Theta] \mid \emptyset[\emptyset] \vdash_{\Sigma[\Sigma],\perp[e]} \mathcal{X}[\ ] : FG[C]\right\}
\end{aligned}$$

All our relations are parameterised by type variable environments Θ, effect hierarchies Σ, effects e for computations, λ_{mon} types X and $\rho = \left\langle \rho_\alpha \subseteq \theta(\alpha) \times \boldsymbol{\lambda}^{\Theta,\Sigma}_{\textbf{mon}}(A_\alpha)\right\rangle_{\alpha\in\Theta}$ for a sequence $\langle \emptyset \mid \Sigma \vdash_k A_\alpha : \textbf{Val}\rangle_{\alpha\in\Theta}$, i.e. ρ is an α-indexed sequence of relations. We then define $R^{v}_{\Theta,\Sigma,e,C}(\rho) \subseteq |[\![C]\!](\theta)| \times \boldsymbol{\lambda}^{\Theta,\Sigma,e}_{\textbf{mon}}(C[A_\alpha/\alpha]_{\alpha\in\Theta})$.

The idea is that if $R^{v}_{\Theta,\Sigma,e,C}(\rho)$ is a relation between denotations and terms, then its $\top\top$-lifting is a relation with possibly more elements, i.e.

$$R_{\Theta,\Sigma,e,C}(\rho) \subseteq (R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top\top} \subseteq |[\![C]\!](\theta)| \times \boldsymbol{\lambda}^{\Theta,\Sigma,e}_{\textbf{mon}}(C[A_\alpha/\alpha]_{\alpha\in\Theta}).$$

The $\top$-lifting of $R_{\Theta,\Sigma,e,C}(\rho)$ is a relation $(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top}$ between denotations of contexts and contexts, i.e $(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top} \subseteq (|[\![C]\!](\theta)| \to |[\![FG]\!](\theta)|) \times \mathcal{X}^{\Theta,\Sigma,e}_{\text{mon}}(FG\left[C[A_\alpha/\alpha]_{\alpha\in\Theta}\right])$.

The relations are defined in figure 4.8.

Lemma 4.7. *The logical relations $R_{\Theta,\Sigma,e,X}(\rho)$ are closed under contextual equivalence. Explicitly: For all $\langle a, P\rangle \in R_{\Theta,\Sigma,e,X}(\rho)$ and $\vdash_{\Sigma,e} Q : X$, $P \simeq Q$ implies $\langle a, Q\rangle \in R_{\Theta,\Sigma,e,X}(\rho)$ and analogously for value relations.*

Proof

For value types X, the proof carries over from CBPV. For computation types, the statement follows immediately from the definition of $\top\top$-lifting. ■

We restate Lemma 2.16 for completeness:

Lemma 4.8. *For all ground types G, $a \in [\![G]\!](\theta)$, and closed value terms $\vdash V, V' : G$:*

$$\langle a, V\rangle, \langle a, V'\rangle \in R_{\Theta,\Sigma,G}(\rho) \implies V \simeq V'$$

Lemma 4.9. *For all Θ, Σ, derivations $(\alpha,\Theta) \mid \Sigma \vdash_k X : \textbf{Comp}_e$ (or $\Theta \mid \Sigma \vdash_k X : \textbf{Val}$), $\langle \emptyset \mid \Sigma \vdash_k A_\alpha : \textbf{Val}\rangle_{\alpha\in\Theta}$, $\theta = \langle [\![A_\alpha]\!](\star)\rangle_{\alpha\in\Theta}$, $\rho = \left\langle R_{\emptyset,\Sigma,A_\alpha}(\star)\right\rangle_{\alpha\in\Theta}$, and $\Theta \mid \Sigma \vdash_k A : \textbf{Val}$ we have*

$$R_{(\alpha,\Theta),\Sigma,e,X}(\rho\left[\alpha \mapsto R_{\emptyset,\Sigma,A[A_\alpha/\alpha]_{\alpha\in\Theta}}(\star)\right]) = R_{\Theta,\Sigma,e,X[A/\alpha]}(\rho)$$

Value relations $\boxed{R_{\Theta,\Sigma,A}(\rho) \subseteq [\![A]\!](\theta) \times \boldsymbol{\lambda}^{\boldsymbol{\Theta},\boldsymbol{\Sigma}}_{\mathbf{mon}}(A)}$

$$R_{\Theta,\Sigma,1}(\rho) \coloneqq \{\langle \star, V\rangle | V \simeq ()\}$$

$$R_{\Theta,\Sigma,A_1\times A_2}(\rho) \coloneqq \left\{\langle\langle a_1, a_2\rangle, V\rangle \middle| \exists V_1, V_2 \forall i. V_i : A_i. \langle a_i, V_i\rangle \in R_{\Sigma,A_i}(\rho), V \simeq (V_1, V_2)\right\}$$

$$R_{\Theta,\Sigma,0}(\rho) \coloneqq \emptyset$$

$$R_{\Theta,\Sigma,A_1+A_2}(\rho) = \bigcup_{i=1,2} \left\{\langle \iota_i a, V\rangle \middle| \exists V' : A_i. \langle a, V'\rangle \in R_{\Theta,\Sigma,A_i}(\rho), V \simeq \mathbf{inj}_i\, V'\right\}$$

$$R_{\Theta,\Sigma,U_e C}(\rho) \coloneqq \{\langle a, V\rangle | \exists M : C. \langle a, M\rangle \in R_{\Theta,\Sigma,e,C}(\rho), V \simeq \{M\}\} \qquad R_{\Theta,\Sigma,\alpha}(\rho) \coloneqq \rho(\alpha)$$

Computation relations $\boxed{R^{v}_{\Theta,\Sigma,e,C}(\rho) \subseteq |[\![C]\!](\theta)| \times \boldsymbol{\lambda}^{\boldsymbol{\Theta},\boldsymbol{\Sigma},\boldsymbol{e}}_{\mathbf{mon}}(C)}$

$$R^{v}_{\Theta,\Sigma,\perp,FA}(\rho) \coloneqq \{\langle a, \mathbf{return}\ V\rangle | \exists \langle a', V\rangle \in R_{\Sigma,A}(\rho), a = \mathbf{return}\ a'\}$$

$$R^{v}_{\Theta,\Sigma,\varepsilon,FA}(\rho) \coloneqq \{\langle a, \mathbf{return}\ V\rangle | \exists \langle a', V\rangle \in R_{\Sigma,A}(\rho), a = \mathbf{return}\ a'\} \cup$$
$$\left\{\langle a, \widehat{\mu}^{\varepsilon}(N)\rangle \middle| (\varepsilon \sim \alpha.C) \in \Sigma, \langle a, N\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho)\right\}$$

$$R^{v}_{\Theta,\Sigma,e,A\to C}(\rho) \coloneqq \{\langle f, \lambda x.M\rangle | \forall \langle a, V\rangle \in R_{\Theta,\Sigma,e,A}(\rho), \langle f(a), (\lambda x.M)\ V\rangle \in R_{\Theta,\Sigma,e,C}(\rho)\}$$

$$R^{v}_{\Theta,\Sigma,e,\top}(\rho) \coloneqq \{\langle \star, \langle\rangle\rangle\} \qquad R^{v}_{\Theta,\Sigma,e,C_1\&C_2}(\rho) \coloneqq \left\{\langle\langle c_1, c_2\rangle, (M_1, M_2)\rangle \middle| \langle c_i, M_i\rangle \in R_{\Theta,\Sigma,e,C_i}(\rho)\right\}$$

$\top$- ***and*** $\top\top$-***liftings*** $\boxed{(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top} \subseteq (|[\![C]\!](\theta)| \to |[\![FG]\!](\theta)|) \times \mathcal{X}^{\boldsymbol{\Theta},\boldsymbol{\Sigma},\boldsymbol{e}}_{mon}(FG[C])}$ ***and*** $\boxed{(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top\top} \subseteq |[\![C]\!](\theta)| \times \boldsymbol{\lambda}^{\boldsymbol{\Sigma}}_{\mathbf{mon}}(C)}$

$$(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top} \coloneqq \left\{\langle f, \mathcal{X}\rangle \middle| \emptyset[\Theta] \mid \emptyset[\emptyset] \vdash_{\Sigma[\Sigma],\perp[e]} \mathcal{X}[\ ] : FG[C]. \forall\langle a, M\rangle \in R^{v}_{\Theta,\Sigma,e,C}(\rho)\ .\right.$$
$$\left. \exists V. \mathcal{X}[M] \simeq \mathbf{return}\ V \wedge \langle fa, \mathbf{return}\ V\rangle \in R^{v}_{\Sigma,\perp,FG}(\rho)\right\}$$

$$(R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top\top} \coloneqq \left\{\langle c, M\rangle \middle| \forall \langle f, \mathcal{X}\rangle \in (R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top}. \exists V. \mathcal{X}[M] \simeq \mathbf{return}\ V \wedge \langle fc, \mathbf{return}\ V\rangle \in R^{v}_{\Sigma,\perp,FG}(\rho)\right\}$$

$$R_{\Theta,\Sigma,e,C}(\rho) \coloneqq (R^{v}_{\Theta,\Sigma,e,C}(\rho))^{\top\top}$$

Figure 4.8: λ_{mon}-calculus logical relations

Lemma 4.10 (basic lemma). *For all* Θ, Σ, *derivations* $\Theta \mid \Gamma \vdash_{\Sigma,e} P : X$, $\langle \emptyset \mid \Sigma \vdash_k A_\alpha : \mathbf{Val} \rangle_{\alpha \in \Theta}$, $\theta = \langle [\![A_\alpha]\!] (\star) \rangle_{\alpha \in \Theta}$, $\rho = \left\langle R_{\emptyset,\Sigma,A_\alpha}(\star) \right\rangle_{\alpha \in \Theta}$, $\gamma \in [\![\Gamma]\!](\theta)$, *and* $\langle V_x \rangle_{x \in \mathrm{Dom}(\Gamma)}$,

if for all $x \in \mathrm{Dom}\ (\Gamma)$*:* $\langle \pi_x \gamma, V_x \rangle \in R_{\Theta,\Sigma,e,\Gamma(x)}(\rho)$ *then* $\left\langle [\![P]\!](\theta)\gamma, P[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} \right\rangle \in R_{\Theta,\Sigma,e,X}(\rho)$

and for all Θ, Σ, $\langle \emptyset \mid \Sigma \vdash_k A_\alpha : \mathbf{Val} \rangle_{\alpha \in \Theta}$, $\theta = \langle [\![A_\alpha]\!] (\star) \rangle_{\alpha \in \Theta}$, $\rho = \left\langle R_{\emptyset,\Sigma,A_\alpha}(\star) \right\rangle_{\alpha \in \Theta}$, *and* $\emptyset \mid \Sigma \vdash_k A, B : \mathbf{Val}$,

$$\begin{aligned}
&\forall(\varepsilon \sim (\alpha.C, N_u, N_b)) \in \Sigma : \\
&\langle [\![N_u]\!](\theta)(\star), N_u \rangle \in R_{(\alpha,\Theta),\Sigma,e,\alpha \to C}(\rho[\alpha \mapsto R_{\emptyset,\Sigma,A}(\star)]) \\
\wedge &\langle [\![N_b]\!](\theta)(\star), N_b \rangle \in R_{(\alpha_1,\alpha_2,\Theta),\Sigma,e,U_e(C[\alpha_1/\alpha]) \to U_e(\alpha_1 \to C[\alpha_2/\alpha]) \to C[\alpha_2/\alpha]}(\rho[\alpha_1 \mapsto R_{\emptyset,\Sigma,A}(\star), \alpha_2 \mapsto R_{\emptyset,\Sigma,B}(\star)])
\end{aligned}$$

Proof

The logical relations for values are not changed, so the proofs are exactly the same as for CBPV.

The interesting computation cases for λ_{mon} are:

(reify) Assume $\Theta \mid \Gamma \vdash_{\Sigma,\varepsilon} N : FA$ and $\left\langle [\![N]\!](\theta)\gamma, N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} \right\rangle \in R_{\Theta,\Sigma,\varepsilon,FA}(\rho)$. Note that $\vdash_k A[A_\alpha/\alpha]_{\alpha \in \Theta} : \mathbf{Val}$. By the induction hypothesis for N_u we know that

$$\langle [\![N_u]\!](\theta)(\star), N_u \rangle \in R_{(\alpha,\Theta),\Sigma,e,\alpha \to C}(\rho \left[\alpha \mapsto R_{\emptyset,\Sigma,A[A_\alpha/\alpha]_{\alpha \in \Theta}}(\star)\right])$$

which is equivalent to $\langle [\![N_u]\!](\theta)(\star), N_u \rangle \in R_{\Theta,\Sigma,e,A \to C[A/\alpha]}(\rho)$ by Lemma 4.9 and show that

$$\left\langle [\![\, [N]^{\varepsilon}]\!](\theta)\gamma,\ [N]^{\varepsilon}[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} \right\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho)$$

which is equivalent to

$$\left\langle [\![N]\!](\theta)(\gamma),\ [N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)}]^{\varepsilon} \right\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho).$$

To do so take an arbitrary $\langle f, \mathcal{X} \rangle \in (R^{v}_{\Theta,\Sigma,e,C[A/\alpha]}(\rho))^{\top}$ and show that

$$\langle f([\![N]\!](\theta)\gamma), \mathbf{return}\ V \rangle \in R_{\Theta,\Sigma,e,F[\bot]G}(\rho) \wedge \mathcal{X}[[N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)}]^{\varepsilon}] \simeq \mathbf{return}\ V$$

for some V. This is the case if $\langle f, \mathcal{X}[[\ ]^{\varepsilon}] \rangle \in (R^{v}_{\Sigma,\varepsilon,FA}(\rho))^{\top}$.

1. Take $\langle a, \mathbf{return}\ V' \rangle \in R^{v}_{\Sigma,\varepsilon,FA}(\rho)$ and show $\langle fa, \mathbf{return}\ V'' \rangle \in R^{v}_{\Sigma,\bot,FG}(\rho)$ and

$$\mathcal{X}[[\mathbf{return}\ V']^{\varepsilon}] \simeq \mathbf{return}\ V''.$$

We have $\mathcal{X}[[\mathbf{return}\ V']^{\varepsilon}] \simeq \mathcal{X}[N_u V']$ by Lemma 4.6. By the inductive hypothesis for N_u, we are done.

2. Take $\langle a, \widehat{\mu}^\varepsilon(N')\rangle \in R^v_{\Sigma,\varepsilon,FA}(\rho)$. We have $\mathcal{X}[\widehat{\mu}^\varepsilon(N')] \simeq \mathcal{X}[N_b\{N'\}\{\lambda x.\mathbf{return}\ x\}] \simeq$ **return** V' with $(fa, \mathbf{return}\ V') \in R^v_{\Theta,\Sigma,\bot,FG}(\rho)$ by the inductive hypotheses for N_u and N_b, Lemma 4.9 and because $\langle a, N'\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho)$.

(reflect) Assume $\Theta \mid \Gamma \vdash_{\Sigma,e} N : C[A/\alpha]$ and $\left\langle [\![N]\!]\,(\theta)\gamma, N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho)$. We have $\left\langle [\![\widehat{\mu}^\varepsilon(N)]\!]\,(\theta)\gamma, \widehat{\mu}^\varepsilon(N)[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Sigma,\varepsilon,FA}(\rho)$ if and only if

$$\left\langle [\![N]\!]\,(\theta)(\gamma), \widehat{\mu}^\varepsilon(N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle \in R_{\Sigma,\varepsilon,FA}(\rho).$$

Let $\langle f, \mathcal{X}\rangle \in (R^v_{\Sigma,\varepsilon,FA}(\rho))^\top$. We have $\left\langle [\![N]\!]\,(\theta)\gamma, \widehat{\mu}^\varepsilon(N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle \in R^v_{\Sigma,\varepsilon,FA}(\rho)$ because $\left\langle [\![N]\!]\,(\theta)\gamma, N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Theta,\Sigma,e,C[A/\alpha]}(\rho)$ and thus

$\mathcal{X}[\widehat{\mu}^\varepsilon(N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})] \simeq \mathbf{return}\ V$ and $\left\langle f([\![N]\!]\,(\theta)\gamma), \mathcal{X}[\widehat{\mu}^\varepsilon(N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})]\right\rangle \in R^v_{\Sigma,\bot,FG}(\rho)$ as needed.

(return) The case for return is not specific for λ_{mon}, but serves as an example for the omitted cases. Assume $\Theta \mid \Gamma \vdash_\Sigma V : A$, $\left\langle [\![V]\!]\,(\theta)\gamma, V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Sigma,A}(\rho)$. We have

$$\begin{aligned} &\left\langle [\![\mathbf{return}\ V]\!]\,(\theta)\gamma, \mathbf{return}\ V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Theta,\Sigma,e,FA}(\rho)\\ \iff &\left\langle \mathbf{return}\ [\![V]\!]\,(\theta)(\gamma), \mathbf{return}\ (V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)})\right\rangle \in R_{\Theta,\Sigma,e,FA}(\rho)\end{aligned}$$

Let $\langle f, \mathcal{X}\rangle \in (R^v_{\Theta,\Sigma,e,FA}(\rho))^\top$. We have $\left\langle \mathbf{return}\ [\![V]\!]\,(\theta)\gamma, \mathbf{return}\ V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R^v_{\Theta,\Sigma,e,FA}(\rho)$ because of the inductive hypothesis, thus

$$\langle f(\mathbf{return}\ [\![V]\!]\,(\theta)\gamma), \mathbf{return}\ V'\rangle \in R_{\Sigma,\bot,FA}(\rho) \wedge \mathcal{X}[\mathbf{return}\ V[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}] \simeq \mathbf{return}\ V'$$

as needed.

(let) Assume the inductive hypothesis for $\Theta \mid \Gamma \vdash_{\Sigma,e} M : FA$ and $\Theta \mid \Gamma, x : A \vdash_{\Sigma,e} N : C$. We have to show that

$$\left\langle [\![M]\!]\,(\theta)(\gamma) \gg\!= \lambda a.\,[\![N]\!]\,(\theta)(\gamma[x \mapsto a]), \mathbf{let}\ x \leftarrow M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\ \mathbf{in}\ N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\right\rangle \in R_{\Theta,\Sigma,e,C}(\rho).$$

Take $\langle f, \mathcal{X}\rangle \in (R^v_{\Theta,\Sigma,e,C}(\rho))^\top$. We have to show that there is a V with

$$\langle f([\![M]\!]\,(\theta)(\gamma) \gg\!= \lambda a.\,[\![N]\!]\,(\theta)(\gamma[x \mapsto a])), \mathbf{return}\ V\rangle \in R_{\Sigma,\bot,FG}(\rho)$$

and

$$\mathcal{X}[\mathbf{let}\ x \leftarrow M[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}\ \mathbf{in}\ N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}] \simeq \mathbf{return}\ V.$$

To do so show that

$$\left\langle \lambda y.f(y \gg\!= \lambda a.\,[\![N]\!]\,(\theta)(\gamma[x \mapsto a])), \mathcal{X}[\mathbf{let}\ x \leftarrow [\ ]\ \mathbf{in}\ N[V_x/x]_{x\in\mathrm{Dom}(\Gamma)}]\right\rangle \in (R^v_{\Theta,\Sigma,e,FA}(\rho))^\top.$$

Take $\langle c, N' \rangle \in R^{v}_{\Theta,\Sigma,e,FA}(\rho)$.

We have $f(c \gg\!= \lambda a. [\![N]\!](\theta)(\gamma[x \mapsto a])) = f(([\![N_b]\!](\theta)(\star))\ c\ (\lambda a. [\![N]\!](\theta)(\gamma[x \mapsto a]))$ and $\mathcal{X}[\mathbf{let}\ x \leftarrow N'\ \mathbf{in}\ N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)}] \simeq \mathcal{X}\left[N_b\ \{N'\}\ \{N[V_x/x]_{x \in \mathrm{Dom}(\Gamma')}\}\right]$ by lemma Lemma 4.5 and Lemma 4.6. By the inductive hypothesis for N_b we are done.

(leteffect) Assume $\Theta \mid \Gamma \vdash_{\Sigma',e} N : D$ and $\left\langle [\![N]\!](\theta)\gamma, N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} \right\rangle \in R_{\Theta,\Sigma',e,D}(\rho)$ for $\Sigma' = \Sigma, \varepsilon \succ e \sim (\alpha.C, N_u, N_b)$.

We have have to show that

$$\left\langle [\![N]\!](\theta)(\gamma), \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ N[V_x/x]_{x \in \mathrm{Dom}(\Gamma)} \right\rangle$$

is in $R_{\Theta,\Sigma,e,D}(\rho) = (R^{v}_{\Theta,\Sigma,e,D}(\rho))^{\top\top}$.

Let $\langle f, \mathcal{X} \rangle \in (R^{v}_{\Theta,\Sigma,e,D}(\rho))^{\top}$. We show that $\langle f, \mathcal{X}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ [\ ]] \rangle$ is in fact in $(R^{v}_{\Sigma',e,D}(\rho))^{\top}$, which directly implies the claim together with the inductive hypothesis.

We only prove the case for $D = FA$ here, all other cases are similar.

So let $D = FA$ and $\langle c, M \rangle \in R^{v}_{\Sigma',e,FA}(\rho)$.

1. If M if of the shape **return** V' we have

$$\mathcal{X}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M] \simeq \mathcal{X}[M]$$

because of Lemma 4.6. Because $\langle f, \mathcal{X} \rangle \in (R^{v}_{\Sigma,e,FA}(\rho))^{\top}$ we find V with $\mathcal{X}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M] \simeq \mathcal{X}[M] \simeq \mathbf{return}\ V$ and $\langle fc, \mathbf{return}\ V \rangle \in R^{v}_{\Sigma,\perp,FG}(\rho)$

2. If $e = \varepsilon'$ and M is of the shape $\widehat{\mu}^{\varepsilon'}(N')$, we can split up $\mathcal{X}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \widehat{\mu}^{\varepsilon'}(N')]$ to

$$M' := \mathcal{X}'[[\mathcal{C}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \widehat{\mu}^{\varepsilon'}(N')]]^{\varepsilon'}]$$

because $\mathcal{X}[M]$ is a $\perp$-computation and $\widehat{\mu}^{\varepsilon'}(N)$ a computation at level $e = \varepsilon'$ and find

$$\begin{aligned} M' &\longrightarrow N_b\ N'\ (\lambda x.\mathcal{C}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \mathbf{return}\ x]) \\ &\simeq N_b\ N'\ (\lambda x.\mathcal{C}[\mathbf{return}\ x]) \\ &\longleftarrow \mathcal{X}'[[\mathcal{C}[\widehat{\mu}^{\varepsilon'}(N')]]^{\varepsilon'}] \\ &= \mathcal{X}[\widehat{\mu}^{\varepsilon'}(N')] \end{aligned}$$

and thus $\mathcal{X}[\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \widehat{\mu}^{\varepsilon'}(N')] \simeq \mathcal{X}[\widehat{\mu}^{\varepsilon'}(N')]$ as wanted.

The case for the effect hierarchies follows immediately by induction.

■

Theorem 4.11 (adequacy). *Denotational equivalence implies contextual equivalence. Explicitly: for all* $\Theta \mid \Gamma \vdash_{\Sigma,e} P, Q : X$*, if* $[\![P]\!] = [\![Q]\!]$ *then* $P \simeq Q$.

Proof

Let $\Theta \mid \Gamma \vdash_{\Sigma,e} P_1, P_2 : X$ be any well-typed phrases satisfying $[\![P_1]\!] = [\![P_2]\!]$ where both $[\![P_i]\!]$ are defined. Consider any closed well-typed ground-returner context $\emptyset[\Theta] \mid \emptyset[\Gamma'] \vdash_{\emptyset[E],\bot[e]} \mathcal{X}[\ ] : \mathsf{FG}[X]$ with $\Gamma' \geqslant \Gamma$ s.t. both $[\![\mathcal{X}[P_i]]\!]$ are defined. By Lemma 4.3, $[\![\mathcal{X}[P_1]]\!]\,(\theta)(\star) = [\![\mathcal{X}[P_2]]\!]\,(\theta)(\star)$. Let c be this common denotation.

Consider any $i \in \{1, 2\}$. By the basic lemma:

$$\langle c, \mathcal{X}[P_i]\rangle \in R_{\emptyset,\emptyset,\bot,\mathsf{FG}}(\star) = (R^{v}_{\emptyset,\emptyset,\bot,\mathsf{FG}}(\star))^{\top\top}$$

Now because $\langle \mathrm{id}, [\ ]\rangle \in (R^{v}_{\emptyset,\emptyset,\bot,\mathsf{FG}}(\star))^{\top}$ we have for each $i \in \{1,2\}$, $\langle c, \mathbf{return}\ V_i\rangle \in R^{v}_{\emptyset,\emptyset,\bot,\mathsf{FG}}(\star)$ with $\mathbf{return}\ V_i \simeq \mathcal{X}[P_i]$ for some V_i. By the definition of $R^{v}_{\emptyset,\emptyset,\bot,\mathsf{FG}}(\star)$, there exist a_i with $\langle a_i, V_i\rangle \in R_{\emptyset,\emptyset,\bot,\mathsf{G}}(\star)$ such that $c = \mathbf{return}\ a_i$. But **return** is the identity for the monad belonging to FG at level $\bot$.

Thus:

$$a_1 = \mathbf{return}\ a_1 = c = \mathbf{return}\ a_2 = a_2$$

Therefore, by Lemma 4.8, $V_1 \simeq V_2$. Therefore:

$$\mathcal{X}[P_1] \simeq \mathbf{return}\ V_1 \simeq \mathbf{return}\ V_2 \simeq \mathcal{X}[P_2]$$

Therefore $\mathcal{X}[P_1] \longrightarrow^{\star} \mathbf{return}\ V$ iff $\mathcal{X}[P_2] \longrightarrow^{\star} \mathbf{return}\ V$, hence $P_1 \simeq P_2$. ■

Filinski also proves Felleisen-style type soundness [22] for his calculus, but via the route of progress and preservation and under the presence of recursion and possible non-termination. Using adequacy, we can also prove strong normalisation for our calculus.

Corollary 4.12 (soundness and strong normalisation). *All well-typed, effect-free closed ground returners reduce to a normal form. Explicitly: for all* $\vdash_{\Sigma,\bot} M : \mathsf{FG}$ *there exists some* $\vdash_{\Sigma} V : G$ *such that:*

$$M \longrightarrow^{\star} \mathbf{return}\ V$$

Proof

By the basic lemma, $\langle [\![M]\!]\,(\theta)\gamma, M\rangle \in R_{\emptyset,\Sigma,\bot,\mathsf{FG}}(\star) = (R^{v}_{\emptyset,\Sigma,\bot,\mathsf{FG}}(\star))^{\top\top}$. Now because

$\langle \mathrm{id}, [\ ] \rangle \in (R^{v}_{\emptyset,\Sigma,\bot,FG}(\star))^{\top}$ we have $\langle [\![M]\!]\,(\theta)\gamma, \textbf{return}\ V \rangle \in R^{v}_{\emptyset,\Sigma,\bot,FG}(\star)$ with $\textbf{return}\ V \simeq M$ for some V. By setting the context in the definition of contextual equivalence to the empty one we find $M \longrightarrow^{*} \textbf{return}\ V$. ■

We have now established that our introduced semantics is adequate. The proof using logical relations was more involved than the proof for λ_{eff}, because we had to use $\top\top$-lifting and incorporate the type variable contexts. We will now use the adequacy results from the last two chapters to compare the expressivness of λ_{mon} and λ_{eff}.

Chapter 5

Expressiveness

We now come to the main contribution of the thesis and analyse the expressive power of λ_{eff} and λ_{mon}. We base this prove on a notion of macro expressability [5]. To express λ_{eff} in λ_{mon}, we have to express the syntactic constructs of λ_{eff} that do not occur in CBPV by macros in terms of λ_{mon}.

We first will define what it means for λ_{eff} to be macro (typed) expressible in λ_{mon} (and vice versa). The concept of macro expressability is adapted from Felleisen [5]. We then prove that λ_{eff} can macro express λ_{mon}, i.e. that there is a structure preserving translation from λ_{mon} to λ_{eff}. Finally, we prove that λ_{mon} cannot typed macro express λ_{eff}, because λ_{mon} only has finitely many observationally different terms whereas λ_{eff} has infinitely many.

5.1 Definition of macro expressability

Felleisen [5] introduced a formal notion of expressiveness of a programming language. His definition of macro expressability is relative: he defines what it means for a programming language to be able to express certain additional constructs. The idea is that an extension $\mathcal{L}'$ of a language $\mathcal{L}$ can be expressed in the language $\mathcal{L}$ itself if $\mathcal{L}'$-programs can be translated to $\mathcal{L}$ programs and the common program structure can be left unchanged.

As most programming languages are Turing-complete (and so are λ_{eff} and λ_{mon} if one adds certain types and recursion), the Church-Turing thesis states that there is always a translation. Macro expressability refines the notion of expressability and requires a local translation, that does not change the program structure.

We extend Felleisens concept and define expressiveness for the two extensions λ_{mon} and λ_{eff} of CBPV.

Definition 5.1. *We say that λ_{mon} is typed macro expressible in λ_{eff} if there is a family of functions $\_{\Sigma} : Terms_{\lambda_{\text{mon}}} \to Terms_{\lambda_{\text{eff}}}$ where the parameter Σ is an effect hierarchy such that:*

1. *The functions* $\underline{}_\Sigma$ *are homomorphic on all* CBPV *constructs, i.e.:*

$$\begin{aligned}
\underline{()}_\Sigma &= () & \underline{V!}_\Sigma &= \underline{V}_\Sigma! \\
\underline{(V_1, V_2)}_\Sigma &= (\underline{V_1}_\Sigma, \underline{V_2}_\Sigma) & \underline{\mathbf{return}\ V}_\Sigma &= \mathbf{return}\ \underline{V}_\Sigma \\
\underline{\mathbf{inj}_i\ V}_\Sigma &= \mathbf{inj}_i\ \underline{V}_\Sigma & \underline{\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N}_\Sigma &= \mathbf{let}\ x \leftarrow \underline{M}_\Sigma\ \mathbf{in}\ \underline{N}_\Sigma \\
\underline{\{M\}}_\Sigma &= \{\underline{M}_\Sigma\} & \underline{\lambda x.M}_\Sigma &= \lambda x.\underline{M}_\Sigma \\
\underline{\mathbf{split}(V, x_1.x_2.M)}_\Sigma &= \mathbf{split}(\underline{V}_\Sigma, x_1.x_2.\underline{M}_\Sigma) & \underline{M\ V}_\Sigma &= \underline{M}_\Sigma\ \underline{N}_\Sigma \\
\underline{\mathbf{case}_0(V)}_\Sigma &= \mathbf{case}_0(\underline{V}_\Sigma) & \underline{\langle M_1, M_2\rangle}_\Sigma &= \left\langle \underline{M_1}_\Sigma, \underline{M_2}_\Sigma \right\rangle \\
\underline{\mathbf{case}(V, x_1.M_1, x_2.M_2)}_\Sigma &= \mathbf{case}(\underline{V}_\Sigma, x_1.\underline{M_1}_\Sigma, x_2.\underline{M_2}_\Sigma) & \underline{\mathbf{prj}_i\ M}_\Sigma &= \mathbf{prj}_i\ \underline{M}_\Sigma
\end{aligned}$$

2. *For the additional constructs of* λ_{mon} *we have:*

$$\begin{aligned}
&\underline{[N]^\varepsilon}_\Sigma = \mathcal{X}_1^\varepsilon[\underline{N}_\Sigma] \\
&\underline{\hat{\mu}^\varepsilon(N)}_\Sigma = \mathcal{X}_2^\varepsilon[\underline{N}_\Sigma] \\
&\underline{\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ N}_\Sigma = \mathcal{X}_3^\varepsilon[\underline{N}_{\Sigma'}]
\end{aligned}$$

for contexts $\mathcal{X}_i$ *of* λ_{eff} *and* $\Sigma' := \Sigma, \varepsilon \sim (\alpha.C, N_u, N_b)$.

3. *If* $\overset{\lambda_{\text{mon}}}{\vdash}_{\emptyset,\perp} P : FG$ *and* P *is proper, then for all proper* $\overset{\lambda_{\text{mon}}}{\vdash}_{\emptyset} V : G$ *it holds that* $P \xrightarrow{\lambda_{\text{mon}}}{}^{*}$ $\mathbf{return}\ V \iff \underline{P}_\emptyset \xrightarrow{\lambda_{\text{eff}}}{}^{*}\ \mathbf{return}\ V$.

4. *For all proper* P *with* $\overset{\lambda_{\text{mon}}}{\vdash}_{\Sigma,e} P : X$ *there is a* λ_{eff}*-type* $\underline{X}$ *and an effect signature* E *with* $\overset{\lambda_{\text{eff}}}{\vdash}_{E} P : \underline{X}$.

Leaving out condition 4 and the semantic definability (i.e. properness) requirement yields the notion of untyped macro expressability of λ_{mon} *in* λ_{eff}.

Definition 5.2. *We say that* λ_{eff} *is typed macro expressible in* λ_{mon} *if there is a function* $\underline{} : \mathit{Terms}_{\lambda_{\text{eff}}} \to \mathit{Terms}_{\lambda_{\text{mon}}}$ *such that:*

1. *The function* $\underline{}$ *is homomorphic on all* CBPV *constructs, as in the last definition.*

2. *For the additional constructs of* λ_{eff} *we have:*

$$\begin{aligned}
\underline{\mathbf{handle}\ N\ \mathbf{with}\ H} &= \mathcal{X}_H[\underline{E}] \\
\underline{\mathrm{op}_V f} &= \mathcal{X}_{\text{op}}[\underline{V}][\underline{f}]
\end{aligned}$$

for λ_{mon} *contexts* $\mathcal{X}_H, \mathcal{X}_{\text{op}}$ *with the last one being a context with two holes, where* $\underline{P}$ *is always proper.*

3. *If* $\overset{\lambda_{\text{eff}}}{\vdash}_{\emptyset} P : FG$, *then* $P \xrightarrow{\lambda_{\text{eff}}}{}^{*}\ \mathbf{return}\ V \iff \underline{P} \xrightarrow{\lambda_{\text{mon}}} \mathbf{return}\ V$.

4. *For all* P *with* $\overset{\lambda_{\text{eff}}}{\vdash}_{E} P : X$ *there is a* λ_{mon}*-type* $\underline{X}$, *an effect hierarchy* Σ *and an effect* $e \in \Sigma$

with $\overset{\lambda_{\text{mon}}}{\vdash}_{\Sigma,e} P : \underline{X}$.

Leaving out condition 4 yields the notion of macro expressability of λ_{eff} *in* λ_{mon}.

5.2 λ_{mon} is macro expressible in λ_{eff}

We first give an untyped translation from λ_{mon} to λ_{eff} and prove a simulation result. To express λ_{mon} in λ_{eff} we use the idea that $\widehat{\mu}^{\varepsilon}(N)$ behaves like an effect operation and $[N]^{\varepsilon}$ like the handling construct.

$$\begin{aligned} \underline{\widehat{\mu}^{\varepsilon}(N)}_{\Sigma} &= \mathrm{op}^{\varepsilon}\ \{\underline{N}_{\Sigma}\}\ (\lambda x.\mathbf{return}\ x)\\ \underline{[N]^{\varepsilon}}_{\Sigma} &= \mathbf{handle}\ \underline{N}_{\Sigma}\ \mathbf{with}\ H^{\Sigma}_{\varepsilon}\\ &\quad \text{for } (\varepsilon \sim (N_u, N_b)) \in \Sigma \text{ and}\\ H^{\Sigma}_{\varepsilon} &:= \left\{\mathbf{return}\ V \mapsto \underline{N_u}_{\Sigma} V,\ \mathrm{op}^{\varepsilon} p\ f \mapsto \underline{N_b}_{\Sigma}\ p\ f\right\}\\ &\quad \cup \left\{\mathrm{op}^{\varepsilon'} p\ f \mapsto \mathrm{op}^{\varepsilon'} p\ f \,\middle|\, \varepsilon \neq \varepsilon' \in E\right\}\\ \underline{\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ N}_{\Sigma} &= \underline{N}_{\Sigma,\varepsilon\sim(N_u,N_b)} \end{aligned}$$

In the second case, if ε does not occur in Σ, the translation is not defined. Note that if $\underline{N}_{\Sigma}$ is defined, $\underline{N}_{\Sigma'}$ is defined for every $\Sigma' \supseteq \Sigma$.

Note that the translation ignores any type information.

We first prove a technical lemma:

Lemma 5.3. *Let* Σ *be an effect hierarchy,* $\mathcal{H}$ *a hoisting context and* $\Theta \mid \Gamma \vdash_{\Sigma,e} \mathcal{H}[\widehat{\mu}^{\varepsilon}(N) : C$. *If* $\varepsilon' \notin \mathit{effects}(\mathcal{H})$ *then for all* $\varepsilon \preceq \varepsilon' \in \Sigma$: $\underline{\mathcal{H}[\widehat{\mu}^{\varepsilon}(N)]}_{\Sigma} \xrightarrow{\lambda_{\text{eff}}}{}^{*}\ \mathrm{op}^{\varepsilon}\ \{N\}(\lambda x.\underline{\mathcal{H}[\mathbf{return}\ x]}_{\Sigma'})$ *for some* $\Sigma' \supseteq \Sigma$

We now show that whenever a program makes a single step in λ_{mon}, the translated program can mirror this via several steps in λ_{eff} (where possibly, if the step comes from a leteffect, the parameter is extended):

Lemma 5.4. *If* $\vdash_{\Sigma,e} P : X$ *for some type* X, *then* $\vdash_{\Sigma} P \xrightarrow{\lambda_{\text{mon}}} P' \implies \underline{P}_{\Sigma} \xrightarrow{\lambda_{\text{eff}}}{}^{*}\ \underline{P'}_{\Sigma'}$ *for some* $\Sigma' \supseteq \Sigma$.

Proof

By induction over $P \xrightarrow{\lambda_{\text{mon}}} P'$. If the reduction is a CBPV reduction, everything follows immediately by the inductive hypotheses. The only interesting cases are:

(reify) Case $[\mathbf{return}\ V]^{\varepsilon} \xrightarrow{\lambda_{\text{mon}}} N_u\ V$. We have

$$\underline{[\mathbf{return}\ V]^{\varepsilon}}_{\Sigma} = \mathbf{handle}\ \mathbf{return}\ V\ \mathbf{with}\ H^{E}_{\varepsilon} \xrightarrow{\lambda_{\text{eff}}} \underline{N_u}_{\Sigma} V = \underline{N_u\ V}_{\Sigma}$$

(reflect) Case $(\varepsilon \sim (\alpha.C, N_u, N_b)) \in \Sigma$ and $[\mathcal{H}[\widehat{\mu}^{\varepsilon}(N)]]^{\varepsilon} \xrightarrow{\lambda_{\text{mon}}} N_b\,\{N\}\,(\lambda x.\,[\mathcal{H}[\mathbf{return}\ x]]^{\varepsilon})$. We have:

$$\begin{aligned}
\underline{[\mathcal{H}[\widehat{\mu}^{\varepsilon}(N)]]^{\varepsilon}}_{\Sigma} &= \mathbf{handle}\ \underline{\mathcal{H}[\widehat{\mu}^{\varepsilon}(N)]}_{\Sigma}\ \mathbf{with}\ H^{\Sigma}_{\varepsilon} \\
&\xrightarrow{\lambda_{\text{eff}}}{}^{*}\ \mathbf{handle}\ \mathrm{op}^{\varepsilon}_{\{\underline{N}_{\Sigma}\}}(\lambda x.\underline{\mathcal{H}[\mathbf{return}\ x]}_{\Sigma'})\ \mathbf{with}\ H^{\Sigma}_{\varepsilon} \quad \text{(Lemma 5.3)} \\
&\xrightarrow{\lambda_{\text{eff}}} \underline{N_b}_{\Sigma}\,\{\underline{N}_{\Sigma}\}\,(\lambda x.\mathbf{handle}\ \underline{\mathcal{H}[\mathbf{return}\ x]}_{\Sigma'}\ \mathbf{with}\ H^{\Sigma}_{\varepsilon}) \\
&= \underline{N_b}_{\Sigma}\,\{\underline{N}_{\Sigma}\}\,(\lambda x.\,\underline{[\mathcal{H}[\mathbf{return}\ x]]^{\varepsilon}}_{\Sigma'}) \\
&= \underline{N_b\,\{N\}\,(\lambda x.\,[\mathcal{H}[\mathbf{return}\ x]]^{\varepsilon})}_{\Sigma'}
\end{aligned}$$

(leteff) Case $\underline{M} \xrightarrow{\lambda_{\text{eff}}} \underline{M}'$ and $\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M \xrightarrow{\lambda_{\text{mon}}} \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M'$. We have for $E' = \varepsilon \sim (\underline{N_u}_{\Sigma}, \underline{N_b}_{\Sigma}), E$:

$$\begin{aligned}
&\underline{\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M}_{\Sigma} \\
= {}&\underline{M}_{E'} \\
\xrightarrow{\lambda_{\text{eff}}} {}&\underline{M'}_{E'} \\
= {}&\underline{\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ M'}_{\Sigma}
\end{aligned}$$

(leteff-return) Case $\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \mathbf{return}\ V \xrightarrow{\lambda_{\text{mon}}} \mathbf{return}\ V$. We have

$$\underline{\mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.C, N_u, N_b)\ \mathbf{in}\ \mathbf{return}\ V}_{\Sigma} = \mathbf{return}\ V = \underline{\mathbf{return}\ V}_{\Sigma}$$

■

We proved that if P makes a step, $\underline{P}_{\emptyset}$ simulates that step in zero or more steps. It is not the case that if $\underline{P}_{\emptyset}$ can make a step, then so can P, because

$$\underline{\mathbf{let}\ y \leftarrow \widehat{\mu}^{\varepsilon}(N)\ \mathbf{in}\ M}_{\Sigma} \longrightarrow \mathrm{op}^{\varepsilon}\,\{\underline{N}_{\Sigma}\}\,(\lambda x.\mathbf{let}\ y \leftarrow \mathbf{return}\ x\ \mathbf{in}\ \underline{M}_{\Sigma})$$

but there is no P' with $\underline{P'}_{\Sigma} = \mathrm{op}^{\varepsilon}\,\{\underline{N}_{\Sigma}\}\,(\lambda x.\mathbf{let}\ y \leftarrow \mathbf{return}\ x\ \mathbf{in}\ \underline{M}_{\Sigma})$. We can however prove the following result:

Lemma 5.5. *If* $\overset{\lambda_{\text{mon}}}{\vdash}_{\emptyset,\perp} P : \mathsf{FG}$*, then* $\underline{P}_{\emptyset} \xrightarrow{\lambda_{\text{eff}}}{}^{*} \mathbf{return}\ V \implies P \xrightarrow{\lambda_{\text{mon}}}{}^{*} \mathbf{return}\ V$

Proof

By $\vdash_{\emptyset,\perp} P : \mathsf{FG}$ and Lemma 4.12 $P \simeq \mathbf{return}\ V'$ for some V', i.e. $P \xrightarrow{\lambda_{\text{mon}}}{}^{*} \mathbf{return}\ V'$. By the last lemma, $\underline{P} \xrightarrow{\lambda_{\text{eff}}}{}^{*} \underline{\mathbf{return}\ V'}_{\Sigma} = \mathbf{return}\ V'$, so $V = V'$ because reduction for λ_{eff} is deterministic.

Thus, $P \xrightarrow{\lambda_{\text{mon}}}{}^{*}$ **return** V. ■

Both lemmas taken together yield the necessary condition 3 of macro expressability:

Corollary 5.6. *If* $\overset{\lambda_{\text{mon}}}{\vdash_{\emptyset,e}} P : FG$, *then* $P \xrightarrow{\lambda_{\text{mon}}}{}^{*}$ **return** $V \iff \underline{P}_{\emptyset} \xrightarrow{\lambda_{\text{eff}}}{}^{*}$ **return** V.

Proof

If $P \xrightarrow{\lambda_{\text{mon}}}{}^{*}$ **return** V, then $\underline{P}_{\emptyset} \xrightarrow{\lambda_{\text{eff}}}{}^{*} \underline{\textbf{return}\ V}_{\Sigma} = \textbf{return}\ V$, the other direction is the corollary. ■

Theorem 5.7. λ_{eff} *can macro express* λ_{mon}

Proof

The function $\underline{\ \ }_{\Sigma}$ obviously fulfills the first and second necessary property of macro expressiveness. The third property was proven in the last lemma. ■

However, the given translation does not suffice to also prove macro typed expressability:

Lemma 5.8. *There is* $\overset{\lambda_{\text{mon}}}{\vdash_{\emptyset,\perp}} P : X$ *such that* $\underline{P}_{\emptyset}$ *is not typeable in* λ_{eff}.

Proof

Look at the program

$$\begin{aligned}&\textbf{leteffect}\ \varepsilon \succ \perp\ \textbf{be}\ (\alpha.F\alpha, \textbf{return}_{\perp}, \gg\!=_{\perp})\ \textbf{in}\\ &\qquad [\textbf{let}\ x \leftarrow \widehat{\mu}^{\varepsilon}(\textbf{return}\ ())\ \textbf{in let}\ y \leftarrow \widehat{\mu}^{\varepsilon}(\textbf{return}\ ((),()))\ \textbf{in return}\ (x,y)]^{\varepsilon}\end{aligned}$$

This program has type $F(1 \times (1 \times 1))$.

The translation of this contains two occurenes of op^{ε}, one where the parameter is of type $U(F1)$, one where it is of type $U(F(1 \times 1))$. This is not allowed in λ_{eff}. ■

The problematic program is pathological, but the very same problem arises for instance when translating the continutation monad in λ_{mon} to λ_{eff}. We conjecture that this problem cannot be simply overcome without changing the type system of λ_{eff}. We describe this conjecture in more detail in Chapter 6.

5.3 λ_{eff} is not macro typed expressible in λ_{mon}

The key idea in proving that λ_{eff} is not expressible in λ_{mon} is that λ_{eff} has infinitely many observationally distinguishable terms, but λ_{mon} only has finitely many obervationally different terms at any type, as the adequate denotational semantics given in section 4.3 is finite.

To do so, we define the syntactic abbreviation $N ; M := \textbf{let}\ \_ \leftarrow N\ \textbf{in}\ M$. Furthermore, we define the generic effect operation $\mathrm{tick} := \mathrm{op}\ ()(\lambda x.\textbf{return}\ x)$ as well as $\mathrm{tick}^0 := \textbf{return}\ ()$ and $\mathrm{tick}^{n+1} := \mathrm{tick} ; \mathrm{tick}^n$. Note that for any n and any E with $(\mathrm{op} : 1 \to 1) \in E$ we have $\vdash_E \mathrm{tick}^n : F1$.

Recall that $\mathbb{F}_n$ is the finite type with n elements and succ the successor function on this type.

Define $\emptyset \vdash H^n : 1\ {}^{\{\mathrm{op}:1\to 1\}}\!\!\Rightarrow^{\emptyset} \mathbb{F}_n$ as $\textbf{return}\ V \mapsto \textbf{return}\ (), \mathrm{op}\ n\ f \mapsto \mathrm{succ}(f())$.

Note that for natural numbers n and m with $m \leqslant n$ we have $\vdash_{\emptyset} \textbf{handle}\ \mathrm{tick}^m\ \textbf{with}\ H^n : \mathbb{F}_n$ and $\textbf{handle}\ \mathrm{tick}^m\ \textbf{with}\ H^n \xrightarrow{\lambda_{\text{eff}}}{}^{*} \textbf{return}\ m_n$.

Lemma 5.9. *For any natural number* n, $m_1 \leqslant n$ *and* $m_2 \leqslant n$ *with* $m_1 \neq m_2$ *there is a handler* H *such that* $\textbf{handle}\ \mathit{tick}^{m_1}\ \textbf{with}\ H \not\simeq \textbf{handle}\ \mathit{tick}^{m_2}\ \textbf{with}\ H$.

Proof

We have $\textbf{handle}\ \mathrm{tick}^{m_i}\ \textbf{with}\ H^n \xrightarrow{\lambda_{\text{eff}}}{}^{*} \textbf{return}\ (m_i)_n$ by the last lemma, but $\textbf{return}\ (m_1)_n \not\simeq \textbf{return}\ (m_2)_n$, thus

$$\textbf{handle}\ \mathrm{tick}^{m_1}\ \textbf{with}\ H \not\simeq \textbf{handle}\ \mathrm{tick}^{m_2}\ \textbf{with}\ H.$$

■

We call two λ_{eff}-terms $\Gamma \vdash_E N, M : X$ obervationally different if $N \not\simeq M$.

Lemma 5.10. λ_{eff} *has countably many obervationally different terms of type*

$$\vdash_k F1 : \textbf{Comp}_{\{\mathrm{op}:1\to 1\}}.$$

We can use this to prove that λ_{eff} is not typed macro expressible in λ_{mon}. Explicitly:

Theorem 5.11. *There is no function* $\underline{\ \ }$: *Terms*$_{\lambda_{\text{eff}}} \to$ *Terms*$_{\lambda_{\text{mon}}}$ *such that* $\underline{\ \ }$ *is homomorphic on all syntactic constructs of* CBPV *and we have for all programs* P*:*

If $\overset{\lambda_{\text{eff}}}{\vdash_{\emptyset}} P : FG$*, then* $\underline{P} \xrightarrow{\lambda_{\text{mon}}}{}^{*} \textbf{return}\ V \iff P \xrightarrow{\lambda_{\text{eff}}}{}^{*} \textbf{return}\ V$

Proof

Consider how terms $\overset{\lambda_{\text{eff}}}{\vdash}_{\{\mathrm{tick}:1\to F1\}} N : F1$ get translated. They have to be terms $\underline{N} : X$ for some type X. Now because λ_{mon} has a finite model (Lemma 4.1), we can only have finitely many obervationally different terms in the type X, say k many. Consider the $k+1$ terms $\mathrm{tick}^0, \ldots, \mathrm{tick}^k$. Their translations cannot all be obervationally different, so we have $n \neq m$ with $n, m \leqslant k$ such that $\underline{\text{tick}^n} \simeq \underline{\text{tick}^m}$. But we have

$$\underline{\textbf{handle}\ \text{tick}^n\ \textbf{with}\ H^k} \xrightarrow{\lambda_{\text{mon}}}{}^{*} \underline{\textbf{return}\ n_k} = \textbf{return}\ n_k$$
$$\underline{\textbf{handle}\ \text{tick}^m\ \textbf{with}\ H^k} \xrightarrow{\lambda_{\text{mon}}}{}^{*} \underline{\textbf{return}\ m_k} = \textbf{return}\ m_k$$

and $n_k \neq m_k$, so $\underline{\textbf{handle}\ \text{tick}^n\ \textbf{with}\ H^k} \not\simeq \underline{\textbf{handle}\ \text{tick}^m\ \textbf{with}\ H^k}$, which is a contradiction. ■

We have now established that λ_{mon} cannot typed macro express λ_{eff}, because λ_{eff} has infinitely many observationally distinguishable terms. The λ_{eff}-calculus can untyped macro express λ_{mon}, by implementing reification as a handler for the reflection effect.

Chapter 6

Conclusion

In this thesis, we compared the macro expressiveness of effect handlers and monadic reflection.

As it turns out, effects and handlers cannot be expressed using monadic reflection. Our proof in section 5.3 is based on a finite, set-theoretic adequate model for the λ_{mon}-calculus and the fact that λ_{eff} has infinitely many observationally different terms. The field of study of the expressiveness of programming languages is extensive, but usually considers positive expressability results. Negative results, i.e. to show that no translation exists, are harder to achieve.

Monadic reflection can be expressed in terms of untyped effect operations and handlers, as proven in section 5.2. We conjecture that there is no translation such that the resulting program is always well-typed. Such a translation would have to implement effect operations and handlers for the continuation monad. However, as the continuation monad is unranked, and effect handlers can only implement ranked monads, we conjecture this to be impossible.

Apart from carrying out this proof, it would be interesting for future work to analyse if our simple type system for λ_{eff} could be, for example, extended with polymorphism, to be able to macro typed express λ_{mon}.

It would also be interesting to analyse further translations. As handlers are a form of a delimited control structure, we want to compare effect handlers and monadic reflection to a calculus with direct access to delimited control features. For all pairs of calculi it would also be interesting to analyse whether, if local translations are not possible, there are still global translations possible.

Bibliography

[1] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[2] Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, July 2001. ISSN 0956-7968. doi: 10.1017/S0956796801004099. URL `http://dx.doi.org/10.1017/S0956796801004099`.

[3] Christian Doczkal and Jan Schwinghammer. Formalizing a strong normalization proof for moggi's computational metalanguage: A case study in isabelle/hol-nominal. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 57–63, New York, NY, USA, 2009. ACM. doi: 10.1145/1577824.1577834. URL `http://doi.acm.org/10.1145/1577824.1577834`.

[4] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL*. ACM, 1988.

[5] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[6] Andrzej Filinski. On the relations between monadic semantics. *Theoret. Comput. Sci.*, 375(1-3), 2007.

[7] Andrzej Filinski. Monads in action. *SIGPLAN Not.*, 45(1):483–494, January 2010. ISSN 0362-1340. doi: 10.1145/1707801.1706354. URL `http://doi.acm.org/10.1145/1707801.1706354`.

[8] Claudio Hermida. *Fibrations, logical predicates and related topics*. PhD thesis, PhD thesis, University of Edinburgh, 1993. Tech. Report ECS-LFCS-93-277. Also available as Aarhus Univ. DAIMI Tech. Report PB-462, 1993.

[9] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *SIGPLAN Not.*, 48(9): 145–158, September 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500590. URL `http://doi.acm.org/10.1145/2544174.2500590`.

[10] Shin-ya Katsumata. Relating computational effects by $\top\top$-lifting. *Inf. Comput.*, 222:228–246, 2013. doi: 10.1016/j.ic.2012.10.014. URL `http://dx.doi.org/10.1016/j.ic.2012.10.014`.

[11] Paul Blain Levy. *Typed Lambda Calculi and Applications: 4th International Conference, TLCA'99 L'Aquila, Italy, April 7–9, 1999 Proceedings*, chapter Call-by-Push-Value: A Subsuming Paradigm, pages 228–243. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48959-7. doi: 10.1007/3-540-48959-2_17. URL `http://dx.doi.org/10.1007/3-540-48959-2_17`.

[12] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[13] Sam Lindley and Ian Stark. Reducibility and ⊤⊤-lifting for computation types. In *Typed Lambda Calculi and Applications: Proceedings of the Seventh International Conference TLCA 2005*, number 3461 in Lecture Notes in Computer Science, pages 262–277. Springer-Verlag, 2005. URL `http://www.inf.ed.ac.uk/~stark/reducibility.html`.

[14] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*. IEEE Computer Society, 1989.

[15] A. M. Pitts and I. D. B. Stark. Higher order operational techniques in semantics. chapter Operational Reasoning for Functions with Local State, pages 227–274. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63168-8. URL `http://dl.acm.org/citation.cfm?id=309656.309671`.

[16] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical. Structures in Comp. Sci.*, 10(3):321–359, June 2000. ISSN 0960-1295. doi: 10.1017/S0960129500003066. URL `http://dx.doi.org/10.1017/S0960129500003066`.

[17] Gordon Plotkin. Lambda-definability and logical relations. 1973.

[18] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FoSSaCS*. Springer-Verlag, 2002.

[19] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.

[20] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP*. Springer-Verlag, 2009.

[21] John C. Reynolds. *Theories of Programming Languages*. Paperback re-issue. Cambridge University Press, 2009. ISBN 9780521106979. URL `https://books.google.co.uk/books?id=2OwlTC4SOccC`.

[22] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1093. URL `http://dx.doi.org/10.1006/inco.1994.1093`.