The Logic and Handling of Algebraic Effects

Matija Pretnar

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2010

Abstract

In the thesis, we explore reasoning about and handling of algebraic effects. Those are computational effects, which admit a representation by an equational theory. Their examples include exceptions, nondeterminism, interactive input and output, state, and their combinations.

In the first part of the thesis, we propose a logic for algebraic effects. We begin by introducing the a-calculus, which is a minimal equational logic with the purpose of exposing distinct features of algebraic effects. Next, we give a powerful logic, which builds on results of the a-calculus. The types and terms of the logic are the ones of Levy's call-by-push-value framework, while the reasoning rules are the standard ones of a classical multi-sorted first-order logic with predicates, extended with predicate fixed points and two principles that describe the universality of free models of the theory representing the effects at hand. Afterwards, we show the use of the logic in reasoning about properties of effectful programs, and in the translation of Moggi's computational λ-calculus, Hennessy-Milner logic, and Moggi's refinement of Pitts's evaluation logic.

In the second part of the thesis, we introduce handlers of algebraic effects. Those not only provide an algebraic treatment of exception handlers, but generalise them to arbitrary algebraic effects. Each such handler corresponds to a model of the theory representing the effects, while the handling construct is interpreted by the homomorphism induced by the universal property of the free model. We use handlers to describe many previously unrelated concepts from both theory and practice, for example CSS renaming and hiding, stream redirection, timeout, and rollback.

Acknowledgements

I would first like to thank my supervisor, Gordon Plotkin. Thank you for all your suggestions and encouragement. Thank you for your detailed comments and for all the patience when you had to repeat them. I hope that at least some of those qualities have passed onto me. Thank you for all the time you let me take away from the studies to spend with my family. It is an honour to be your friend and student.

I would like to thank John Power, my second supervisor, for all the advice he gave when I was starting my studies. I would like to thank Paul Levy and Alex Simpson for showing genuine interest in my work, for their friendship, and for all the time and effort they put in examining my thesis.

I would like to thank Andrej Bauer for all the support he has offered and continues to offer. Thank you for the distractions you provided when I was stuck with my thesis. They kept me interested in science and I look forward to working by your side.

I would like to thank EPSRC, Microsoft Research in Silicon Valley, the Slovenian Government, and the Slovenian Academy of Sciences and Arts for generously funding my studies. I would like to thank Monika-Jeannette Lekuse and Andrew Finnie for sorting out all the paperwork, especially when I was on my leave of absence. I would like to thank the library in Škofja Loka and the Department of mathematics in Ljubljana for giving me a place to write the thesis.

Then, I would like to thank all the friends I met in Edinburgh: Charles, Christophe, Daniel, Dominik, Ezra, Ingo, Jaro, Jorge, Kirsty, Michael, Páidí, and Zuzanna. It would have been horribly boring without you and I do hope to see you again soon. I would also like to thank Paul Taylor for his intriguing ideas and his excellent package for drawing diagrams.

I would like to thank my parents. You never ceased to support me, even when I decided to go abroad — I know it must have been hard for you.

I would like to thank my brother. Everything is easy with you around.

Finally, I would like to thank Mojca, my beautiful wife. Thank you for all the support I got whenever I needed it. Thank you for all the sacrifices you made to be with me in Edinburgh. Thank you for bravely weathering both my absence and my absentminded presence. This thesis has been looming over us ever since we met. Now, as it comes to an end, I will try to make it worthwhile.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except in technical preliminaries (Chapter 2) and where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Parts of the results have appeared in joint work with Gordon Plotkin [PP08, PP09].

Matija Pretnar

Roku.

Contents

Chapter 1

Introduction

1.1 Historical background

Few areas of computer science are as influential as the study of the λ-calculus, whose results range from the purest theory to mainstream applications. It lead to new fields such as domain theory [GHK$^+$03]; it provided a base for most program logics such as LCF [Sco93]; and it inspired functional programming languages such as ML [MTHM97] or HASKELL [PJ03], which started as academic projects, but matured and are becoming more and more popular and influential.

However, the main obstacle to the widespread adoption of the λ-calculus in programming languages lies in the lack of treatment of *computational effects* such as user interaction, memory management, runtime errors, and so on.

Many approaches tried to bring theory closer to practice, but usually focused on a single kind of effect. Examples are Hoare logic [Hoa69] for state or Hennessy-Milner logic [HM85] for concurrency. Such specific approaches to effects are double-edged: they do yield very elaborate results, for example proofs of correctness of algorithms, but those results are almost impossible to combine when one wants to treat more than one effect.

Then, in seminal work [Mog89], Moggi proposed a uniform representation of all computational effects by monads [Mog91, BHM00]. This paved the way for more general program logics, for example Pitts's evaluation logic [Pit91], and gave an elegant method to introduce computational effects to HASKELL, otherwise a pure functional language. Still, with their encapsulation of effects, monads abstract away from many valuable details, and combining monads for different effects is a nuisance.

To account for the sources of effects, Plotkin and Power suggested the representation of effects by equational theories [PP01, PP03, PP04]. Intuitively, each occurrence of an effect can be seen as a branching point in the execution of a program, with as many branches as there are possible outcomes of an effect. Each source of an effect is then represented by an operation, whose arity is the number of outcomes and whose arguments represent the branches. The properties of those sources are described by equations between terms built from the corresponding operations.

For example, nondeterminism is represented by a binary operation, which represents the nondeterministic choice, and by equations, which state its idempotency, commutativity, and associativity. With the notable exception of continuations [FSDF93], almost all computational effects have such a representation, and those we call *algebraic effects*.

So far, the approach proved to be successful: it gave an elegant denotational semantics of effectful programs in terms of Lawvere theories [PP03], it demonstrated how such representation induces the usual monadic one [PP02], and it provided simple ways of combining effects [HPP06].

However, the approach lacked a logic, which would account for the algebraic nature of effects, and was unsuccessful in providing a treatment of exception handlers, which fail to be algebraic effects [PP03].

1.2 Aims of the thesis

The first aim of the thesis is to give a comprehensive and powerful logic of algebraic effects. To capture the distinct features of effects, we first introduce a minimal equational logic, and then expand it into a rich first-order logic with predicates. To show the power of the resulting logic, we use it to derive properties of effectful programs, and to embrace other approaches by translating their syntax into ours and deriving the translations of their axioms in our logic.

The second aim of the thesis is to give an algebraic treatment of exception handlers. As it turns out, each such handler corresponds to a (not necessarily free) model of the equational theory for exceptions. This idea is then further expanded to handlers of arbitrary algebraic effects. In order to emphasise the importance of generalised handlers, we show how they describe previously unrelated concepts from both theory and practice.

The last aim of the thesis is to consolidate the findings into a logic for effects and their handlers, and use it to derive the known properties of exception handlers and discuss their generalisation to arbitrary effects.

1.3 Structure of the thesis

In Chapter 2, we give standard definitions of multi-sorted equational theories [Grä79, Jac99], multi-sorted first-order theories [End00, Jac99], and Lawvere theories [Bor94, Pow06]. Then, we describe the algebraic representation of effects, and provide examples of algebraic effects and the corresponding theories.

In Chapter 3, we introduce the a-calculus and use it to point out the basic properties of algebraic effects. The a-calculus is a minimal equational logic with a clear separation between values, effects, and computations, along the lines of Levy's call-by-push-value approach [Lev06a]. After giving a denotational semantics in terms of models of Lawvere theories, we give the reasoning rules of the equational logic, and show how each term is equivalent to one in canonical form. Finally, we use this result to prove some simple properties of programs and show the completeness of the equational logic.

Afterwards, we start developing the general logic, and in Chapter 4, we describe its language. We generalise the description of effects by extending the operations with parameters and binding, and by equipping the equations with side conditions. The term language of the logic is an extension of the a-calculus with other call-by-push-value primitives and computation variables, while the propositions and predicates of the logic are the ones of first-order logic, together with predicate fixed points and quantifiers over both values and computations. The interpretation of the logic builds on countable Lawvere theories and their models in the category of sets.

In Chapter 5, we list all the reasoning rules of the logic: the structural reasoning rules, rules for connectives and predicates, equations for call-by-push-value constructs, a principle of computational induction, and a free model principle, with the last two expressing the universal property of the free model. In addition to providing the reasoning rules of the logic, we also show their soundness with respect to the semantics of the logic in sets.

Chapter 6 further develops the logic and describes how to obtain translations of other approaches. We generalise the results obtained in the a-calculus, explore the properties of the free model principle, and define local and global modalities. Then, we give a translation of Moggi's computational λ-calculus [Mog89], of Hennessy-Milner logic [HM85], and of Moggi's variant [Mog95] of Pitts's evaluation logic [Pit91]. For the latter, the translation preserves the provability only for a well-behaved subset of algebraic effects.

With Chapter 7, we shift our focus to the study of handlers. We give an algebraic treatment of exception handlers in terms of induced homomorphisms from the free model, and generalise the approach to arbitrary effects. This introduces a novel difficulty: such handlers have to obey the same equations as the effects they handle. We try to resolve this by giving two separate language for describing and using the handlers, thus delegating the control of correctness to the meta-level. We conclude the chapter with a list of various examples of handlers from both theory and practice, for example stream redirection, CCS renaming and hiding [HM85], timeout, and rollback.

In Chapter 8 we bring the two parts of the thesis together, showing how to extend the logic with handlers, in particular how to relate handlers to the free model principle, and how to generalise the existing properties of sequencing and exception handlers to ones of the handling construct.

Then, in Chapter 9, we extend our development with recursion. We extend the description of effects with inequalities, introduce computation fixed points to the term language, and generalise the semantics to one in terms of ω-chain complete partial orders. Then, we single out admissible predicates, which can be interpreted in terms of sub-cpos, limit the principle of computational induction to admissible predicates, and add the principle of Scott induction, also limited to admissible predicates.

Finally, in Chapter 10, we summarise our results and discuss future work.

Chapter 2

Technical preliminaries

We begin by summarising multi-sorted equational [Grä79, Jac99] and first-order theories [End00, Jac99], Lawvere theories [Bor94, Pow06], and the algebraic representation of effects [PP01, PP03].

In the technical preliminaries, we also introduce a number of abbreviations that will be used throughout the thesis. We introduce each notation only once, but use it in numerous variants. For example, although we only say that $\boldsymbol{s}$ abbreviates a sequence of sorts $s_1, \dots, s_n$, we shall use the same notation to abbreviate a sequence of terms $t_1, \dots, t_n$ by $\boldsymbol{t}$, and other sequences in a similar way.

2.1 Equational theories

All effects treated in the thesis are representable with countable equational theories. However, to grasp the underlying ideas, the a-calculus is built in the framework of finitary equational theories, which we introduce first. In Section 2.1.2, we are going to generalise those to countable theories, and in Section 4.1, we are going to provide a convenient finitary syntax for describing such countable theories.

Definition 2.1 A *signature* Σ *of a (multi-sorted) equational theory* consists of:

- a set of *sorts* s,
- a set of *function symbols* f,
- an assignment $\mathsf{f} : (s_1, \dots, s_n) \to s$ of *argument sorts* $s_1, \dots, s_n$ and a *result sort* s to each function symbol f.

When the list of argument sorts is empty, we write $\mathsf{f}:s$ instead of $\mathsf{f}:() \to s$, and call f a *constant symbol*. We often abbreviate $s_1,\ldots,s_n$ by $\boldsymbol{s}$.

Definition 2.2 Take a countably infinite set of *variables* x. Then, the set of *terms* t is given by the following grammar:

$$t ::= x \mid \mathsf{f}(\boldsymbol{t}) .$$

When the list of arguments is empty, we sometimes write f instead of $\mathsf{f}()$. In addition to $\mathsf{f}(\boldsymbol{t})$, we shall abbreviate $\mathsf{f}(t_1,\ldots,t_n)$ by $\mathsf{f}(t_i)_i$.

For distinct variables $x_1,\ldots,x_n$ and terms $t_1,\ldots,t_n$ and t, we define the term $t[t_1/x_1,\ldots,t_n/x_n]$, sometimes abbreviated by $t[\boldsymbol{t}/\boldsymbol{x}]$ or $t[t_i/x_i]_i$, to be the term obtained by the standard simultaneous substitution of x_i by t_i in t [End00].

Definition 2.3 A *context* Γ is a finite list $x_1:s_1,\ldots,x_n:s_n$, sometimes abbreviated by $\boldsymbol{x}:\boldsymbol{s}$ or $(x_i:s_i)_i$, of distinct variables x_i, each paired with a single sort s_i.

Definition 2.4 A *typing judgement* $\Gamma \vdash t:s$ states that a term t has a sort s in a context Γ. The typing judgements are given inductively by the following rules:

$$\frac{}{\Gamma \vdash x:s}\ (x:s \in \Gamma)\,, \qquad \frac{\Gamma \vdash \boldsymbol{t}:\boldsymbol{s}}{\Gamma \vdash \mathsf{f}(\boldsymbol{t}):s}\ (\mathsf{f}:(\boldsymbol{s}) \to s \in \Sigma)\,,$$

where for $\boldsymbol{t} = t_1,\ldots,t_n$ and $\boldsymbol{s} = s_1,\ldots,s_n$, the typing judgement $\Gamma \vdash \boldsymbol{t}:\boldsymbol{s}$ states that $\Gamma \vdash t_i:s_i$ holds for all $1 \leqslant i \leqslant n$.

Note that given Γ and t, there is a unique sort s such that $\Gamma \vdash t:s$ holds, and that the typing judgement has a unique derivation. We shall usually talk about a term $\Gamma \vdash t:s$, by which we shall mean a term t such that $\Gamma \vdash t:s$ holds.

Lemma 2.5 (Substitution) *Take a term $\boldsymbol{x}:\boldsymbol{s} \vdash t:s$ and terms $\Gamma \vdash \boldsymbol{t}:\boldsymbol{s}$. Then, we have*

$$\Gamma \vdash t[\boldsymbol{t}/\boldsymbol{x}]:s .$$

Definition 2.6 A *(multi-sorted) equational theory* $\mathcal{T}$ over a signature Σ is a set of *equations* $\Gamma \vdash t =_s t'$ between terms $\Gamma \vdash t : s$ and $\Gamma \vdash t' : s$, closed under the following rules:

- replacement:

$$\frac{\Gamma \vdash_{\mathcal{T}} t_i =_{s_i} t_i' \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash_{\mathcal{T}} t[\boldsymbol{t}/\boldsymbol{x}] =_s t[\boldsymbol{t}'/\boldsymbol{x}]} \quad (\boldsymbol{x}{:}\boldsymbol{s} \vdash t{:}s)\,,$$

- substitution:

$$\frac{\boldsymbol{x}{:}\boldsymbol{s} \vdash_{\mathcal{T}} t =_s t'}{\Gamma \vdash_{\mathcal{T}} t[\boldsymbol{t}/\boldsymbol{x}] =_s t'[\boldsymbol{t}/\boldsymbol{x}]} \quad (\Gamma \vdash \boldsymbol{t}{:}\boldsymbol{s})\,,$$

- reflexivity, symmetry, and transitivity of equality:

$$\frac{}{\Gamma \vdash_{\mathcal{T}} t =_s t} \quad (\Gamma \vdash t{:}s)\,, \qquad \frac{\Gamma \vdash_{\mathcal{T}} t =_s t'}{\Gamma \vdash_{\mathcal{T}} t' =_s t}\,, \qquad \frac{\Gamma \vdash_{\mathcal{T}} t =_s t' \qquad \Gamma \vdash_{\mathcal{T}} t' =_s t''}{\Gamma \vdash_{\mathcal{T}} t =_s t''}\,,$$

where $\Gamma \vdash_{\mathcal{T}} t =_s t'$ means that the equation $\Gamma \vdash t =_s t'$ is in the theory $\mathcal{T}$. If $\mathcal{T}$ is obtained by closing a set $\mathcal{A}$ under the above rules, we call $\mathcal{A}$ *the axiomatisation of* $\mathcal{T}$.

We write

$$\Gamma \vdash_{\mathcal{T}} t_1 =_s t_2 =_s \cdots =_s t_{n-1} =_s t_n$$

for the sequence of equations

$$\Gamma \vdash_{\mathcal{T}} t_1 =_s t_2 \qquad \cdots \qquad \Gamma \vdash_{\mathcal{T}} t_{n-1} =_s t_n\,,$$

which by transitivity imply the equation $\Gamma \vdash_{\mathcal{T}} t_1 =_s t_n$.

Definition 2.7 Let $\mathcal{T}$ and $\mathcal{T}'$ be equational theories over the same signature Σ. We say that $\mathcal{T}'$ is an *extension of* $\mathcal{T}$, if $\mathcal{T} \subseteq \mathcal{T}'$. If $\mathcal{T} \subsetneq \mathcal{T}'$, we call the extension *proper*.

Definition 2.8 An equational theory $\mathcal{T}$ is:

- *trivial*, if it contains only equations of the form $\Gamma \vdash t =_s t$,
- *consistent*, if there exists a sort s such that

  $$x{:}s, x'{:}s \vdash_{\mathcal{T}} x =_s x'$$

  does not hold,
- *Hilbert-Post complete*, if it is consistent and has no consistent proper extensions.

Definition 2.9 Let $\mathcal{C}$ be a category with finite products. An *interpretation* $\mathcal{I}$ of a signature Σ in $\mathcal{C}$ is given by:

- an object $[\![s]\!]_{\mathcal{I}}$ in $\mathcal{C}$ for each sort $s \in \Sigma$,
- and a morphism

  $$[\![\mathsf{f}]\!]_{\mathcal{I}} \colon [\![s_1]\!]_{\mathcal{I}} \times \cdots \times [\![s_n]\!]_{\mathcal{I}} \to [\![s]\!]_{\mathcal{I}}$$

  for each function symbol $\mathsf{f}:(s_1,\ldots,s_n) \to s \in \Sigma$.

We can extend an interpretation to:

- contexts $\Gamma = x_1 : s_1, \ldots, x_n : s_n$ with objects

  $$[\![\Gamma]\!]_{\mathcal{I}} = [\![s_1]\!]_{\mathcal{I}} \times \cdots \times [\![s_n]\!]_{\mathcal{I}} ,$$

- typing judgements $\Gamma \vdash t : s$ with morphisms

  $$[\![\Gamma \vdash t : s]\!]_{\mathcal{I}} \colon [\![\Gamma]\!]_{\mathcal{I}} \to [\![s]\!]_{\mathcal{I}} ,$$

  given recursively on the (unique) derivation of the judgement by

  $$[\![\Gamma \vdash x_i : s_i]\!]_{\mathcal{I}} = \mathrm{pr}_i ,$$
  $$[\![\Gamma \vdash \mathsf{f}(t_i)_i : s]\!]_{\mathcal{I}} = [\![\mathsf{f}]\!]_{\mathcal{I}} \circ \langle [\![\Gamma \vdash t_i : s_i]\!]_{\mathcal{I}} \rangle_i ,$$

  where pr_i denotes the i-th projection morphism and $\langle f_i \rangle_i$ denotes the tuple of morphisms $f_1, \ldots, f_n$.

When we are dealing with a single interpretation $\mathcal{I}$, we write $[\![-]\!]$ instead of $[\![-]\!]_{\mathcal{I}}$.

Definition 2.10 Let $\mathcal{T}$ be an equational theory over a signature Σ, and let $\mathcal{C}$ be a category with finite products. An interpretation $\mathcal{M}$ of Σ is a *model of* $\mathcal{T}$ if

$$\Gamma \vdash_{\mathcal{T}} t =_s t' \qquad \text{implies} \qquad [\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}} \colon [\![\Gamma]\!]_{\mathcal{M}} \to [\![s]\!]_{\mathcal{M}} .$$

A *homomorphism* ϑ between such models $\mathcal{M}$ and $\mathcal{M}'$ is a family of morphisms $\vartheta_s \colon [\![s]\!]_{\mathcal{M}} \to [\![s]\!]_{\mathcal{M}'}$ for each sort s, such that

$$\begin{tikzcd}
{[\![s_1]\!]_{\mathcal{M}} \times \cdots \times [\![s_n]\!]_{\mathcal{M}}} \arrow[r, "\vartheta_{s_1} \times \cdots \times \vartheta_{s_n}"] \arrow[d, "{[\![\mathsf{f}]\!]_{\mathcal{M}}}"'] & {[\![s_1]\!]_{\mathcal{M}'} \times \cdots \times [\![s_n]\!]_{\mathcal{M}'}} \arrow[d, "{[\![\mathsf{f}]\!]_{\mathcal{M}'}}"] \\
{[\![s]\!]_{\mathcal{M}}} \arrow[r, "\vartheta_s"'] & {[\![s]\!]_{\mathcal{M}'}}
\end{tikzcd}$$

commutes for each $\mathsf{f}:(s_1,\ldots,s_n) \to s \in \Sigma$.

Models of $\mathcal{T}$ in $\mathcal{C}$, together with homomorphisms, form a category $\mathrm{Mod}_{\mathcal{T}}(\mathcal{C})$ in the obvious way [Bor94].

Proposition 2.11 *Take a signature Σ, its interpretation $\mathcal{M}$, and a theory $\mathcal{T}$ over Σ with an axiomatisation $\mathcal{A}$. If $[\![t_1]\!] = [\![t_2]\!]$ holds for all equations $\Gamma \vdash t_1 =_s t_2 \in \mathcal{A}$, then $\mathcal{M}$ is a model of $\mathcal{T}$.*

2.1.1 Single-sorted theories

When a signature Σ contains only a single sort s, we drop any mention of it in bindings, type assignments, or equations. Then, we refer to function symbols as *operation symbols* op, and write their arity as $\mathsf{op}{:}n$ instead of as

$$\mathsf{op}{:}(\underbrace{s,\ldots,s}_{n}) \to s\,.$$

For a given model $\mathcal{M}$, we call $[\![s]\!]_{\mathcal{M}}$ the *carrier* of $\mathcal{M}$.

Given a set A, we can construct the *free model* FA of a theory $\mathcal{T}$ over a signature Σ. Let Σ_{A} be the signature Σ extended with a constant $a{:}0$ for each $a \in A$, and let $\mathcal{T}_{\mathrm{A}}$ be the theory over the extended signature, with no non-trivial equations other than the ones that follow from $\mathcal{T}$. For the carrier of FA, we take the set of all equivalence classes $[\vdash t]$ of closed terms $\vdash t$ over Σ_{A}, factored by the equality of $\mathcal{T}_{\mathrm{A}}$. Then, $[\![\mathsf{op}]\!]_{FA}$ is given by

$$[\![\mathsf{op}]\!]_{FA}([\vdash t_1],\ldots[\vdash t_n]) =_{\mathrm{def}} [\vdash \mathsf{op}(t_1,\ldots,t_n)]\,.$$

Since the provable equality is a congruence, this defines an operation on equivalence classes, and it is straightforward to check that such a family of operations gives a model of $\mathcal{T}$. A similar construction can be done in any locally finitely presentable category with finite products [Bor94].

Proposition 2.12 *Take a set $A = \{a_1,\ldots,a_n\}$ and a single-sorted equational theory $\mathcal{T}$. Then,*

$$x_1,\ldots,x_n \vdash_{\mathcal{T}} t = t' \qquad \textit{if and only if} \qquad [\![t[a_i/x_i]_i]\!]_{FA} = [\![t'[a_i/x_i]_i]\!]_{FA}\,.$$

Proof The proof follows from the construction of the free model on A. □

2.1.2 Countable theories

Definition 2.13 A *signature* Σ *of a countable (multi-sorted) equational theory* consists of:

- a set of *sorts* s,
- a set of *function symbols* f,
- an assignment $\mathsf{f}:(\boldsymbol{s}) \to s$ of a countable list of *argument sorts* $\boldsymbol{s}$ and a *result sort* s to each function symbol f.

Note that when discussing countable theories, we use $\boldsymbol{s}$ and similar notations to denote countable lists.

Terms, contexts, typing rules, and substitution, are routinely adapted to the countable case. Terms are countably branching, as function symbols have a countable number of arguments, and well-founded, as they are built inductively. Since terms can contain a countably infinite number of variables, the contexts are also infinite. An interpretation of a signature Σ now has to be given in a category with countable products.

A countable version of an equational theory is, as before, defined to be the set $\mathcal{T}$ of equations, closed under the following infinitary rules:

- replacement:
$$\frac{\Gamma \vdash_{\mathcal{T}} t_i =_{s_i} t'_i \quad (1 \leqslant i)}{\Gamma \vdash_{\mathcal{T}} t[\boldsymbol{t}/\boldsymbol{x}] =_s t[\boldsymbol{t}'/\boldsymbol{x}]} \quad (\boldsymbol{x}:\boldsymbol{s} \vdash t:s)\,,$$
- substitution:
$$\frac{\boldsymbol{x}:\boldsymbol{s} \vdash_{\mathcal{T}} t =_s t'}{\Gamma \vdash_{\mathcal{T}} t[\boldsymbol{t}/\boldsymbol{x}] =_s t[\boldsymbol{t}'/\boldsymbol{x}]} \quad (\Gamma \vdash \boldsymbol{t}:\boldsymbol{s})\,,$$
- reflexivity, symmetry, and transitivity of equality:

$$\frac{}{\Gamma \vdash_{\mathcal{T}} t =_s t} \quad (\Gamma \vdash t:s)\,, \qquad \frac{\Gamma \vdash_{\mathcal{T}} t =_s t'}{\Gamma \vdash_{\mathcal{T}} t' =_s t}\,, \qquad \frac{\Gamma \vdash_{\mathcal{T}} t =_s t' \qquad \Gamma \vdash_{\mathcal{T}} t' =_s t''}{\Gamma \vdash_{\mathcal{T}} t =_s t''}\,.$$

Models of $\mathcal{T}$ in a category $\mathcal{C}$ with countable products, together with homomorphisms, both defined similarly as before, again form a category $\mathrm{Mod}_{\mathcal{T}}(\mathcal{C})$, and if an interpretation makes all the equations in $\mathcal{A}$ sound, it is again a model of the theory. Similarly, we can construct free models in the category **Set** of sets or any locally countably presentable category with countable products.

2.2 First-order theories

Definition 2.14 A *signature* Σ *of a (multi-sorted) first-order theory* consists of:

- a set of *sorts* s,
- a set of *function symbols* f,
- an assignment $\mathsf{f}:(\boldsymbol{s}) \to s$ of *argument sorts* $\boldsymbol{s}$ and a *result sort* s to each function symbol f,
- a set of *relation symbols* rel,
- an assignment $\mathsf{rel}:(\boldsymbol{s})$ of *argument sorts* $\boldsymbol{s}$ to each relation symbol rel.

Definition 2.15 Taking a countably infinite set of variables x, *terms* t are given in the same way as for an equational theory, while *formulae* φ are given by the following grammar:

$$\varphi ::= \mathsf{rel}(\boldsymbol{t}) \mid t_1 =_s t_2 \mid \top \mid \varphi_1 \wedge \varphi_2 \mid \bot \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x{:}s.\,\varphi \mid \exists x{:}s.\,\varphi\,.$$

We define negation by $\neg\varphi =_{\text{def}} \varphi \Rightarrow \bot$.

For distinct variables $\boldsymbol{x}$, terms $\boldsymbol{t}$, and a formula φ, we define $\varphi[\boldsymbol{t}/\boldsymbol{x}]$ to be the formula, obtained by the standard capture-avoiding simultaneous substitution of variables x_i by t_i in φ [End00].

We define contexts $\Gamma = \boldsymbol{x}{:}\boldsymbol{s}$ and typing judgements $\Gamma \vdash t{:}s$ in the same way as in multi-sorted equational theories.

Definition 2.16 A *formula typing judgement* $\Gamma \vdash \varphi{:}\mathbf{form}$ states that a formula φ is well-typed in a context Γ. The formula typing judgements are given inductively by the following rules:

$$\frac{\Gamma \vdash \boldsymbol{t} : \boldsymbol{s}}{\Gamma \vdash \mathsf{rel}(\boldsymbol{t}) : \mathbf{form}} \; (\mathsf{rel} : (\boldsymbol{s}) \in \Sigma)\,, \qquad \frac{\Gamma \vdash t : s \qquad \Gamma \vdash t' : s}{\Gamma \vdash t =_s t' : \mathbf{form}}\,, \qquad \frac{}{\Gamma \vdash \top : \mathbf{form}}\,,$$

$$\frac{\Gamma \vdash \varphi_1 : \mathbf{form} \qquad \Gamma \vdash \varphi_2 : \mathbf{form}}{\Gamma \vdash \varphi_1 \wedge \varphi_2 : \mathbf{form}}\,, \qquad \frac{}{\Gamma \vdash \bot : \mathbf{form}}\,,$$

$$\frac{\Gamma \vdash \varphi_1 : \mathbf{form} \qquad \Gamma \vdash \varphi_2 : \mathbf{form}}{\Gamma \vdash \varphi_1 \vee \varphi_2 : \mathbf{form}}\,, \qquad \frac{\Gamma \vdash \varphi_1 : \mathbf{form} \qquad \Gamma \vdash \varphi_2 : \mathbf{form}}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2 : \mathbf{form}}\,,$$

$$\frac{\Gamma, x : s \vdash \varphi : \mathbf{form}}{\Gamma \vdash \forall x : s.\ \varphi : \mathbf{form}}\,, \qquad \frac{\Gamma, x : s \vdash \varphi : \mathbf{form}}{\Gamma \vdash \exists x : s.\ \varphi : \mathbf{form}}\,.$$

Lemma 2.17 (Substitution) *Take a formula* $\boldsymbol{x} : \boldsymbol{s} \vdash \varphi : \mathbf{form}$ *and terms* $\Gamma \vdash \boldsymbol{t} : \boldsymbol{s}$. *Then, we have*

$$\Gamma \vdash \varphi[\boldsymbol{t}/\boldsymbol{x}] : \mathbf{form}\,.$$

From the formulae, we build *judgements* $\Gamma \mid \Psi \vdash \varphi$, where Ψ is a set of *hypotheses* $\varphi_1, \dots, \varphi_n$ such that $\Gamma \vdash \varphi_i : \mathbf{form}$ holds for $1 \leqslant i \leqslant n$, and $\Gamma \vdash \varphi : \mathbf{form}$ is the *conclusion*.

Definition 2.18 A *multi-sorted first-order theory* $\mathcal{T}$ over a signature Σ is a set of judgements, closed under the following rules [Jac99]:

- hypothesis:

$$\frac{}{\Gamma \mid \Psi, \varphi \vdash_{\mathcal{T}} \varphi}\,,$$

- replacement:

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} t_i =_{s_i} t'_i \quad (1 \leqslant i \leqslant n) \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi[\boldsymbol{t}/\boldsymbol{x}]}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi[\boldsymbol{t}'/\boldsymbol{x}]} \; (\boldsymbol{x} : \boldsymbol{s} \vdash \varphi : \mathbf{form})\,,$$

- substitution:

$$\frac{\boldsymbol{x} : \boldsymbol{s} \mid \Psi \vdash_{\mathcal{T}} \varphi}{\Gamma \mid \Psi[\boldsymbol{t}/\boldsymbol{x}] \vdash_{\mathcal{T}} \varphi[\boldsymbol{t}/\boldsymbol{x}]} \; (\Gamma \vdash \boldsymbol{t} : \boldsymbol{s})\,,$$

- reflexivity, symmetry, and transitivity of equality:

$$\frac{}{\Gamma \mid \Psi \vdash_{\mathcal{T}} t =_s t}\ (\Gamma \vdash t{:}s)\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} t =_s t'}{\Gamma \mid \Psi \vdash_{\mathcal{T}} t' =_s t}\,,$$

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} t =_s t' \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}} t' =_s t''}{\Gamma \mid \Psi \vdash_{\mathcal{T}} t =_s t''}\,,$$

- truth and falsehood:

$$\frac{}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \top}\,, \qquad \frac{}{\Gamma \mid \Psi, \bot \vdash_{\mathcal{T}} \varphi}\,,$$

- introduction and elimination of conjunction:

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_2}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \wedge \varphi_2}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \wedge \varphi_2}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \wedge \varphi_2}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_2}\,,$$

- introduction and elimination of disjunction:

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \vee \varphi_2}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_2}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \vee \varphi_2}\,, \qquad \frac{\Gamma \mid \Psi, \varphi_1 \vdash_{\mathcal{T}} \varphi \qquad \Gamma \mid \Psi, \varphi_2 \vdash_{\mathcal{T}} \varphi}{\Gamma \mid \Psi, \varphi_1 \vee \varphi_2 \vdash_{\mathcal{T}} \varphi}\,,$$

- introduction and elimination of implication:

$$\frac{\Gamma \mid \Psi, \varphi_1 \vdash_{\mathcal{T}} \varphi_2}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \Rightarrow \varphi_2}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1 \Rightarrow \varphi_2 \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_1}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi_2}\,,$$

- introduction and elimination of universal quantification:

$$\frac{\Gamma, x{:}s \mid \Psi \vdash_{\mathcal{T}} \varphi}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \forall x{:}s.\, \varphi}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \forall x{:}s.\, \varphi}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi[t/x]}\ (\Gamma \vdash t{:}s)\,,$$

- introduction and elimination of existential quantification:

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi[t/x]}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \exists x{:}s.\, \varphi}\ (\Gamma \vdash t{:}s)\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}} \exists x{:}s.\, \varphi' \qquad \Gamma, x{:}s \mid \Psi, \varphi' \vdash_{\mathcal{T}} \varphi}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi}\,,$$

- reductio ad absurdum:

$$\frac{\Gamma \mid \Psi, \neg\varphi \vdash_{\mathcal{T}} \bot}{\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi}\,,$$

where $\Gamma \mid \Psi \vdash_{\mathcal{T}} \varphi$ means that the judgement $\Gamma \mid \Psi \vdash \varphi$ is in the theory $\mathcal{T}$. If $\mathcal{T}$ is obtained by closing a set $\mathcal{A}$ under the above rules, we call $\mathcal{A}$ *the axiomatisation of* $\mathcal{T}$.

Throughout the thesis, we omit any conditions on the freshness of bound variables, as they are guaranteed by the restrictions on the contexts. For a more elaborate discussion of such restrictions, see Remark 3.22

Definition 2.19 An *interpretation* $\mathcal{I}$ of a signature Σ is given by:

- a set $[\![s]\!]_{\mathcal{I}}$ for each sort $s \in \Sigma$;
- a map
$$[\![\mathsf{f}]\!]_{\mathcal{I}} \colon [\![s_1]\!]_{\mathcal{I}} \times \cdots \times [\![s_n]\!]_{\mathcal{I}} \to [\![s]\!]_{\mathcal{I}}$$
for each function symbol $\mathsf{f} : (s_1, \ldots, s_n) \to s \in \Sigma$;
- and a subset
$$[\![\mathsf{rel}]\!]_{\mathcal{I}} \subseteq [\![s_1]\!]_{\mathcal{I}} \times \cdots \times [\![s_n]\!]_{\mathcal{I}}$$
for each relation symbol $\mathsf{rel} : (s_1, \ldots, s_n) \in \Sigma$.

We extend the interpretations to contexts and typed terms just as in multi-sorted equational theories, while formula typing judgements $\Gamma \vdash \varphi : \mathbf{form}$ are interpreted by subsets $[\![\Gamma \vdash \varphi : \mathbf{form}]\!] \subseteq [\![\Gamma]\!]$ according to Tarski's semantics, given recursively on the (unique) derivation of the judgement by:

$$\begin{aligned}
[\![\mathsf{rel}(t_1, \ldots, t_n)]\!] &= \{\boldsymbol{\gamma} \in [\![\Gamma]\!] \mid \langle [\![t_1]\!](\boldsymbol{\gamma}), \ldots, [\![t_n]\!](\boldsymbol{\gamma}) \rangle \in [\![\mathsf{rel}]\!]\}\,, \\
[\![t_1 =_s t_2]\!] &= \{\boldsymbol{\gamma} \in [\![\Gamma]\!] \mid [\![t_1]\!](\boldsymbol{\gamma}) = [\![t_2]\!](\boldsymbol{\gamma})\}\,, \\
[\![\top]\!] &= [\![\Gamma]\!]\,, \\
[\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!] \cap [\![\varphi_2]\!]\,, \\
[\![\bot]\!] &= \emptyset\,, \\
[\![\varphi_1 \vee \varphi_2]\!] &= [\![\varphi_1]\!] \cup [\![\varphi_2]\!]\,, \\
[\![\varphi_1 \Rightarrow \varphi_2]\!] &= \{\boldsymbol{\gamma} \in [\![\Gamma]\!] \mid \boldsymbol{\gamma} \in [\![\varphi_1]\!] \text{ implies } \boldsymbol{\gamma} \in [\![\varphi_2]\!]\}\,, \\
[\![\forall x{:}s.\, \varphi]\!] &= \{\boldsymbol{\gamma} \in [\![\Gamma]\!] \mid \langle \boldsymbol{\gamma}, \gamma \rangle \in [\![\Gamma, x{:}s \vdash \varphi{:}\mathbf{form}]\!] \text{ for all } \gamma \in [\![s]\!]\}\,, \\
[\![\exists x{:}s.\, \varphi]\!] &= \{\boldsymbol{\gamma} \in [\![\Gamma]\!] \mid \langle \boldsymbol{\gamma}, \gamma \rangle \in [\![\Gamma, x{:}s \vdash \varphi{:}\mathbf{form}]\!] \text{ for some } \gamma \in [\![s]\!]\}\,,
\end{aligned}$$

where we abbreviate $[\![\Gamma \vdash \varphi{:}\mathbf{form}]\!]$ by $[\![\varphi]\!]$.

Definition 2.20 Let $\mathcal{T}$ be a multi-sorted first-order theory over a signature Σ. An interpretation $\mathcal{M}$ of Σ is a *model of* $\mathcal{T}$ if

$$\Gamma \mid \varphi_1, \ldots, \varphi_n \vdash_{\mathcal{T}} \varphi \qquad \text{implies} \qquad [\![\varphi_1]\!]_{\mathcal{M}} \cap \cdots \cap [\![\varphi_n]\!]_{\mathcal{M}} \subseteq [\![\varphi]\!]_{\mathcal{M}} \,.$$

Proposition 2.21 *Take a signature* Σ*, its interpretation* $\mathcal{M}$*, and a theory* $\mathcal{T}$ *over* Σ *with an axiomatisation* $\mathcal{A}$*. If*

$$[\![\varphi_1]\!]_{\mathcal{M}} \cap \cdots \cap [\![\varphi_n]\!]_{\mathcal{M}} \subseteq [\![\varphi]\!]_{\mathcal{M}} \,.$$

holds for all judgements $\Gamma \mid \varphi_1, \ldots, \varphi_n \vdash \varphi$ *in* $\mathcal{A}$*, then* $\mathcal{M}$ *is a model of* $\mathcal{T}$*.*

2.3 Lawvere theories

Definition 2.22 Take small categories $\mathcal{C}$ and $\mathcal{C}'$ and a functor $F \colon \mathcal{C} \to \mathcal{C}'$. We say that F is *bijective-on-objects*, if its object-component is a bijection.

Assuming that both $\mathcal{C}$ and $\mathcal{C}'$ have finite products, we say that F is a *(strict) product preserving* functor if the image

$$\begin{tikzcd} FA & F(A \times B) \arrow[l, "F\mathrm{pr}_1"'] \arrow[r, "F\mathrm{pr}_2"] & FB \end{tikzcd}$$

of a (specified) product diagram in $\mathcal{C}$ is a (specified) product diagram in $\mathcal{C}'$.

Definition 2.23 A *Lawvere theory* is a small category $\mathcal{L}$ with finite products, together with a strict product preserving bijective-on-objects functor $J \colon \aleph_0^{\mathrm{op}} \to \mathcal{L}$, where the category $\aleph_0$ is a skeleton [ML71] of the category of all finite sets and maps between them [Pow06].

The objects of $\aleph_0$ are $[0], [1], [2], \ldots$, where $[i]$ is the representative of a finite set with i elements. A morphism $p \colon [m] \to [n]$ can be represented by a m-tuple $\langle p_1, \ldots, p_m \rangle$, where $1 \leqslant p_i \leqslant n$ for each $1 \leqslant i \leqslant m$.

The category $\aleph_0$ has finite sums, in particular, the sum of objects $[m]$ and $[n]$ is the object $[m+n]$. Hence, its dual $\aleph_0^{\mathrm{op}}$ has finite products, in particular, the object $[n]$ is equal to a product $[1] \times \cdots \times [1]$ of n copies of $[1]$.

Remark 2.24 Usually, a Lawvere theory is defined to be a small category with finite products and with objects $A_0, A_1, \ldots$, such that A_n is equal to $(A_1)^n$ [Bor94]. Although we prefer to give the definition of a Lawvere theory in a different way, which is easier to generalise, the two definitions coincide. (If we want a correspondence with Lawvere theories where A_n is isomorphic to $(A_1)^n$, we have to omit the strictness condition in Definition 2.23.)

On one hand, take a small category $\mathcal{L}$ with finite products and a strict product preserving bijective-on-objects functor $J\colon \aleph_0^{\mathrm{op}} \to \mathcal{L}$. Since J is bijective-on-objects, the objects of $\mathcal{L}$ are exactly the objects $J[0], J[1], J[2], \ldots$. As J is strict product preserving, $J[n]$ is equal to $(J[1])^n$. We usually identify $J[n]$ with $[n]$.

On the other hand, taking a small category with finite products and objects $\{A_i\}_{i\in\mathbb{N}}$, we define J as the functor that maps an object $[n]$ to A_n and a morphism $p\colon [m] \leftarrow [n]$ to $Jp\colon A_m \to A_n$, defined by

$$A_m = (A_1)^m \xrightarrow{\langle \mathrm{pr}_{p_1}, \ldots, \mathrm{pr}_{p_n} \rangle} (A_1)^n = A_n \,.$$

Because $[m] \times [n] = [m+n]$, we have

$$J([m] \times [n]) = A_{m+n} = (A_1)^{m+n} = (A_1)^m \times (A_1)^n = A_m \times A_n = J[m] \times J[n] \,,$$

hence J is strict product preserving, and it is obviously bijective-on-objects.

It is easy to check that the two constructions yield an isomorphism of categories.

Definition 2.25 A *model* of a Lawvere theory $\mathcal{L}$ in the category $\mathcal{C}$ with finite products is a finite-product preserving functor $M\colon \mathcal{L} \to \mathcal{C}$, and a *homomorphism* between models M and M' is a natural transformation $h\colon M \to M'$. Models of $\mathcal{L}$ in $\mathcal{C}$, together with homomorphisms, form a category $\mathrm{Mod}_{\mathcal{L}}(\mathcal{C})$.

Definition 2.26 For models M_1 and M_2, the *product model* $M_1 \times M_2$ is defined by

$$(M_1 \times M_2)[n] = M_1[n] \times M_2[n] \,,$$
$$(M_1 \times M_2)p = M_1 p \times M_2 p \,.$$

If for an object A, the exponent $(M[1])^A$ exists, the *exponent model* M^A is defined by

$$(M^A)[n] = (M[n])^A \,,$$
$$(M^A)p = (Mp)^A \,.$$

It is easy to check that if $(M[1])^A$ exists, so does $(M[n])^A \cong ((M[1])^A)^n$, and that both the product and the exponent models are finite-product preserving functors.

The category $\mathrm{Mod}_{\mathcal{L}}(\mathcal{C})$ is equipped with a forgetful functor $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathcal{C}) \to \mathcal{C}$, which maps a model M to an object $UM = M[1]$ and a homomorphism $h\colon M \to M'$ to its component $h_{[1]}\colon UM \to UM'$.

When $\mathcal{C}$ is locally finitely presentable (see [Bor94] for the definition), the free model construction, sketched in Section 2.1.1, gives a functor $F\colon \mathcal{C} \to \mathrm{Mod}_{\mathcal{L}}(\mathcal{C})$, which is the left adjoint of U [Bor94].

Example 2.27 The category **Set** of all sets and maps between them is a locally finitely presentable category.

The functors U and F form a strong (see Definition 2.28 below) monad T on $\mathcal{C}$. We write η and ε for the unit and the co-unit of the adjunction, and μ for the monad multiplication.

Definition 2.28 A *strength* of a monad (T,η,μ) is a natural transformation

$$\mathrm{st}_{A,B}\colon A \times TB \to T(A \times B)\,,$$

that is natural in both components and makes the following diagrams commute:

$$\begin{tikzcd}
\mathbf{1}\times TA \arrow[r, "\mathrm{st}_{\mathbf{1},A}"] \arrow[d, "\mathrm{pr}_2"'] & T(\mathbf{1}\times A) \arrow[dl, "T\mathrm{pr}_2"] \\
TA &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (A\times B)\times TC \arrow[r, "\mathrm{st}_{A\times B,C}"] & T((A\times B)\times C) \arrow[d, "\cong"] \\
A\times(B\times TC) \arrow[ur, "\cong"] \arrow[r, "\mathrm{id}_A\times\mathrm{st}_{B,C}"'] & A\times T(B\times C) \arrow[r, "\mathrm{st}_{A,B\times C}"'] & T(A\times(B\times C))
\end{tikzcd}$$

$$\begin{tikzcd}
A\times TB \arrow[r, "\mathrm{st}_{A,B}"] & T(A\times B) \\
A\times B \arrow[u, "\mathrm{id}_A\times\eta_B"] \arrow[ur, "\eta_{A\times B}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & A\times TB \arrow[rr, "\mathrm{st}_{A,B}"] & & T(A\times B) \\
A\times T^2B \arrow[ur, "\mathrm{id}_A\times\mu_B"] \arrow[rr, "\mathrm{st}_{A,TB}"'] & & T(A\times TB) \arrow[r, "T\mathrm{st}_{A,B}"'] & T^2(A\times B) \arrow[u, "\mu_{A\times B}"']
\end{tikzcd}$$

Definition 2.29 Take objects A and B, and a model M. Then, for every morphism $f\colon A\times B \to UM$, we define its *lifting* $f^\dagger$ to be the morphism

$$U\varepsilon_M \circ UFf \circ \mathrm{st}_{A,B}\colon A\times UFB \to UM\,.$$

Lemma 2.30 *For any morphism $f\colon A \times B \to UM$, we have*

$$f^\dagger \circ (\mathrm{id}_A \times \eta_B) = f\ .$$

Proof We have

$$\begin{aligned} & f^\dagger \circ (\mathrm{id}_A \times \eta_B) \\ &= U\varepsilon_M \circ UFf \circ \mathrm{st}_{A,B} \circ (\mathrm{id}_A \times \eta_B) && \text{(by definition)} \\ &= U\varepsilon_M \circ UFf \circ \eta_{A\times B} && \text{(by definition of strength)} \\ &= U\varepsilon_M \circ \eta_{UM} \circ f && \text{(by naturality of } \eta) \\ &= f && \text{(as } U\varepsilon_M \text{ and } \eta_{UM} \text{ are inverse [ML71])}\ . \end{aligned}$$

□

2.3.1 Algebraic operations

Definition 2.31 Let $\mathcal{L}$ be a Lawvere theory and let $\mathcal{C}$ be a category with finite products such that the forgetful functor $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathcal{C}) \to \mathcal{C}$ has a left adjoint F. A family of maps $a_{FA}\colon (UFA)^n \to UFA$, where A ranges over all objects of $\mathcal{C}$, is an *algebraic operation of arity n*, if for any morphism $f\colon A \times B \to UFC$, the following diagram commutes.

$$\begin{tikzcd} A \times (UFB)^n \arrow[r, "\langle f^\dagger \circ (\mathrm{id}_A \times \mathrm{pr}_i)\rangle_{i=1}^n"] \arrow[d, "\mathrm{id}_A \times a_{FB}"'] & (UFC)^n \arrow[d, "a_{FC}"] \\ A \times UFB \arrow[r, "f^\dagger"'] & UFC \end{tikzcd}$$

The forgetful functor has a left adjoint if the category $\mathcal{C}$ is locally finitely presentable.

Let $\mathfrak{M}$ be the smallest set of models that contains all the free models FA, where A ranges over the objects of $\mathcal{C}$, and is closed under products and exponentials. Given an algebraic operation $\{a_{FA}\}_A$, we can recursively extend it to other models from $\mathfrak{M}$ by

$$a_{M_1 \times M_2} =_{\mathrm{def}} a_{M_1} \times a_{M_2} \qquad \text{and} \qquad a_{M^A} =_{\mathrm{def}} (a_M)^A\ .$$

Then, such a family behaves as an algebraic operation.

Proposition 2.32 *Let $\mathcal{L}$ be a Lawvere theory and let $\mathcal{C}$ be a category with finite products such that the forgetful functor $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathcal{C}) \to \mathcal{C}$ has a left adjoint F. Take a family of maps $a_M\colon (UM)^n \to UM$, where M ranges over all models in the family $\mathfrak{M}$ defined above. Then, for any morphism $f\colon A \times B \to UM$, the following diagram commutes.*

$$\begin{tikzcd}
A \times (UFB)^n \arrow[r, "\langle f^\dagger \circ (\mathrm{id}_A \times \mathrm{pr}_i)\rangle_{i=1}^n"] \arrow[d, "\mathrm{id}_A \times a_{FB}"'] & (UM)^n \arrow[d, "a_M"] \\
A \times UFB \arrow[r, "f^\dagger"'] & UM
\end{tikzcd}$$

Proof We proceed by an induction on the construction of M. The proof is routine. □

Lemma 2.33 *Assume that the category $\mathcal{C}$ is cartesian closed and that the forgetful functor $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathcal{C}) \to \mathcal{C}$ has a left adjoint F. Then, the family of maps*

$$\{(FA)p\colon (UFA)^n \to UFA\}_A$$

is an algebraic operation for all morphisms $p\colon [n] \to [1]$.

Proof Take an arbitrary morphism $f\colon A \times B \to UFC$. By transposing it, we obtain a morphism $B \to (UFC)^A$. Since $(UFC)^A = U(FC)^A$, the adjunction yields a homomorphism $\hat{f}\colon FB \to (FC)^A$. Because of the naturality of $\hat{f}$, the diagram

$$\begin{tikzcd}
(FB)[n] \arrow[r, "\hat{f}_{[n]}"] \arrow[d, "(FB)p"'] & (FC)^A[n] \arrow[d, "(FC)^A p"] \\
(FB)[1] \arrow[r, "\hat{f}_{[1]}"'] & (FC)^A[1]
\end{tikzcd}$$

commutes, from which it follows that the family $\{(FA)p\}_A$ is indeed an algebraic operation. □

In **Set** and other suitable categories, the Yoneda embedding induces a bijection between algebraic operations and maps in the Kleisli category of the monad UF [PP03]. In particular, each algebraic operation $\{a_{FA}\colon (UFA)^n \to UFA\}_A$ corresponds to a map $\mathrm{gen}_a\colon \mathbf{1} \to UF\mathbf{n}$, called the *generic effect of a*, where the set $\mathbf{n} = \mathbf{1} + \cdots + \mathbf{1}$ is the disjoint sum of n terminal objects $\mathbf{1}$.

2.3.2 Relationship to equational theories

Each single-sorted equational theory $\mathcal{T}$ induces a Lawvere theory $\mathcal{L}_{\mathcal{T}}$ in a canonical way [Bor94]. First, fix a sequence of variables $x_1, x_2, \ldots$. Then, for the objects of $\mathcal{L}_{\mathcal{T}}$, we take the objects $[0], [1], [2], \ldots$ of the category $\aleph_0^{\mathrm{op}}$.

Because $[m]$ is isomorphic to $[1]^m$, each morphism $[n] \to [m]$ is equivalent to a m-tuple of morphism $[n] \to [1]$. For those, we take equivalence classes $[x_1, \ldots, x_n \vdash t]$ of terms, modulo the equality in $\mathcal{T}$.

The composition of morphisms

$$\langle [x_1, \ldots, x_n \vdash t_1], \ldots, [x_1, \ldots, x_n \vdash t_m] \rangle \colon [n] \to [m]$$

and

$$\langle [x_1, \ldots, x_m \vdash t'_1], \ldots, [x_1, \ldots, x_m \vdash t'_k] \rangle \colon [m] \to [k]$$

is defined to be

$$\langle [x_1, \ldots, x_n \vdash t'_1[t_i/x_i]_i], \ldots, [x_1, \ldots, x_n \vdash t'_k[t_i/x_i]_i] \rangle \colon [n] \to [k] \,,$$

and the identity morphism on $[n]$ is defined to be

$$\langle [x_1, \ldots, x_n \vdash x_1], \ldots, [x_1, \ldots, x_n \vdash x_n] \rangle \,.$$

The identity laws are easy to check, while the proof (and statement of) associativity is much more cumbersome.

For an arbitrary single-sorted equational theory $\mathcal{T}$, the categories $\mathrm{Mod}_{\mathcal{T}}(\mathcal{C})$ and $\mathrm{Mod}_{\mathcal{L}_{\mathcal{T}}}(\mathcal{C})$ are equivalent [Pow06].

2.3.3 Countable theories

Definition 2.34 A *countable Lawvere theory* is a small category $\mathcal{L}$ with countable products and a strict countable-product preserving bijective-on-objects functor $J \colon \aleph_1^{\mathrm{op}} \to \mathcal{L}$, where $\aleph_1$ is a skeleton [ML71] of the category of all countable sets and maps between them [Pow06].

Definition 2.35 A *model* of a countable Lawvere theory $\mathcal{L}$ in a category $\mathcal{C}$ with countable products is a countable-product preserving functor $M \colon \mathcal{L} \to \mathcal{C}$, and a *homomorphism* between such models M and M' is a natural transformation $h \colon M \to M'$.

Models of $\mathcal{L}$ in $\mathcal{C}$, together with homomorphisms, form a category $\mathrm{Mod}_{\mathcal{L}}(\mathcal{C})$. As before, each countable single-sorted equational theory $\mathcal{T}$ induces a canonical countable Lawvere theory $\mathcal{L}_{\mathcal{T}}$ such that the categories $\mathrm{Mod}_{\mathcal{T}}(\mathcal{C})$ and $\mathrm{Mod}_{\mathcal{L}_{\mathcal{T}}}(\mathcal{C})$ are equivalent [Pow06].

If $\mathcal{C}$ is locally countably presentable, the forgetful functor $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathcal{C}) \to \mathcal{C}$ has a left adjoint $F\colon \mathcal{C} \to \mathrm{Mod}_{\mathcal{L}}(\mathcal{C})$, and the two functors form a strong monad $UF\colon \mathcal{C} \to \mathcal{C}$ [Pow06].

Example 2.36 An *ω-chain complete partial order* $(A, \leqslant)$ (or an ω-cpo) is a set A, equipped with a partial order $\leqslant$, such that any countable increasing chain $a_1 \leqslant a_2 \leqslant \ldots$ has a supremum $\bigvee_i a_i$ [GHK+03].

For ω-cpos $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we say that a map $f\colon A \to B$ is *continuous*, if:

- it is monotone, that is $a \leqslant_A a'$ implies $f(a) \leqslant_B f(a)$;
- it preserves suprema of increasing chains, that is $f(\bigvee_i a_i) = \bigvee_i f(a_i)$ (since f is monotone, the elements $f(a_i)$ form a chain).

Then, ω-cpos and continuous maps between them form a category ω-**Cpo**, which is locally countably presentable.

2.4 Algebraic effects

As suggested by Plotkin and Power, computational effects may be represented by single-sorted countable equational theories [PP01, PP02]. For each effect, we give a set of its sources, which we represent by operation symbols. Then, each occurrence of those sources represents a branching point in the execution of the program, where the number of branches reflects the number of possible outcomes, and is captured in the arity of the operation representing the source. Finally, the properties of effects are described by an equational theory.

To distinguish this use of equational theories from the standard use in describing the values of the underlying system, we adopt a different syntax, which we will also use in the remainder of the thesis when describing effects. The variables are labelled by z and the contexts by Z. Finally, the terms of the theory are called *effect terms* e. The algebraic representation is obtained as follows.

2.4.1 Exceptions

Take a set E of exceptions. Since the evaluation of a program stops when an exception is encountered, we take a signature containing a nullary operation symbol $\mathsf{raise}_{\mathsf{exc}} : 0$ for each exception $\mathsf{exc} \in E$. For the theory, we take the trivial theory. The induced monad T then maps a set A to the set $A+E$, the disjoint sum of sets A and E.

A notion that accompanies exceptions is *exception handling*. In its simplest form, it is represented by a *handling construct* $\mathsf{handle}_{\mathsf{exc}}(e_1, e_2)$ for each exception exc, which intuitively evaluates as e_1, unless its evaluation raises exc, in which case it evaluates as e_2.

However, exception handling is not an algebraic effect, as the handling construct is not an algebraic operation. If any family of maps $\{h_{FA}\}_A$ were to represent exception handling, a map $h_{FA} : (A+E)^2 \to A+E$ should obey the following two equations:

$$
\begin{aligned}
(h_{FA})(\mathrm{in}_1(a), z) &= \mathrm{in}_1(a)\,, \\
(h_{FA})(\mathrm{in}_2(\mathsf{exc}), z) &= z\,,
\end{aligned}
$$

where $\mathrm{in}_1 : A \to A+E$ and $\mathrm{in}_2 : E \to A+E$ are the inclusion maps.

Now take A to be $\{a, b\}$ and set $f(a) = \mathrm{in}_2(\mathsf{exc})$ and $f(b) = \mathrm{in}_1(b)$. Then, we get

$$
\begin{aligned}
&(h_{FA} \circ (f^\dagger)^2)(\mathrm{in}_1(a), \mathrm{in}_1(b)) \\
&= h_{FA}(f^\dagger(\mathrm{in}_1(a)), f^\dagger(\mathrm{in}_1(b))) \\
&= h_{FA}(\mathrm{in}_2(\mathsf{exc}), \mathrm{in}_1(b)) && \text{(by Lemma 2.30)} \\
&= \mathrm{in}_1(b) && \text{(as } h \text{ represents the handling construct)}\,.
\end{aligned}
$$

However, if the family $\{h_{FA}\}_A$ is an algebraic operation, we have

$$
\begin{aligned}
&(h_{FA} \circ (f^\dagger)^2)(\mathrm{in}_1(a), \mathrm{in}_1(b)) \\
&= (f^\dagger \circ h_{FA})(\mathrm{in}_1(a), \mathrm{in}_1(b)) && \text{(since } \{h_{FA}\}_A \text{ is algebraic)} \\
&= f^\dagger(h_{FA}(\mathrm{in}_1(a), \mathrm{in}_1(b))) \\
&= f^\dagger(\mathrm{in}_1(a)) && \text{(as } h \text{ represents the handling construct)} \\
&= \mathrm{in}_2(\mathsf{exc})\,.
\end{aligned}
$$

Although exception handling fails to be an algebraic effect, it has an algebraic treatment in terms of handlers of algebraic effects, a novel concept we discuss in Chapter 7.

2.4.2 Nondeterminism

To describe nondeterminism, we take a signature containing a binary operation symbol $\mathsf{or}:2$, which represents nondeterministic choice. We write $\mathsf{or}(e_1, e_2)$ in the infix form $e_1 \cup e_2$. The theory for nondeterminism is the theory of semi-lattices, given by the following axiomatisation:

$$z \vdash z \cup z = z\,,$$
$$z_1, z_2 \vdash z_1 \cup z_2 = z_2 \cup z_1\,,$$
$$z_1, z_2, z_3 \vdash (z_1 \cup z_2) \cup z_3 = z_1 \cup (z_2 \cup z_3)\,.$$

The induced monad for nondeterminism maps a set A to the set $\mathcal{F}^+(A)$ of finite non-empty subsets of A.

To show that the given theory accurately describes nondeterminism, we first observe that nondeterministic choice satisfies all the semi-lattice equations. To show that it cannot satisfy any other equation, we prove that the theory is Hilbert-Post complete: an addition of any other equation, not already present in the theory, would make the theory inconsistent.

Proposition 2.37 (Plotkin) *The theory for nondeterminism is Hilbert-Post complete.*

Proof We first observe that due to the semi-lattice equations, each effect term e can be put in a canonical form $z_1 \cup \cdots \cup z_n$. Now take an arbitrary extension $\mathcal{T}'$ of the theory $\mathcal{T}$ for nondeterminism, and an arbitrary equation $Z \vdash e = e' \in \mathcal{T}'$, where, without loss of generality, both e and e' are in canonical form.

If both sides of the equation contain exactly the same variables, then the two sides are equal and the equation is already present in $\mathcal{T}$. If not, there is a variable z, present only on (say) the left hand side. We take a fresh variable z', and substitute all variables except z by z'. If z is the only variable present on the left hand side, we get

$$z, z' \vdash_{\mathcal{T}'} z = z'\,,$$

if it is not, we get

$$z, z' \vdash_{\mathcal{T}'} z \cup z' = z'\,,$$

which by a simultaneous substitution of z' for z and z for z' entails

$$z, z' \vdash_{\mathcal{T}'} z' \cup z = z\,,$$

and so

$$z, z' \vdash_{\mathcal{T}'} z = z'\,.$$

Hence the extension $\mathcal{T}'$ is inconsistent, therefore the theory for nondeterminism is Hilbert-Post complete. □

2.4.3 Interactive input and output

To describe interactive input and output on a countable alphabet A, we take a signature containing an operation symbol $\mathsf{input} : |A|$ and an operation symbol $\mathsf{output}_a : 1$ for each character $a \in A$, and the trivial equational theory.

The meaning behind the operation symbols is as follows: $\mathsf{input}(e_a)_a$ represents a computation that waits for user's input, and proceeds as e_a if the user entered the character a, while $\mathsf{output}_a(e)$ represents a computation that outputs a and proceeds as e. For example,

$$\mathsf{input}(\mathsf{output}_a(\mathsf{output}_a(z)))_a$$

represents a computation that waits for the user's input, repeats it twice and then proceeds as z. Above, the term $\mathsf{input}(e_a)_a$ abbreviates the countably branching term

$$\mathsf{input}(e_{a_i})_i = \mathsf{input}(e_{a_1}, e_{a_2}, \ldots)\,,$$

where $a_1, a_2, \ldots$ is some injective enumeration of elements of A. We use a similar convention for other suitable index sets.

2.4.4 Time

To describe the passing of time, we take a signature containing a unary operation symbol $\mathsf{tick} : 1$, which represents the passing of a fixed amount of time, and the trivial theory. The induced monad maps a set A to the set $A \times \mathbb{N}$.

An alternative would be to take a signature consisting of a unary operation symbol $\mathsf{tick}_r : 1$ for each non-negative real number $r \in \mathbb{R}^+$ (or other suitable monoid), and the theory, generated by equations

$$z \vdash \mathsf{tick}_0(z) = z\,,$$

$$z \vdash \mathsf{tick}_{r_1}(\mathsf{tick}_{r_2}(z)) = \mathsf{tick}_{r_1+r_2}(z)\,.$$

In this case, the induced monad maps a set A to the set $A \times \mathbb{R}^+$ (or $A \times M$ for a more general monoid M).

2.4.5 State

To describe state with a finite set L of locations and a countable set D of data, we take a signature containing an operation symbol $\mathsf{lookup}_\ell : |D|$ for each location $\ell \in L$, and a unary operation symbol $\mathsf{update}_{\ell,d} : 1$ for each location $\ell \in L$ and datum $d \in D$.

If we took an infinite set L of locations, the induced monad would not be the standard one for state. Since the elements of the free model are built inductively from operations, they represent computations that only update a finite number of locations at a time. In contrast, the elements of the standard monad represent computations that can perform an arbitrary modification of the state.

The effect term $\mathsf{lookup}_\ell(e_d)_d$ represents a computation that looks up the contents of location ℓ and proceeds as e_d if the stored contents is d. The effect term $\mathsf{update}_{\ell,d}(e)$ represents a computation that updates the location ℓ with d and proceeds as e. For example, an effect term

$$\mathsf{lookup}_\ell(\mathsf{update}_{\ell',d}(z))_d$$

represents a computation that copies the contents of ℓ to ℓ' and proceeds as z.

State is represented by a theory $\mathcal{T}$, generated by the following seven families of equations, where for the sake of clarity we omit the contexts:

$$\begin{aligned}
\mathsf{lookup}_\ell(\mathsf{update}_{\ell,d}(z))_d &= z\,, \\
\mathsf{lookup}_\ell(\mathsf{lookup}_\ell(z_{dd'})_{d'})_d &= \mathsf{lookup}_\ell(z_{dd})_d\,, \\
\mathsf{update}_{\ell,d}(\mathsf{lookup}_\ell(z_{d'})_{d'}) &= \mathsf{update}_{\ell,d}(z_d)\,, \\
\mathsf{update}_{\ell,d}(\mathsf{update}_{\ell,d'}(z)) &= \mathsf{update}_{\ell,d'}(z)\,, \\
\mathsf{lookup}_\ell(\mathsf{lookup}_{\ell'}(z_{dd'})_{d'})_d &= \mathsf{lookup}_{\ell'}(\mathsf{lookup}_\ell(z_{dd'})_d)_{d'} && (\ell \neq \ell')\,, \\
\mathsf{update}_{\ell,d}(\mathsf{lookup}_{\ell'}(z_{d'})_{d'}) &= \mathsf{lookup}_{\ell'}(\mathsf{update}_{\ell,d}(z_{d'}))_{d'} && (\ell \neq \ell')\,, \\
\mathsf{update}_{\ell,d}(\mathsf{update}_{\ell',d'}(z)) &= \mathsf{update}_{\ell',d'}(\mathsf{update}_{\ell,d}(z)) && (\ell \neq \ell')\,.
\end{aligned}$$

Then, the induced monad maps a set A to $(S \times A)^S$, where $S = D^L$.

The first four equations describe the behaviour of operations on a single location: the first one says that updating a location with its current contents does not modify the state; the second one says that state does not change between two consecutive lookups; the third one says that state is determined immediately after an update; and the fourth one says that the second update overwrites the first one. The last three equations state that operations on different locations commute.

Like the theory for nondeterminism, the theory for state both describes properties of state and is Hilbert-Post complete. As before, the proof proceeds by first showing that each term has a canonical form, and then analysing equations between terms in canonical form. For the sake of simplicity, let us assume a single location ℓ, and write lookup and update_d instead of lookup_ℓ and $\mathsf{update}_{\ell,d}$.

Proposition 2.38 *For every term $Z \vdash e$, there exists a map $f : D \to D$ and a collection of variables $(z_d)_{d \in D}$ such that $z_d \in Z$ for all $d \in D$ and*

$$Z \vdash_{\mathcal{T}} e = \mathsf{lookup}(\mathsf{update}_{f(d)}(z_d))_d \,.$$

Proof Let us proceed by an induction on the structure of e:

- if $e = z$ for some variable z, then $f(d) = d$ and $z_d = z$, and we have

$$Z \vdash_{\mathcal{T}} z = \mathsf{lookup}(\mathsf{update}_d(z))_d \,;$$

- if $e = \mathsf{update}_{d'}(e')$ for some e', then by the induction hypothesis, we have

$$Z \vdash_{\mathcal{T}} e' = \mathsf{lookup}(\mathsf{update}_{f(d)}(z_d))_d$$

  and

$$\begin{aligned} Z \vdash_{\mathcal{T}} & \mathsf{update}_{d'}(e') \\ &= \mathsf{update}_{d'}(\mathsf{lookup}(\mathsf{update}_{f(d)}(z_d))_d) \\ &= \mathsf{update}_{d'}(\mathsf{update}_{f(d')}(z_{d'})) \\ &= \mathsf{update}_{f(d')}(z_{d'}) \\ &= \mathsf{lookup}(\mathsf{update}_{f(d')}(z_{d'}))_d \,; \end{aligned}$$

- if $e = \mathsf{lookup}(e_d)_d$ for some family of effect terms $(e_d)_{d \in D}$, then by the induction hypothesis, we have

$$Z \vdash_{\mathcal{T}} e_d = \mathsf{lookup}(\mathsf{update}_{f_d(d')}(z_{dd'}))_{d'}$$

for all $d \in D$ and

$$\begin{aligned} Z \vdash_{\mathcal{T}} & \mathsf{lookup}(e_d)_d \\ & = \mathsf{lookup}(\mathsf{lookup}(\mathsf{update}_{f_d(d')}(z_{dd'}))_{d'})_d \\ & = \mathsf{lookup}(\mathsf{update}_{f_d(d)}(z_{dd}))_d \,. \end{aligned}$$

□

Lemma 2.39 *If $Z \vdash_{\mathcal{T}} \mathsf{update}_{d_1}(z) = \mathsf{update}_{d_2}(z')$, then $Z \vdash_{\mathcal{T}} z = z'$.*

Proof By substituting $\mathsf{update}_d(z)$ for z and $\mathsf{update}_d(z')$ for z', we get

$$Z \vdash_{\mathcal{T}} \mathsf{update}_d(z) = \mathsf{update}_{d_1}(\mathsf{update}_d(z)) = \mathsf{update}_{d_2}(\mathsf{update}_d(z')) = \mathsf{update}_d(z')$$

for all $d \in D$. Hence

$$Z \vdash_{\mathcal{T}} z = \mathsf{lookup}(\mathsf{update}_d(z))_d = \mathsf{lookup}(\mathsf{update}_d(z'))_d = z' \,.$$

□

Proposition 2.40 (Plotkin) *The theory for state is Hilbert-Post complete.*

Proof Take an extension $\mathcal{T}'$ of the theory for state, and an arbitrary equation $Z \vdash e = e'$ in $\mathcal{T}'$. Without loss of generality, this equation is of the form

$$Z \vdash \mathsf{lookup}(\mathsf{update}_{f(d)}(z_d))_d = \mathsf{lookup}(\mathsf{update}_{f'(d)}(z'_d))_d \,.$$

Then, for all $d \in D$, we have

$$\begin{aligned} Z \vdash_{\mathcal{T}'} & \mathsf{update}_{f(d)}(z_d) \\ & = \mathsf{update}_d(\mathsf{update}_{f(d)}(z_d)) \\ & = \mathsf{update}_d(\mathsf{lookup}(\mathsf{update}_{f(d')}(z_{d'}))_{d'}) \\ & = \mathsf{update}_d(\mathsf{lookup}(\mathsf{update}_{f'(d')}(z'_{d'}))_{d'}) \\ & = \mathsf{update}_d(\mathsf{update}_{f'(d)}(z'_d)) \\ & = \mathsf{update}_{f'(d)}(z'_d) \,, \end{aligned}$$

and from Lemma 2.39, we get $Z \vdash_{\mathcal{T}'} z_d = z'_d$.

If $f(d) = f'(d)$ and $z_d = z'_d$ for all $d \in D$, the additional equation is already in the theory $\mathcal{T}$. Otherwise, the equations fail for some d'. On the one hand, if $z_{d'}$ and $z'_{d'}$ are distinct variables, the resulting theory is inconsistent, since we have $Z \vdash_{\mathcal{T}'} z_d = z'_d$ for all $d \in D$, including d'.

On the other hand, we have $f(d') \neq f'(d')$. Take a fresh set of distinct variables $\{z''_d\}_{d \in D}$ and substitute $\mathsf{lookup}(z''_d)_d$ for $z_{d'}$ and $z'_{d'}$. Then, we get

$$\begin{aligned}
\{z''_d\}_{d \in D} \vdash_{\mathcal{T}'} \mathsf{update}_{f(d')}(z''_{f(d')}) \\
&= \mathsf{update}_{f(d')}(\mathsf{lookup}(z''_d)_d) \\
&= \mathsf{update}_{f(d')}(z_d)[\mathsf{lookup}(z''_d)_d/z_d, \mathsf{lookup}(z''_d)_d/z'_d] \\
&= \mathsf{update}_{f'(d')}(z'_d)[\mathsf{lookup}(z''_d)_d/z_d, \mathsf{lookup}(z''_d)_d/z'_d] \\
&= \mathsf{update}_{f'(d')}(\mathsf{lookup}(z''_d)_d) \\
&= \mathsf{update}_{f'(d')}(z''_{f'(d')})\,.
\end{aligned}$$

Similarly as before, we get $Z \vdash_{\mathcal{T}'} z''_{f(d')} = z''_{f'(d')}$, and the extension $\mathcal{T}'$ is inconsistent. Hence, the theory for state is Hilbert-Post complete. □

In the case of multiple locations, the canonical form is a series of lookups, which read all the locations, followed by a series of updates, which set the locations to their final state, followed by a variable. In order to get a fixed ordering of operations in the normal form, we use the equations for commutativity between operations on different locations. The rest of the proof generalises accordingly.

Chapter 3

The a-calculus

We start our study with the *a-calculus*. It is a minimal calculus, which builds on the algebraic representation of effects, and which describes the most basic features of effectful computations: caused effects, returned values, and evaluation order. The latter is especially important: as the evaluation of computations depends on the environment (computer memory, user input, . . .), different evaluation orders cause different results.

For that reason, we base the structure of the a-calculus on Levy's call-by-push-value approach, which has a clean syntactic separation between values, where the evaluation order does not matter, and computations, where it does. However, we add a third layer for effects, and expand Levy's slogan: "A value is, a computation does," with a third statement: "an effect occurs." This reflects a view of values as interchangeable timeless entities, passed around in execution; of computations as instructions setting the path of the execution, including triggering of effects; and of effects as the consequences of execution on the environment.

Although the a-calculus could easily be generalised to account for countable equational theories, we limit it to finitary ones for the sake of simplicity of exposition.

3.1 Syntax

Since the a-calculus focuses on computations, we restrict its values to ones described in terms of an equational theory.

Definition 3.1 The *base theory* $\mathfrak{T}_{\text{base}}$ is a multi-sorted equational theory over a *base signature* Σ_{base}. Its sorts are called *base types* β, its variables *base variables* x, its contexts *base contexts* Γ, and its terms *base terms* b.

Example 3.2 To describe natural numbers with addition and multiplication, we take a base signature, consisting of base type **nat** and function symbols $\mathsf{zero}\colon\mathbf{nat}$, $\mathsf{succ}\colon(\mathbf{nat})\to\mathbf{nat}$, $\mathsf{plus}\colon(\mathbf{nat},\mathbf{nat})\to\mathbf{nat}$, and $\mathsf{times}\colon(\mathbf{nat},\mathbf{nat})\to\mathbf{nat}$, which we write in the usual notation.

Then, we take a base theory, generated by the following axiomatisation:

$$\begin{aligned} 0+x &= x\,, & 0\cdot x &= 0\,,\\ \mathsf{succ}(x_1)+x_2 &= \mathsf{succ}(x_1+x_2)\,, & \mathsf{succ}(x_1)\cdot x_2 &= x_1\cdot x_2+x_2\,. \end{aligned}$$

The effects at hand are also given by a finitary equational theory, in the way described in Section 2.4.

Definition 3.3 The *effect theory* $\mathcal{T}_{\text{eff}}$ is a single-sorted equational theory over an *effect signature* Σ_{eff}. Its variables are called *effect variables* z, its function symbols *operation symbols* op, its contexts *effect contexts* Z, and its terms *effect terms* e.

By building on this description of values and effects, the a-calculus describes the computations that return those values and cause those effects.

Definition 3.4 The sets of *computation types* $\underline{\tau}$ and *computation terms* t are given by the following grammar:

$$\begin{aligned} \underline{\tau} &::= F\beta\,,\\ t &::= \mathsf{op}(\boldsymbol{t}) \mid \mathsf{return}\,b \mid t\,\mathsf{to}\,x\colon\!\beta.\,t'\,, \end{aligned}$$

where in $t\,\mathsf{to}\,x\colon\!\beta.\,t'$, the variable x is bound in t', according to the usual conventions [End00].

The computation type $F\beta$ is the type of computations that return values of type β, and is named so because it is interpreted by a free model of the effect theory $\mathcal{T}_{\text{eff}}$. The grammar of computation terms reflects the three basic properties of effectful programs: operations $\mathsf{op}(\boldsymbol{t})$ reflect caused effects, returned base terms $\mathsf{return}\,b$ reflect returned values, and sequencing $t\,\mathsf{to}\,x\colon\!\beta.\,t'$ reflects evaluation order.

We do not give a formal operational semantics of the a-calculus. However, the intuitive computational meaning behind computation terms is as follows: a term $\mathsf{op}(t_i)_i$ represents a computation that triggers the effect, represented by an operation symbol op, and then, depending on its outcome $1 \leqslant i \leqslant n$, proceeds as t_i; a term $\mathsf{return}\, b$ represents the computation that returns the value b; and a term $t\, \mathsf{to}\, x{:}\beta.\, t'$ represents a computation that evaluates t, binds the result to x and proceeds as t'. To get an idea about a formal operational treatment, one can take a look at the operational semantics of call-by-push-value [Lev06a] and of PCF with algebraic operations [PP01].

Example 3.5 An example of computation term is

$$\begin{aligned}&\mathsf{or}(\mathsf{return}\,2, \mathsf{return}\,3)\,\mathsf{to}\, x_1{:}\mathbf{int}.\,(\\&\mathsf{or}(\mathsf{return}\,5, \mathsf{return}\,7)\,\mathsf{to}\, x_2{:}\mathbf{int}.\\&\quad \mathsf{return}(x_1 \cdot x_2))\,,\end{aligned}$$

representing a computation that first nondeterministically chooses between 2 and 3, binds the result to x_1, then chooses between 5 and 7, binds the result to x_2, and finally returns $x_1 \cdot x_2$. Intuitively, the computation nondeterministically returns one of 10, 14, 15, or 21.

Definition 3.6 For distinct base variables $\boldsymbol{x}$, base terms $\boldsymbol{b}$, and a computation term t, the computation term $t[\boldsymbol{b}/\boldsymbol{x}]$ is the term obtained by the standard simultaneous capture-avoiding substitution of x_i by b_i in t [End00].

Effect terms serve as templates for computation terms. One obtains a computation term from an effect term by replacing its effect variables by computation terms.

Definition 3.7 Let e be an effect term with free variables $z_1, \ldots, z_n$, and let $t_1, \ldots, t_n$ be computation terms. Then, the *instantiation* $e[t_i/z_i]_i$ is defined recursively on the structure of e by:

$$\begin{aligned}z_j[t_i/z_i]_i &= t_j\,,\\ \mathsf{op}(e_j)_j[t_i/z_i]_i &= \mathsf{op}(e_j[t_i/z_i]_i)_j\,.\end{aligned}$$

Example 3.8 The computation term $\mathsf{or}(\mathsf{or}(\mathsf{return}\,1, \mathsf{return}\,2), \mathsf{return}\,3)$ is a result of an instantiation

$$\mathsf{or}(\mathsf{or}(z_1, z_2), z_3)[\mathsf{return}\,1/z_1, \mathsf{return}\,2/z_2, \mathsf{return}\,3/z_3]\,.$$

Note that a term can be a result of multiple instantiations. For example, the above term can also result from the instantiation

$$\mathsf{or}(z_1, z_2)[\mathsf{or}(\mathsf{return}\,1, \mathsf{return}\,2)/z_1, \mathsf{return}\,3/z_2]\,.$$

Definition 3.9 A *typing judgement* $\Gamma \vdash t{:}\underline{\tau}$ states that a term t has a type $\underline{\tau}$ in a context Γ, which consists of distinct base variables x_i, each paired with a single base type β_i. The typing judgements are given inductively by the following rules:

$$\frac{\Gamma \vdash t_i{:}\underline{\tau} \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash \mathsf{op}(t_1, \ldots, t_n){:}\underline{\tau}}\ (\mathsf{op}{:}n \in \Sigma_{\mathsf{eff}})\,, \qquad \frac{\Gamma \vdash b{:}\beta}{\Gamma \vdash \mathsf{return}\,b{:}F\beta}\,,$$

$$\frac{\Gamma \vdash t{:}F\beta \qquad \Gamma, x{:}\beta \vdash t'{:}\underline{\tau}}{\Gamma \vdash t\,\mathsf{to}\,x{:}\beta.\,t'{:}\underline{\tau}}\,.$$

Note that the computation term $\mathsf{raise}_{\mathsf{exc}}$ can have an arbitrary computation type $\underline{\tau}$, because the typing hypotheses are vacuously satisfied — this behaviour can be observed in strongly typed functional programming languages such as ML or HASKELL, where exceptions can be raised at any point in the program, no matter what type the surrounding expression expects.

Given a context Γ, a computation term t, and a computation type $\underline{\tau}$, the derivation of $\Gamma \vdash t{:}\underline{\tau}$, if it exists, is uniquely determined, and hence so is its interpretation. For this reason, the bound variable x in sequencing $t\,\mathsf{to}\,x{:}\beta.\,t'$ has to be explicitly typed. Usually, its type is determined by the type of t, but since t can have an arbitrary type, $\Gamma \vdash t\,\mathsf{to}\,x.\,t'{:}\underline{\tau}$ could have more than one typing derivation, and its semantics would not be uniquely defined. Informally, however, we often abbreviate $t\,\mathsf{to}\,x{:}\beta.\,t'$ by $t\,\mathsf{to}\,x.\,t'$.

Lemma 3.10 (Substitution) *Take a computation term* $\boldsymbol{x} : \boldsymbol{\beta} \vdash t : \underline{\tau}$ *and base terms* $\Gamma \vdash \boldsymbol{b}{:}\boldsymbol{\beta}$. *Then, we have*

$$\Gamma \vdash t[\boldsymbol{b}/\boldsymbol{x}]{:}\underline{\tau}\,.$$

Proof We proceed by an induction on the structure of t. The proof is straightforward. □

Lemma 3.11 (Instantiation) *Let* $z_1,\ldots,z_n \vdash e$ *be an effect term and let* $\Gamma \vdash t_i : \underline{\tau}$ *be a computation term for all* $1 \leqslant i \leqslant n$*. Then, we have*

$$\Gamma \vdash e[\boldsymbol{t}/\boldsymbol{z}] : \underline{\tau}\,.$$

Proof We proceed by an induction on the structure of e. The proof is straightforward. □

3.2 Semantics

Fix a model $\mathcal{M}$ of the base theory $\mathcal{T}_{\mathrm{base}}$ in **Set** and let $\mathcal{L}$ be the Lawvere theory, induced by the effect theory $\mathcal{T}_{\mathrm{eff}}$. Then, each operation symbol $\mathsf{op} : n \in \Sigma_{\mathrm{eff}}$ is interpreted by a morphism $[\![\mathsf{op}]\!]_{\mathcal{L}} : [n] \to [1]$ in $\mathcal{L}$.

Each computation type $F\beta$ is interpreted by the free model $F[\![\beta]\!]$, where F is the free model functor. Note that while $[\![\beta]\!]$ is a set, as given in Definition 2.9, the interpretation $[\![\underline{\tau}]\!]$ of a computation type $\underline{\tau}$ is a model, hence a functor, which maps morphisms in $\mathcal{L}$ into operations on $U[\![\underline{\tau}]\!]$. In particular, for each computation type $\underline{\tau}$ and each operation symbol $\mathsf{op}{:}n \in \Sigma_{\mathrm{eff}}$, we get a map

$$[\![\underline{\tau}]\!]([\![\mathsf{op}]\!]_{\mathcal{L}}) \colon U[\![\underline{\tau}]\!]^n \to U[\![\underline{\tau}]\!]\,.$$

A typing judgement $\Gamma \vdash t{:}\underline{\tau}$ is interpreted by a map

$$[\![\Gamma \vdash t{:}\underline{\tau}]\!] : [\![\Gamma]\!] \to U[\![\underline{\tau}]\!]\,,$$

defined inductively on its derivation by:

$$\begin{aligned}
[\![\Gamma \vdash \mathsf{op}(t_1,\ldots,t_n){:}\underline{\tau}]\!] &= [\![\underline{\tau}]\!]([\![\mathsf{op}]\!]_{\mathcal{L}}) \circ \langle [\![t_1]\!],\ldots,[\![t_n]\!]\rangle\,,\\
[\![\Gamma \vdash \mathsf{return}\, b : \underline{\tau}]\!] &= \eta_{[\![\beta]\!]} \circ [\![b]\!]\,,\\
[\![\Gamma \vdash t\, \mathsf{to}\, x{:}\beta.\, t'{:}\underline{\tau}]\!] &= [\![t']\!]^{\dagger} \circ \langle \mathrm{id}_{\Gamma}, [\![t]\!]\rangle\,,
\end{aligned}$$

where, judgements are abbreviated to terms on the right, interpretation $[\![\Gamma]\!]$ is defined component-wise as in multi-sorted theories, and

$$[\![t']\!]^{\dagger} : [\![\Gamma]\!] \times UF[\![\beta]\!] \to U[\![\underline{\tau}]\!]$$

is the lifting of

$$[\![t']\!] : [\![\Gamma]\!] \times [\![\beta]\!] \to U[\![\underline{\tau}]\!]\,,$$

as given in Definition 2.29.

Lemma 3.12 *Take a computation term* $\boldsymbol{x}{:}\boldsymbol{\beta} \vdash t{:}\underline{\tau}$ *and base terms* $\Gamma \vdash \boldsymbol{b}{:}\boldsymbol{\beta}$*. Then, we have*

$$[\![t[\boldsymbol{b}/\boldsymbol{x}]]\!] = [\![t]\!] \circ \langle [\![b_i]\!] \rangle_i \,.$$

Proof We proceed by an induction on the structure of t. The proof is straightforward. □

Lemma 3.13 *Let* $\Gamma \vdash t_i : \underline{\tau}$ *be a computation term for all* $1 \leqslant i \leqslant n$ *and take an effect term* $z_1,\ldots,z_n \vdash e$*. Then, we have*

$$[\![e[t_i/z_i]_i]\!] = [\![\underline{\tau}]\!]([\![e]\!]) \circ \langle [\![t_i]\!] \rangle_i \,.$$

Proof We proceed by an induction on the structure of e. The proof is straightforward. □

Note that the semantics of the a-calculus can easily be given much more generally in any locally presentable and cartesian closed category $\mathcal{C}$. We need the local presentability to guarantee the existence of the free model construction, finite products to interpret contexts, and exponentials to ensure that operation symbols are interpreted by algebraic operations (see Lemma 2.33).

3.3 Equational logic

Definition 3.14 The *equational logic of the a-calculus* is the smallest set of equations $\Gamma \vdash t =_{\underline{\tau}} t'$ between computation terms $\Gamma \vdash t : \underline{\tau}$ and $\Gamma \vdash t' : \underline{\tau}$, closed under the following rules:

- reflexivity, symmetry, and transitivity of equality:

$$\frac{}{\Gamma \vdash_a t =_{\underline{\tau}} t}\,, \qquad \frac{\Gamma \vdash_a t =_{\underline{\tau}} t'}{\Gamma \vdash_a t' =_{\underline{\tau}} t}\,, \qquad \frac{\Gamma \vdash_a t =_{\underline{\tau}} t' \qquad \Gamma \vdash_a t' =_{\underline{\tau}} t''}{\Gamma \vdash_a t =_{\underline{\tau}} t''}\,,$$

- congruence for operations:

$$\frac{\Gamma \vdash_a t_i =_{\underline{\tau}} t'_i \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash_a \mathsf{op}(\boldsymbol{t}) =_{\underline{\tau}} \mathsf{op}(\boldsymbol{t}')} \quad (\mathsf{op}{:}n \in \Sigma_{\mathrm{eff}})\,,$$

- congruence for sequencing:

$$\frac{\Gamma \vdash_a t_1 =_{F\beta} t'_1 \qquad \Gamma, x{:}\beta \vdash_a t_2 =_{\underline{\tau}} t'_2}{\Gamma \vdash_a t_1 \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} t'_1 \,\mathsf{to}\, x.\, t'_2}\,,$$

- inheritance from the base theory:

$$\frac{\Gamma \vdash_{\mathcal{T}_{\text{base}}} b =_{\beta} b'}{\Gamma \vdash_{a} \mathsf{return}\, b =_{F\beta} \mathsf{return}\, b'}\,,$$

- inheritance from the effect theory:

$$\frac{z_1, \ldots, z_n \vdash_{\mathcal{T}_{\text{eff}}} e = e'}{\Gamma \vdash_{a} e[t_i/z_i]_i =_{\underline{\tau}} e'[t_i/z_i]_i} \quad (\Gamma \vdash t_i : \underline{\tau} \quad (1 \leqslant i \leqslant n))\,,$$

- β-equivalence of sequencing:

$$\frac{}{\Gamma \vdash_{a} \mathsf{return}\, b\, \mathsf{to}\, x.\, t =_{\underline{\tau}} t[b/x]}\,,$$

- algebraicity of operations:

$$\frac{}{\Gamma \vdash_{a} \mathsf{op}(t_i)_i\, \mathsf{to}\, x.\, t =_{\underline{\tau}} \mathsf{op}(t_i\, \mathsf{to}\, x.\, t)_i} \quad (\mathsf{op} : n \in \Sigma_{\text{eff}})\,,$$

where $\Gamma \vdash_a t =_{\underline{\tau}} t'$ means that the equation $\Gamma \vdash t =_{\underline{\tau}} t'$ is in the equational theory of the a-calculus. In the rules, we have omitted the hypotheses that ensure that equations are well-formed.

Lemma 3.15 (Substitution) *For any equation $\boldsymbol{x} : \boldsymbol{\beta} \vdash_a t =_{\underline{\tau}} t'$ and base terms $\Gamma \vdash \boldsymbol{b} : \boldsymbol{\beta}$, we have*

$$\Gamma \vdash_a t[\boldsymbol{b}/\boldsymbol{x}] =_{\underline{\tau}} t'[\boldsymbol{b}/\boldsymbol{t}]\,.$$

Proof We proceed by an induction on the derivation of $\boldsymbol{x} : \boldsymbol{\beta} \vdash_a t =_{\underline{\tau}} t'$. The proof is routine. □

Since the a-calculus has no variables for computation terms, we cannot use the replacement rule as in the multi-sorted equational logic. Instead, we have to write out the congruence rules for each constructor.

In addition to those rules, we have two rules, used to inherit equations from the base and effect theories, and two equational schemas, which describe the behaviour of sequencing. The first one is the usual β-equivalence, while understanding the second one requires some computational intuition.

Intuitively, the evaluation of $\mathsf{op}(t_i)_i\, \mathsf{to}\, x.\, t$ starts by evaluating $\mathsf{op}(t_i)_i$. This begins with an occurrence of the effect represented by the operation $\mathsf{op} : n$. Then, depending on the outcome of this effect, say i, the term t_i is evaluated. The resulting value is bound to x, and then, t is evaluated.

On the other hand, the evaluation of $\mathsf{op}(t_i \,\mathsf{to}\, x.\, t)_i$ begins with an occurrence of the effect represented by the operation $\mathsf{op}\!:\!n$. Then, depending on the outcome of this effect, say i, the term $t_i \,\mathsf{to}\, x.\, t$ is evaluated. This proceeds by evaluating t_i, binding the resulting value to x, and evaluating in t. Since the two evaluations proceed in the same way, we deem them equivalent, which is exactly what the second schema states.

Proposition 3.16 (Soundness) *If* $\Gamma \vdash_a t =_{\underline{\tau}} t'$, *then* $\llbracket t \rrbracket = \llbracket t' \rrbracket$.

Proof We do an induction on the derivation of the equation. For the congruence rules and the inheritance from the base theory, the proof is straightforward. For substitution rule, we employ Lemma 3.12, while for inheritance from the effect theory, we employ Lemma 3.13.

For the β-equivalence of sequencing, we have

$$\begin{aligned}
&\llbracket \mathsf{return}\, b \,\mathsf{to}\, x.\, t \rrbracket && \\
&= \llbracket t \rrbracket^{\dagger} \circ \langle \mathrm{id}_\Gamma, \eta_{\llbracket \beta \rrbracket} \circ \llbracket b \rrbracket \rangle && \text{(by definition)} \\
&= \llbracket t \rrbracket \circ \langle \mathrm{id}_\Gamma, \llbracket b \rrbracket \rangle && \text{(by Lemma 2.30)} \\
&= \llbracket t[b/x] \rrbracket && \text{(by Lemma 3.12)}\ .
\end{aligned}$$

And for the algebraicity of operations, we have

$$\begin{aligned}
&\llbracket \mathsf{op}(t_i)_i \,\mathsf{to}\, x.\, t \rrbracket && \\
&= \llbracket t \rrbracket^{\dagger} \circ (\mathrm{id}_\Gamma \times \llbracket F\beta \rrbracket(\llbracket \mathsf{op} \rrbracket)) \circ \langle \mathrm{id}_\Gamma, \langle \llbracket t_i \rrbracket \rangle_i \rangle && \text{(by definition)} \\
&= \llbracket \underline{\tau} \rrbracket(\llbracket \mathsf{op} \rrbracket) \circ \langle \llbracket t \rrbracket^{\dagger} \circ \langle \mathrm{id}_\Gamma, \llbracket t_i \rrbracket \rangle \rangle_i && \text{(by Lemma 2.33)} \\
&= \llbracket \mathsf{op}(t_i \,\mathsf{to}\, x.\, t)_i \rrbracket && \text{(by definition)}\ .
\end{aligned}$$

□

The majority of results in the a-calculus follow from the properties of the free model, and are obtained by observing that computation terms can be put in a canonical form, composed only of operations and returned values. This is possible only due to the simplicity of the a-calculus. In a more powerful calculus such as the term calculus of the logic, discussed in Chapter 4, such a canonical form does not exist. For that reason, the logic, discussed in Chapter 5, includes a principle of computational induction, which generalises the approach with canonical forms.

Definition 3.17 A computation term $\Gamma \vdash t : \underline{\tau}$ is in *canonical form*, if it is of the form:

- $\Gamma \vdash \mathsf{return}\, b : F\beta$ for some base term $\Gamma \vdash b : \beta$;
- $\Gamma \vdash \mathsf{op}(t_i)_i : \underline{\tau}$, where computation terms $\Gamma \vdash t_i : \underline{\tau}$ are in canonical form for all $1 \leqslant i \leqslant n$.

To simplify the proof that all computation terms are equivalent to a term in canonical form, we first consider a special case of a sequencing with both terms already in canonical form.

Lemma 3.18 *If $\Gamma \vdash t_1 : F\beta$ and $\Gamma, x : \beta \vdash t_2 : \underline{\tau}$ are in canonical form, there exists a term $\Gamma \vdash t : \underline{\tau}$ in canonical form, such that*

$$\Gamma \vdash_a t_1 \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} t\,.$$

Proof We proceed by an induction on the canonical form of t_1:

- if $t_1 = \mathsf{return}\, b$ for some base term b, then

  $$\Gamma \vdash_a \mathsf{return}\, b \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} t_2[b/x]\,,$$

  and it is easy to show that if t_2 is in canonical form, so is $t_2[b/x]$;
- if $t_1 = \mathsf{op}(t'_i)_i$ for some operation symbol $\mathsf{op} : n \in \Sigma_{\text{eff}}$ and computation terms t'_i in canonical form, then

  $$\Gamma \vdash_a \mathsf{op}(t'_i)_i \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} \mathsf{op}(t'_i \,\mathsf{to}\, x.\, t_2)_i\,.$$

  By the induction hypothesis, there exist computation terms t''_i in canonical form such that $\Gamma \vdash_a t'_i \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} t''_i$ for $1 \leqslant i \leqslant n$, hence

  $$\Gamma \vdash_a \mathsf{op}(t'_i)_i \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} \mathsf{op}(t''_i)_i\,.$$

□

Proposition 3.19 *For each computation term $\Gamma \vdash t:\underline{\tau}$, there exists a computation term $\Gamma \vdash t':\underline{\tau}$ in canonical form, such that $\Gamma \vdash_a t =_{\underline{\tau}} t'$.*

Proof We do an induction on the derivation of $\Gamma \vdash t:\underline{\tau}$:

- if $t = \mathsf{return}\, b$ for some base term $\Gamma \vdash b : \beta$, then t is already in canonical form;

- if $t = \mathsf{op}(t_i)_i$ for some operation symbol $\mathsf{op}:n \in \Sigma_{\mathrm{eff}}$ and computation terms t_i, then by induction hypothesis, there exist computation terms t'_i in canonical form such that $\Gamma \vdash_a t_i =_{\underline{\tau}} t'_i$ for all $1 \leqslant i \leqslant n$, hence
$$\Gamma \vdash_a t =_{\underline{\tau}} \mathsf{op}(t'_i)_i\ ,$$
and $\mathsf{op}(t'_i)_i$ is in canonical form.

- if $t = t_1 \,\mathsf{to}\, x.\, t_2$ for some computation terms $\Gamma \vdash t_1 : \beta$ and $\Gamma, x : \beta \vdash t_2 : \underline{\tau}$, then by induction hypothesis, there exist computation terms t'_1 and t'_2 in canonical form, such that $\Gamma \vdash_a t_1 =_{F\beta} t'_1$ and $\Gamma, x : \beta \vdash_a t_2 =_{\underline{\tau}} t'_2$. Then, by Lemma 3.18, there exists a term t' in canonical form such that
$$\Gamma \vdash_a t'_1 \,\mathsf{to}\, x.\, t'_2 =_{\underline{\tau}} t'\ ,$$
hence
$$\Gamma \vdash_a t =_{\underline{\tau}} t_1 \,\mathsf{to}\, x.\, t_2 =_{\underline{\tau}} t'_1 \,\mathsf{to}\, x.\, t'_2 =_{\underline{\tau}} t'\ .$$

□

Note that the canonical forms are not unique unless both the base and the effect theories are trivial. Still, the existence of canonical forms leads to proofs, simpler than the ones by structural induction. Examples are proofs of η-equivalence and associativity of sequencing — two schemas that are usually taken as axioms [Mog91, Lev06a].

Proposition 3.20 (**η-equivalence of sequencing**) *For any computation terms $\Gamma \vdash t:F\beta$, we have*
$$\Gamma \vdash_a t \,\mathsf{to}\, x.\ \mathsf{return}\, x =_{F\beta} t\ .$$

Proof According to Proposition 3.19, we can assume without loss of generality that t is in canonical form. Let us proceed by induction on the structure of that canonical form:

- if $t = \mathsf{return}\, b$ for some base term $\Gamma \vdash b : \beta$, then

$$\begin{aligned} &\Gamma \vdash_a \mathsf{return}\, b\, \mathsf{to}\, x.\ \mathsf{return}\, x \\ &\quad =_{F\beta} (\mathsf{return}\, x)[b/x] && \text{(by β-equivalence)} \\ &\quad =_{F\beta} \mathsf{return}\, b && \text{(by definition of substitution)} \ ; \end{aligned}$$

- if $t = \mathsf{op}(t_i)_i$ for some operation symbol $\mathsf{op} : n \in \Sigma_{\text{eff}}$ and computation terms $\Gamma \vdash t_i : F\beta$ in canonical form for $1 \leqslant i \leqslant n$, then

$$\begin{aligned} &\Gamma \vdash_a \mathsf{op}(t_i)_i\, \mathsf{to}\, x.\ \mathsf{return}\, x \\ &\quad =_{F\beta} \mathsf{op}(t_i\, \mathsf{to}\, x.\ \mathsf{return}\, x)_i && \text{(by algebraicity of operations)} \\ &\quad =_{F\beta} \mathsf{op}(t_i)_i && \text{(by the induction hypothesis)} \ . \end{aligned}$$

□

Proposition 3.21 (Associativity of sequencing) *The equation schema*

$$\Gamma \vdash_a t_1\, \mathsf{to}\, x_1.\ (t_2\, \mathsf{to}\, x_2.\ t) =_{\underline{\tau}} (t_1\, \mathsf{to}\, x_1.\ t_2)\, \mathsf{to}\, x_2.\ t$$

is derivable for all $\Gamma \vdash t_1 : F\beta_1$, $\Gamma, x_1 : \beta_1 \vdash t_2 : F\beta_2$, *and* $\Gamma, x_2 : \beta_2 \vdash t : \underline{\tau}$.

Remark 3.22 The usual condition that x_1 does not appear free in t is implemented by the restrictions on contexts. Since each context contains at most one occurrence of each variable, x_1 does not appear in Γ, otherwise t_2 would not be well-typed. Similarly x_2 does not appear in t_2, otherwise $t_2\, \mathsf{to}\, x_2.\ t$ would not be well-typed, hence x_1 and x_2 are distinct, and x_1 does appear free in t. We use the same reasoning throughout the thesis and omit explicit restrictions on free variables.

Proof According to Proposition 3.19, we can assume without loss of generality that t_1 is in canonical form. Let us proceed by induction on the structure of that canonical form:

- if $t_1 = \mathsf{return}\, b_1$ for some base term $\Gamma \vdash b_1 : \beta_1$, we have

$$\begin{aligned} &\Gamma \vdash_a \mathsf{return}\, b\, \mathsf{to}\, x_1.\ (t_2\, \mathsf{to}\, x_2.\ t) \\ &\quad =_{\underline{\tau}} (t_2\, \mathsf{to}\, x_2.\ t)[b/x_1] && \text{(by β-equivalence)} \\ &\quad =_{\underline{\tau}} t_2[b/x_1]\, \mathsf{to}\, x_2.\ t && \text{(since x_1 does not appear in t)} \\ &\quad =_{\underline{\tau}} (\mathsf{return}\, b\, \mathsf{to}\, x_1.\ t_2)\, \mathsf{to}\, x_2.\ t && \text{(by β-equivalence)} \ ; \end{aligned}$$

- if $t_1 = \mathsf{op}(t'_i)_i$ for some operation symbol $\mathsf{op}{:}n \in \Sigma_{\text{eff}}$ and computation terms $\Gamma \vdash t'_i{:}F\beta_1$ in canonical form for $1 \leqslant i \leqslant n$, we have

$$\begin{aligned}
&\Gamma \vdash_a \mathsf{op}(t'_i)_i \,\mathsf{to}\, x_1.\, (t_2 \,\mathsf{to}\, x_2.\, t) && \\
&\quad =_{\underline{\tau}} \mathsf{op}(t'_i \,\mathsf{to}\, x_1.\, (t_2 \,\mathsf{to}\, x_2.\, t))_i && \text{(by algebraicity of operations)} \\
&\quad =_{\underline{\tau}} \mathsf{op}((t'_i \,\mathsf{to}\, x_1.\, t_2) \,\mathsf{to}\, x_2.\, t)_i && \text{(by induction hypothesis)} \\
&\quad =_{\underline{\tau}} (\mathsf{op}(t'_i \,\mathsf{to}\, x_1.\, t_2)_i) \,\mathsf{to}\, x_2.\, t && \text{(by algebraicity of operations)} \\
&\quad =_{\underline{\tau}} (\mathsf{op}(t'_i)_i \,\mathsf{to}\, x_1.\, t_2) \,\mathsf{to}\, x_2.\, t && \text{(by algebraicity of operations)}\ .
\end{aligned}$$

□

As seen in the last proof, associativity of sequencing is a consequence of the algebraicity of operations. There are other properties of operations, which are reflected in sequencing, for example commutativity is derivable when the effect theory is commutative.

Proposition 3.23 *If the effect theory $\mathcal{T}_{\text{eff}}$ is* commutative*, that is*

$$Z \vdash_{\mathcal{T}_{\text{eff}}} \mathsf{op}(\mathsf{op}'(z_{ij})_j)_i = \mathsf{op}'(\mathsf{op}(z_{ij})_i)_j$$

holds for all operations $\mathsf{op}{:}n, \mathsf{op}{:}n' \in \Sigma_{\text{eff}}$, then

$$\Gamma \vdash_a t_1 \,\mathsf{to}\, x_1.\, (t_2 \,\mathsf{to}\, x_2.\, t) =_{\underline{\tau}} t_2 \,\mathsf{to}\, x_2.\, (t_1 \,\mathsf{to}\, x_1.\, t)$$

holds for all $\Gamma \vdash t_1{:}F\beta_1$, $\Gamma \vdash t_2{:}F\beta_2$, and $\Gamma, x_1{:}\beta_1, x_2{:}\beta_2 \vdash t{:}F\underline{\tau}$.

Proof The idea of the proof is that we use algebraicity of operations to move operation symbols to the outside of the computation term, where we can reorder them due to commutativity of the effect theory. Then, using the induction hypothesis, we can reorder the arguments, and finally we move the operation symbols back to the inside.

According to Proposition 3.19, we can assume that t_1 is in canonical form and proceed by induction on its structure:

- if $t_1 = \mathsf{return}\, b_1$ for some base term $\Gamma \vdash b_1{:}\beta_1$, we have:

$$\begin{aligned}
&\Gamma \vdash_a \mathsf{return}\, b \,\mathsf{to}\, x_1.\, (t_2 \,\mathsf{to}\, x_2.\, t) && \\
&\quad =_{\underline{\tau}} (t_2 \,\mathsf{to}\, x_2.\, t)[b/x_1] && \text{(by β-equivalence)} \\
&\quad =_{\underline{\tau}} t_2 \,\mathsf{to}\, x_2.\, t[b/x_1] && \text{(since x_1 does not appear in t_2)} \\
&\quad =_{\underline{\tau}} t_2 \,\mathsf{to}\, x_2.\, (\mathsf{return}\, b_1 \,\mathsf{to}\, x_1.\, t) && \text{(by β-equivalence)}\ ;
\end{aligned}$$

- otherwise, we have $t_1 = \mathsf{op}(t_{1i})_i$ for some operation symbol $\mathsf{op}\!:\!n \in \Sigma_{\text{eff}}$ and computation terms $\Gamma \vdash t_{1i}\!:\!F\beta_1$ such that

$$\Gamma \vdash_a t_{1i}\,\mathsf{to}\,x_1.\,(t_2\,\mathsf{to}\,x_2.\,t) =_{\underline{\tau}} t_2\,\mathsf{to}\,x_2.\,(t_{1i}\,\mathsf{to}\,x_1.\,t)$$

holds for all t_2. Next, assume that t_2 is in canonical form and proceed by induction on its structure to show that

$$\Gamma \vdash_a \mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,(t_2\,\mathsf{to}\,x_2.\,t) =_{\underline{\tau}} t_2\,\mathsf{to}\,x_2.\,(\mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,t)$$

holds:

  - if $t_2 = \mathsf{return}\,b_2$ for some base term $\Gamma \vdash b_2\!:\!\beta_2$, the argument is similar as in the base case for t_1;
  - otherwise, we have $t_2 = \mathsf{op}(t_{2j})_j$ for some operation symbol $\mathsf{op}'\!:\!n' \in \Sigma_{\text{eff}}$ and computation terms $\Gamma \vdash t_{2i}\!:\!F\beta_2$ such that

$$\Gamma \vdash_a \mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,(t_{2j}\,\mathsf{to}\,x_2.\,t) =_{\underline{\tau}} t_{2j}\,\mathsf{to}\,x_2.\,(\mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,t)$$

holds. Then, we have

$$\begin{aligned}
&\Gamma \vdash_a \mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,(\mathsf{op}'(t_{2j})_j\,\mathsf{to}\,x_2.\,t) && \\
&\quad =_{\underline{\tau}} \mathsf{op}(t_{1i}\,\mathsf{to}\,x_1.\,(\mathsf{op}'(t_{2j})_j\,\mathsf{to}\,x_2.\,t))_i && \text{(by algebraicity of operations)}\\
&\quad =_{\underline{\tau}} \mathsf{op}(\mathsf{op}'(t_{2j})_j\,\mathsf{to}\,x_2.\,(t_{1i}\,\mathsf{to}\,x_1.\,t))_i && \text{(by hypothesis of induction on } t_1\text{)}\\
&\quad =_{\underline{\tau}} \mathsf{op}(\mathsf{op}'(t_{2j}\,\mathsf{to}\,x_2.\,(t_{1i}\,\mathsf{to}\,x_1.\,t))_j)_i && \text{(by algebraicity of operations)}\\
&\quad =_{\underline{\tau}} \mathsf{op}'(\mathsf{op}(t_{2j}\,\mathsf{to}\,x_2.\,(t_{1i}\,\mathsf{to}\,x_1.\,t))_i)_j && \text{(by commutativity of } \mathcal{T}_{\text{eff}}\text{)}\\
&\quad =_{\underline{\tau}} \mathsf{op}'(\mathsf{op}(t_{1i}\,\mathsf{to}\,x_1.\,(t_{2j}\,\mathsf{to}\,x_2.\,t))_i)_j && \text{(by hypothesis of induction on } t_1\text{)}\\
&\quad =_{\underline{\tau}} \mathsf{op}'(\mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,(t_{2j}\,\mathsf{to}\,x_2.\,t))_j && \text{(by algebraicity of operations)}\\
&\quad =_{\underline{\tau}} \mathsf{op}'(t_{2j}\,\mathsf{to}\,x_2.\,(\mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,t))_j && \text{(by hypothesis of induction on } t_2\text{)}\\
&\quad =_{\underline{\tau}} \mathsf{op}'(t_{2j})_j\,\mathsf{to}\,x_2.\,(\mathsf{op}(t_{1i})_i\,\mathsf{to}\,x_1.\,t) && \text{(by algebraicity of operations)}\ .
\end{aligned}$$

Note that we could also proceed by induction on the sum of sizes of t_1 and t_2, and that the resulting proof would be shorter. However, an advantage of a proof with structural induction is that it easily adapts to a proof with the principle of induction in the logic. □

Example 3.24 The effect theory for nondeterminism is commutative, as we have

$$\vdash_{\mathcal{T}} \mathsf{or}(\mathsf{or}(z_{11}, z_{12}), \mathsf{or}(z_{21}, z_{22})) = \mathsf{or}(\mathsf{or}(z_{11}, z_{21}), \mathsf{or}(z_{12}, z_{22}))\,.$$

Hence, in absence of other effects, the order of evaluation of nondeterministic computations does not matter. Note that commutativity of the effect theory is a consequence of both commutativity *and* associativity of or.

Theorem 3.25 (Completeness) *If for computation terms $\Gamma \vdash t : \underline{\tau}$ and $\Gamma \vdash t' : \underline{\tau}$, we have $[\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}}$ for all models $\mathcal{M}$ of the base theory $\mathcal{T}_{\text{base}}$, then $\Gamma \vdash_a t =_{\underline{\tau}} t'$.*

Proof First, assume that the effect theory $\mathcal{T}_{\text{eff}}$ is consistent. If not, we immediately get $\Gamma \vdash_a t =_{\underline{\tau}} t'$ using the rule for inheritance from the effect theory.

Now, take a model $\mathcal{M}$ of the base theory, defined by setting $[\![\beta]\!]_{\mathcal{M}}$ to be the set of equivalence classes $[\Gamma \vdash b : \beta]$ of base terms b of base type β in the context Γ, modulo the provable equality of $\mathcal{T}_{\text{base}}$.

Due to Proposition 3.19, we can assume without loss of generality that both t and t' are in canonical form. Furthermore, say that $\underline{\tau} = F\beta$ for some base type β. Now, we can construct effect terms $(z_i)_i \vdash e$ and $(z_i)_i \vdash e'$, and base terms $\Gamma \vdash b_i : \beta$ such that

$$\Gamma \vdash_a t =_{F\beta} e[\mathsf{return}\, b_i/z_i]_i\,,$$
$$\Gamma \vdash_a t' =_{F\beta} e'[\mathsf{return}\, b_i/z_i]_i\,.$$

Note that we instantiate two different effect terms with the same set of computation terms return b_i. Furthermore, we may assume that $[\![b_i]\!]_{\mathcal{M}} \neq [\![b_j]\!]_{\mathcal{M}}$ for $i \neq j$, as otherwise, the definition of the model $\mathcal{M}$ implies $\Gamma \vdash_{\mathcal{T}_{\text{base}}} b_i =_\beta b_j$, and we may replace b_i by b_j.

Next, let $A = \{[\![b_i]\!]_{\mathcal{M}}\}_i \subset [\![\beta]\!]_{\mathcal{M}}$. By Lemma 3.13, we have

$$\begin{aligned} &[\![t]\!]_{\mathcal{M}} \\ &= [\![e[\mathsf{return}\, b_i/z_i]_i]\!]_{\mathcal{M}} \\ &= [\![F\beta]\!]_{\mathcal{M}}([\![e]\!]_{\mathcal{L}}) \circ \langle [\![\mathsf{return}\, b_i]\!]_{\mathcal{M}}\rangle_i \\ &= j \circ [\![e[[\![b_i]\!]_{\mathcal{M}}/z_i]_i]\!]_{FA}\,, \end{aligned}$$

where j is the canonical embedding of the carrier $U[\![FA]\!]$ into the carrier $U[\![F\beta]\!]$.

A similar equality holds for t', hence we get

$$j \circ [\![e[[\![b_i]\!]_{\mathcal{M}}/z_i]_i]\!]_{FA} = [\![t]\!]_{\mathcal{M}} = [\![t']\!]_{\mathcal{M}} = j \circ [\![e'[[\![b_i]\!]_{\mathcal{M}}/z_i]_i]\!]_{FA}\,.$$

Since the effect theory is consistent, j is injective. Thus, Proposition 2.12 implies

$$(z_i)_i \vdash_{\mathcal{T}_{\text{eff}}} e = e' \,.$$

Using the rule for inheritance from the effect theory, we get $\Gamma \vdash_a t =_{\underline{\tau}} t'$. □

Chapter 4

Language of the logic

The main purpose of the a-calculus is to emphasise the distinct features of algebraic effects: the layered structure, the interpretation of computations with models of Lawvere theories, and the algebraicity of operations. Now, taking these observations into account, we extend the a-calculus to a more powerful logic for algebraic effects.

We start by giving the language of the logic, which we build in three stages: first, we represent the values and effects of the underlying system; then, we introduce the terms of the logic; and, finally, we give the propositions and the predicates of the logic.

We give the semantics of the logic in the category **Set**. It is locally countably presentable, hence we can interpret free models of countable Lawvere theories. It is cartesian closed and cocomplete, so we can interpret the rest of the type system. And it gives a simple semantics to propositions and predicates, even with their fixed points.

The question of a logic over a general category is beyond the scope of this thesis. Still, if we want to model recursion, the semantics easily adapts to one in the category ω-**Cpo**, as shown in Chapter 9. Furthermore, a substantial amount of development has been done on combining call-by-push-value with polymorphism [MS09], or representing local state with Lawvere theories [PP02], and both could be used to introduce these features to the logic.

4.1 The effect theory

We generalise the approach taken in the a-calculus and describe the built-in values with a first-order rather than an equational theory.

Definition 4.1 The *base theory* $\mathcal{T}_{\text{base}}$ is a multi-sorted first-order theory over a *base signature* Σ_{base}. Its sorts are called *base types* β, its variables *base variables* x, its contexts *base contexts* Γ, its terms *base terms* b, and its formulae *base formulae* φ. Out of the base types, we select a subset of *arity types* α, which are the types of outcomes of effects.

As in the a-calculus, the interpretation of the base theory completely determines the interpretation of the rest of the logic. However, in order to get an interpretation of operations in countable Lawvere theories, we have to restrict their outcomes to countable sets.

Definition 4.2 A model $\mathcal{M}$ of the base theory $\mathcal{T}_{\text{base}}$ is a *model of the logic*, if it maps arity sorts to countable sets.

In the development of the logic, we assume a fixed model $\mathcal{M}$.

The effects at hand are also given by a modification of a single-sorted equational theory, which provides a finitary notation for describing effects that are given by an infinite family of operations, have an infinite number of outcomes, or are described by an infinite number of equations [PP03].

Definition 4.3 An *effect signature* Σ_{eff} consists of a finite list of *operation symbols* op, together with a (possibly empty) list of *parameter base types* $\boldsymbol{\beta}$, and a (possibly empty) list of lists of *argument arity types* $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$, written as

$$\mathsf{op} \colon \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \, .$$

We omit the semicolon when $\boldsymbol{\beta}$ is empty, and we write n instead of $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ when all the $\boldsymbol{\alpha}_i$ are empty.

The intuition behind the generalised operations is as follows. Instead of having a set of nearly identical operations, for example $\mathsf{update}_{\ell,d} \colon 1$ for each location ℓ and datum d, we take a single operation $\mathsf{update} \colon \mathbf{loc}, \mathbf{dat}; 1$ with parameter types $\mathbf{loc}$ of locations and $\mathbf{dat}$ of data. Next, if we were to describe a memory holding an infinite set of data by routinely generalising the operations to countable ones, we would be left with an infinitary syntax [PP01]. Instead, we allow arguments of operations to be dependent on values of arity types. For example: $\mathsf{lookup}:\mathbf{loc};\mathbf{dat}$ has, in addition to a location parameter, an argument that depends on the datum, stored in the location.

Remark 4.4 The choice of the operation symbol signature seems arbitrary — why have a single list of parameter types, but multiple lists of argument types? In seeking generality, one might consider arbitrary first-order combination of base types. Due to the distributivity of products over sums, each such combination is equivalent to a sum of products of base types. Hence, the argument types of operations are already in the most general form, just expressed in a notation without products and sums. Now, suppose we have an operation op with a parameter type

$$\prod_{i_1} \beta_{i_1} + \cdots + \prod_{i_m} \beta_{i_m} .$$

To give such a parameter is the same as to give an index j and a parameter from $\prod_{i_j} \beta_{i_j}$. Thus, op can be represented by m operation symbols op_j with parameter types $\prod_{i_j} \beta_{i_j}$, which is again the generality achieved in our syntax.

Definition 4.5 Take a countably infinite set of *effect variables* z. Then, the set of *effect terms* e is given by the following grammar:

$$e ::= z(\boldsymbol{b}) \mid \mathsf{op}_{\boldsymbol{b}}(\boldsymbol{x}_1.\, e_1, \ldots, \boldsymbol{x}_n.\, e_n)$$

When the list $\boldsymbol{x}_i$ is empty, we write e_i instead of $\boldsymbol{x}_i.\, e_i$.

To reflect the dependency on values, we type effect terms in a base context Γ and an *effect context* Z, consisting of effect variables $z:(\boldsymbol{\alpha})$, each associated to a list of arity types.

Definition 4.6 An *effect typing judgement* $\Gamma;Z \vdash e$ states that an effect term e is well-typed in Γ and Z. Effect typing judgements are given inductively by the following inductive rules:

$$\frac{\Gamma \vdash \boldsymbol{b}:\boldsymbol{\alpha}}{\Gamma;Z \vdash z(\boldsymbol{b})} \quad (z:(\boldsymbol{\alpha}) \in Z) ,$$

$$\frac{\Gamma \vdash \boldsymbol{b}:\boldsymbol{\beta} \qquad \Gamma, \boldsymbol{x}_i:\boldsymbol{\alpha}_i;\, Z \vdash e_i \quad (1 \leqslant i \leqslant n)}{\Gamma;Z \vdash \mathsf{op}_{\boldsymbol{b}}(\boldsymbol{x}_1.\, e_1, \ldots, \boldsymbol{x}_n.\, e_n)} \quad (\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n) .$$

To describe the case when an equation holds only for a particular subset of parameters, we equip equations between effect terms with side-conditions, which are base formulae, as defined in Section 2.2.

Definition 4.7 An *effect theory* $\mathcal{T}_{\mathsf{eff}}$ is a finite collection of *conditional equations*

$$\Gamma; Z \vdash e = e' \, (\varphi) \, ,$$

between effect terms $\Gamma; Z \vdash e$ and $\Gamma; Z \vdash e'$ with a *side condition* $\Gamma \vdash \varphi : \mathbf{form}$. We write $\Gamma; Z \vdash_{\mathcal{T}_{\mathsf{eff}}} e = e' \, (\varphi)$ if the equation $\Gamma; Z \vdash e = e' \, (\varphi)$ is in $\mathcal{T}_{\mathsf{eff}}$.

Altogether, this allows us to limit ourselves to a finite list of operations and a finite list of equations describing them, which then allows finitary rules in the logic.

Remark 4.8 Note that the effect theory is just a collection of conditional equations rather than a reasoning system. Although there is a way of stating the effect theory in terms of a conditional equational theory [Plo06], we choose a simpler path and move all the reasoning into the logic.

Now, we take another look at the examples of effect theories for various effects in the a-calculus, and adapt them to the presentation in the logic.

Example 4.9 (Exceptions) To describe a set of exceptions E, the base signature Σ_{base} consists of a base type **exc** and a constant symbol $\mathsf{exc} : \mathbf{exc}$ for each $\mathsf{exc} \in E$, while the base theory $\mathcal{T}_{\mathsf{base}}$ is trivial.

The effect signature Σ_{eff} contains a nullary operation symbol $\mathsf{raise} : \mathbf{exc}; 0$, while the effect theory $\mathcal{T}_{\mathsf{eff}}$ is empty. Then, an effect term $\mathsf{raise}_{\mathsf{exc}}$ represents the computation that raises exception exc.

Example 4.10 (Nondeterminism) The description of nondeterminism is the same as before, except that for the effect theory, we take the three equations which state that $\mathsf{or} : 2$ is associative, commutative, and idempotent. To justify that this collection of equations is an accurate description of nondeterminism, we observe that it describes properties of the nondeterministic choice and that it abbreviates an equational theory, which is Hilbert-Post complete.

Example 4.11 (Interactive input and output) To describe interactive input and output on a (now not necessarily finite) alphabet A, we take a base signature

Σ_{base}, consisting of a base type **char** and appropriate constant symbols $\mathsf{a}:\mathbf{char}$ for each $\mathsf{a} \in A$, and the trivial base theory $\mathcal{T}_{\text{base}}$.

We take an effect signature Σ_{eff}, containing an operation symbol $\mathsf{input}:\mathbf{char}$ and an operation symbol $\mathsf{output}:\mathbf{char};1$, together with the empty effect theory.

Then a computation that waits for the user's input, repeats it twice and then proceeds as z is represented by the effect term

$$\mathsf{input}(a.\, \mathsf{output}_a(\mathsf{output}_a(z)))\,.$$

Example 4.12 (Time) We can represent time with a unary operation symbol $\mathsf{tick}:1$ as before. Alternatively, we can first represent integers (or other suitable monoid) with the base signature and theory as in the a-calculus. Then, we take an effect signature Σ_{eff}, containing a unary operation symbol $\mathsf{tick}:\mathbf{int};1$ and an effect theory $\mathcal{T}_{\text{eff}}$, containing the following two equations:

$$z \vdash \mathsf{tick}_0(z) = z\,,$$
$$x_1:\mathbf{int}, x_2:\mathbf{int}; z \vdash \mathsf{tick}_{x_1}(\mathsf{tick}_{x_2}(z)) = \mathsf{tick}_{x_1+x_2}(z)\,.$$

Whereas in the a-calculus, the arithmetic in the representation of time had to be done on the meta-level, the current presentation allows arbitrary base terms as parameters.

Example 4.13 (State) For state, the base signature Σ_{base} contains a base type **loc** of memory locations, an arity type **dat** of data, and appropriate function and relation symbols to represent the locations and data, while the effect signature Σ_{eff} consists of operation symbols $\mathsf{lookup}:\mathbf{loc};\mathbf{dat}$ and $\mathsf{update}:\mathbf{loc},\mathbf{dat};1$.

Then, a computation that copies the content of ℓ to ℓ' and proceeds as z, can be represented by an effect term

$$\mathsf{lookup}_\ell(d.\, \mathsf{update}_{\ell',d}(z))\,.$$

The effect theory comprises the following conditional equations, again omitting the context [Plo06]:

$$
\begin{aligned}
\mathsf{lookup}_{\ell}(d.\,\mathsf{lookup}_{\ell}(d'.\,z(d,d'))) &= \mathsf{lookup}_{\ell}(d.\,z(d,d))\,, \\
\mathsf{lookup}_{\ell}(d.\,\mathsf{update}_{\ell,d}(z)) &= z\,, \\
\mathsf{update}_{\ell,d}(\mathsf{lookup}_{\ell}(d'.\,z(d'))) &= \mathsf{update}_{\ell,d}(z(d))\,, \\
\mathsf{update}_{\ell,d}(\mathsf{update}_{\ell,d'}(z)) &= \mathsf{update}_{\ell,d'}(z)\,, \\
\mathsf{lookup}_{\ell}(d.\,\mathsf{lookup}_{\ell'}(d'.\,z(d,d'))) &= \mathsf{lookup}_{\ell'}(d'.\,\mathsf{lookup}_{\ell}(d.\,z(d,d'))) && (\ell \neq \ell')\,, \\
\mathsf{update}_{\ell,d}(\mathsf{lookup}_{\ell'}(d'.\,z(d'))) &= \mathsf{lookup}_{\ell'}(d'.\,\mathsf{update}_{\ell,d}(z(d'))) && (\ell \neq \ell')\,, \\
\mathsf{update}_{\ell,d}(\mathsf{update}_{\ell',d'}(z)) &= \mathsf{update}_{\ell',d'}(\mathsf{update}_{\ell,d}(z)) && (\ell \neq \ell')\,.
\end{aligned}
$$

Although the effect theory $\mathcal{T}_{\text{eff}}$ is not an equational theory, it is an abbreviation for a countable one as follows.

For each operation symbol $\mathsf{op}\colon \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n \in \Sigma_{\text{eff}}$ and each tuple $\boldsymbol{a} \in \llbracket \boldsymbol{\beta} \rrbracket$, take an operation symbol $\mathsf{op}_{\boldsymbol{a}}$ of countable arity $\sum_i |\llbracket \boldsymbol{\alpha}_i \rrbracket|$. Then each $\Gamma; Z \vdash e$ and each $\boldsymbol{\gamma} \in \llbracket \Gamma \rrbracket$ gives rise to a term $Z' \vdash e^{\boldsymbol{\gamma}}$, where Z' consists of variables $z^{\boldsymbol{a}}$ for each $z{:}(\boldsymbol{\alpha}) \in Z$ and $\boldsymbol{a} \in \llbracket \boldsymbol{\alpha} \rrbracket$, and $e^{\boldsymbol{\gamma}}$ is recursively defined by

$$
\begin{aligned}
(z(\boldsymbol{b}))^{\boldsymbol{\gamma}} &= z^{\llbracket \boldsymbol{b} \rrbracket(\boldsymbol{\gamma})}\,, \\
(\mathsf{op}_{\boldsymbol{b}}(\boldsymbol{x}_i.\, e_i)_i)^{\boldsymbol{\gamma}} &= \mathsf{op}_{\llbracket \boldsymbol{b} \rrbracket(\boldsymbol{\gamma})}((e_i^{\boldsymbol{\gamma}}[\boldsymbol{a}_i/\boldsymbol{x}_i])_{\boldsymbol{a}_i \in \llbracket \boldsymbol{\alpha}_i \rrbracket})_i\,.
\end{aligned}
$$

Example 4.14 To clarify the construction, take the effect theory for state, as described in Example 4.13. If we have an effect term

$$
\ell{:}\mathbf{loc}; z{:}(\mathbf{dat}) \vdash \mathsf{lookup}_{\ell}(d.\,\mathsf{update}_{\ell,d}(z(d)))\,,
$$

an element $l \in \llbracket \mathbf{loc} \rrbracket$, and if $\llbracket \mathbf{dat} \rrbracket = \{d_1, d_2, \dots\}$, then $Z' = z_{d_1}, z_{d_2}, \dots$ and

$$
(\mathsf{lookup}_{\ell}(d.\,\mathsf{update}_{\ell,d}(z(d))))^{l} = \mathsf{lookup}_{l}(\mathsf{update}_{l,d_1}(z_{d_1}), \mathsf{update}_{l,d_2}(z_{d_2}), \dots)\,.
$$

In the case of operation symbols with multiple lists of argument types, we proceed similarly.

The equational theory is generated by equations $Z' \vdash e_{\boldsymbol{\gamma}} = e'_{\boldsymbol{\gamma}}$ for any equation $\Gamma; Z \vdash e = e'\ (\varphi)$ in $\mathcal{T}_{\text{eff}}$ and any $\boldsymbol{\gamma} \in \llbracket \varphi \rrbracket \subset \llbracket \Gamma \rrbracket$.

Note that for the sake of simplicity, we will see each model M of the induced Lawvere theory $\mathcal{L}$ as a set UM, together with a map

$$
\mathsf{op}_M\colon \llbracket \boldsymbol{\beta} \rrbracket \times UM^{\llbracket \boldsymbol{\alpha}_1 \rrbracket} \times \cdots \times UM^{\llbracket \boldsymbol{\alpha}_n \rrbracket} \to UM
$$

for each $\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$, defined by

$$\mathsf{op}_M(\boldsymbol{a},f_1,\ldots,f_n) =_{\mathrm{def}} M(\mathsf{op}_{\boldsymbol{a}})(f_1(\boldsymbol{a}_1)_{\boldsymbol{a}_1\in[\![\boldsymbol{\alpha}_1]\!]},\ldots,f_n(\boldsymbol{a}_n)_{\boldsymbol{a}_n\in[\![\boldsymbol{\alpha}_n]\!]})\,.$$

Then, the following naturality result, analogous to Lemma 2.33, holds.

Proposition 4.15 *For any* $\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$ *and any map* $f\colon A\times B \to UFC$, *the diagram*

$$\begin{tikzcd}
A\times[\![\boldsymbol{\beta}]\!]\times\prod_{i=1}^{n}(UFB)^{[\![\boldsymbol{\alpha}_i]\!]} \arrow[r, "\tilde{f}"] \arrow[d, "\mathrm{id}_A\times\mathsf{op}_{FB}"'] & [\![\boldsymbol{\beta}]\!]\times\prod_{i=1}^{n}(UFC)^{[\![\boldsymbol{\alpha}_i]\!]} \arrow[d, "\mathsf{op}_{FC}"] \\
A\times UFB \arrow[r, "f^{\dagger}"'] & UFC
\end{tikzcd}$$

commutes, where

$$\tilde{f}(a,\boldsymbol{a},f_1,\ldots,f_n) = \langle \boldsymbol{a}, \boldsymbol{a}_1 \mapsto f^{\dagger}(a,f_1(\boldsymbol{a}_1)),\ldots,\boldsymbol{a}_n \mapsto f^{\dagger}(a,f_n(\boldsymbol{a}_n))\rangle\,.$$

Proof Take arbitrary $\boldsymbol{a}\in[\![\boldsymbol{\beta}]\!]$ and $f_i\in(UFB)^{[\![\boldsymbol{\alpha}_i]\!]}$ for $1\leqslant i\leqslant n$. By a straightforward countable generalisation of Lemma 2.33, we show that the family

$$\{FA[\![\mathsf{op}_{\boldsymbol{a}}]\!] : (UFA)_A^{\sum_{i=1}^{n}|[\![\boldsymbol{\alpha}_i]\!]|} \to UFA\}_A$$

is an algebraic operation. Hence, the following diagram commutes

$$\begin{tikzcd}
A\times(UFB)^{\sum_{i=1}^{n}|[\![\boldsymbol{\alpha}_i]\!]|} \arrow[r, "\langle f^{\dagger}\circ(\mathrm{id}_A\times\mathrm{pr}_j)\rangle_{j=1}^{\sum_{i=1}^{n}|[\![\boldsymbol{\alpha}_i]\!]|}"] \arrow[d, "\mathrm{id}_A\times FB[\![\mathsf{op}_{\boldsymbol{a}}]\!]"'] & (UFC)^{\sum_{i=1}^{n}|[\![\boldsymbol{\alpha}_i]\!]|} \arrow[d, "FC[\![\mathsf{op}_{\boldsymbol{a}}]\!]"] \\
A\times UFB \arrow[r, "f^{\dagger}"'] & UFC
\end{tikzcd}$$

and the proposition holds. □

This result can be extended to other models M, obtained by products and exponentials of the free models.

4.2 Term language

The term language of the logic for algebraic effects is based on Levy's call-by-push-value approach [Lev06a]. A minor difference is that in call-by-push-value one has no base types, but compensates for them with indexed products and sums, which our language does not have. A more significant difference is that for the purpose of the logic we allow variables over both values and computations.

Definition 4.16 The sets of *value types* σ and *computation types* $\underline{\tau}$ are given by the following grammar:

$$\begin{aligned} \sigma &::= \beta \mid \mathbf{1} \mid \sigma_1 \times \sigma_2 \mid \mathbf{0} \mid \sigma_1 + \sigma_2 \mid U\underline{\tau}\,, \\ \underline{\tau} &::= F\sigma \mid \underline{\mathbf{1}} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \sigma \to \underline{\tau}\,. \end{aligned}$$

The meaning behind value types is the usual one, except for $U\underline{\tau}$, which represents the type of thunked (or frozen) computations of type $\underline{\tau}$. One can imagine storing the code of a computation for the purpose of executing it later. The computation type $F\sigma$ has the same meaning as in the α-calculus: it is the type of computations that return values of type σ. As for finite products, $\underline{\mathbf{1}}$ is the unit type, and $\underline{\tau}_1 \times \underline{\tau}_2$ is the type of pairs of computations. Those do not evaluate sequentially, as one might expect, but only after selecting one of the components with a projection. The function type $\sigma \to \underline{\tau}$ is the type of computations of type $\underline{\tau}$ that expect a value of type σ before evaluating. We abbreviate $\sigma_1 \to \cdots \to \sigma_n \to \underline{\tau}$ by $\boldsymbol{\sigma} \to \underline{\tau}$.

Definition 4.17 Take disjoint countably infinite sets of *value variables* x and *computation variables* y. The sets of *value terms* v and *computation terms* t are given by the following grammar:

$$\begin{aligned} v &::= x \mid \mathsf{f}(\boldsymbol{v}) \mid \star \mid \langle v_1, v_2 \rangle \mid \mathsf{fst}\, v \mid \mathsf{snd}\, v \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \mathsf{thunk}\, t\,, \\ t &::= y \mid \mathsf{force}\, v \mid \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_1.\, t_1, \dots, \boldsymbol{x}_n.\, t_n) \mid \mathsf{if}\, \mathsf{rel}(\boldsymbol{v})\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2 \mid \\ &\qquad \mathsf{return}\, v \mid t\, \mathsf{to}\, x{:}\sigma.\, t' \mid \mathsf{zero}\, v \mid \mathsf{match}\, v\, \mathsf{with}\, \mathsf{inl}\, x_1{:}\sigma_1.\, t_1, \mathsf{inr}\, x_2{:}\sigma_2.\, t_2 \mid \\ &\qquad \underline{\star} \mid \langle t_1, t_2 \rangle \mid \mathsf{fst}\, t \mid \mathsf{snd}\, t \mid \lambda x{:}\sigma.\, t \mid t v\,, \end{aligned}$$

variables $\boldsymbol{x}_i$ in operation application are bound in t_i, where variable x in sequencing is bound in t', variables x_1 and x_2 in pattern matching are bound in t_1 and t_2, respectively, and variable x in lambda abstraction is bound in t, all according to the standard conventions of α-equivalence and renaming of bound variables in order to avoid clash of variables.

The meaning behind most of the value and computation terms is again standard. The value term $\mathsf{thunk}\, t$ represents the thunk of the computation term t, while the computation term $\mathsf{force}\, v$ represents the inverse procedure. Although we form sums of value terms, we match them in computation terms. In particular, $\mathsf{zero}\, v$ is a computation term that represents the unique map from the zero type, while $\mathsf{match}\, v\, \mathsf{with}\, \mathsf{inl}\, x_1 : \sigma_1.\ t_1, \mathsf{inr}\, x_2 : \sigma_2.\ t_2$ is the binary pattern matching construct. Often, we write conditionals of the form $\mathsf{if}\, v_1 = v_2\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2$. This is an abbreviation for $\mathsf{if}\, \mathsf{eq}(v_1, v_2)\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2$, where we have assumed that the base signature contains a relation symbol $\mathsf{eq} : (\beta, \beta)$ for an appropriate base type β and that the base theory contains the axiom

$$x_1 : \beta, x_2 : \beta \vdash \mathsf{eq}(x_1, x_2) \Leftrightarrow (x_1 =_\beta x_2)\,.$$

As in the a-calculus, bound variables in sequencing and pattern matching construct have to be explicitly typed in order to ensure a unique derivation of typing judgements. However, we often omit those types and write $t\, \mathsf{to}\, x.\ t'$ and $\mathsf{match}\, v\, \mathsf{with}\, \mathsf{inl}\, x_1.\ t_1, \mathsf{inr}\, x_2.\ t_2$ instead. Bound variables in operations do not have an explicit type assignment because their type is determined by the arity of operation symbols.

Instead of $\lambda x_1 : \sigma_1.\ \ldots \lambda x_n : \sigma_n.\ t$, we write $\lambda \boldsymbol{x} : \boldsymbol{\sigma}.\ t$, and instead of $(\ldots(t v_1)\ldots) v_n$ we write $\boldsymbol{t v}$. By using nested binary sums and zero type, we can define arbitrary finitary sums of n values, with injections inj_i for $1 \leqslant i \leqslant n$, and a pattern matching construct

$$\mathsf{match}\, x\, \mathsf{with} (\mathsf{inj}_i\, x_i.\ t_i)_i$$

as an abbreviation for nested binary pattern matching constructs and zero maps.

A *value context* Γ is a list

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

of distinct value variables, each paired to a single value type, and a *computation context* Δ is a list

$$y_1 : \underline{\tau}, \ldots, y_m : \underline{\tau}_m$$

of distinct computation variables, each paired to a single computation type.

Definition 4.18 A *value typing judgement* $\Gamma; \Delta \vdash v : \sigma$ states that a value term v has a value type σ in a value context Γ and a computation context Δ, while a *computation typing judgement* $\Gamma; \Delta \vdash t : \underline{\tau}$ states the analogue for a computation term t and a computation type $\underline{\tau}$. The typing judgements are given inductively by the following rules:

$$\frac{}{\Gamma;\Delta\vdash x{:}\sigma}\ (x{:}\sigma\in\Gamma),\qquad \frac{\Gamma;\Delta\vdash \boldsymbol{v}{:}\boldsymbol{\beta}}{\Gamma;\Delta\vdash \mathsf{f}(\boldsymbol{v}){:}\beta}\ (\mathsf{f}{:}(\boldsymbol{\beta})\to\beta\in\Sigma_{\mathrm{base}}),\qquad \frac{}{\Gamma;\Delta\vdash\star{:}\mathbf{1}},$$

$$\frac{\Gamma;\Delta\vdash v_1{:}\sigma_1\qquad \Gamma;\Delta\vdash v_2{:}\sigma_2}{\Gamma;\Delta\vdash\langle v_1,v_2\rangle{:}\sigma_1\times\sigma_2},\qquad \frac{\Gamma;\Delta\vdash v{:}\sigma_1\times\sigma_2}{\Gamma;\Delta\vdash \mathsf{fst}\,v{:}\sigma_1},\qquad \frac{\Gamma;\Delta\vdash v{:}\sigma_1\times\sigma_2}{\Gamma;\Delta\vdash \mathsf{snd}\,v{:}\sigma_2},$$

$$\frac{\Gamma;\Delta\vdash v{:}\sigma_1}{\Gamma;\Delta\vdash \mathsf{inl}\,v{:}\sigma_1+\sigma_2},\qquad \frac{\Gamma;\Delta\vdash v{:}\sigma_2}{\Gamma;\Delta\vdash \mathsf{inr}\,v{:}\sigma_1+\sigma_2},\qquad \frac{\Gamma;\Delta\vdash t{:}\underline{\tau}}{\Gamma;\Delta\vdash \mathsf{thunk}\,t{:}U\underline{\tau}},$$

$$\frac{}{\Gamma;\Delta\vdash y{:}\underline{\tau}}\ (y{:}\underline{\tau}\in\Delta),\qquad \frac{\Gamma;\Delta\vdash v{:}U\underline{\tau}}{\Gamma;\Delta\vdash \mathsf{force}\,v{:}\underline{\tau}},$$

$$\frac{\Gamma;\Delta\vdash \boldsymbol{v}{:}\boldsymbol{\beta}\qquad \Gamma,\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i;\,\Delta\vdash t_i{:}\underline{\tau}\quad (1\leqslant i\leqslant n)}{\Gamma;\Delta\vdash \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\,t_i)_i{:}\underline{\tau}}\ (\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n\in\Sigma_{\mathrm{eff}}),$$

$$\frac{\Gamma\vdash\varphi{:}\mathbf{form}\qquad \Gamma;\Delta\vdash t_1{:}\underline{\tau}\qquad \Gamma;\Delta\vdash t_2{:}\underline{\tau}}{\Gamma;\Delta\vdash \mathsf{if}\,\mathsf{rel}(\boldsymbol{v})\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2{:}\underline{\tau}},\qquad \frac{\Gamma;\Delta\vdash v{:}\sigma}{\Gamma;\Delta\vdash \mathsf{return}\,v{:}F\sigma},$$

$$\frac{\Gamma;\Delta\vdash t{:}F\sigma\qquad \Gamma,x{:}\sigma;\,\Delta\vdash t'{:}\underline{\tau}}{\Gamma;\Delta\vdash t\,\mathsf{to}\,x{:}\sigma.\,t'{:}\underline{\tau}},\qquad \frac{\Gamma;\Delta\vdash v{:}\mathbf{0}}{\Gamma;\Delta\vdash \mathsf{zero}\,v{:}\underline{\tau}},$$

$$\frac{\Gamma;\Delta\vdash v{:}\sigma_1+\sigma_2\qquad \Gamma,x_1{:}\sigma_1;\,\Delta\vdash t_1{:}\underline{\tau}\qquad \Gamma,x_2{:}\sigma_2;\,\Delta\vdash t_2{:}\underline{\tau}}{\Gamma;\Delta\vdash \mathsf{match}\,v\,\mathsf{with}\,\mathsf{inl}\,x_1{:}\sigma_1.\,t_1,\mathsf{inr}\,x_2{:}\sigma_2.\,t_2{:}\underline{\tau}},\qquad \frac{}{\Gamma;\Delta\vdash\underline{\star}{:}\underline{\mathbf{1}}},$$

$$\frac{\Gamma;\Delta\vdash t_1{:}\underline{\tau}_1\qquad \Gamma;\Delta\vdash t_2{:}\underline{\tau}_2}{\Gamma;\Delta\vdash\langle t_1,t_2\rangle{:}\underline{\tau}_1\times\underline{\tau}_2},\qquad \frac{\Gamma;\Delta\vdash t{:}\underline{\tau}_1\times\underline{\tau}_2}{\Gamma;\Delta\vdash \mathsf{fst}\,t{:}\underline{\tau}_1},\qquad \frac{\Gamma;\Delta\vdash t{:}\underline{\tau}_1\times\underline{\tau}_2}{\Gamma;\Delta\vdash \mathsf{snd}\,t{:}\underline{\tau}_2},$$

$$\frac{\Gamma,x{:}\sigma;\,\Delta\vdash t{:}\underline{\tau}}{\Gamma;\Delta\vdash\lambda x{:}\sigma.\,t{:}\sigma\to\underline{\tau}},\qquad \frac{\Gamma;\Delta\vdash t{:}\sigma\to\underline{\tau}\qquad \Gamma;\Delta\vdash v{:}\sigma}{\Gamma;\Delta\vdash t v{:}\underline{\tau}}.$$

For each operation $\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n$, we define a computation term $\mathsf{gen}_{\mathsf{op}}$, called the *generic effect of* op [PP03], defined by

$$\mathsf{gen}_{\mathsf{op}} =_{\mathrm{def}} \lambda\boldsymbol{x}{:}\boldsymbol{\beta}.\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,\mathsf{return}\,\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle)_i : \boldsymbol{\beta}\to F\big(\textstyle\prod\boldsymbol{\alpha}_1+\cdots+\prod\boldsymbol{\alpha}_n\big),$$

where $\prod\boldsymbol{\alpha}$ abbreviates $\alpha_1\times\cdots\times\alpha_n$.

Example 4.19 Corresponding generic effects for state are

$$\mathsf{gen}_{\mathsf{lookup}} =_{\mathsf{def}} \lambda\ell{:}\mathbf{loc}.\ \mathsf{lookup}_\ell(d.\ \mathsf{return}\, d){:}\mathbf{loc} \to F\mathbf{dat}$$

and

$$\mathsf{gen}_{\mathsf{update}} =_{\mathsf{def}} \lambda\ell{:}\mathbf{loc}, d{:}\mathbf{dat}.\ \mathsf{update}_{\ell,d}(\mathsf{return}\,\star){:}\mathbf{loc} \to \mathbf{dat} \to F\mathbf{1}\,.$$

Usually, one writes $!\ell$ instead of $\mathsf{gen}_{\mathsf{lookup}}\,\ell$ to denote a computation that returns the datum stored in ℓ, and $\ell := d$ instead of $(\mathsf{gen}_{\mathsf{update}}\,\ell)d$ to denote a computation that sets the location ℓ to d and returns an element of the unit type.

Value types σ are interpreted by sets $[\![\sigma]\!]$, while computation types $\underline{\tau}$ are interpreted by models $[\![\underline{\tau}]\!]$ of the countable Lawvere theory $\mathcal{L}$, induced by the infinitary equational theory, generated by the effect theory $\mathcal{T}_{\mathsf{eff}}$.

The value types are interpreted in the obvious way, with the type $U\underline{\tau}$ being interpreted by $U[\![\underline{\tau}]\!]$, where $U\colon \mathrm{Mod}_{\mathcal{L}}(\mathbf{Set}) \to \mathbf{Set}$ is the forgetful functor. The computation types are interpreted by

$$\begin{aligned}
[\![F\sigma]\!] &= F[\![\sigma]\!]\,,\\
[\![\underline{\mathbf{1}}]\!] &= \mathbf{1}\,,\\
[\![\underline{\tau}_1 \times \underline{\tau}_2]\!] &= [\![\underline{\tau}_1]\!] \times [\![\underline{\tau}_2]\!]\,,\\
[\![\sigma \to \underline{\tau}]\!] &= [\![\underline{\tau}]\!]^{[\![\sigma]\!]}\,,
\end{aligned}$$

where F is the free model functor, $\mathbf{1}$ is the final model, and the models $M_1 \times M_2$ and M^A are defined as in Section 2.3.

Contexts $x_1{:}\sigma_1,\dots,x_n{:}\sigma_n$ are interpreted by a set $[\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$, while contexts $y_1{:}\underline{\tau}_1,\dots,y_n{:}\underline{\tau}_n$ are interpreted by a set $U[\![\underline{\tau}_1]\!] \times \cdots \times U[\![\underline{\tau}_n]\!]$.

Value terms $\Gamma;\Delta \vdash v{:}\sigma$ are interpreted by functions $[\![v]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to [\![\sigma]\!]$ and computation terms $\Gamma;\Delta \vdash t{:}\underline{\tau}$ are interpreted by functions $[\![t]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to U[\![\underline{\tau}]\!]$. all defined mutually recursively on the typing judgement by:

$$\begin{aligned}
[\![x_1{:}\sigma_1,\dots,x_n{:}\sigma_n;\ \Delta \vdash x_i{:}\sigma_i]\!] &= \mathrm{pr}_i \circ \mathrm{pr}_1\,,\\
[\![\Gamma;\Delta \vdash \mathsf{f}(\boldsymbol{v}){:}\beta]\!] &= [\![\mathsf{f}]\!] \circ [\![\boldsymbol{v}]\!]\,,\\
[\![\Gamma;\Delta \vdash \star{:}\mathbf{1}]\!] &= !_{[\![\Gamma]\!] \times [\![\Delta]\!]}\,,
\end{aligned}$$

where $!_{[\![\Gamma]\!]\times[\![\Delta]\!]} : [\![\Gamma]\!] \times [\![\Delta]\!] \to \mathbf{1}$ is the unique map to the final object,

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \langle v_1, v_2\rangle : \sigma_1 \times \sigma_2]\!] &= \langle [\![v_1]\!], [\![v_2]\!]\rangle\ , \\
[\![\Gamma;\Delta \vdash \mathsf{fst}\, v : \sigma_1]\!] &= \mathrm{pr}_1 \circ [\![v]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{snd}\, v : \sigma_2]\!] &= \mathrm{pr}_2 \circ [\![v]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{inl}\, v : \sigma_1 + \sigma_2]\!] &= \mathrm{in}_1 \circ [\![v]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{inr}\, v : \sigma_1 + \sigma_2]\!] &= \mathrm{in}_2 \circ [\![v]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{thunk}\, t : U\underline{\tau}]\!] &= [\![t]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{force}\, v : \underline{\tau}]\!] &= [\![v]\!]\ , \\
[\![\Gamma; y_1 : \underline{\tau}_1, \ldots, y_n : \underline{\tau}_n \vdash y_i : \underline{\tau}_i]\!] &= \mathrm{pr}_i \circ \mathrm{pr}_2\ , \\
[\![\Gamma;\Delta \vdash \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i : \underline{\tau}]\!] &= \langle \boldsymbol{\gamma}, \boldsymbol{\delta}\rangle \mapsto \mathsf{op}_{[\![\underline{\tau}]\!]}([\![\boldsymbol{v}]\!](\boldsymbol{\gamma},\boldsymbol{\delta}), \widehat{[\![t_1]\!]}, \ldots, \widehat{[\![t_n]\!]})\ ,
\end{aligned}
$$

where $\widehat{[\![t_i]\!]} \in U[\![\underline{\tau}]\!]^{[\![\boldsymbol{\alpha}_i]\!]}$ is the transpose of $[\![t_i]\!](\boldsymbol{\gamma}, -, \boldsymbol{\delta})\colon [\![\boldsymbol{\alpha}_i]\!] \to U[\![\underline{\tau}]\!]$,

$$
[\![\Gamma;\Delta \vdash \mathsf{if}\ \mathsf{rel}(\boldsymbol{v})\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2 : \underline{\tau}]\!] = \langle \boldsymbol{\gamma}, \boldsymbol{\delta}\rangle \mapsto \begin{cases} [\![t_1]\!](\boldsymbol{\gamma},\boldsymbol{\delta}) & \text{if } [\![\boldsymbol{v}]\!](\langle \boldsymbol{\gamma},\boldsymbol{\delta}\rangle) \in [\![\mathsf{rel}]\!] \\ [\![t_2]\!](\boldsymbol{\gamma},\boldsymbol{\delta}) & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \mathsf{return}\, v : F\sigma]\!] &= \eta_{[\![\sigma]\!]} \circ [\![v]\!]\ , \\
[\![\Gamma;\Delta \vdash t\, \mathsf{to}\, x : \sigma.\, t' : \underline{\tau}]\!] &= [\![t']\!]^{\dagger} \circ \langle \mathrm{id}_{[\![\Gamma]\!]\times[\![\Delta]\!]}, [\![t]\!]\rangle\ , \\
[\![\Gamma;\Delta \vdash \mathsf{zero}\, v : \underline{\tau}]\!] &= ¡_{U[\![\underline{\tau}]\!]} \circ [\![v]\!]\ ,
\end{aligned}
$$

where $¡_{U[\![\underline{\tau}]\!]} : \mathbf{0} \to U[\![\underline{\tau}]\!]$ is the unique map from the initial object,

$$
[\![\Gamma;\Delta \vdash \mathsf{match}\, v\, \mathsf{with}\, \mathsf{inl}\, x_1 : \sigma_1.\, t_1, \mathsf{inr}\, x_2 : \sigma_2.\, t_2 : \underline{\tau}]\!] = [[\![t_1]\!], [\![t_2]\!]] \circ \psi \circ \langle \mathsf{prj}_1, [\![v]\!], \mathsf{prj}_2\rangle\ ,
$$

where $[[\![t_1]\!], [\![t_2]\!]]\colon [\![\sigma_1]\!] + [\![\sigma_2]\!] \to U[\![\underline{\tau}]\!]$ is the co-tuple of $[\![t_1]\!]$ and $[\![t_2]\!]$, and ψ is the canonical isomorphism

$$
[\![\Gamma]\!] \times ([\![\sigma_1]\!] + [\![\sigma_2]\!]) \times [\![\Delta]\!] \to [\![\Gamma]\!] \times [\![\sigma_1]\!] \times [\![\Delta]\!] + [\![\Gamma]\!] \times [\![\sigma_2]\!] \times [\![\Delta]\!]\ ,
$$

$$
\begin{aligned}
[\![\Gamma;\Delta \vdash \underline{\star} : \underline{\mathbf{1}}]\!] &= !_{[\![\Gamma]\!]\times[\![\Delta]\!]}\ , \\
[\![\Gamma;\Delta \vdash \langle t_1, t_2\rangle : \underline{\tau}_1 \times \underline{\tau}_2]\!] &= \langle [\![t_1]\!], [\![t_2]\!]\rangle\ , \\
[\![\Gamma;\Delta \vdash \mathsf{fst}\, t : \underline{\tau}_1]\!] &= \mathrm{pr}_1 \circ [\![t]\!]\ , \\
[\![\Gamma;\Delta \vdash \mathsf{snd}\, t : \underline{\tau}_2]\!] &= \mathrm{pr}_2 \circ [\![t]\!]\ , \\
[\![\Gamma;\Delta \vdash \lambda x : \sigma.\, t : \sigma \to \underline{\tau}]\!] &= \widehat{[\![t]\!]}\ , \\
[\![\Gamma;\Delta \vdash t v : \underline{\tau}]\!] &= \mathrm{ev}_{[\![\sigma]\!], U[\![\underline{\tau}]\!]} \circ \langle [\![t]\!], [\![v]\!]\rangle\ ,
\end{aligned}
$$

where $\mathrm{ev}_{[\![\sigma]\!],U[\![\underline{\tau}]\!]} : U[\![\underline{\tau}]\!]^{[\![\sigma]\!]} \times [\![\sigma]\!] \to U[\![\underline{\tau}]\!]$ is the evaluation map.

Remark 4.20 Computation terms can also be interpreted by morphisms in the co-Kleisli category of the adjunction $F \dashv U$, as their interpretations are of the form $A \times UM \to UN$, where $A = \prod_i [\![\sigma_i]\!]$ and $UM = U\prod_j [\![\underline{\tau}_j]\!] \cong \prod_j U[\![\underline{\tau}_j]\!]$. By transposition, we get a morphism of the form $UM \to U(N^A)$, and by adjunction to one of the form $FUM \to N^A$, which is a morphism in the co-Kleisli category. This differs from the standard interpretation of effectful call-by-value computations, which uses the Kleisli category [Mog91].

4.3 Judgements

Approaches to reasoning about computations can be grouped into two different classes [Pnu77]. On one hand, we have *endogenous* approaches, where the validity $t \vDash \varphi$ of propositions φ is studied with regard to a computation t, for example as in Hennessy-Milner logic [HM85]. On the other hand, we have *exogenous* approaches, where computations occur inside propositions, and the validity $\vdash \varphi$ is global, for example as in Pitts's evaluation logic [Pit91].

Because we strive to obtain a very general logic, we take the exogenous approach: it can express the endogenous one by translating $t \vDash \varphi$ to $\vdash \varphi^*(t)$ for a suitable predicate φ^*. For that reason, we find it convenient for our logic to have predicates in addition to propositions.

Definition 4.21 Take a countably infinite set of*predicate variables P*. Then the sets of*propositions* φ and*predicates* π is given by the following grammar:

$$
\begin{aligned}
\varphi ::= {} & \pi(\boldsymbol{v};\boldsymbol{t}) \mid \mathsf{rel}(\boldsymbol{v}) \mid v_1 =_\sigma v_2 \mid t_1 =_{\underline{\tau}} t_2 \mid \\
& \top \mid \varphi_1 \wedge \varphi_2 \mid \bot \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \\
& \forall x{:}\sigma.\, \varphi \mid \exists x{:}\sigma.\, \varphi \mid \forall y{:}\underline{\tau}.\, \varphi \mid \exists y{:}\underline{\tau}.\, \varphi \mid , \\
\pi ::= {} & P \mid (\boldsymbol{x}{:}\boldsymbol{\sigma};\boldsymbol{y}{:}\underline{\boldsymbol{\tau}}).\, \varphi \mid \nu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi \mid \mu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi\,.
\end{aligned}
$$

Most of the propositions and predicates represent standard logical constructions. A proposition $\pi(\boldsymbol{v};\boldsymbol{t})$ represents an application of a predicate π to value terms $\boldsymbol{v}$ and computation terms $\boldsymbol{t}$, while $\mathsf{rel}(\boldsymbol{v})$ represents an application of a relation symbol $\mathsf{rel}{:}(\boldsymbol{\beta}) \in \Sigma_{\mathrm{base}}$. We define negation by $\neg\varphi =_{\mathrm{def}} \varphi \Rightarrow \bot$.

Predicates are constructed using predicate variables, abstracted propositions, or fixed point constructors. The main purpose of the latter is to express various global modalities.

For distinct predicate variables $\boldsymbol{P}$, predicates $\boldsymbol{\pi}$, and a proposition φ, we define $\varphi[\boldsymbol{\pi}/\boldsymbol{P}]$ to be the proposition, obtained by the standard simultaneous substitution of variables P_i by π_i in φ. A predicate $\pi[\boldsymbol{\pi}/\boldsymbol{P}]$ is defined analogously.

Definition 4.22 A variable P is *positive* (*negative*) in φ, if:

- it does not occur in φ;
- φ is of the form $\pi(\boldsymbol{v};\boldsymbol{t})$, and P is positive (negative) in π;
- φ is of the form $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ and P is positive (negative) in φ_1 and φ_2;
- φ is of the form $\varphi_1 \Rightarrow \varphi_2$ and P is negative (positive) in φ_1 and positive (negative) in φ_2;
- φ is of the form $\forall x\!:\!\sigma.\,\varphi'$, $\exists x\!:\!\sigma.\,\varphi'$, $\forall y\!:\!\underline{\tau}.\,\varphi'$, or $\exists y\!:\!\underline{\tau}.\,\varphi'$ and P is positive (negative) in φ';

and positive (negative) in π, if

- it is equal to π;
- π is of the form $(\boldsymbol{x}\!:\!\boldsymbol{\sigma};\boldsymbol{y}\!:\!\underline{\boldsymbol{\tau}}).\,\varphi$, and P is positive (negative) in φ;
- π is of the form $\nu P'\!:\!(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi'$ or $\mu P'\!:\!(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi'$ for some $P' \neq P$ and P is positive (negative) in π'.

A *predicate context* Π is list of distinct predicate variables $P\!:\!\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$, each paired with a list of value types $\boldsymbol{\sigma}$ and a list of computation types $\underline{\boldsymbol{\tau}}$.

Definition 4.23 *Proposition typing judgement* $\Gamma;\Delta;\Pi \vdash \varphi\!:\!\mathbf{prop}$ states that in a value context Γ, a computation context Δ, and a predicate context Π, a proposition φ is well-typed. *Predicate typing judgement* $\Gamma;\Delta;\Pi \vdash \pi\!:\!\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$ states the analogue for a predicate π. The typing judgements are given inductively by the following rules:

$$\frac{\Gamma;\Delta\vdash \boldsymbol{v}:\boldsymbol{\sigma} \qquad \Gamma;\Delta\vdash \boldsymbol{t}:\underline{\boldsymbol{\tau}} \qquad \Gamma;\Delta;\Pi\vdash \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}{\Gamma;\Delta;\Pi\vdash \pi(\boldsymbol{v};\boldsymbol{t}):\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta\vdash \boldsymbol{v}:\boldsymbol{\beta}}{\Gamma;\Delta;\Pi\vdash \mathsf{rel}(\boldsymbol{v}):\mathbf{prop}}\;(\mathsf{rel}:(\boldsymbol{\beta})\in\Sigma_{\mathrm{base}})\,,\qquad \frac{\Gamma;\Delta\vdash v_1:\sigma \qquad \Gamma;\Delta\vdash v_2:\sigma}{\Gamma;\Delta;\Pi\vdash v_1 =_\sigma v_2:\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta\vdash t_1:\underline{\tau} \qquad \Gamma;\Delta\vdash t_2:\underline{\tau}}{\Gamma;\Delta;\Pi\vdash t_1 =_{\underline{\tau}} t_2:\mathbf{prop}}\,,\qquad \frac{}{\Gamma;\Delta;\Pi\vdash \top:\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta;\Pi\vdash \varphi_1:\mathbf{prop} \qquad \Gamma;\Delta;\Pi\vdash \varphi_2:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \varphi_1\wedge\varphi_2:\mathbf{prop}}\,,\qquad \frac{}{\Gamma;\Delta;\Pi\vdash \bot:\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta;\Pi\vdash \varphi_1:\mathbf{prop} \qquad \Gamma;\Delta;\Pi\vdash \varphi_2:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \varphi_1\vee\varphi_2:\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta;\Pi\vdash \varphi_1:\mathbf{prop} \qquad \Gamma;\Delta;\Pi\vdash \varphi_2:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \varphi_1\Rightarrow\varphi_2:\mathbf{prop}}\,,\qquad \frac{\Gamma,x:\sigma;\ \Delta;\Pi\vdash \varphi:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \forall x:\sigma.\ \varphi:\mathbf{prop}}\,,$$

$$\frac{\Gamma,x:\sigma;\Delta;\Pi\vdash \varphi:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \exists x:\sigma.\ \varphi:\mathbf{prop}}\,,\qquad \frac{\Gamma;\Delta,y:\underline{\tau};\ \Pi\vdash \varphi:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \forall y:\underline{\tau}.\ \varphi:\mathbf{prop}}\,,$$

$$\frac{\Gamma;\Delta,y:\underline{\tau};\ \Pi\vdash \varphi:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash \exists y:\underline{\tau}.\ \varphi:\mathbf{prop}}\,,\qquad \frac{}{\Gamma;\Delta;\Pi\vdash P:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}\;(P:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})\in\Pi)\,,$$

$$\frac{\Gamma,\boldsymbol{x}:\boldsymbol{\sigma};\ \Delta,\boldsymbol{y}:\underline{\boldsymbol{\tau}};\ \Pi\vdash \varphi:\mathbf{prop}}{\Gamma;\Delta;\Pi\vdash (\boldsymbol{x}:\boldsymbol{\sigma};\boldsymbol{y}:\underline{\boldsymbol{\tau}}).\ \varphi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}\,,$$

$$\frac{\Gamma;\Delta;\Pi,P:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})\vdash \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}{\Gamma;\Delta;\Pi\vdash \nu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\ \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}\;(P\text{ is positive in }\pi)\,,$$

$$\frac{\Gamma;\Delta;\Pi,P:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})\vdash \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}{\Gamma;\Delta;\Pi\vdash \mu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\ \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})}\;(P\text{ is positive in }\pi)\,.$$

When any of the contexts is empty, we also omit the corresponding semicolon. Note that there is an evident inclusion taking $\Gamma\vdash\varphi:\mathbf{form}$ into $\Gamma;\Delta;\Pi\vdash\varphi:\mathbf{prop}$, hence we treat base formulae as a subset of the propositions of the logic.

Having given the syntax of propositions and predicates, we describe judgements of the logic. These are of the form

$$\Gamma;\Delta;\Pi \mid \Psi \vdash \varphi\,,$$

where Ψ is a set of *hypotheses* $\varphi_1,\dots,\varphi_n$ and φ is the *conclusion*, with all well-typed propositions in the contexts $\Gamma;\Delta;\Pi$. We write $\Gamma;\Delta;\Pi \vdash \varphi$ when the set of hypotheses is empty.

We interpret predicate contexts

$$\Pi = P_1\!:\!\mathbf{prop}(\boldsymbol{\sigma}_1;\underline{\boldsymbol{\tau}}_1),\dots,P_n\!:\!\mathbf{prop}(\boldsymbol{\sigma}_n;\underline{\boldsymbol{\tau}}_n)$$

by sets

$$[\![\Pi]\!] = \mathcal{P}([\![\boldsymbol{\sigma}_1]\!] \times U[\![\underline{\boldsymbol{\tau}}_1]\!]) \times \cdots \times \mathcal{P}([\![\boldsymbol{\sigma}_n]\!] \times U[\![\underline{\boldsymbol{\tau}}_n]\!])\,.$$

To interpret predicate fixed points $\nu P\!:\!(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\varphi$ and $\mu P\!:\!(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\varphi$ we are going to employ an easy adaptation of Tarski's theorem with parameters, which states that every monotone map on a complete lattice has a greatest post-fixed point and a least pre-fixed point [Tar55].

Theorem 4.24 *Take a map $F: \mathcal{P}(A_1)\times\cdots\times\mathcal{P}(A_n)\times\mathcal{P}(A) \to \mathcal{P}(A)$, monotone in the last argument. Then, there exists maps*

$$\nu_F: \mathcal{P}(A_1) \times \cdots \times \mathcal{P}(A_n) \to \mathcal{P}(A)\,,$$

and

$$\mu_F: \mathcal{P}(A_1) \times \cdots \times \mathcal{P}(A_n) \to \mathcal{P}(A)$$

such that:

- *for any $\boldsymbol{U} \in \mathcal{P}(A_1)\times\cdots\times\mathcal{P}(A_n)$, the map $F(\boldsymbol{U},-): \mathcal{P}(A) \to \mathcal{P}(A)$ has a greatest post-fixed point $\nu_F(\boldsymbol{U})$ and a least pre-fixed point $\mu_F(\boldsymbol{U})$;*
- *if F is monotone (anti-monotone) in $\mathcal{P}(A_i)$, then so are ν_F and μ_F.*

Then, we interpret propositions $\Gamma;\Delta;\Pi \vdash \varphi\!:\!\mathbf{prop}$ by subsets

$$[\![\varphi]\!] \subseteq [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]\,,$$

and predicates $\Gamma;\Delta;\Pi \vdash \pi\!:\!\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$ by maps

$$[\![\varphi]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!] \to \mathcal{P}([\![\boldsymbol{\sigma}]\!] \times U[\![\underline{\boldsymbol{\tau}}]\!])\,,$$

defined recursively on the derivation of the typing judgement.

Propositions are interpreted as in multi-sorted first-order theories in Section 2.2, with the predicate application interpreted by

$$[\![\Gamma;\Delta;\Pi \vdash \pi(\boldsymbol{v};\boldsymbol{t})\colon\mathbf{prop}]\!] = \{\langle\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\Gamma]\!]\times[\![\Delta]\!]\times[\![\Pi]\!] \mid \langle[\![\boldsymbol{v}]\!],[\![\boldsymbol{t}]\!]\rangle(\boldsymbol{\gamma},\boldsymbol{\delta}) \in [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})\}\,,$$

while predicates $\Gamma;\Delta;(P_j:\mathbf{prop}(\boldsymbol{\sigma}_j;\underline{\boldsymbol{\tau}}_j))_j \vdash \pi:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$ are interpreted as follows:

$$\begin{aligned}
[\![P_i]\!] &= \mathrm{pr}_i \circ \mathrm{pr}_3\,,\\
[\![(\boldsymbol{x}{:}\boldsymbol{\sigma};\boldsymbol{y}{:}\underline{\boldsymbol{\tau}}).\,\varphi]\!] &= \langle\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \mapsto \{\langle\boldsymbol{\gamma}',\boldsymbol{\delta}'\rangle \mid \langle\boldsymbol{\gamma},\boldsymbol{\gamma}',\boldsymbol{\delta},\boldsymbol{\delta}',\boldsymbol{U}\rangle \in [\![\varphi]\!]\}\,,\\
[\![\nu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi]\!] &= \langle\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \mapsto \begin{cases} \nu_{[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-)} & \text{if } [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-) \text{ is monotone}\,,\\ \emptyset & \text{otherwise}\,,\end{cases}\\
[\![\mu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi]\!] &= \langle\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \mapsto \begin{cases} \mu_{[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-)} & \text{if } [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-) \text{ is monotone}\,,\\ [\![\boldsymbol{\sigma}]\!]\times U[\![\underline{\boldsymbol{\tau}}]\!] & \text{otherwise}\,.\end{cases}
\end{aligned}$$

Lemma 4.25 *If a predicate variable P is positive (negative) in*

$$\Gamma;\Delta;\Pi,P:\mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}) \vdash \pi{:}\mathbf{prop}(\boldsymbol{\sigma}';\underline{\boldsymbol{\tau}}')\,,$$

then for any $\langle\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\Gamma]\!]\times[\![\Delta]\!]\times[\![\Pi]\!]$, the map

$$[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-)\colon \mathcal{P}([\![\boldsymbol{\sigma}]\!]\times U[\![\underline{\boldsymbol{\tau}}]\!]) \to \mathcal{P}([\![\boldsymbol{\sigma}']\!]\times U[\![\underline{\boldsymbol{\tau}}']\!])$$

is monotone (anti-monotone).

If in addition, $\boldsymbol{\sigma} = \boldsymbol{\sigma}'$, $\underline{\boldsymbol{\tau}} = \underline{\boldsymbol{\tau}}'$, and P is positive in π, then

$$[\![\nu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}) \qquad \textit{and} \qquad [\![\mu P:(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$$

are the greatest and least fixed points of $[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U},-)$.

Proof The proof proceeds by a straightforward induction on the structure of the predicate in question, employing Theorem 4.24 to show that the condition of monotonicity in the definition of interpretation of predicate fixed points is indeed satisfied. □

Definition 4.26 A judgement $\Gamma;\Delta;\Pi \mid \varphi_1,\ldots,\varphi_n \vdash \varphi$ is *sound* if

$$\bigcap_{i=1}^{n} [\![\varphi_i]\!] \subseteq [\![\varphi]\!]\,.$$

Chapter 5

Reasoning rules of the logic

The logic for algebraic effects is the smallest collection of judgements closed under the reasoning rules listed in this chapter. We write $\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi$ when the judgement $\Gamma;\Delta;\Pi \mid \Psi \vdash \varphi$ is in the logic for algebraic effects.

For clarity, we split the reasoning rules into the following groups:

- reasoning rules for propositions and predicates,
- reasoning rules describing equality,
- equations for call-by-push-value constructs,
- two algebraic principles describing universality of the free model.

In all the rules, we omit the hypotheses that ensure that judgements are well-typed.

5.1 Propositions and predicates

First, we give the standard reasoning rules for a classical first-order logic with fixed points:

- hypothesis:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi,\varphi \vdash_L \varphi}\,,$$

- substitution of value terms:

$$\frac{\boldsymbol{x}:\boldsymbol{\sigma};\Delta;\Pi \mid \Psi \vdash_L \varphi}{\Gamma;\Delta;\Pi \mid \Psi[\boldsymbol{v}/\boldsymbol{x}] \vdash_L \varphi[\boldsymbol{v}/\boldsymbol{x}]} \quad (\Gamma;\Delta;\Pi \vdash \boldsymbol{v}:\boldsymbol{\sigma})\,,$$

- substitution of computation terms:

$$\frac{\Gamma;\boldsymbol{y}:\underline{\boldsymbol{\tau}};\Pi \mid \Psi \vdash_L \varphi}{\Gamma;\Delta;\Pi \mid \Psi[\boldsymbol{t}/\boldsymbol{y}] \vdash_L \varphi[\boldsymbol{t}/\boldsymbol{y}]} \quad (\Gamma;\Delta;\Pi \vdash \boldsymbol{t}:\underline{\boldsymbol{\tau}}),$$

- substitution of predicates:

$$\frac{\Gamma;\Delta;(P_i:\mathbf{prop}(\boldsymbol{\sigma}_i;\underline{\boldsymbol{\tau}}_i))_i \mid \Psi \vdash_L \varphi}{\Gamma;\Delta;\Pi \mid \Psi[\boldsymbol{\pi}/\boldsymbol{P}] \vdash_L \varphi[\boldsymbol{\pi}/\boldsymbol{P}]} \quad (\Gamma;\Delta;\Pi \vdash \pi_i:\mathbf{prop}(\boldsymbol{\sigma}_i;\underline{\boldsymbol{\tau}}_i) \quad (1 \leqslant i \leqslant n)),$$

- truth and falsehood:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \top}, \qquad \frac{}{\Gamma;\Delta;\Pi \mid \Psi, \bot \vdash_L \varphi},$$

- conjunction introduction and elimination:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_2}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \wedge \varphi_2}, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \wedge \varphi_2}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1},$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \wedge \varphi_2}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_2},$$

- disjunction introduction and elimination:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \vee \varphi_2}, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_2}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \vee \varphi_2},$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi, \varphi_1 \vdash_L \varphi \qquad \Gamma;\Delta;\Pi \mid \Psi, \varphi_2 \vdash_L \varphi}{\Gamma;\Delta;\Pi \mid \Psi, \varphi_1 \vee \varphi_2 \vdash_L \varphi},$$

- implication introduction and elimination:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi, \varphi_1 \vdash_L \varphi_2}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \Rightarrow \varphi_2}, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1 \Rightarrow \varphi_2 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_1}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi_2},$$

- introduction and elimination of universal quantification over values:

$$\frac{\Gamma, x:\sigma;\, \Delta;\Pi \mid \Psi \vdash_L \varphi}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \forall x:\sigma.\, \varphi}, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \forall x:\sigma.\, \varphi}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi[v/x]} \quad (\Gamma;\Delta \vdash v:\sigma),$$

- introduction and elimination of universal quantification over computations:

$$\frac{\Gamma;\Delta,y{:}\underline{\tau};\,\Pi\mid\Psi\vdash_L\varphi}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\forall y{:}\underline{\tau}.\,\varphi}\,,\qquad \frac{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\forall y{:}\underline{\tau}.\,\varphi}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi[t/y]}\quad(\Gamma;\Delta\vdash y{:}\underline{\tau})\,,$$

- introduction and elimination of existential quantification over values:

$$\frac{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi[v/x]}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\exists x{:}\sigma.\,\varphi}\quad(\Gamma;\Delta\vdash v{:}\sigma)\,,$$

$$\frac{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\exists x{:}\sigma.\,\varphi'\qquad\Gamma,x{:}\sigma;\,\Delta;\Pi\mid\Psi,\varphi'\vdash_L\varphi}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi}\,,$$

- introduction and elimination of existential quantification over computations:

$$\frac{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi[t/y]}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\exists y{:}\underline{\tau}.\,\varphi}\quad(\Gamma;\Delta\vdash t{:}\underline{\tau})\,,$$

$$\frac{\Gamma;\Delta,y{:}\underline{\tau};\,\Pi\mid\Psi,\varphi'\vdash_L\varphi\qquad\Gamma;\Delta;\Pi\mid\Psi\vdash_L\exists y{:}\underline{\tau}.\,\varphi'}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi}\,,$$

- reductio ad absurdum:

$$\frac{\Gamma;\Delta;\Pi\mid\Psi,\neg\varphi\vdash_L\bot}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L\varphi}\,.$$

- application of abstracted propositions:

$$\frac{}{\Gamma;\Delta;\Pi\mid\Psi\vdash_L((\boldsymbol{x}{:}\boldsymbol{\sigma};\boldsymbol{y}{:}\underline{\boldsymbol{\tau}}).\,\varphi)(\boldsymbol{v};\boldsymbol{t})\Leftrightarrow\varphi[\boldsymbol{v}/\boldsymbol{x},\boldsymbol{t}/\boldsymbol{y}]}\,,$$

  where $\varphi\Leftrightarrow\varphi' =_{\mathrm{def}} (\varphi\Rightarrow\varphi')\wedge(\varphi'\Rightarrow\varphi)$,

- greatest fixed point of a predicate:

$$\frac{}{\Gamma;\Delta;\Pi\mid(\nu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi)(\boldsymbol{v};\boldsymbol{t})\vdash_L\pi[\nu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi/P](\boldsymbol{v};\boldsymbol{t})}\,,$$

$$\frac{\Gamma,\boldsymbol{x}{:}\boldsymbol{\sigma};\,\Delta,\boldsymbol{y}{:}\underline{\boldsymbol{\tau}};\,\Pi\mid\pi'(\boldsymbol{x};\boldsymbol{y})\vdash_L\pi[\pi'/P](\boldsymbol{x};\boldsymbol{y})}{\Gamma,\boldsymbol{x}{:}\boldsymbol{\sigma};\,\Delta,\boldsymbol{y}{:}\underline{\boldsymbol{\tau}};\,\Pi\mid\pi'(\boldsymbol{x};\boldsymbol{y})\vdash_L(\nu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\,\pi)(\boldsymbol{x};\boldsymbol{y})}\,,$$

- least fixed point of a predicate:

$$\frac{}{\Gamma;\Delta;\Pi \mid \pi[\mu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi/P](\boldsymbol{v};\boldsymbol{t}) \vdash_L (\mu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi)(\boldsymbol{v};\boldsymbol{t})}\,,$$

$$\frac{\Gamma,\boldsymbol{x}{:}\boldsymbol{\sigma};\ \Delta,\boldsymbol{y}{:}\underline{\boldsymbol{\tau}};\ \Pi \mid \pi[\pi'/P](\boldsymbol{x};\boldsymbol{y}) \vdash_L \pi'(\boldsymbol{x};\boldsymbol{y})}{\Gamma,\boldsymbol{x}{:}\boldsymbol{\sigma};\ \Delta,\boldsymbol{y}{:}\underline{\boldsymbol{\tau}};\ \Pi \mid (\mu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi)(\boldsymbol{x};\boldsymbol{y}) \vdash_L \pi'(\boldsymbol{x};\boldsymbol{y})}\,.$$

5.2 Equality

Then, we state the usual structural properties of equality:

- reflexivity, symmetry, and transitivity of value equality:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_\sigma v}{} \quad (\Gamma;\Delta;\Pi \mid \Psi \vdash v{:}\sigma), \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_\sigma v'}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v' =_\sigma v}\,,$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_\sigma v' \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L v' =_\sigma v''}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_\sigma v''}\,,$$

- reflexivity, symmetry, and transitivity of computation equality:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t} \quad (\Gamma;\Delta \vdash t{:}\underline{\tau})\,, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t'}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t' =_{\underline{\tau}} t}\,,$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t' \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L t' =_{\underline{\tau}} t''}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t''}\,,$$

- replacement for values:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_\sigma v' \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi[v/x]}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi[v'/x]}\,,$$

- replacement for computations:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t' \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi[t/y]}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \varphi[t'/y]}\,.$$

Since the context remains fixed in the replacement rules, we have to give an extensionality rule for each construct that employs variable binding:

- extensionality of operations:

$$\frac{\Gamma, \boldsymbol{x}_i : \boldsymbol{\alpha}_i;\ \Delta; \Pi \mid \Psi \vdash_L t_i =_{\underline{\tau}} t'_i \quad (1 \leqslant i \leqslant n)}{\Gamma; \Delta; \Pi \mid \Psi \vdash_L \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t'_i)_i} \quad (\mathsf{op}\colon \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n)\,,$$

- extensionality of sequencing:

$$\frac{\Gamma, x\colon\sigma;\ \Delta; \Pi \mid \Psi \vdash_L t' =_{\underline{\tau}} t''}{\Gamma; \Delta; \Pi \mid \Psi \vdash_L t \,\mathsf{to}\, x\colon\sigma.\, t' =_{\underline{\tau}} t \,\mathsf{to}\, x\colon\sigma.\, t''}\,,$$

- extensionality of pattern matching:

$$\frac{\Gamma, x_1\colon\sigma_1;\ \Delta; \Pi \mid \Psi \vdash_L t_1 =_{\underline{\tau}} t'_1 \qquad \Gamma, x_2\colon\sigma_2;\ \Delta; \Pi \mid \Psi \vdash_L t_2 =_{\underline{\tau}} t'_2}{\Gamma; \Delta; \Pi \mid \Psi \vdash_L \mathsf{match}\, v \,\mathsf{with}\, \mathsf{inl}\, x_1.\, t_1, \mathsf{inr}\, x_2.\, t_2 =_{\underline{\tau}} \mathsf{match}\, v \,\mathsf{with}\, \mathsf{inl}\, x_1.\, t'_1, \mathsf{inr}\, x_2.\, t'_2}\,,$$

- extensionality of abstraction:

$$\frac{\Gamma, x\colon\sigma;\ \Delta; \Pi \mid \Psi \vdash_L t =_{\underline{\tau}} t'}{\Gamma; \Delta; \Pi \mid \Psi \vdash_L \lambda x\colon\sigma.\, t =_{\sigma \to \underline{\tau}} \lambda x\colon\sigma.\, t'}\,.$$

As in the a-calculus, we inherit propositions from the base theory $\mathcal{T}_{\mathrm{base}}$ and instantiate equations from the effect theory $\mathcal{T}_{\mathrm{eff}}$:

- inheritance from the base theory:

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\mathrm{base}}} \varphi}{\Gamma; \Delta; \Pi \mid \Psi, \Psi' \vdash_L \varphi}\,,$$

- inheritance from the effect theory:

$$\frac{\Gamma; (z_i \colon (\boldsymbol{\alpha}_i))_i \vdash_{\mathcal{T}_{\mathrm{eff}}} e = e' \ (\varphi)}{\Gamma; \Delta; \Pi \mid \Psi, \varphi \vdash_L e[(\boldsymbol{x}_i).\, t_i/z_i]_i =_{\underline{\tau}} e'[(\boldsymbol{x}_i).\, t_i/z_i]_i}\,, \quad (\Gamma, \boldsymbol{x}_i : \boldsymbol{\alpha}_i;\ \Delta \vdash t_i : \underline{\tau})_i$$

where the *instantiation* $\Gamma; \Delta \vdash e[(\boldsymbol{x}_i).\, t_i/z_i]_i : \underline{\tau}$ of an effect term $\Gamma; Z \vdash e$ by computation terms $\Gamma, \boldsymbol{x}_i : \boldsymbol{\alpha}_i;\ \Delta \vdash t_i : \underline{\tau}$, for each $z_i : (\boldsymbol{\alpha}_i) \in Z$, is defined structurally by

$$z_j(\boldsymbol{v})[(\boldsymbol{x}_i).\, t_i/z_i]_i = t_j[\boldsymbol{v}/\boldsymbol{x}_j]\,,$$

$$\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}'_j.\, t'_j)_j[(\boldsymbol{x}_i).\, t_i/z_i]_i = \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}'_j.\, t'_j[(\boldsymbol{x}_i).\, t_i/z_i]_i)_j\,.$$

Note that in the a-calculus, we combined the rule for inheritance from the base theory with the congruence rule for the return construct, as the base theory consisted of exactly all equations between base terms. Now, there are more value terms than just the base terms, so the two rules have to be split.

5.3 Call-by-push-value constructs

All the value and computation terms, except for conditionals and sequencing, are described by the standard call-by-push-value equations [Lev06a]:

- η-equivalence for the value unit:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v =_{\mathbf{1}} \star}\,,$$

- β-equivalences for value products:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{fst}\langle v_1, v_2\rangle =_{\sigma_1} v_1}\,, \qquad \frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{snd}\langle v_1, v_2\rangle =_{\sigma_2} v_2}\,,$$

- η-equivalence for value products:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \langle \mathsf{fst}\, v, \mathsf{snd}\, v\rangle =_{\sigma_1 \times \sigma_2} v}\,,$$

- η-equivalence for the computation unit:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\mathbf{1}}} \underline{\star}}\,,$$

- β-equivalences for computation products:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{fst}\langle t_1, t_2\rangle =_{\underline{\tau}_1} t_1}\,, \qquad \frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{snd}\langle t_1, t_2\rangle =_{\underline{\tau}_2} t_2}\,,$$

- η-equivalence for computation products:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \langle \mathsf{fst}\, t, \mathsf{snd}\, t\rangle =_{\underline{\tau}_1 \times \underline{\tau}_2} t}\,,$$

- β-equivalence for thunks:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{force}(\mathsf{thunk}\, t) =_{\underline{\tau}} t}\,,$$

- η-equivalence for thunks:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{thunk}(\mathsf{force}\, v) =_{U\underline{\tau}} v}\,,$$

- β-equivalence for functions:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L (\lambda x{:}\sigma.\ t)v =_{\underline{\tau}} t[v/x]},$$

- η-equivalence for functions:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \lambda x{:}\sigma.\ tx =_{\sigma\to\underline{\tau}} t},$$

- emptiness of the zero type:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \forall x{:}\mathbf{0}.\ \bot},$$

- β-equivalences for value sums:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{match}\,\mathsf{inl}\,v\,\mathsf{with}\,\mathsf{inl}\,x_1.\ t_1, \mathsf{inr}\,x_2.\ t_2 =_{\underline{\tau}} t_1[v/x_1]},$$

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{match}\,\mathsf{inr}\,v\,\mathsf{with}\,\mathsf{inl}\,x_1.\ t_1, \mathsf{inr}\,x_2.\ t_2 =_{\underline{\tau}} t_2[v/x_2]},$$

- cases of value sums:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L (\exists x_1{:}\sigma_1.\ v =_{\sigma_1+\sigma_2} \mathsf{inl}\,x_1) \vee (\exists x_2{:}\sigma_2.\ v =_{\sigma_1+\sigma_2} \mathsf{inr}\,x_2)}$$

- β-equivalences for conditionals:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{rel}(\boldsymbol{v})}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{if}\,\mathsf{rel}(\boldsymbol{v})\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 =_{\underline{\tau}} t_1},$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \neg\mathsf{rel}(\boldsymbol{v})}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{if}\,\mathsf{rel}(\boldsymbol{v})\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 =_{\underline{\tau}} t_2}.$$

Note that η-equivalence for conditionals, which states

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{if}\,\mathsf{rel}(\boldsymbol{v})\,\mathsf{then}\,t\,\mathsf{else}\,t =_{\underline{\tau}} t},$$

is provable using reductio ad absurdum.

Then, we specify the behaviour of operations on computation types:

- operations on product types:

$$\frac{(\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n)}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \langle \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_{1i})_i, \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_{2i})_i\rangle =_{\underline{\tau}_1 \times \underline{\tau}_2} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, \langle t_{1i}, t_{2i}\rangle)_i},$$

- operations on function types:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \lambda x{:}\sigma.\, \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i =_{\sigma \to \underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, \lambda x{:}\sigma.\, t_i)_i}\ (\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n)\,.$$

Note that the trivial behaviour of operations on the unit computation type $\underline{\mathbf{1}}$ is already determined by η-equivalence $\Gamma;\Delta;\Pi \mid \Psi \vdash_L t =_{\underline{\mathbf{1}}} \underline{\star}$.

Finally, we describe the behaviour of sequencing with equational schemas, which are a generalisation of the ones in the a-calculus:

- β-equivalence of sequencing:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{return}\, v\, \mathsf{to}\, x{:}\sigma.\, t =_{\underline{\tau}} t[v/x]},$$

- algebraicity of operations:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i\, \mathsf{to}\, x{:}\sigma.\, t =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i\, \mathsf{to}\, x{:}\sigma.\, t)_i}\ (\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n)\,.$$

We omit η-equivalence for sequencing from the axioms as it is derivable (see Proposition 6.1).

5.4 Algebraic principles

In contrast to the a-calculus, the logic is complex and its terms do not have a canonical form. Hence, instead of structural induction on canonical forms, we employ a *principle of computational induction*, which is motivated by Proposition 3.19 and states that every computation term of type $F\sigma$ is either a returned value, or built from other computation terms using operations.

Then, for a predicate $\Gamma;\Delta;\Pi \vdash \pi{:}\mathbf{prop}(F\sigma)$, we have:

- principle of computational induction:

$$\frac{\Gamma, x{:}\sigma;\Delta;\Pi \vdash_L \pi(\mathsf{return}\, x) \qquad \mathfrak{C}_{\mathsf{op}} \quad (\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathsf{eff}})}{\Gamma;\Delta;\Pi \vdash_L \pi(t)}\ (\Gamma;\Delta \vdash t{:}F\sigma)\,,$$

where the induction case $\mathfrak{C}_{\mathsf{op}}$ is an abbreviation for

$$\Gamma, \boldsymbol{x}{:}\boldsymbol{\beta};\ \Delta, (y_i : \boldsymbol{\alpha}_i \to F\sigma)_i;\ \Pi \mid (\forall \boldsymbol{x}_i : \boldsymbol{\alpha}_i.\ \pi(y_i \boldsymbol{x}_i))_i \vdash_L \pi(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i \boldsymbol{x}_i)_i)\,.$$

The induction hypotheses $\forall \boldsymbol{x}_i : \boldsymbol{\alpha}_i.\ \pi(y_i \boldsymbol{x}_i)$ state that all the continuations y_i of $\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i \boldsymbol{x}_i)_i$ satisfy π for all outcomes $\boldsymbol{x}_i$ of the triggered effect.

Example 5.1 For exceptions, the effect signature Σ_{eff} contains a single nullary operation $\mathsf{raise}{:}\mathbf{exc}$, thus computational induction is of the form

$$\frac{\Gamma, x{:}\sigma; \Delta; \Pi \vdash_L \pi(\mathsf{return}\, x) \qquad \Gamma, \mathsf{exc}{:}\mathbf{exc}; \Delta; \Pi \mid \cdot \vdash_L \pi(\mathsf{raise}_{\mathsf{exc}})}{\Gamma; \Delta; \Pi \vdash_L \pi(t)}$$

and states that if π holds for all returned values and for all raised exceptions, then it holds for all computations.

Example 5.2 For nondeterminism, the effect signature Σ_{eff} contains a single binary operation $\mathsf{or}{:}2$ with the corresponding induction case $\mathfrak{C}_{\mathsf{or}}$

$$\Gamma; \Delta, y_1{:}F\sigma, y_2{:}F\sigma; \Pi \mid \pi(y_1), \pi(y_2) \vdash_L \pi(\mathsf{or}(y_1, y_2))\,.$$

This states that if π holds for both y_1 and y_2, it holds for $\mathsf{or}(y_1, y_2)$ as well.

Example 5.3 For global state, the induction case $\mathfrak{C}_{\mathsf{lookup}}$ for $\mathsf{lookup}{:}\mathbf{loc}; \mathbf{dat}$ is of the form

$$\Gamma, \ell{:}\mathbf{loc}; \Delta, y{:}\mathbf{dat} \to F\sigma; \Pi \mid \forall d{:}\mathbf{dat}.\ \pi(y\, d) \vdash_L \pi(\mathsf{lookup}_\ell(d.\ y\, d))$$

and states that if π holds for $y\, d$ for any outcome d, then it holds for $\mathsf{lookup}_\ell(d.\ y\, d)$ for all locations ℓ.

The case $\mathfrak{C}_{\mathsf{update}}$ for $\mathsf{update}{:}\mathbf{loc}, \mathbf{dat}; 1$ is of the form

$$\Gamma, \ell{:}\mathbf{loc}, d{:}\mathbf{dat}; \Delta, y{:}F\sigma; \Pi \mid \pi(y) \vdash_L \pi(\mathsf{update}_{\ell,d}(y))$$

and states that if π holds for y, it holds for $\mathsf{update}_{\ell,d}(y)$ for all locations ℓ and data d.

Note that the finiteness of the signature Σ_{eff} is crucial in the formulation of the induction principle, as an infinite signature would lead to an infinite number of hypotheses $\mathfrak{C}_{\mathsf{op}}$ in the induction rule.

We next present a *free model principle* that expresses the universal property of the free model: for any set A, any model M, and any map $f : A \to UM$, there exists a unique homomorphism $\hat{f}\colon FA \to M$ such that $U\hat{f} \circ \eta_A = f$. To state this in our logic, we first describe models and homomorphisms.

To express the existence of a homomorphism in the logic, we employ the existential quantifier over the function type. But as function types have computation types for codomains, we have to limit the models to ones with carriers of the form $U\underline{\tau}$ rather than arbitrary sets. Although this is cumbersome, Proposition 6.6 shows that if we give a model on $F\sigma$, where operations map returned values into returned values, and if we have a map into $F\sigma$, with the image restricted to returned values, then the image of the induced homomorphism is also restricted to returned values. This enables us to simulate a model on some sets, for example the one used in Lemma 6.18. It is unlikely, however, that all models, in particular the ones with infinitary operations, can be simulated in this way.

Fix contexts Γ and Δ, which will serve as parameters. Recall that a model is given by its carrier UM together with appropriate maps

$$\mathsf{op}_M\colon [\![\boldsymbol{\beta}]\!] \times \prod_i UM^{[\![\boldsymbol{\alpha}_i]\!]} \to UM$$

for each $\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$. Such a map is described by a computation term

$$\Gamma,\boldsymbol{x}\colon\boldsymbol{\beta};\ \Delta,(y_i\colon\boldsymbol{\alpha}_i \to \underline{\tau})_i \vdash t_{\mathsf{op}}\colon\underline{\tau}\,.$$

Next, we state when a given family of terms $\{t_{\mathsf{op}}\}_{\mathsf{op}}$ satisfies all the equations $\Gamma';Z' \vdash e = e'\ (\varphi)$ of the effect theory $\mathcal{T}_{\mathrm{eff}}$. Take any effect term $\Gamma';Z' \vdash e$ and a context Δ' consisting of computation variables $y'_j\colon\boldsymbol{\alpha}_j \to \underline{\tau}$ for each effect variable $z'_j\colon(\boldsymbol{\alpha}_j) \in Z'$. Then, we define a computation term $\Gamma,\Gamma';\Delta,\Delta' \vdash e[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}}\colon\underline{\tau}$ by

$$\begin{aligned} z'_j(\boldsymbol{v})[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}} &= y'_j\boldsymbol{v}\,, \\ \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\,e_i)_i[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}} &= t_{\mathsf{op}}[\boldsymbol{v}/\boldsymbol{x},(\lambda\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i.\,e_i[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}}/y_i)_i]\,. \end{aligned}$$

Then, we define the proposition $\{t_{\mathsf{op}}\colon\underline{\tau}\}_{\mathsf{op}\in\Sigma_{\mathrm{eff}}}$ models $\mathcal{T}_{\mathrm{eff}}$ to be the conjunction

$$\bigwedge_{\Gamma';Z'\vdash_{\mathcal{T}_{\mathrm{eff}}}e=e'\,(\varphi)} \varphi \Rightarrow \forall\boldsymbol{y}'.\, e[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}} =_{\underline{\tau}} e'[t_{\mathsf{op}}/\mathsf{op}]_{\mathsf{op}}\,.$$

For $y\colon\sigma \to \underline{\tau}$ and $\hat{y}\colon UF\sigma \to \underline{\tau}$, we define the proposition $\hat{y}$ $\mathsf{extends}$ y to be

$$\forall x\colon\sigma.\ \hat{y}(\mathsf{thunk}\,\mathsf{return}\,x) = yx\,,$$

and the proposition $\hat{y}$ $\mathsf{homomorphism}$ to be the conjunction

$$\bigwedge_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n\in\Sigma_{\mathrm{eff}}} \forall\boldsymbol{x}\colon\boldsymbol{\beta}.\ \forall(y'_i\colon\boldsymbol{\alpha}_1 \to \underline{\tau})_i.\ \hat{y}(\mathsf{thunk}\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}'_i.\,y'_i\boldsymbol{x}'_i)_i) =_{\underline{\tau}} t_{\mathsf{op}}[(\lambda\boldsymbol{x}'_i\colon\boldsymbol{\alpha}_i.\,\hat{y}(\mathsf{thunk}\,y'_i\boldsymbol{x}'_i))/y_i]_i\,.$$

Then, we have

- free model principle:

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \{t_{\mathsf{op}} : \underline{\tau}\}_{\mathsf{op} \in \Sigma_{\mathrm{eff}}} \ \mathsf{models}\ \mathcal{T}_{\mathrm{eff}}}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L \forall y{:}\sigma \to \underline{\tau}.\ \exists \hat{y}{:}UF\sigma \to \underline{\tau}.\ \hat{y}\ \mathsf{extends}\ y \wedge \hat{y}\ \mathsf{homomorphism}} \ .$$

As in the case of principle of computational induction, the finiteness of both the signature Σ_{eff} and the effect theory $\mathcal{T}_{\mathrm{eff}}$ is crucial in the formulation of the free model principle.

One might expect the statement of the uniqueness of the induced homomorphism. However, as Proposition 6.5 shows, this is a consequence of the principle of computational induction.

5.5 Soundness

Proposition 5.4 *If* $\Gamma;\Delta;\Pi \mid \varphi_1,\dots,\varphi_n \vdash_L \varphi$ *holds, then*

$$\bigcap_{i=1}^{n} [\![\varphi_i]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}) \subseteq [\![\varphi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$$

for all $\langle \boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$.

Proof We proceed by an induction on the derivation of $\Gamma;\Delta;\Pi \mid \varphi_1,\dots,\varphi_n \vdash_L \varphi$. We omit the standard cases where we have used a reasoning rule for equality, or any of the standard rules for propositions and predicates. Next, the rules for the greatest fixed point are sound since $[\![\nu P : (\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi]\!]$ is defined to be the greatest post-fixed point, and hence the greatest fixed point of the operator on $\mathcal{P}([\![\boldsymbol{\sigma}]\!] \times U[\![\underline{\boldsymbol{\tau}}]\!])$, induced by $[\![\pi]\!]$. A similar argument holds for the least fixed point.

Then, it is straightforward to show the soundness of most of the equations for call-by-push-value constructs [Lev06a]. To show the soundness of the equation that describes the behaviour of an operation $\mathsf{op} : \boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n$ on product type $\underline{\tau}_1 \times \underline{\tau}_2$, we take an arbitrary $\langle \boldsymbol{\gamma},\boldsymbol{\delta}\rangle \in [\![\Gamma]\!] \times [\![\Delta]\!]$. Then, we have

$$\begin{aligned}
&[\![\langle \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_{1i})_i, \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_{2i})_i\rangle]\!](\boldsymbol{\gamma},\boldsymbol{\delta}) && \\
&= \langle \mathsf{op}_{[\![\underline{\tau}_1]\!]}([\![\boldsymbol{v}]\!](\boldsymbol{\gamma},\boldsymbol{\delta}), \widehat{[\![t_{1i}]\!]})_i, \mathsf{op}_{[\![\underline{\tau}_2]\!]}([\![\boldsymbol{v}]\!](\boldsymbol{\gamma},\boldsymbol{\delta}), \widehat{[\![t_{2i}]\!]})_i\rangle && \text{(by definition)} \\
&= \mathsf{op}_{[\![\underline{\tau}_1]\!] \times [\![\underline{\tau}_2]\!]}([\![\boldsymbol{v}]\!](\boldsymbol{\gamma},\boldsymbol{\delta}), \langle \widehat{[\![t_{1i}]\!]}, \widehat{[\![t_{2i}]\!]}\rangle)_i && \text{(by product model structure)} \\
&= [\![\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, \langle t_{1i}, t_{2i}\rangle)_i]\!](\boldsymbol{\gamma},\boldsymbol{\delta}) && \text{(by definition)}\ .
\end{aligned}$$

The case for the definition of operations on $\sigma \to \underline{\tau}$ proceeds similarly.

The case of β-equivalence for sequencing is the same as in the a-calculus, while the proof of algebraicity of operations proceeds as in the a-calculus, except that we employ Proposition 4.15 instead of Lemma 2.33 to show that operations commute with liftings.

To show the soundness of the induction principle for a predicate

$$\Gamma;\Delta;\Pi \vdash \pi{:}\mathbf{prop}(F\sigma)\,,$$

we fix $\langle \boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$. Then, the base case of the induction principle shows that $\eta\colon [\![\sigma]\!] \to UF[\![\sigma]\!]$ is a composition of $i\colon [\![\sigma]\!] \to [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$ and the inclusion $j\colon [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta}) \to UF[\![\sigma]\!]$. The step case shows that $[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$ has a model structure, inherited from $UF[\![\sigma]\!]$.

$$\begin{tikzcd}
UF[\![\sigma]\!] \arrow[r, dotted, "\mathrm{id}_{UF[\![\sigma]\!]}"] \arrow[dr, dotted, "\bar{\imath}"] & UF[\![\sigma]\!] \\
{[\![\sigma]\!]} \arrow[u, "\eta"] \arrow[r, "i"'] & {[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})} \arrow[u, "j"']
\end{tikzcd}$$

Because $F[\![\sigma]\!]$ is the free model on $[\![\sigma]\!]$, there exists a unique

$$\bar{\imath}\colon UF[\![\sigma]\!] \to [\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$$

such that $i = \bar{\imath}\circ\eta$. Since we have $j\circ\bar{\imath}\circ\eta = j\circ i = \eta$, we get $j\circ\bar{\imath} = \mathrm{id}_{UF[\![\sigma]\!]}$ from the universality of the model. Since j is a retraction and a monomorphism, it is an isomorphism, and $[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}) = UF[\![\sigma]\!]$.

Finally, the free model principle is, following its derivation, a direct transliteration of the universality of free models (without the guarantee of the uniqueness), hence the proof of its soundness is straightforward. □

Proposition 5.5 *The equational theory, induced by the effect theory $\mathcal{T}_{\mathrm{eff}}$, is non-trivial if and only if the* consistency *proposition*

$$\forall x_1,x_2{:}\sigma.\ \mathsf{return}\,x_1 =_{F\sigma} \mathsf{return}\,x_2 \Rightarrow x_1 =_\sigma x_2$$

is sound for all value types σ.

Proof If the induced equational theory is non-trivial, the construction of the free model implies that the unit map $\eta_{[\![\sigma]\!]}\colon [\![\sigma]\!] \to UF[\![\sigma]\!]$ is a monomorphism, hence

$$[\![\mathsf{return}\,x_1]\!] = \eta_{[\![\sigma]\!]}\circ[\![x_1]\!] = \eta_{[\![\sigma]\!]}\circ[\![x_2]\!] = [\![\mathsf{return}\,x_2]\!]$$

implies $[\![x_1]\!] = [\![x_2]\!]$. On the other hand, if the effect theory is trivial, the free model collapses to a single point, and if $[\![\sigma]\!]$ has more than one point (if $\sigma = \mathbf{1} + \mathbf{1}$, for example), the unit map fails to be injective and the consistency proposition is not sound. □

Chapter 6

Development and applications of the logic

After describing the logic, we collect a few examples of its use. We first go through the properties of sequencing we proved in the α-calculus, restate them in a non-schematic way and prove them using the principle of computational induction. Next, we show some additional properties of the free model principle and define local and global modalities. Finally, we give a conservative translation of Moggi's computational λ-calculus [Mog89], Hennessy-Milner logic [HM85], and Pitts's evaluation logic [Pit91] into our logic.

6.1 Sequencing

As in the α-calculus, η-equivalence and associativity of sequencing are derivable. In addition, those properties can be stated in a stronger, non-schematic way.

Proposition 6.1 *The following holds:*

$$y{:}F\sigma \vdash_L y\,\mathsf{to}\,x.\ \mathsf{return}\,x =_{F\sigma} y\,.$$

Proof We proceed by the principle of computational induction. Take

$$\pi =_{\mathrm{def}} (y{:}F\sigma).\ y\,\mathsf{to}\,x.\ \mathsf{return}\,x =_{F\sigma} y\,.$$

Then, the base case $x'{:}\sigma \vdash_L \pi(\mathsf{return}\,x')$ is equivalent to

$$x'{:}\sigma \vdash_L \mathsf{return}\,x'\,\mathsf{to}\,x.\ \mathsf{return}\,x =_{F\sigma} \mathsf{return}\,x'\,,$$

which is derivable by the β-equivalence of the sequencing.

For the step case for an operation $\mathsf{op}\!:\!\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$, we have to show the validity of the judgement

$$\boldsymbol{x}\!:\!\boldsymbol{\beta};(y_i\!:\!\boldsymbol{\alpha}_i \to F\sigma)_i \mid \varphi^1_{\mathsf{op}},\ldots,\varphi^n_{\mathsf{op}} \vdash_L \pi(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i)_i)\,,$$

where

$$\varphi^i_{\mathsf{op}} =_{\mathsf{def}} \forall \boldsymbol{x}_i\!:\!\boldsymbol{\alpha}_i.\, y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\,\mathsf{return}\,x =_{F\sigma} y_i\boldsymbol{x}_i\,.$$

Then, we get

$$\begin{aligned}
&\Gamma;\Delta \vdash_L \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i)_i\,\mathsf{to}\,x.\,\mathsf{return}\,x \\
&\quad =_{F\sigma} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\,\mathsf{return}\,x) && \text{(by algebraicity of operations)} \\
&\quad =_{F\sigma} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i) && \text{(by the induction hypotheses)}\,.
\end{aligned}$$

Hence, by the induction principle, we get $y\!:\!F\sigma \vdash_L \pi(y)$. □

We can see that the proof resembles the one of Proposition 3.20, except that it uses the principle of computational induction rather than the induction over the structure of canonical forms. Similarly, we can prove a non-schematic version of Proposition 3.21.

Proposition 6.2 *The following holds:*

$$\begin{aligned}
&y_1\!:\!F\sigma_1, y_2\!:\!\sigma_1 \to F\sigma_2, y\!:\!\sigma_2 \to \underline{\tau} \vdash_L \\
&\quad y_1\,\mathsf{to}\,x_1.\,(y_2x_1\,\mathsf{to}\,x_2.\,yx_2) = (y_1\,\mathsf{to}\,x_1.\,y_2x_1)\,\mathsf{to}\,x_2.\,yx_2\,.
\end{aligned}$$

In the a-calculus, the commutativity of sequencing is a consequence of the commutativity of the effect theory. But in the logic, the effect theory is just a set of equations. For that reason, it is possible that the induced equational theory is commutative even though the effect theory does not contain all the equations describing commutativity between operations. But since we cannot observe this in the logic, we simplify things and restate the commutativity condition at any given computation type. Then, the proof proceeds as in Proposition 3.23. In the case that all the necessary commutativity equations are present in the effect theory, this assumptions follow immediately by inheritance from the effect theory.

Proposition 6.3 *If*

$$\begin{aligned}
&\boldsymbol{x}\!:\!\boldsymbol{\beta},\boldsymbol{x}'\!:\!\boldsymbol{\beta}';(y_{ii'}\!:\!\boldsymbol{\alpha}_i \to \boldsymbol{\alpha}'_{i'} \to \underline{\tau})_{1\le i\le n, 1\le i'\le n'} \vdash_L \\
&\quad \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,\mathsf{op}'_{\boldsymbol{x}'}(\boldsymbol{x}'_{i'}.\,y_{ii'}\boldsymbol{x}_i\boldsymbol{x}'_{i'})_{i'})_i =_{\underline{\tau}} \mathsf{op}'_{\boldsymbol{x}'}(\boldsymbol{x}'_{i'}.\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_{ii'}\boldsymbol{x}_i\boldsymbol{x}'_{i'})_i)_{i'}
\end{aligned}$$

holds for all operations $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$ *and* $\mathsf{op}'{:}\boldsymbol{\beta}';\boldsymbol{\alpha}'_1,\ldots,\boldsymbol{\alpha}'_{n'}$*, then*

$$\begin{aligned}&y_1{:}F\sigma_1, y_2{:}F\sigma_2, y{:}\sigma_1 \to \sigma_2 \to \underline{\tau} \vdash_L \\ &\quad y_1\,\mathsf{to}\,x_1.\,(y_2\,\mathsf{to}\,x_2.\,yx_1x_2) =_{\underline{\tau}} y_2\,\mathsf{to}\,x_2.\,(y_1\,\mathsf{to}\,x_1.\,yx_1x_2)\end{aligned}$$

is derivable.

In particular, the commutativity condition holds when the effect theory $\mathcal{T}_{\mathrm{eff}}$ contains the equation

$$\begin{aligned}&\boldsymbol{x}{:}\boldsymbol{\beta};\boldsymbol{x}'{:}\boldsymbol{\beta}';(z_{ii'}{:}(\boldsymbol{\alpha}_i,\boldsymbol{\alpha}'_{i'}))_{1\leqslant i\leqslant n,1\leqslant i'\leqslant n'} \vdash \\ &\quad \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,\mathsf{op}'_{\boldsymbol{x}'}(\boldsymbol{x}'_{i'}.\,z_{ii'}(\boldsymbol{x}_i,\boldsymbol{x}'_{i'}))_{i'})_i = \mathsf{op}'_{\boldsymbol{x}'}(\boldsymbol{x}'_{i'}.\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,z_{ii'}(\boldsymbol{x}_i,\boldsymbol{x}'_{i'}))_i)_{i'}\end{aligned}$$

for all operations $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$ and $\mathsf{op}'{:}\boldsymbol{\beta}';\boldsymbol{\alpha}'_1,\ldots,\boldsymbol{\alpha}'_{n'}$ in Σ_{eff}.

Finally, we show a logical counterpart to the fact that algebraic operations are recoverable from their generic effects [PP03]. Although operations seem more general, this is not surprising because the generic effect captures both the effect and its outcome.

Proposition 6.4 *The equation*

$$\Gamma;\Delta \vdash_L \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\,t_i)_i =_{\underline{\tau}} \mathsf{gen}_{\mathsf{op}}\,\boldsymbol{v}\,\mathsf{to}\,x.\ \mathsf{match}\,x\,\mathsf{with}(\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle.\,t_i)_i$$

holds for an arbitrary operation symbol $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$.

Proof We have:

$$\begin{aligned}&\Gamma;\Delta \vdash_L \mathsf{gen}_{\mathsf{op}}\,\boldsymbol{v}\,\mathsf{to}\,x.\ \mathsf{match}\,x\,\mathsf{with}(\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle.\,t_i)_i \\ &\quad =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_j.\ \mathsf{return}\,\mathsf{inj}_j\langle\boldsymbol{x}_j\rangle)_j\,\mathsf{to}\,x.\ \mathsf{match}\,x\,\mathsf{with}(\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle.\,t_i)_i \\ &\quad =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_j.\ \mathsf{return}\,\mathsf{inj}_j\langle\boldsymbol{x}_j\rangle\,\mathsf{to}\,x.\ \mathsf{match}\,x\,\mathsf{with}(\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle.\,t_i)_i)_j \\ &\quad =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_j.\ \mathsf{match}\,\mathsf{inj}_j\langle\boldsymbol{x}_j\rangle\,\mathsf{with}(\mathsf{inj}_i\langle\boldsymbol{x}_i\rangle.\,t_i)_i)_j \\ &\quad =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_j.\,t_j)_j\,.\end{aligned}$$

□

6.2 Free model principle

The free model principle does not state the universal property of the free model in its entirety: it omits the uniqueness of the induced homomorphism because it follows from the principle of induction.

Proposition 6.5 *Take Δ containing $y{:}\sigma \to \underline{\tau}$, $\hat{y}_1{:}UF\sigma \to \underline{\tau}$, and $\hat{y}_2{:}UF\sigma \to \underline{\tau}$. As in the statement of the free model principle, define*

$$\begin{aligned}\varphi_m &= \{t_{\mathsf{op}} : \underline{\tau}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}}\ \mathsf{models}\ \mathcal{T}_{\mathsf{eff}}\,,\\ \varphi_1 &= \hat{y}_1\ \mathsf{extends}\ y \wedge \hat{y}_1\ \mathsf{homomorphism}\,,\\ \varphi_2 &= \hat{y}_2\ \mathsf{extends}\ y \wedge \hat{y}_2\ \mathsf{homomorphism}\ .\end{aligned}$$

Then, we have

$$\Gamma;\Delta;\Pi \mid \Psi,\varphi_m,\varphi_1,\varphi_2 \vdash_L \forall y'{:}F\sigma.\ \hat{y}_1(\mathsf{thunk}\,y') =_{\underline{\tau}} \hat{y}_2(\mathsf{thunk}\,y')$$

Proof We proceed by the principle of computational induction. For the base case, we take $y' = \mathsf{return}\,x$ for some $x:\sigma$. By the definition of $\hat{y}_j$ $\mathsf{extends}$ y for $j \in \{1,2\}$, we get

$$\Gamma;\Delta;\Pi \mid \Psi,\varphi_m,\varphi_1,\varphi_2 \vdash_L \hat{y}_1(\mathsf{thunk}\,\mathsf{return}\,x) =_{\underline{\tau}} yx =_{\underline{\tau}} \hat{y}_2(\mathsf{thunk}\,\mathsf{return}\,x)\,.$$

Next, take an operation $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$ and assume that

$$\hat{y}_1(\mathsf{thunk}(y_i'\boldsymbol{x}_i)) =_{\underline{\tau}} \hat{y}_2(\mathsf{thunk}(y_i'\boldsymbol{x}_i))$$

for all $y_i'{:}\boldsymbol{\alpha}_i \to F\sigma$ and $\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i$ for $1 \leqslant i \leqslant n$. Then, by $\hat{y}_j$ $\mathsf{homomorphism}$ for $j \in \{1,2\}$, we get

$$\begin{aligned}&\hat{y}_1(\mathsf{thunk}(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i'\boldsymbol{x}_i)_i))\\ &=_{\underline{\tau}} t_{\mathsf{op}}[\lambda\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \hat{y}_1(\mathsf{thunk}\,y_i'\boldsymbol{x}_i)/y_i]_i\\ &=_{\underline{\tau}} t_{\mathsf{op}}[\lambda\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \hat{y}_2(\mathsf{thunk}\,y_i'\boldsymbol{x}_i)/y_i]_i\\ &=_{\underline{\tau}} \hat{y}_2(\mathsf{thunk}(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i'\boldsymbol{x}_i)_i))\,,\end{aligned}$$

which proves the step case. □

Although we can only give models on sets of the form $U[\![\underline{\tau}]\!]$ for some $\underline{\tau}$, we can simulate a model on a set $[\![\sigma']\!]$. To do so, we give a model on $UF[\![\sigma']\!]$ for which the set of returned values from $[\![\sigma']\!]$ is closed under operations.

Proposition 6.6 *Take Δ containing $y{:}\sigma \to F\sigma'$ and $\hat{y}{:}UF\sigma \to F\sigma'$. As before,* *define*

$$
\begin{aligned}
\varphi_m &= \{t_{\mathsf{op}} : F\sigma'\}_{\mathsf{op}\in\Sigma_{\mathrm{eff}}}\ \mathsf{models}\ \mathcal{T}_{\mathrm{eff}}\,,\\
\varphi &= \hat{y}\ \mathsf{extends}\ y \wedge \hat{y}\ \mathsf{homomorphism}\,,\\
\varphi' &= \forall x{:}\sigma.\ \exists x'{:}\sigma'.\ yx =_{F\sigma'} \mathsf{return}\, x'\,,\\
\varphi_{\mathsf{op}} &= \forall \boldsymbol{x}{:}\boldsymbol{\beta}; y_1{:}\boldsymbol{\alpha}_1 \to F\sigma', \ldots, y_n{:}\boldsymbol{\alpha}_n \to F\sigma'.\\
&\qquad \bigwedge_{i=1}^{n} (\forall \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \exists x_i{:}\sigma'.\ y_i\boldsymbol{x}_i =_{F\sigma'} \mathsf{return}\, x_i) \Rightarrow \exists x'{:}\sigma'.\ t_{\mathsf{op}} =_{F\sigma'} \mathsf{return}\, x'\,.
\end{aligned}
$$

Then, we have

$$\Gamma;\Delta;\Pi \mid \Psi, \varphi_m, \varphi, \varphi', \bigwedge_{\mathsf{op}} \varphi_{\mathsf{op}} \vdash_L \forall y'{:}F\sigma.\ \exists x'{:}\sigma'.\ \hat{y}(\mathsf{thunk}\, y') = \mathsf{return}\, x'\,.$$

Proof For the base case, we take $y' = \mathsf{return}\, x$ for some $x{:}\sigma$. By $\hat{y}$ extends y, we get

$$\hat{y}(\mathsf{thunk}(\mathsf{return}\, x)) =_{F\sigma'} yx\,,$$

hence by φ', there exists $x'{:}\sigma'$ such that

$$yx =_{F\sigma'} \mathsf{return}\, x'\,.$$

For the step case, take an operation $\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ and assume that for $1 \leqslant i \leqslant n$, $y'_i : \boldsymbol{\alpha}_i \to F\sigma'$ and $\boldsymbol{x}_i : \boldsymbol{\alpha}_i$, there exists a $x'_i{:}\sigma'$ such that

$$\hat{y}(\mathsf{thunk}(y'_i\boldsymbol{x}_i)) =_{F\sigma'} \mathsf{return}\, x'_i\,.$$

Then, by $\hat{y}$ homomorphism, and by φ_{op}, there exists $x'{:}\sigma$ such that

$$\hat{y}(\mathsf{thunk}\, \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y'_i\boldsymbol{x}_i)) =_{F\sigma'} t_{\mathsf{op}}[\lambda \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \hat{y}(\mathsf{thunk}\, y'_i\boldsymbol{x}_i)/y_i]_i =_{F\sigma'} \mathsf{return}\, x'\,.$$

□

6.3 Modalities

We define local modalities in order to reason about the structure of computations. Note that because of the exogenous approach to logic, modalities are operators on predicates, rather than propositions.

Definition 6.7 For a predicate $\pi{:}\mathbf{prop}(\sigma)$, a *pureness necessity modality* $[\downarrow]$ and a *pureness possibility modality* $\langle\downarrow\rangle$ are defined by:

$$
\begin{aligned}
[\downarrow](\pi) &=_{\mathrm{def}} (y{:}F\sigma).\ \forall x{:}\sigma.\ y =_{F\sigma} \mathsf{return}\, x \Rightarrow \pi(x)\,,\\
\langle\downarrow\rangle(\pi) &=_{\mathrm{def}} (y{:}F\sigma).\ \exists x{:}\sigma.\ y =_{F\sigma} \mathsf{return}\, x \wedge \pi(x)\,.
\end{aligned}
$$

A pureness modality transforms a predicate on a value type σ into a predicate on a computation type $F\sigma$. The notation for the pureness modality follows the notation for Moggi's pureness predicate $t \downarrow \sigma$, which is expressible in terms of the pureness modality as $\langle\downarrow\rangle((x{:}\sigma).\ \top)(t)$.

Definition 6.8 For an operation $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$ and a predicate

$$\pi{:}\mathbf{prop}(\boldsymbol{\beta};\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,\boldsymbol{\alpha}_n \to \underline{\tau})\,.$$

an *operation necessity modality* $[\mathsf{op}]$ and an *operation possibility modality* $\langle\mathsf{op}\rangle$ are defined by:

$$[\mathsf{op}](\pi) =_{\mathrm{def}} (y{:}\underline{\tau}).\ \forall \boldsymbol{x}{:}\boldsymbol{\beta}, y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau}.\ y =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i \Rightarrow \pi(\boldsymbol{x};\boldsymbol{y})\,,$$
$$\langle\mathsf{op}\rangle(\pi) =_{\mathrm{def}} (y{:}\underline{\tau}).\ \exists \boldsymbol{x}{:}\boldsymbol{\beta}, y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau}.\ y =_{\underline{\tau}} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i \wedge \pi(\boldsymbol{x};\boldsymbol{y})\,.$$

Example 6.9 If we take the effect theory for exceptions, then $[\mathsf{raise}]((y{:}\underline{\tau}).\ \bot)(t)$ is equivalent to $\forall \mathsf{exc}{:}\mathbf{exc}.\ \neg(t =_{\underline{\tau}} \mathsf{raise}_{\mathsf{exc}})$ and hence states that t does not raise an exception. On the other hand, $\langle\mathsf{raise}\rangle((y{:}\underline{\tau}).\ \top)(t)$ states that t does raise an exception.

For a predicate $\pi{:}\mathbf{prop}(\underline{\tau})$, we define $[-](\pi)$ to be

$$(y{:}\underline{\tau}).\ \bigwedge_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}} [\mathsf{op}]((\boldsymbol{x}{:}\boldsymbol{\beta}, y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau}).\ \bigwedge_{i=1}^{n} \forall \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \pi(y_i\boldsymbol{x}_i))(y)\,,$$

and $\langle-\rangle(\pi)$ to be

$$(y{:}\underline{\tau}).\ \bigvee_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}} \langle\mathsf{op}\rangle((\boldsymbol{x}{:}\boldsymbol{\beta}, y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau}).\ \bigvee_{i=1}^{n} \exists \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \pi(y_i\boldsymbol{x}_i))(y)\,.$$

Informally, we say that computation terms t_i are *immediate continuations* of $\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\ t_i)_i$. Then, we get the notion of a *continuation* by taking the reflexive and transitive closure of the notion of the immediate continuations. A *run* of a computation term t is a sequence of computation terms $t = t_1, t_2, \ldots, t_n$ such that t_i is an immediate continuation of t_{i-1}. These notions serve only to illustrate the meaning of modalities, and should not be confused with continuations [FSDF93].

Then, intuitively, $[-](\pi)(t)$ states that all immediate continuations of t satisfy π, no matter what the outcome of the effect was, while $\langle-\rangle(\pi)(t)$ states that there exists an immediate continuation of t that satisfies π.

Remark 6.10 The (immediately) derivable two-way introduction and elimination rules for necessity and possibility modalities are

$$\begin{array}{c} \Gamma;\Delta,y{:}F\sigma;\Pi \mid \Psi \vdash_L [\downarrow](\pi)(y) \\ \hline\hline \Gamma,x{:}\sigma;\Delta;\Pi \mid \Psi[\mathsf{return}\,x/y] \vdash_L \pi(x) \end{array},$$

$$\begin{array}{c} \Gamma;\Delta,y{:}F\sigma;\Pi \mid \Psi,\langle\downarrow\rangle(\pi)(y) \vdash_L \varphi \\ \hline\hline \Gamma,x{:}\sigma;\Delta;\Pi \mid \Psi[\mathsf{return}\,x/y],\pi(x) \vdash_L \varphi[\mathsf{return}\,x/y] \end{array}$$

for pureness modalities, and

$$\begin{array}{c} \Gamma;\Delta,y{:}\underline{\tau};\Pi \mid \Psi \vdash_L [\mathsf{op}](\pi)(y) \\ \hline\hline \Gamma,\boldsymbol{x}{:}\boldsymbol{\beta};\Delta,y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau};\Pi \mid \Psi[\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i/y] \vdash_L \pi(\boldsymbol{x};\boldsymbol{y}) \end{array},$$

$$\begin{array}{c} \Gamma;\Delta,y{:}\underline{\tau};\Pi \mid \Psi,\langle\mathsf{op}\rangle(\pi)(y) \vdash_L \varphi \\ \hline\hline \Gamma,\boldsymbol{x}{:}\boldsymbol{\beta};\Delta,y_1{:}\boldsymbol{\alpha}_1 \to \underline{\tau},\ldots,y_n{:}\boldsymbol{\alpha}_n \to \underline{\tau};\Pi \mid \Psi[\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i/y],\pi(\boldsymbol{x};\boldsymbol{y}) \vdash_L \varphi[\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i/y] \end{array}$$

for operation modalities, corresponding to $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$.

From the adjoint form of those rules, one can see that in the categorical approach to logic [Jac99], pureness and operation modalities are quantifiers corresponding to the returning of values and to operations, respectively.

To extend local to global reasoning, we use predicate fixed points to define global modalities.

Definition 6.11 For a predicate $\pi : \mathbf{prop}(\underline{\tau})$, the *global necessity modality* $\Box$ is defined to be

$$\Box\pi =_{\mathrm{def}} \nu P{:}(\underline{\tau}).\,(y{:}\underline{\tau}).\,\pi(y) \wedge [-](P)(y)\,,$$

while the *global possibility modality* $\Diamond$ is defined to be

$$\Diamond\pi =_{\mathrm{def}} \mu P{:}(\underline{\tau}).\,(y{:}\underline{\tau}).\,\pi(y) \vee \langle-\rangle(P)(y)\,.$$

From the introduction and elimination rules for predicate fixed points, we can immediately derive the following rules for global modalities:

$$\frac{\Gamma;\Delta;\Pi \mid \pi'(y) \vdash_L \pi(y) \wedge [-](\pi')(y)}{\Gamma;\Delta;\Pi \mid \pi'(y) \vdash_L \Box\pi(y)}\,, \qquad \frac{}{\Gamma;\Delta;\Pi \mid \Box\pi(t) \vdash_L \pi(t) \wedge [-]\Box\pi(t)}\,,$$

$$\frac{\Gamma;\Delta;\Pi \mid \pi(y) \vee \langle-\rangle(\pi')(y) \vdash_L \pi'(y)}{\Gamma;\Delta;\Pi \mid \Diamond\pi(y) \vdash_L \pi'(y)}\,, \qquad \frac{}{\Gamma;\Delta;\Pi \mid \pi(t) \vee \langle-\rangle(\Diamond\pi)(t) \vdash_L (\Diamond\pi)(t)}\,.$$

Intuitively, $\Box\pi(t)$ states that all continuations of t satisfy π, while $\Diamond\pi(t)$ states that there exists a continuation of t that satisfies π. Since continuations are obtained by a reflexive and transitive closure of immediate continuations, we may expect that the global modalities satisfy the rules of $S4$ modal logic.

Proposition 6.12 *The following rules are derivable:*

- (K)
$$\frac{}{\Gamma;\Delta;\Pi \mid \Box((y:\underline{\tau}).\, \pi_1(y) \Rightarrow \pi_2(y))(t) \vdash_L \Box\pi_1(t) \Rightarrow \Box\pi_2(t)}\,,$$
- (N)
$$\frac{}{\Gamma;\Delta;\Pi \mid \forall y:\underline{\tau}.\, \pi(y) \vdash_L \forall y:\underline{\tau}.\, \Box\pi(y)}\,,$$
- (T)
$$\frac{}{\Gamma;\Delta;\Pi \mid \Box\pi(t) \vdash_L \pi(t)}\,,$$
- (4)
$$\frac{}{\Gamma;\Delta;\Pi \mid \Box\pi(t) \vdash_L \Box\Box\pi(t)}\,.$$

Dual properties, of course, hold for the possibility modality.

Proof

- (K) Take
$$\pi'(y) =_{\mathrm{def}} \Box((y:\underline{\tau}).\, \pi_1(y) \Rightarrow \pi_2(y))(y) \wedge \Box\pi_1(y)$$
and assume that $\pi'(y)$ holds. To show that $\Box\pi_2(y)$ holds, we have to show that $\pi'(y)$ implies both $\pi_2(y)$ and $[-](\pi')(y)$. On one hand, $\Box\pi(y)$ implies $\pi(y)$ for any π, so we immediately get $\pi_2(y)$. On the other hand, $\Box\pi(y)$ also implies $[-]\Box\pi(y)$, hence $\pi'(y)$ implies
$$[-](\Box((y:\underline{\tau}).\, \pi_1(y) \Rightarrow \pi_2(y)))(y) \wedge [-](\Box\pi_1(y))(y)\,.$$
Since necessity modalities commute with conjunctions, we get $[-](\pi')(y)$. Thus $\pi'(y)$ implies $\Box\pi_2(y)$. We finish the argument by substituting t for y.

- (N) Assume $\forall y:\underline{\tau}.\, \pi(y)$ and define π' to be $(y':\underline{\tau}).\, \forall y:\underline{\tau}.\, \pi(y)$. Since $\pi'(y)$ implies both $\pi(y)$ and $[-](\pi')(y)$, it implies $\Box\pi(y)$.

- (T) Next, assuming that $\Box\pi(y)$ holds, we immediately get that $\pi(y)$ holds as well. We prove the judgement by substituting t for y.
- (4) Assume that $\Box\pi(y)$ holds, and define $\pi'(y)$ to be $\Box\pi(y)$. Since $\Box\pi(y)$ implies both $\Box\pi(y)$ and $[-]\Box\pi(y)$, we have

  $$\Gamma;\Delta;\Pi \mid \pi'(y) \vdash_L \Box\pi(y) \wedge [-]\Box\pi(y)\,,$$

  hence $\Gamma;\Delta;\Pi \mid \pi'(y) \vdash_L \Box\Box\pi(y)$. We finish the argument by substituting t for y.

□

We can also define other global modalities known from computational tree logic [HR04], for example

$$\mathrm{AF}\pi =_{\mathrm{def}} \mu P:(\underline{\tau}).\,(y:\underline{\tau}).\,\pi(y) \vee (\langle - \rangle((y:\underline{\tau}).\,\top)(y) \wedge [-](P)(y))\,,$$

which states that for any run of the computation, π holds at some point, or

$$\mathrm{EG}\pi =_{\mathrm{def}} \nu P:(\underline{\tau}).\,(y:\underline{\tau}).\,\pi(y) \wedge ([-]((y:\underline{\tau}).\,\bot)(y) \vee \langle - \rangle(P)(y))\,.$$

which states that a computation has a run for which π continues to hold.

6.4 Computational λ-calculus

We now give a conservative translation of a reasonable restriction of Moggi's computational λ-calculus [Mog89]. We shall translate its types, terms, propositions, and judgements into the ones of the logic, and show that translation preserves provability.

6.4.1 Definition

Moggi's computational λ-calculus is an equational logic, equipped with a pureness predicate. Given a signature Σ_{t} of base types β, the types σ are given by the following grammar:

$$\sigma ::= \beta \mid \mathbf{1} \mid \sigma_1 \times \sigma_2 \mid T\sigma \mid \sigma_1 \rightarrow \sigma_2\,.$$

Then, given a signature Σ_{f} of function symbols $\mathsf{f}:\sigma_1 \to \sigma_2$, the terms t are given by:

$$t ::= x \mid \mathsf{f}(t) \mid [t] \mid \mu(t) \mid \star \mid \langle t_1, t_2 \rangle \mid \pi_1(t) \mid \pi_2(t) \mid \mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, t' \mid \lambda x{:}\sigma.\, t \mid tt'\,.$$

Terms are typed as $\Gamma \vdash_{\lambda_c} t : \sigma$ in a context $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$, according to the following rules:

$$\frac{}{\Gamma \vdash_{\lambda_c} x : \sigma} \; (x : \sigma \in \Gamma), \qquad \frac{\Gamma \vdash_{\lambda_c} t : \sigma_1}{\Gamma \vdash_{\lambda_c} \mathsf{f}(t) : \sigma_2} \; (\mathsf{f} : \sigma_1 \to \sigma_2 \in \Sigma_\mathsf{f}), \qquad \frac{\Gamma \vdash_{\lambda_c} t : \sigma}{\Gamma \vdash_{\lambda_c} [t] : T\sigma},$$

$$\frac{\Gamma \vdash_{\lambda_c} t : T\sigma}{\Gamma \vdash_{\lambda_c} \mu(t) : \sigma}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} \star : \mathbf{1}}, \qquad \frac{\Gamma \vdash_{\lambda_c} t_1 : \sigma_1 \qquad \Gamma \vdash_{\lambda_c} t_2 : \sigma_2}{\Gamma \vdash_{\lambda_c} \langle t_1, t_2 \rangle : \sigma_1 \times \sigma_2},$$

$$\frac{\Gamma \vdash_{\lambda_c} t : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\lambda_c} \pi_1(t) : \sigma_1}, \qquad \frac{\Gamma \vdash_{\lambda_c} t : \sigma_1 \times \sigma_2}{\Gamma \vdash_{\lambda_c} \pi_2(t) : \sigma_2}, \qquad \frac{\Gamma \vdash_{\lambda_c} t : \sigma \qquad \Gamma, x : \sigma \vdash_{\lambda_c} t' : \sigma'}{\Gamma \vdash_{\lambda_c} \mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, t' : \sigma'},$$

$$\frac{\Gamma, x : \sigma \vdash_{\lambda_c} t : \sigma'}{\Gamma \vdash_{\lambda_c} \lambda x : \sigma .\, t : \sigma \to \sigma'}, \qquad \frac{\Gamma \vdash_{\lambda_c} t : \sigma \to \sigma' \qquad \Gamma \vdash_{\lambda_c} t' : \sigma}{\Gamma \vdash_{\lambda_c} t t' : \sigma'}.$$

To differentiate between values and computations, the computational λ-calculus has a pureness predicate $t \downarrow \sigma$, which states that a computation t of type σ causes no effects. The rules for the pureness predicate are

$$\frac{\Gamma \vdash_{\lambda_c} t \downarrow \sigma \qquad \Gamma, x : \sigma \vdash_{\lambda_c} t' \downarrow \sigma'}{\Gamma \vdash_{\lambda_c} t'[t/x] \downarrow \sigma'}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} x \downarrow \sigma}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} [t] \downarrow T\tau},$$

$$\frac{}{\Gamma \vdash_{\lambda_c} \star \downarrow \mathbf{1}}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} \langle x_1, x_2 \rangle \downarrow \tau_1 \times \tau_2}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} \pi_1(x) \downarrow \tau_1}, \qquad \frac{}{\Gamma \vdash_{\lambda_c} \pi_2(x) \downarrow \tau_2},$$

$$\frac{}{\Gamma \vdash_{\lambda_c} \lambda x : \sigma .\, t \downarrow \sigma \to \tau},$$

while the rules for equality are the ones stating that it is a congruence, and

- (=-subst)

$$\frac{\Gamma \vdash_{\lambda_c} t \downarrow \sigma \qquad \Gamma, x : \sigma \vdash_{\lambda_c} t_1 =_{\sigma'} t_2}{\Gamma \vdash_{\lambda_c} t_1[t/x] =_{\sigma'} t_2[t/x]},$$

- (unit)

$$\frac{}{\Gamma \vdash_{\lambda_c} \mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, x =_\sigma t},$$

- (ass)

$$\frac{}{\Gamma \vdash_{\lambda_c} \mathsf{let}\, x_2\, \mathsf{be}\, (\mathsf{let}\, x_1\, \mathsf{be}\, t_1\, \mathsf{in}\, t_2)\, \mathsf{in}\, t =_\sigma \mathsf{let}\, x_1\, \mathsf{be}\, t_1\, \mathsf{in} (\mathsf{let}\, x_2\, \mathsf{be}\, t_2\, \mathsf{in}\, t)},$$

- (let-β)

$$\frac{}{\Gamma \vdash_{\lambda_c} \mathsf{let}\,x\,\mathsf{be}\,x'\,\mathsf{in}\,t =_\sigma t[x'/x]}\,,$$

- (let-f)

$$\frac{}{\Gamma \vdash_{\lambda_c} \mathsf{f}(t) =_{\sigma_2} \mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}\,\mathsf{f}(x)}\,,$$

- (T-β)

$$\frac{}{\Gamma \vdash_{\lambda_c} \mu([t]) =_\sigma t}\,,$$

- (T-η)

$$\frac{}{\Gamma \vdash_{\lambda_c} [\mu(x)] =_{T\sigma} x}\,,$$

- ($\mathbf{1}$-η)

$$\frac{}{\Gamma \vdash_{\lambda_c} x =_{\mathbf{1}} \star}\,,$$

- (let-$\langle - \rangle$)

$$\frac{}{\Gamma \vdash_{\lambda_c} \langle t_1, t_2 \rangle =_{\sigma_1 \times \sigma_2} \mathsf{let}\,x_1\,\mathsf{be}\,t_1\,\mathsf{in}\,\mathsf{let}\,x_2\,\mathsf{be}\,t_2\,\mathsf{in}\,\langle x_1, x_2 \rangle}\,,$$

- ($\times$-β)

$$\frac{}{\Gamma \vdash_{\lambda_c} \pi_i(\langle x_1, x_2 \rangle) =_{\sigma_i} x_i} \quad (1 \leqslant i \leqslant 2)\,,$$

- ($\times$-η)

$$\frac{}{\Gamma \vdash_{\lambda_c} \langle \pi_1(x), \pi_2(x) \rangle =_{\sigma_1 \times \sigma_2} x}\,,$$

- (let-λ)

$$\frac{}{\Gamma \vdash_{\lambda_c} tt' =_\sigma \mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}\,\mathsf{let}\,x'\,\mathsf{be}\,t'\,\mathsf{in}\,xx'}\,,$$

- (β)

$$\frac{}{\Gamma \vdash_{\lambda_c} (\lambda x{:}\sigma.\, t)x =_\sigma t}\,,$$

- (η)

$$\frac{}{\Gamma \vdash_{\lambda_c} \lambda x{:}\sigma.\, x'x =_\sigma x'} \quad (x \neq x')\,.$$

Note that because of the lack of distinction between values and computations, the evaluation of pairs and applications has an implicit order, determined by equations (let-$\langle - \rangle$) and (let-λ), in which the left subterm is evaluated first.

6.4.2 Translation

Before giving the translation, note that in the computational λ-calculus, primitive functions $\mathsf{f}:\sigma \to \sigma'$ are not limited to base types, but accept arguments of arbitrary types and return computations that can cause effects. We are not going to give a translation of the calculus over such general signature, but for one where the signature Σ_{f} is restricted to pure functions $\mathsf{f}:\prod \boldsymbol{\beta} \to \beta$ and generic effects $\mathsf{gen}_{\mathsf{op}}:\prod \boldsymbol{\beta} \to T(\prod \boldsymbol{\alpha})$ (for more general generic effects, one would add sum types to Moggi's language). This restriction is in the line with the main premise of our approach, which is that algebraic operations give an adequate representation of effects.

Then, we take a base signature, consisting of all the base types $\beta \in \Sigma_{\mathrm{t}}$, and of function symbols $\mathsf{f}:(\boldsymbol{\beta}) \to \beta$ for each pure function $\mathsf{f}:\prod \boldsymbol{\beta} \to \beta \in \Sigma_{\mathrm{f}}$. Next, we take an effect signature, consisting of operations $\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}$ for each generic effect $\mathsf{gen}_{\mathsf{op}}:\prod \boldsymbol{\beta} \to T(\prod \boldsymbol{\alpha}) \in \Sigma_{\mathrm{f}}$.

For translating types, we first observe that, unlike in our approach, terms $\Gamma \vdash_{\lambda_{\mathrm{c}}} t:\sigma$ represent computations that return values of type σ. Hence, we translate types σ to value types σ^* as

$$\begin{aligned}
\beta^* &= \beta\,, \\
(\sigma_1 \times \sigma_2)^* &= \sigma_1^* \times \sigma_2^*\,, \\
\mathbf{1}^* &= \mathbf{1}\,, \\
(\sigma \to \sigma')^* &= U(\sigma^* \to F\sigma'^*)\,, \\
(T\sigma)^* &= UF\sigma^*\,,
\end{aligned}$$

and contexts $\Gamma = x_1:\sigma_1,\ldots,x_n:\sigma_n$ as

$$\Gamma^* = x_1:\sigma_1^*,\ldots,x_n:\sigma_n^*\,.$$

Then, we translate terms t to computation terms t^* as

$$\begin{aligned}
x^* &= \mathsf{return}\, x\,,\\
\mathsf{f}(t)^* &= t^*\,\mathsf{to}\, x.\ \mathsf{return}\,\mathsf{f}(\mathsf{prj}_i\, x)_i\,,\\
\mathsf{gen}_{\mathsf{op}}(t)^* &= t^*\,\mathsf{to}\, x.\ \mathsf{return}\,\mathsf{thunk}(\mathsf{gen}_{\mathsf{op}}\, x)\,,\\
[t]^* &= \mathsf{return}\,\mathsf{thunk}\, t^*\,,\\
\mu(t)^* &= t^*\,\mathsf{to}\, x.\ \mathsf{force}\, x\,,\\
\star^* &= \mathsf{return}\,\star\,,\\
\langle t_1, t_2\rangle^* &= t_1^*\,\mathsf{to}\, x_1.\ t_2^*\,\mathsf{to}\, x_2.\ \mathsf{return}\langle x_1, x_2\rangle\,,\\
\pi_1(t)^* &= t^*\,\mathsf{to}\, x.\ \mathsf{return}\,\mathsf{fst}\, x\,,\\
\pi_2(t)^* &= t^*\,\mathsf{to}\, x.\ \mathsf{return}\,\mathsf{snd}\, x\,,\\
(\mathsf{let}\, x\,\mathsf{be}\, t\,\mathsf{in}\, t')^* &= t^*\,\mathsf{to}\, x.\ t'^*\,,\\
(\lambda x{:}\sigma.\ t)^* &= \mathsf{return}\,\mathsf{thunk}\,\lambda x{:}\sigma^*.\ t^*\,,\\
(tt')^* &= t^*\,\mathsf{to}\, x.\ t'^*\,\mathsf{to}\, x'.\ (\mathsf{force}\, x) x'\,.
\end{aligned}$$

Proposition 6.13 *If* $\Gamma \vdash_{\lambda_c} t{:}\sigma$ *then* $\Gamma^* \vdash t^*{:}F\sigma^*$.

Proof We proceed by a routine induction on the derivation of the typing judgement. □

With the typing judgements preserved, we show how the logic of the computational λ-calculus is embraced in our logic.

Lemma 6.14 *If there exists a value term* v *such that* $\Gamma \vdash_L t^* =_{F\sigma^*} \mathsf{return}\, v$, *then* $\Gamma \vdash_L (t'[t/x])^* =_{F\sigma'^*} t'^*[v/x]$.

Proof We proceed by a straightforward induction on the structure of t'. □

Proposition 6.15 *If* $\Gamma \vdash_{\lambda_c} t \downarrow \sigma$, *then there exists a value term* $\Gamma^* \vdash v{:}\sigma^*$, *such that* $\Gamma \vdash_L t^* =_{F\sigma^*} \mathsf{return}\, v$.

Proof We proceed by an induction on the derivation of $\Gamma \vdash_{\lambda_c} t\downarrow\sigma$. If we consider any axiom of the form $\Gamma \vdash_{\lambda_c} t\downarrow\sigma$, we observe that t^* is always equivalent to $\mathsf{return}\, v$ for some value term v. Next, consider the case when the last applied rule was

$$\frac{\Gamma \vdash_{\lambda_c} t\downarrow\sigma \qquad \Gamma, x{:}\sigma \vdash_{\lambda_c} t'\downarrow\sigma'}{\Gamma \vdash_{\lambda_c} t'[t/x]\downarrow\sigma'}\,.$$

By the induction hypothesis, we get value terms v and v' such that

$$\Gamma^* \vdash_L t^* =_{F\sigma^*} \mathsf{return}\, v \qquad \text{and} \qquad \Gamma^*, x{:}\sigma^* \vdash_L t'^* =_{F\sigma'^*} \mathsf{return}\, v'\,.$$

Then, by Lemma 6.14 and by value substitution, we get

$$\Gamma^* \vdash_L (t'[t/x])^* =_{F\sigma'^*} \mathsf{return}\, v'[v/x] \,.$$

□

Theorem 6.16 *If* $\Gamma \vdash_{\lambda_c} t_1 =_\sigma t_2$*, then* $\Gamma^* \vdash_L t_1^* =_{F\sigma^*} t_2^*$*.*

Proof We again proceed by an induction on the derivation of $\Gamma \vdash_{\lambda_c} t_1 =_\sigma t_2$. The case, when the last applied rule was (=-subst) proceeds similarly as the step case in the proof of Proposition 6.15, while the cases of (let-$\langle - \rangle$) and (let-λ) are routine. We consider the other cases in turn.

- For (unit), we have

$$\Gamma^* \vdash_L (\mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, x)^* =_{F\sigma^*} t^*\, \mathsf{to}\, x.\ \mathsf{return}\, x =_{F\sigma^*} t^*$$

  by Proposition 6.1.

- For (ass), we use Proposition 6.2 to show

$$\begin{aligned} &\Gamma^* \vdash_L (\mathsf{let}\, x_2\, \mathsf{be} (\mathsf{let}\, x_1\, \mathsf{be}\, t_1\, \mathsf{in}\, t_2) \mathsf{in}\, t)^* \\ &\quad =_{\sigma^*} (t_1^*\, \mathsf{to}\, x_1.\ t_2^*) \mathsf{to}\, x_2.\ t^* \\ &\quad =_{\sigma^*} t_1^*\, \mathsf{to}\, x_1. (t_2^*\, \mathsf{to}\, x_2.\ t^*) \\ &\quad = (\mathsf{let}\, x_1\, \mathsf{be}\, t_1\, \mathsf{in} (\mathsf{let}\, x_2\, \mathsf{be}\, t_2\, \mathsf{in}\, t))^* \,. \end{aligned}$$

- For (let-β), we have

$$\begin{aligned} &\Gamma^* \vdash_L (\mathsf{let}\, x\, \mathsf{be}\, x'\, \mathsf{in}\, t)^* \\ &\quad =_{F\sigma^*} \mathsf{return}\, x'\, \mathsf{to}\, x.\ t^* \\ &\quad =_{F\sigma^*} t^*[x'/x] \\ &\quad =_{F\sigma^*} (t[x'/x])^* \,. \end{aligned}$$

- (let-f) For a pure function $\mathsf{f} \colon \prod \boldsymbol{\beta} \to \beta$, we have:

$$\begin{aligned} &\Gamma^* \vdash_L \mathsf{f}(t)^* \\ &\quad =_\beta t^*\, \mathsf{to}\, x.\ \mathsf{return}\, \mathsf{f}(\mathsf{prj}_i\, x)_i \\ &\quad =_\beta t^*\, \mathsf{to}\, x.\ \mathsf{return}\, x\, \mathsf{to}\, x'.\ \mathsf{return}\, \mathsf{f}(\mathsf{prj}_i\, x')_i \qquad \text{(by β-equivalence)} \\ &\quad =_\beta (\mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, \mathsf{f}(x))^* \,, \end{aligned}$$

  while the case with a generic effect proceeds similarly.

- For (T-β), we have

$$\begin{aligned}
&\Gamma^* \vdash_L (\mu([t]))^* \\
&\quad =_{F\sigma^*} \mathsf{return}\,\mathsf{thunk}\,t^*\,\mathsf{to}\,x.\ \mathsf{force}\,x \\
&\quad =_{F\sigma^*} \mathsf{force}\,\mathsf{thunk}\,t^* \\
&\quad =_{F\sigma^*} t^*\ .
\end{aligned}$$

- For (T-η), we have

$$\begin{aligned}
&\Gamma^* \vdash_L [\mu(x)]^* \\
&\quad =_{FUF\sigma^*} \mathsf{return}\,\mathsf{thunk}(\mathsf{return}\,x\,\mathsf{to}\,x'.\ \mathsf{force}\,x') \\
&\quad =_{FUF\sigma^*} \mathsf{return}\,\mathsf{thunk}\,\mathsf{force}\,x \\
&\quad =_{FUF\sigma^*} \mathsf{return}\,x\ .
\end{aligned}$$

- ($\mathbf{1}$-η) From $\Gamma \vdash_{\lambda_c} x =_{\mathbf{1}} \star$, it follows that $\Gamma \vdash_{\lambda_c} x{:}\mathbf{1}$ holds, hence $x{:}\mathbf{1} \in \Gamma$. This implies $x{:}\mathbf{1} \in \Gamma^*$, thus $\Gamma^* \vdash_L x =_{\mathbf{1}} \star$ and

$$\Gamma^* \vdash_L \mathsf{return}\,x =_{F\mathbf{1}} \mathsf{return}\,\star\ .$$

- For ($\times$-β), we have

$$\begin{aligned}
&\Gamma^* \vdash_L \pi_1(\langle x_1, x_2\rangle)^* \\
&\quad =_{F\sigma_1^*} (\mathsf{return}\,x_1\,\mathsf{to}\,x_1'.\ \mathsf{return}\,x_2\,\mathsf{to}\,x_2'.\ \mathsf{return}\langle x_1', x_2'\rangle)\,\mathsf{to}\,x.\ \mathsf{fst}\,x \\
&\quad =_{F\sigma_1^*} \mathsf{return}\langle x_1, x_2\rangle\,\mathsf{to}\,x.\ \mathsf{return}\,\mathsf{fst}\,x \\
&\quad =_{F\sigma_1^*} \mathsf{return}\,\mathsf{fst}\langle x_1, x_2\rangle \\
&\quad =_{F\sigma_1^*} \mathsf{return}\,x_1\ ,
\end{aligned}$$

  for the first projection, while the case of the second projection proceeds similarly.

- For ($\times$-η), we have

$$\begin{aligned}
&\Gamma^* \vdash_L \langle \pi_1(x), \pi_2(x)\rangle^* \\
&\quad =_{F(\sigma_1\times\sigma_2)^*} (\mathsf{return}\,x\,\mathsf{to}\,x_1'.\ \mathsf{return}\,\mathsf{fst}\,x_1')\,\mathsf{to}\,x_1. \\
&\qquad\quad (\mathsf{return}\,x\,\mathsf{to}\,x_2'.\ \mathsf{return}\,\mathsf{snd}\,x_2')\,\mathsf{to}\,x_2.\ \mathsf{return}\langle x_1, x_2\rangle \\
&\quad =_{F(\sigma_1\times\sigma_2)^*} \mathsf{return}\,\mathsf{fst}\,x\,\mathsf{to}\,x_1.\ \mathsf{return}\,\mathsf{snd}\,x\,\mathsf{to}\,x_2.\ \mathsf{return}\langle x_1, x_2\rangle \\
&\quad =_{F(\sigma_1\times\sigma_2)^*} \mathsf{return}\langle \mathsf{fst}\,x, \mathsf{snd}\,x\rangle \\
&\quad =_{F(\sigma_1\times\sigma_2)^*} \mathsf{return}\,x \\
&\quad =_{F(\sigma_1\times\sigma_2)^*} x^*\ .
\end{aligned}$$

- For (β), we have

$$\begin{aligned}
\Gamma^* \vdash_L & ((\lambda x{:}\sigma.\, t)x')^* \\
&=_{\sigma^*} \mathsf{return}(\mathsf{thunk}(\lambda x{:}\sigma^* t^*.\,))\,\mathsf{to}\,x_1.\ \mathsf{return}\,x'\,\mathsf{to}\,x_2.\,(\mathsf{force}\,x_1)x_2 \\
&=_{\sigma^*} (\mathsf{force}(\mathsf{thunk}(\lambda x{:}\sigma^*.\, t^*)))x' \\
&=_{\sigma^*} (\lambda x{:}\sigma^*.\, t^*)x' \\
&=_{\sigma^*} t^*[x'/x] \\
&=_{\sigma^*} (t[x'/x])^* \,.
\end{aligned}$$

- And for (η), we have

$$\begin{aligned}
\Gamma^* \vdash_L & (\lambda x{:}\sigma.\, x'x)^* \\
&=_{\sigma\to\sigma'^*} \mathsf{return}(\mathsf{thunk}(\lambda x{:}\sigma^*.\ \mathsf{return}\,x'\,\mathsf{to}\,x_1.\ \mathsf{return}\,x\,\mathsf{to}\,x_2.\,(\mathsf{force}\,x_1)x_2)) \\
&=_{\sigma\to\sigma'^*} \mathsf{return}(\mathsf{thunk}(\lambda x{:}\sigma^*.\,(\mathsf{force}\,x')x)) \\
&=_{\sigma\to\sigma'^*} \mathsf{return}(\mathsf{thunk}\,\mathsf{force}\,x') \\
&=_{\sigma\to\sigma'^*} \mathsf{return}\,x' \\
&=_{\sigma\to\sigma'^*} x'^* \,.
\end{aligned}$$

□

6.5 Hennessy-Milner logic

6.5.1 Definition

Hennessy-Milner logic [HM85] is an endogenous logic, which examines whether a given CCS *process* p satisfies a certain *property* φ. The processes and properties are given by the following grammar,

$$\begin{aligned}
p,q,r &::= 0 \mid a.p \mid p+q \\
\varphi &::= \top \mid \bot \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [a](\varphi) \mid \langle a\rangle(\varphi)\,,
\end{aligned}$$

where a ranges over a set of actions A. In the above grammar, we have omitted renaming, hiding, and parallel composition. As seen in Section 7.4.3, the first two can be described using handlers, while the third poses problems for our framework. We have also omitted recursion, which could be done along the lines, presented in Chapter 9.

Note that the properties of Hennessy-Milner logic are usually given by truth, conjunction, possibility modality $\langle a \rangle$, and negation, while falsehood, disjunction, and necessity modality are defined. We take an equivalent approach, which proves more suitable for our logic, and recursively define negation $\neg\varphi$ by

$$\begin{aligned}
\neg\top &= \bot\,, \\
\neg\bot &= \top\,, \\
\neg(\varphi_1 \wedge \varphi_2) &= \neg\varphi_1 \vee \neg\varphi_2\,, \\
\neg(\varphi_1 \vee \varphi_2) &= \neg\varphi_1 \wedge \neg\varphi_2\,, \\
\neg[a](\varphi) &= \langle a \rangle(\neg\varphi)\,, \\
\neg\langle a \rangle(\varphi) &= [a](\neg\varphi)\,.
\end{aligned}$$

In terms of the *transition relation* $p \xrightarrow{a} q$, given inductively by

$$\frac{}{a.p \xrightarrow{a} p}\,, \qquad \frac{p \xrightarrow{a} r}{p + q \xrightarrow{a} r}\,, \qquad \frac{q \xrightarrow{a} r}{p + q \xrightarrow{a} r}\,,$$

we define *satisfiability* $p \vDash \varphi$ by

$$\begin{aligned}
&p \vDash \top && \text{always}\,, \\
&p \vDash \bot && \text{never}\,, \\
&p \vDash \varphi_1 \wedge \varphi_2 && \text{if } p \vDash \varphi_1 \text{ and } p \vDash \varphi_2\,, \\
&p \vDash \varphi_1 \vee \varphi_2 && \text{if } p \vDash \varphi_1 \text{ or } p \vDash \varphi_2\,, \\
&p \vDash [a](\varphi) && \text{if } q \vDash \varphi \text{ for all } q \text{ such that } p \xrightarrow{a} q\,, \\
&p \vDash \langle a \rangle(\varphi) && \text{if } q \vDash \varphi \text{ for some } q \text{ such that } p \xrightarrow{a} q\,.
\end{aligned}$$

6.5.2 Translation

Take the base signature Σ_{base} with a base type of actions **act**, and appropriate constant symbols $a : \mathbf{act}$ for all the actions $a \in A$. We interpret **act** by A, and constant symbols a by the corresponding actions. For the base theory, we take standard first-order logic without any additional axioms.

Then, take the effect signature Σ_{eff}, consisting of operations $\mathsf{nil} : 0$, $\mathsf{act} : \mathbf{act}; 1$, and or:2; and the effect theory, given by the following equations:

$$\begin{aligned}\mathsf{or}(z_1,\mathsf{or}(z_2,z_3)) &= \mathsf{or}(\mathsf{or}(z_1,z_2),z_3)\,,\\ \mathsf{or}(z,z) &= z\,,\\ \mathsf{or}(z_1,z_2) &= \mathsf{or}(z_2,z_1)\,,\\ \mathsf{or}(z,\mathsf{nil}) &= z\,.\end{aligned}$$

We then translate each process p to a computation term $\vdash p^* : F\mathbf{0}$, given by:

$$\begin{aligned}0^* &= \mathsf{nil}\,,\\ (a.p)^* &= \mathsf{act}_a(p^*)\,,\\ (p+q)^* &= \mathsf{or}(p^*,q^*)\,.\end{aligned}$$

Take any two processes p and q, composed using the three given operations. Then, they are bisimilar, which we write as $p \simeq q$, if and only if they are provably equal in the equational theory, given by the above four equations [HM85].

We translate each property φ into a predicate $\vdash \varphi^* : \mathbf{prop}(F\mathbf{0})$, given by

$$\begin{aligned}\top^* &= (y{:}F\mathbf{0}).\ \top\,,\\ \bot^* &= (y{:}F\mathbf{0}).\ \bot\,,\\ (\varphi_1 \wedge \varphi_2)^* &= (y{:}F\mathbf{0}).\ \varphi_1^*(y) \wedge \varphi_2^*(y)\,,\\ (\varphi_1 \vee \varphi_2)^* &= (y{:}F\mathbf{0}).\ \varphi_1^*(y) \vee \varphi_2^*(y)\,,\\ ([a](\varphi))^* &= [\mathsf{or}]((y_1{:}F\mathbf{0}, y_2{:}F\mathbf{0}).\ [\mathsf{act}_a](\varphi^*)(y_1))\,,\\ (\langle a\rangle(\varphi))^* &= \langle\mathsf{or}\rangle((y_1{:}F\mathbf{0}, y_2{:}F\mathbf{0}).\ \langle\mathsf{act}_a\rangle(\varphi^*)(y_1))\,.\end{aligned}$$

To show the derivability of translated judgements of Hennessy-Milner logic, we first have to prove a couple of technical lemmas.

Lemma 6.17 *If we have $p \xrightarrow{a} q$ for processes p and q, there exists a process r such that*

$$\vdash_L p^* =_{F\mathbf{0}} (a.q + r)^*\,.$$

Proof We proceed by induction on the derivation of $p \xrightarrow{a} q$. If $a.p \xrightarrow{a} p$, we have $\vdash_L a.p^* =_{F\mathbf{0}} (a.p + 0)^*$. If $p + q \xrightarrow{a} r$ because $p \xrightarrow{a} r$, we get $\vdash_L p^* =_{F\mathbf{0}} (a.r + s)^*$ by the induction hypothesis, hence $\vdash_L (p+q)^* =_{F\mathbf{0}} (a.r + (s+q))^*$. In the other case, we proceed in the same way. □

Lemma 6.18 *For any process $p = \sum_{i=1}^{n} a_i.p_i$ and an action a we have*

$$y_1{:}\boldsymbol{F0}, y_2{:}\boldsymbol{F0} \mid p^* =_{\boldsymbol{F0}} \mathsf{or}(\mathsf{act}_a(y_1), y_2) \vdash_L \bigvee_{a_i = a} y_1 =_{\boldsymbol{F0}} p_i^* \, .$$

Proof We proceed by a proof by contradiction inside the logic. We are going to assume that both $p^* =_{\boldsymbol{F0}} \mathsf{or}(\mathsf{act}_a(y_1), y_2)$ and $\bigwedge_{a_i=a} \neg(y =_{\boldsymbol{F0}} p_i^*)$ hold. Next, using the free model principle, we are going to construct a (rather contrived) model of the effect theory. Then, we are going to show that the induced homomorphism from the initial model maps p^* and $\mathsf{or}(\mathsf{act}_a(y_1), y_2)$ into different elements, which is in contradiction with the assumptions.

To gain some intuition, we describe the model before employing the free model principle. First, take an action w that does not occur in p. Then, for the carrier of the model, take the (finite) set of all bisimulation equivalence classes $[q_1 + \cdots + q_m]$, where each q_j is either a subterm of p or of the form $w.\mathsf{nil}$. This set has evident semi-lattice with a zero structure, while $a.[q_1 + \cdots + q_m]$ is defined to be $[a.(q_1 + \cdots + q_m)]$ if $a.(q_1 + \cdots + q_m)$ is a subterm of p and $[w.\mathsf{nil}]$ otherwise.

The same construction in the logic goes as follows. Let $\underline{\tau}$ be the computation type $\boldsymbol{F}\sum_{i=1}^{|UM|} \mathbf{1}$, where UM is the carrier set of the above model. We are going to label closed terms by the appropriate equivalence classes $[q_1 + \cdots + q_m]$. Using nested pattern matching constructs and injections, we first define the operations on $\sum_{i=1}^{|UM|} \mathbf{1}$. Using sequencing, we extend those to operations on $\boldsymbol{F}\sum_{i=1}^{|UM|} \mathbf{1}$. The definition of those operations on terms other than returned values is a bit arbitrary, but that is not important because Proposition 6.6 allows us to ignore them.

Next, take the unique homomorphism $\hat{y}$ from $\boldsymbol{F0}$ to M, which extends the zero map from $\mathbf{0}$ to M. Then, the free model principle yields $\vdash_L \hat{y}p^* =_{\underline{\tau}} \mathsf{return}[p]$ and

$$\vdash_L \hat{y}\mathsf{or}(\mathsf{act}_a(y_1), y_2) =_{\underline{\tau}} t_{\mathsf{or}}(t_{\mathsf{act}_a}(\hat{y}y_1), \hat{y}y_2) \, ,$$

hence

$$\vdash_L \mathsf{return}[p] =_{\underline{\tau}} t_{\mathsf{or}}(t_{\mathsf{act}_a}(\hat{y}y_1), \hat{y}y_2) \, .$$

Now, as the two sides are equal, we use the definition of t_{act_a} and t_{or} together with the standard rules for value sums to show that $\hat{y}y_1$ has to be equal to $[p_i]$ for some $a_i = a$. However, this is in contradiction with our assumption. □

Theorem 6.19 *Take a process p and a property φ. Then, $p \vDash \varphi$ holds if and only if $\vdash_L \varphi^*(p^*)$ holds.*

Proof First, assume that $p \vDash \varphi$ holds and proceed by induction on φ. The cases when $\varphi = \top$ and $\varphi = \bot$ are trivial. If $\varphi = \varphi_1 \wedge \varphi_2$, we have $p \vDash \varphi$ if and only if $p \vDash \varphi_1$ and $p \vDash \varphi_2$. By induction hypothesis, we get $\vdash_L \varphi_1^*(p^*)$ and $\vdash_L \varphi_2^*(p^*)$, hence also $\vdash_L \varphi^*(p^*)$. For the disjunctive case, we proceed similarly.

If $\varphi = [a](\varphi')$, we have $p \simeq \sum_{i=1}^n a_i.p_i$ for some actions a_i and processes p_i, such that $p_i \vDash \varphi'$ if $a = a_i$. By Lemma 6.18, we get

$$y{:}F\mathbf{0}, y'{:}F\mathbf{0} \mid p^* =_{F\mathbf{0}} \mathsf{or}(\mathsf{act}_a(y), y') \vdash_L \bigvee_{a_i = a} y =_{F\mathbf{0}} p_i^* \,,$$

which together with the induction hypothesis implies

$$y{:}F\mathbf{0}, y'{:}F\mathbf{0} \mid p^* =_{F\mathbf{0}} \mathsf{or}(\mathsf{act}_a(y), y') \vdash_L \varphi'^*(y) \,,$$

which is equivalent to $\vdash_L \varphi^*(p^*)$.

And if $\varphi = \langle a \rangle(\varphi')$ holds, there exists a process q such that $p \xrightarrow{a} q$ and $q \vDash \varphi'$. Then, by Lemma 6.17, there exists a process r such that $\vdash_L p^* =_{F\mathbf{0}} a.q + r^*$, while from the induction hypothesis, we get $\vdash_L \varphi^*(q^*)$. Together, this implies $\vdash_L \varphi^*(P^*)$.

Next, assume $\vdash_L \varphi^*(p^*)$. From soundness of interpretation, we get that $[\![\varphi^*(p^*)]\!] = \mathbf{1}$. Let us show by induction on φ that $[\![\varphi^*(p^*)]\!] = \mathbf{1}$ implies $p \vDash \varphi$. The cases with truth, falsehood, conjunction, and disjunction are immediate. In the case when $\varphi = \bot$, we use the fact that the effect theory is consistent to show that $[\![\varphi^*(p^*)]\!]$ is the empty set.

If $\varphi = [a](\varphi')$, take an arbitrary q such that $p \xrightarrow{a} q$. By Lemma 6.17, there exists a process r such that $\vdash_L p^* =_{F\mathbf{0}} (a.q + r)^*$. From the soundness of interpretation, we get $[\![p^* =_{F\mathbf{0}} (a.q + r)^*]\!] = \mathbf{1}$, which together with $[\![\varphi^*(p^*)]\!] = \mathbf{1}$ implies $[\![\varphi'^*(q^*)]\!] = \mathbf{1}$. By the induction hypothesis, we get $q \vDash \varphi'$, and so $p \vDash \varphi$.

Finally, if $\varphi = \langle a \rangle(\varphi')$, the soundness of interpretation implies that there exist two processes q and r such that

$$\begin{aligned} [\![p^* =_{F\mathbf{0}} \mathsf{or}(\mathsf{act}_a(q^*), r^*)]\!] &= \mathbf{1} \,, \\ [\![\varphi'^*(q^*)]\!] &= \mathbf{1} \,, \end{aligned}$$

hence $p \simeq a.q + r$ and $q \vDash \varphi'$, therefore $p \vDash \varphi$. □

6.6 Evaluation logic

Evaluation logic [Pit91] was introduced by Pitts as a way of reasoning about computations in terms of values they return. We are, however, not going to focus on this logic, but on one of Moggi's suggested refinements [Mog95], because its semantics is uniform and as such comparable to ours, and because the axioms for necessity are stronger than the ones of Pitts. Moggi also suggested a more general variant of evaluation logic [Mog94], however we omit it from our discussion because it has a weaker set of axioms and a semantics that does not apply to ω-cpos.

6.6.1 Definition

Evaluation logic builds on Moggi's computational metalanguage [Mog91], but fixes a single monad T. Given a signature Σ_{t} of base types β, the types σ are given by:

$$\sigma ::= \beta \mid \mathbf{1} \mid \sigma_1 \times \sigma_2 \mid T\sigma \mid \sigma_1 \to \sigma_2 \,.$$

Next, given a signature Σ_{f} of function symbols $\mathsf{f}{:}\sigma_1 \to \sigma_2$, the terms t are given by:

$$t ::= x \mid \mathsf{f}(t) \mid [t] \mid \star \mid \langle t_1, t_2 \rangle \mid \pi_1(t) \mid \pi_2(t) \mid \mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, t' \mid \lambda x{:}\sigma.\, t \mid tt' \,.$$

Terms are typed as $\Gamma \vdash_{\mathrm{ev}} t{:}\sigma$ in a context $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ according to the same rules as in computational λ-calculus, with the following exception:

$$\frac{\Gamma \vdash_{\mathrm{ev}} t{:}T\sigma \qquad \Gamma, x{:}\sigma \vdash_{\mathrm{ev}} t'{:}T\sigma'}{\Gamma \vdash_{\mathrm{ev}} \mathsf{let}\, x\, \mathsf{be}\, t\, \mathsf{in}\, t'{:}T\sigma'}$$

Note that the meaning behind $t{:}\sigma$ differs. In the computational λ-calculus, t is a computation that returns values of type σ, while in the computational metalanguage, it is a value of type $T\sigma$.

Then, building on the computational metalanguage, the propositions φ of evaluation logic are given by the following grammar:

$$\varphi ::= t_1 = t_2 \mid \top \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x{:}\sigma.\, \varphi \mid [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi) \,,$$

where in the last two propositions, x is bound in φ.

Informally, the *(necessity) evaluation modality* $[\mathsf{let}\, x\, \mathsf{be}\, t](\varphi)$ states that every value computed by the computation term t satisfies φ. For example, if the effect at hand is nondeterminism, then $[\mathsf{let}\, x\, \mathsf{be}\, t](\varphi)$ holds if and only if all values computed by t satisfy φ; if it is exceptions, then $[\mathsf{let}\, x\, \mathsf{be}\, t](\varphi)$ holds if and only if t satisfies φ when it does not raise an exception.

In addition to the necessity modality, Pitts's evaluation logic [Pit91] gives a possibility modality $\langle \mathsf{let}\, x\, \mathsf{be}\, t\rangle(\varphi)$, which states that t can return a value that satisfies φ. We follow Moggi and omit it from our discussion, as we can define it as

$$\langle \mathsf{let}\, x\, \mathsf{be}\, t\rangle(\varphi) =_{\mathrm{def}} \neg[\mathsf{let}\, x\, \mathsf{be}\, t](\neg\varphi)$$

and translate it accordingly. For the same reasons, we omit the discussion of falsehood, disjunction, and existential quantifier.

Proposition are typed as $\Gamma \vdash_{\mathrm{ev}} \varphi{:}\mathbf{prop}$ in the standard way, with the modality typed by:

$$\frac{\Gamma \vdash_{\mathrm{ev}} t{:}T\sigma \qquad \Gamma, x{:}\sigma \vdash_{\mathrm{ev}} \varphi{:}\mathbf{prop}}{\Gamma \vdash_{\mathrm{ev}} [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi){:}\mathbf{prop}}\,.$$

Judgements of evaluation logic are of the form $\Gamma \mid \Psi \vdash \varphi$, where Ψ is a set of *hypotheses* $\Gamma \vdash \varphi_i : \mathbf{prop}$ and $\Gamma \vdash \varphi : \mathbf{prop}$ is the *conclusion*. If a judgement $\Gamma \mid \Psi \vdash \varphi$ is derivable in evaluation logic, we write $\Gamma \mid \Psi \vdash_{\mathrm{ev}} \varphi$. We write an *equivalence* $\Gamma \mid \Psi \dashv\vdash_{\mathrm{ev}} \varphi$ when both $\Gamma \mid \Psi \vdash_{\mathrm{ev}} \varphi$ holds and $\Gamma \mid \varphi \vdash_{\mathrm{ev}} \varphi_i$ holds for any $\varphi_i \in \Psi$.

Omitting the standard reasoning rules of first-order logic (as presented in Section 2.2) together with weakening, the reasoning rules for modalities are:

- ($\Box$-$\vdash$)

$$\frac{\Gamma, x{:}\sigma \mid \Psi, \varphi \vdash_{\mathrm{ev}} \varphi'}{\Gamma \mid \Psi, [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi) \vdash_{\mathrm{ev}} [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi')}\,,$$

- ($\Box$-η)

$$\frac{}{\Gamma, x{:}\sigma \mid \varphi \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\, x\, \mathsf{be}\, [x]](\varphi)}\,,$$

- ($\Box$-let)

$$\frac{}{\Gamma \mid [\mathsf{let}\, x_1\, \mathsf{be}\, t_1][\mathsf{let}\, x_2\, \mathsf{be}\, t_2](\varphi) \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\, x_2\, \mathsf{be}\, (\mathsf{let}\, x_1\, \mathsf{be}\, t_1\, \mathsf{in}\, t_2)](\varphi)}\,,$$

- ($\Box$-$\top$)

$$\frac{}{\Gamma \mid \top \vdash_{\mathrm{ev}} [\mathsf{let}\, x\, \mathsf{be}\, t](\top)}\,,$$

- ($\Box$-$\wedge$)

$$\frac{}{\Gamma \mid [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi_1), [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi_2) \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\, x\, \mathsf{be}\, t](\varphi_1 \wedge \varphi_2)}\,,$$

- (□-⇒)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x\,\mathsf{be}\,t](\varphi_1 \Rightarrow \varphi_2) \dashv\vdash_{\mathrm{ev}} \varphi_1 \Rightarrow [\mathsf{let}\,x\,\mathsf{be}\,t](\varphi_2)},$$

- (□-∀)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x\,\mathsf{be}\,t](\forall x' {:} \sigma'.\, \varphi) \dashv\vdash_{\mathrm{ev}} \forall x' {:} \sigma'.\, [\mathsf{let}\,x\,\mathsf{be}\,t](\varphi)},$$

- (□-=)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x\,\mathsf{be}\,t](t_1 = t_2) \vdash_{\mathrm{ev}} (\mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}\,t_1) = (\mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}\,t_2)}.$$

The above rules are slightly different than the ones given by Moggi. In particular, we omit the rules

- (□-μ)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x_1\,\mathsf{be}\,t_1][\mathsf{let}\,x_2\,\mathsf{be}\,x_1](\varphi) \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\,x_2\,\mathsf{be}\,(\mathsf{let}\,x_1\,\mathsf{be}\,t_1\,\mathsf{in}\,x_1)](\varphi)},$$

- (□-T)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x_1\,\mathsf{be}\,t_1]\varphi[t_2/x_2] \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\,x_2\,\mathsf{be}\,(\mathsf{let}\,x_1\,\mathsf{be}\,t_1\,\mathsf{in}\,[t_2])](\varphi)},$$

- (□-st)

$$\frac{}{\Gamma \mid [\mathsf{let}\,x\,\mathsf{be}\,t]\varphi[\langle x_1, x_2\rangle/x'] \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\,x'\,\mathsf{be}(\mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}[\langle x_1, x_2\rangle])](\varphi)},$$

as they are derivable. First, (□-μ) is an instance of (□-let) if we set t_2 to be x_1. Then, if we substitute t_2 for x in (□-η) and apply (□-$\vdash$) in both directions, we get

$$\frac{}{\Gamma \mid [\mathsf{let}\,x_1\,\mathsf{be}\,t_1]\varphi[t_2/x_2] \dashv\vdash_{\mathrm{ev}} [\mathsf{let}\,x_1\,\mathsf{be}\,t_1][\mathsf{let}\,x_2\,\mathsf{be}\,[t_2]]\varphi},$$

which together with (□-μ) implies (□-T). Finally, (□-st) is an instance of (□-T) if we set t_2 to be $\langle x_1, x_2\rangle$.

Note that Moggi gives a variant of some rules, marked by an asterisk, where a one-way judgement is replaced by an equivalence. Wherever possible, we gave a variant with an equivalence, therefore we omit the asterisks from the labels.

6.6.2 Translation

As in the translation of the computational λ-calculus, we limit our translation to ones where signatures are restricted to pure functions $\mathsf{f}:\prod\boldsymbol{\beta}\to\beta$ and generic effects $\mathsf{gen}_{\mathsf{op}}:\prod\boldsymbol{\beta}\to T(\prod\boldsymbol{\alpha})$. Furthermore, since function types of our logic have computation types for codomains, we restrict the function types in the metalanguage to ones of the form $\sigma\to T\sigma'$. An alternative would be to add a value type of functions $\sigma\to\sigma'$ with value type codomains to our logic.

Then, we take a base signature, consisting of all the base types $\beta\in\Sigma_{\mathrm{t}}$, and of function symbols $\mathsf{f}:(\boldsymbol{\beta})\to\beta$ for each pure function $\mathsf{f}:\prod\boldsymbol{\beta}\to\beta\in\Sigma_{\mathrm{f}}$. Next, we take an effect signature, consisting of operations $\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}$ for each generic effect $\mathsf{gen}_{\mathsf{op}}:\prod\boldsymbol{\beta}\to T(\prod\boldsymbol{\alpha})\in\Sigma_{\mathrm{f}}$, and the empty effect theory

We translate types of evaluation logic to value types as:

$$\begin{aligned}
\beta^* &= \beta\,,\\
(\sigma_1\times\sigma_2)^* &= \sigma_1^*\times\sigma_2^*\,,\\
\mathbf{1}^* &= \mathbf{1}\,,\\
(\sigma\to T\sigma')^* &= U(\sigma^*\to F\sigma'^*)\,,\\
(T\sigma)^* &= UF\sigma^*\,,
\end{aligned}$$

terms to value terms as:

$$\begin{aligned}
x^* &= x\,,\\
\mathsf{f}(t)^* &= \mathsf{f}(t^*)\,,\\
\mathsf{gen}_{\mathsf{op}}(t)^* &= \mathsf{thunk}(\mathsf{gen}_{\mathsf{op}}(t^*))\,,\\
[t]^* &= \mathsf{thunk}(\mathsf{return}\,t^*)\,,\\
\star^* &= \star\,,\\
\langle t_1,t_2\rangle^* &= \langle t_1^*,t_2^*\rangle\,,\\
\pi_1(t)^* &= \mathsf{fst}\,t^*\,,\\
\pi_2(t)^* &= \mathsf{snd}\,t^*\,,\\
(\mathsf{let}\,x\,\mathsf{be}\,t\,\mathsf{in}\,t')^* &= \mathsf{thunk}((\mathsf{force}\,t^*)\,\mathsf{to}\,x.\,(\mathsf{force}\,t'^*))\,,\\
(\lambda x{:}\sigma.\,t)^* &= \mathsf{thunk}\,\lambda x{:}\sigma^*.\,t^*\,,\\
(tt')^* &= \mathsf{force}(t^*)t'^*\,,
\end{aligned}$$

and propositions as:

$$\begin{aligned}
(t_1 = t_2)^* &= (t_1^* = t_2^*)\,, \\
\top^* &= \top\,, \\
(\varphi_1 \wedge \varphi_2)^* &= \varphi_1^* \wedge \varphi_2^*\,, \\
(\varphi_1 \Rightarrow \varphi_2)^* &= \varphi_1^* \Rightarrow \varphi_2^*\,, \\
(\forall x{:}\sigma.\, \varphi)^* &= \forall x{:}\sigma^*.\, \varphi^*\,, \\
([\mathsf{let}\, x\, \mathsf{be}\, t](\varphi))^* &= \Box([\downarrow](x{:}\sigma^*).\, \varphi^*)(\mathsf{force}\, t^*)\,.
\end{aligned}$$

To prove the soundness of the translation, we are going to use the fact that the satisfiability of a necessity modality propagates to all subcomputations and returned values.

Lemma 6.20 *The* pureness balance proposition

$$\Gamma; \Delta; \Pi \mid \Box([\downarrow]\pi)(\mathsf{return}\, v) \vdash_L \pi(v)$$

and the operation balance proposition

$$\Gamma; \Delta; \Pi \mid \Box([\downarrow]\pi)(\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i) \vdash_L \bigwedge_i \forall \boldsymbol{x}_i : \boldsymbol{\alpha}_i.\, \Box([\downarrow]\pi)(t_i)$$

both hold.

Proof The proof is immediate from the definition of the modalities. □

Proposition 6.21 *The translations of* ($\Box$-$\vdash$), ($\Box$-$\top$), ($\Box$-$\wedge$), ($\Box$-$\Rightarrow$), ($\Box$-$\forall$), *and* ($\Box$-$=$) *are derivable in our logic.*

Proof

- To show that the translation of ($\Box$-$\vdash$) is derivable, we observe that

  $$\forall x{:}\sigma.\, \pi(x) \Rightarrow \pi'(x)$$

  implies

  $$\forall y{:}F\sigma.\, ([\downarrow]\pi)(y) \Rightarrow ([\downarrow]\pi')(y)\,.$$

  Together with Proposition 6.12, this implies

  $$\forall y{:}F\sigma.\, \Box([\downarrow]\pi)(y) \Rightarrow \Box([\downarrow]\pi')(y)$$

  and so the translation of ($\Box$-$\vdash$).

- The proofs of the cases ($\Box$-$\top$), ($\Box$-$\wedge$), ($\Box$-$\Rightarrow$), and ($\Box$-$\forall$) are all alike, so let us give only the one for ($\Box$-$\wedge$). Set $\pi_i =_{\mathrm{def}} (x{:}\sigma^*).\ \varphi_i^*$ for $i = 1,2$. Then, ($\Box$-$\wedge$) gets translated as

  $$\Gamma^* \mid \Box([\downarrow]\pi_1)(\mathsf{force}\,t^*), \Box([\downarrow]\pi_2)(\mathsf{force}\,t^*) \vdash \Box([\downarrow](x{:}\sigma^*).\ (\pi_1(x) \wedge \pi_2(x)))(\mathsf{force}\,t^*)$$

  and its converse. First, assume that $\Box([\downarrow]\pi_1)(y)$ and $\Box([\downarrow]\pi_2)(y)$ hold. To show the conclusion, we use the introduction rule for the predicate fixed point defining $\Box$. Hence, we need to prove that

  $$[\downarrow]((x{:}\sigma^*).\ \pi_1(x) \wedge \pi_2(x))(y)$$

  and

  $$[-]((y{:}F\sigma^*).\ \Box([\downarrow]\pi_1)(y) \wedge \Box([\downarrow]\pi_2)(y))(y)$$

  hold. As $\Box([\downarrow]\pi_i)(y)$ entails $([\downarrow]\pi_i)(y)$, we immediately get the first condition. For the second condition, $\Box([\downarrow]\pi_i)(y)$ entails $[-](\Box([\downarrow]\pi_i))(y)$. It is straightforward to check that the modality $[-]$ commutes with the conjunction, hence

  $$[-](\Box([\downarrow]\pi_1))(y) \wedge [-](\Box([\downarrow]\pi_2))(y)$$

  entails

  $$[-]((y{:}F\sigma^*).\ \Box([\downarrow]\pi_1)(y) \wedge \Box([\downarrow]\pi_2)(y))(y)\,.$$

  The other direction follows from ($\Box$-$\vdash$) and the elimination rules for conjunction.

- ($\Box$-$=$) gets translated as

  $$\begin{gathered}\Gamma^* \mid \Box([\downarrow](x{:}\sigma^*).\ \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*)(\mathsf{force}\,t^*) \\ \vdash t^*\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^* = \mathsf{force}\,t^*\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*\,.\end{gathered}$$

  To show derivability, take

  $$\begin{aligned}\pi =_{\mathrm{def}} (y{:}F\sigma).\ &\Box([\downarrow](x{:}\sigma^*).\ \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*)(y) \\ &\Rightarrow y\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^* = y\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*\end{aligned}$$

  and proceed by the principle of computational induction on y.

- For the base case, we have

$$
\begin{aligned}
&\Box([\downarrow](x{:}\sigma^*).\ \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*)(\mathsf{return}\,x)\\
&\Rightarrow \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*\\
&\text{(by the pureness balance proposition)}\\
&\Rightarrow \mathsf{return}\,x\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^* = \mathsf{return}\,x\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*\\
&\text{(by β-reduction for sequencing)}\,.
\end{aligned}
$$

- For the step case, take $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$ and assume that $\pi(y_i\boldsymbol{x}_i)$ holds for all $y_i{:}(\boldsymbol{\alpha}_i)\to \boldsymbol{F}\sigma$ and $\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i$. Then, we have

$$
\begin{aligned}
&\Box([\downarrow](x{:}\sigma^*).\ \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*)(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i)\\
&\Rightarrow \bigwedge_i \forall \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ (\Box([\downarrow](x{:}\sigma^*).\ \mathsf{force}\,t_1^* = \mathsf{force}\,t_2^*))(y_i\boldsymbol{x}_i)\\
&\text{(by the operation balance proposition)}\\
&\Rightarrow \bigwedge_i \forall \boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^* = y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*\\
&\text{(by the induction hypothesis)}\\
&\Rightarrow \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^*)_i = \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*)_i\\
&\text{(by congruence)}\\
&\Rightarrow \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_1^* = \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i\,\mathsf{to}\,x.\ \mathsf{force}\,t_2^*\\
&\text{(by algebraicity of operations)}\,.
\end{aligned}
$$

Hence, $\pi(y)$ holds for all $y{:}\boldsymbol{F}\sigma$. We finish the proof by substituting $\mathsf{force}\,t^*$ for y.

□

Unfortunately, the translations of axioms (□-η) and (□-let) are not derivable in our logic.

Example 6.22 Take the base theory $\mathcal{T}_{\mathrm{base}}$ of natural numbers, equipped with a relation symbol $\mathsf{iszero}{:}(\mathbf{nat})$ and the expected axioms, and the effect theory $\mathcal{T}_{\mathrm{eff}}$ for state. Then, we have

$$
\begin{aligned}
&\vdash_L \mathsf{return}\,0\\
&=_{\boldsymbol{F}\mathbf{int}} \mathsf{lookup}_\ell(d.\ \mathsf{return}\,0)\\
&=_{\boldsymbol{F}\mathbf{int}} \mathsf{lookup}_\ell(d.\ \mathsf{lookup}_\ell(d'.\ \mathsf{if}\,d = d'\,\mathsf{then}\,\mathsf{return}\,0\,\mathsf{else}\,\mathsf{return}\,1))\,.
\end{aligned}
$$

Now, although we have $\vdash_{\mathcal{T}_{\text{base}}} \mathsf{iszero}(0)$ holds, the proposition

$$([\mathsf{let}\, x\, \mathsf{be}\, \mathsf{return}\, 0](\mathsf{iszero}(x)))^*$$

fails because $\vdash_L \mathsf{iszero}(1)$ fails, while the translation of ($\Box$-η) states that $\mathsf{iszero}(x)$ holds for all possible returned values x.

Hence the translation of the axiom ($\Box$-η) is even refutable in our logic for some base and effect theories. The reason for the failure of translation lies in the fact that we could contaminate a term with values that would never be returned in its evaluation. For this reason, we limit ourselves to a well-behaved subset of equational theories where such contamination is not possible.

Remark 6.23 An alternative is to define the translation of the evaluation modality as

$$\begin{aligned}&\mu X\!:\!(F\sigma).\,(y\!:\!F\sigma).\,[\downarrow](\pi)(y)\vee\\ &\qquad\bigvee_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n\in\Sigma_{\text{eff}}}\langle\mathsf{op}\rangle((\boldsymbol{x}\!:\!\boldsymbol{\beta}, y_1\!:\!\boldsymbol{\alpha}_1\to\underline{\tau},\ldots,y_n\!:\!\boldsymbol{\alpha}_n\to\underline{\tau}).\,\bigwedge_{i=1}^{n}\forall\boldsymbol{x}_i\!:\!\boldsymbol{\alpha}_i.\,X(y_i\boldsymbol{x}_i))(y)\,.\end{aligned}$$

Then, the translation of $[\mathsf{let}\, x\, \mathsf{be}\, t](\varphi)$ states that there exists a computation, equivalent to t, whose leaves all satisfy φ. This does give the correct semantics for all effects mentioned above. Unfortunately, it does not help us in the embrace of evaluation logic, as most of the translated axioms end in a form where no progress can be made by applying rules of the logic.

6.6.3 Balanced theories

Definition 6.24 An equation $Z \vdash e_1 = e_2$ of a (countable) single-sorted equational theory $\mathcal{T}$ is *balanced*, if a variable $z \in Z$ occurs in e_1 if and only if it occurs in e_2. A theory $\mathcal{T}$ is *balanced* if all the equations $Z \vdash_{\mathcal{T}} e_1 = e_2$ are balanced.

For a given model of the base theory $\mathcal{T}_{\text{base}}$, an effect theory $\mathcal{T}_{\text{eff}}$ is *balanced* if the induced countable equational theory, constructed as in Section 4.1, is balanced.

The theories for exceptions, nondeterminism, interactive input and output and time are all balanced. Furthermore, if two theories are balanced, so is their sum and tensor product [HPP06]. Also, note that the reasoning rules of equational theories preserve balanced equations, hence a theory is balanced if all its axioms are balanced.

Remark 6.25 The restriction to balanced theories is related to the restriction to *simple monads* in HASCASL [SM04] and global evaluation logic [GSM06]. In particular, if an equational theory is balanced, the induced monad is simple, but note that the state monad is simple even though the effect theory for state is not balanced.

Also, monads induced by balanced theories correspond exactly to *collection monads* [Man98], which are used to model various collection classes such as lists, sets, trees, or bags.

Lemma 6.26 *If an equational theory $\mathcal{T}$ is balanced, it is equationally consistent.*

Proof If the equational theory $\mathcal{T}$ is balanced, we cannot have $x_1, x_2 \vdash_{\mathcal{T}} x_1 = x_2$ as both x_1 and x_2 occur on only one side of the equation. □

Recall that for a given set X, the set TX is constructed as the set of equivalence classes $[\vdash e]$ of closed terms $\vdash e$, built using operations from Σ_{eff} and generators from X, modulo the equality of the infinitary equational theory $\mathcal{T}$, generated by the effect theory $\mathcal{T}_{\text{eff}}$.

For such a term e, we can define its *support* $\operatorname{supp} e \subseteq X$ by

$$\operatorname{supp} a = \{a\} \quad (a \in X)$$
$$\operatorname{supp} \mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n) = \bigcup_i \operatorname{supp} f_i \,,$$

where for a function $f \colon [\![\boldsymbol{\alpha}]\!] \to TX$, we define $\operatorname{supp} f =_{\text{def}} \bigcup_{\boldsymbol{a} \in [\![\boldsymbol{\alpha}]\!]} \operatorname{supp} f \boldsymbol{a}$.

If the effect theory is balanced, we have $\operatorname{supp} e = \operatorname{supp} e'$ whenever $\vdash_{\mathcal{T}} e = e'$. Thus, for $\delta \in TX$, we can define $\operatorname{supp} \delta$ to be $\operatorname{supp} e$ for any e such that $[\![e]\!] = \delta$. By the construction of TX, such e always exists.

Lemma 6.27 *Take a set X and $A \subseteq X$. Then, for any $[e] \in TX$, we have*

$$\operatorname{supp}[e] \subseteq A \qquad \textit{if and only if} \qquad [e] \in TA \,.$$

Proof We proceed by induction on the structure of terms of the infinitary equational theory.

- For the base step, we have $\operatorname{supp}[a] = \{a\}$, hence $\operatorname{supp}[a] \subseteq A$ if and only if $a \in A$, which is equivalent to $[a] \in TA$ as the theory is consistent.

- For the induction step, assume that $\operatorname{supp}[\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)] \subseteq A$. Since we have $\operatorname{supp}[\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)] = \bigcup_i \operatorname{supp}[f_i]$, we have $\operatorname{supp}[f_i] \subseteq A$ for all i. By the induction hypothesis, we get $[f_i \boldsymbol{a}_i] \in TA$ for all $1 \leqslant i \leqslant n$ and $\boldsymbol{a}_i \in [\![\boldsymbol{\alpha}_i]\!]$, hence $[\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)] \in TA$. On the other hand, if we have

$$[\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)] \in TA$$

then by the construction of $\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)$, it follows that

$$\operatorname{supp}[\mathsf{op}(\boldsymbol{a}; f_1, \dots, f_n)] \subseteq A\ .$$

□

Lemma 6.28 *Assume that the effect theory $\mathfrak{T}_{\text{eff}}$ is balanced, and take a predicate*

$$\Gamma; \Delta; \Pi \vdash \pi : \mathbf{prop}(\sigma)\ .$$

Then for any $\langle \boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{U} \rangle \in [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$, the set $[\![\Box([\downarrow]\pi)]\!](\boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{U})$ is equal to the free model $T([\![\pi]\!](\boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{U}))$ of the equational theory, generated by $\mathfrak{T}_{\text{eff}}$, over the set of generators $[\![\pi]\!](\boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{U})$.

Proof To simplify the proof, we write $[\![-]\!]$ instead of $[\![-]\!](\boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{U})$. Since

$$\Box([\downarrow]\pi) =_{\text{def}} \nu P : (F\sigma).\ (y : F\sigma).\ ([\downarrow]\pi)(y) \wedge [-](P)(y)\,,$$

the set $[\![\Box([\downarrow]\pi)]\!]$ is defined to be the largest set $U \subseteq T[\![\sigma]\!]$ such that:

1. if $\eta_{[\![\sigma]\!]}(a) \in U$ for some $a \in [\![\sigma]\!]$, then $a \in [\![\pi]\!]$;
2. if for any $\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n$, any $\boldsymbol{a} \in [\![\boldsymbol{\beta}]\!]$, and any $f_i : [\![\boldsymbol{\alpha}_i]\!] \to T[\![\sigma]\!]$, we have $\mathsf{op}_{F\sigma}(\boldsymbol{a}; f_1, \dots, f_n) \in U$, then $f_i(a_i) \in U$ for any $1 \leqslant i \leqslant n$ and all $a_i \in [\![\boldsymbol{\alpha}_i]\!]$.

We first show that $T[\![\pi]\!]$ satisfies the two conditions. Since $\eta_{[\![\sigma]\!]}(a) = [a]$, we have $\eta_{[\![\sigma]\!]}(a) \in T[\![\pi]\!]$. From Lemma 6.27, it follows that $\operatorname{supp} \eta_{[\![\sigma]\!]}(a) = \{a\} \subseteq [\![\pi]\!]$, hence $a \in [\![\pi]\!]$.

Then, assuming $\mathsf{op}_{F\sigma}(\boldsymbol{a}; f_1, \dots, f_n) \in T[\![\pi]\!]$, we get

$$\operatorname{supp} \mathsf{op}_{F\sigma}(\boldsymbol{a}; f_1, \dots, f_n) = \bigcup_i \operatorname{supp} f_i \subseteq [\![\pi]\!]\ ,$$

which implies $\operatorname{supp} f_i \subseteq [\![\pi]\!]$, hence $f_i a_i \in T[\![\pi]\!]$ for all $a_i \in [\![\boldsymbol{\alpha}_i]\!]$ and $1 \leqslant i \leqslant n$.

On the other hand, take a set $U \subseteq T[\![\sigma]\!]$ that satisfies the above two conditions. Let us show by induction on e that $[e] \in U$ implies $[e] \in T[\![\pi]\!]$ for any $[e] \in T[\![\sigma]\!]$.

First, if $[a] = \eta_{[\![\sigma]\!]}(a) \in U$ holds for some $a \in [\![\sigma]\!]$, the first condition implies $a \in [\![\pi]\!]$, which further implies $[a] \in T[\![\pi]\!]$.

Next, take $[\mathsf{op}(\boldsymbol{a};f_1,\ldots,f_n)] \in U$. The second condition implies $[f_i(a_i)] \in U$ for any $1 \leqslant i \leqslant n$ and all $\boldsymbol{a}_i \in [\![\boldsymbol{\alpha}_i]\!]$. By the induction hypothesis, we get that $[f_i(\boldsymbol{a}_i)] \in T[\![\pi]\!]$ for any $1 \leqslant i \leqslant n$ and all $\boldsymbol{a}_i \in [\![\boldsymbol{\alpha}_i]\!]$. And as $T[\![\pi]\!]$ is a free model, we get

$$\mathsf{op}_{F\sigma}(\boldsymbol{a};[f_1],\ldots,[f_n]) = [\mathsf{op}(\boldsymbol{a};f_1,\ldots,f_n)] \in T[\![\pi]\!] \ .$$

Thus, $T[\![\pi]\!]$ is the greatest set that satisfies the above two conditions and is as such equivalent to $[\![\Box([\downarrow]\pi)]\!]$. □

Corollary 6.29 *If the effect theory $\mathcal{T}_{\text{eff}}$ is balanced, the converse*

$$\Gamma;\Delta;\Pi \mid \pi(v) \vdash_L \Box([\downarrow]\pi)(\mathsf{return}\, v)$$

of the pureness balance proposition, and the converse

$$\Gamma;\Delta;\Pi \mid \bigwedge_i \forall \boldsymbol{x}_i : \boldsymbol{\alpha}_i .\, \Box([\downarrow]\pi)(t_i) \vdash_L \Box([\downarrow]\pi)(\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\, t_i)_i)$$

of the operation balance proposition, both defined in Lemma 6.20, are both sound.

Remark 6.30 Note that the converse of the pureness balance proposition entails the consistency proposition

$$\forall x_1, x_2 : \sigma.\ \mathsf{return}\, x_1 = \mathsf{return}\, x_2 \Rightarrow x_1 = x_2 \,,$$

given in Proposition 5.5, if one takes $v =_{\text{def}} x_2$ and $\pi =_{\text{def}} (x_1 : \sigma).\, x_1 =_\sigma x_2$.

Since the balance propositions are sound for balanced effect theories, we may add them to our logic to obtain a translation of evaluation logic.

Theorem 6.31 *Assume the converses of pureness and operation balance propositions. Then, if $\Gamma \mid \Psi \vdash_{ev} \varphi$ holds, so does $\Gamma^* \mid \Psi^* \vdash_L \varphi^*$.*

Proof We proceed by induction on the derivation of $\Gamma \mid \Psi \vdash_{\text{ev}} \varphi$.

- The proofs of the cases ($\Box$-$\vdash$), ($\Box$-$\top$), ($\Box$-$\wedge$), ($\Box$-$\Rightarrow$), ($\Box$-$\forall$), ($\Box$-$=$) were already treated in Proposition 6.21.
- ($\Box$-η) gets translated as

$$\Gamma^* \mid \varphi^*(x) \vdash \Box([\downarrow]\varphi^*)(\mathsf{return}\, x)$$

  and its converse. The second direction follows from the pureness balance proposition, while the first direction follows from its converse.

- (□-let) gets translated as

$$\begin{aligned}&\Gamma^* \mid \Box([\downarrow](x_1{:}\sigma_1).\ \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*))(\mathsf{force}\,t_1^*)\\ &\quad\vdash \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_1^*\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)\end{aligned}$$

  and its converse. To show the equivalence, take

$$\begin{aligned}\pi =_{\mathrm{def}} (y{:}F\sigma).\ &\Box([\downarrow](x_1{:}\sigma_1).\ \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*))(y)\\ &\Leftrightarrow \Box([\downarrow]\varphi^*)(y\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)\end{aligned}$$

  and proceed by the principle of computational induction on y.

  - For the base case, we have

$$\begin{aligned}&\Box([\downarrow](x_1{:}\sigma_1).\ \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*))(\mathsf{return}\,x)\\ &\Leftrightarrow \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*)[x/x_1]\\ &\text{(by the balance proposition and its converse)}\\ &\Leftrightarrow \Box([\downarrow]\varphi^*(\mathsf{force}\,t_2^*[x/x_1]))\\ &\text{(since } x_1 \text{ is not free in } \varphi)\\ &\Leftrightarrow \Box([\downarrow]\varphi^*)(\mathsf{return}\,x\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)\\ &\text{(by } \beta\text{-reduction for sequencing)}\,.\end{aligned}$$

  - For the step case, take $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$ and assume that $\pi(y_i\boldsymbol{x}_i)$ holds for all $y_i{:}(\boldsymbol{\alpha}_i)\to F\sigma$ and $\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i$. Then, we have

$$\begin{aligned}&\Box([\downarrow](x_1{:}\sigma_1).\ \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*))(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i)\\ &\Leftrightarrow \bigwedge_i \forall\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \Box([\downarrow](x_1{:}\sigma_1).\ \Box([\downarrow]\varphi^*)(\mathsf{force}\,t_2^*))(y_i\boldsymbol{x}_i)\\ &\text{(by the balance proposition and its converse)}\\ &\Leftrightarrow \bigwedge_i \forall\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\ \Box([\downarrow]\varphi^*)(y_i\boldsymbol{x}_i\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)\\ &\text{(by the induction hypothesis)}\\ &\Leftrightarrow \Box([\downarrow]\varphi^*)(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)_i)\\ &\text{(by the balance proposition and its converse)}\\ &\Leftrightarrow \Box([\downarrow]\varphi^*)(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i\boldsymbol{x}_i)_i\,\mathsf{to}\,x_1.\ \mathsf{force}\,t_2^*)\\ &\text{(by algebraicity of operations)}\,.\end{aligned}$$

  Hence, $\pi(y)$ holds for all $y{:}F\sigma$. We finish the proof by substituting $\mathsf{force}\,t^*$ for y.

□

Chapter 7

Handlers of algebraic effects

We now turn to the other aim of the thesis: giving an algebraic treatment of exception handlers. We shall first relate each exception handler to a model of the effect theory for exceptions, and describe handling in terms of induced homomorphisms from the free model. Then, we shall generalise this treatment to other algebraic effects, note the difficulties that arise with the generalisation, and suggest a possible solution.

7.1 Exception handlers

We start our study with exception handlers, because they are an established concept [BK01, Lev06b] and also because exceptions provide the simplest example of algebraic effects. Note that in this section, we do not present a calculus of handlers, but work informally in the context of the a-calculus in order to focus on the exposition of ideas.

Recall that we represent a finite set of exceptions E by a finite effect signature Σ_{eff}, consisting of a nullary operation symbol $\mathsf{raise}_{\mathsf{exc}}\!:\!0$ for each $\mathsf{exc} \in E$, and by the trivial effect theory $\mathcal{T}_{\mathsf{eff}}$. Then, a computation term $\Gamma \vdash t\!:\!F\sigma$ is interpreted by a map $[\![t]\!] : [\![\Gamma]\!] \to [\![\sigma]\!] + E$. In particular, we have $[\![\mathsf{return}\, v]\!] = \mathrm{in}_1 \circ [\![v]\!]$ and $[\![\mathsf{raise}_{\mathsf{exc}}]\!] = \mathrm{in}_2 \circ \mathrm{k}_{\mathsf{exc}}$, where $\mathrm{k}_{\mathsf{exc}} : [\![\Gamma]\!] \to E$ is the constant map that maps each $a \in [\![\Gamma]\!]$ to $\mathsf{exc} \in E$.

Let us extend computation terms with an *exception handling construct*

$$\mathsf{try}\, t\, \mathsf{with}\, \{\mathsf{exc}_i = t_i\}_i \ ,$$

where t is the *handled term* and $\{\mathsf{exc}_i = t_i\}_i$ is the *handler*. The handling construct evaluates as t, unless its evaluation raises an exception exc_i for some i, in which case, it evaluates as t_i.

The behaviour of the handling construct is thus described by the following:

$$\begin{aligned}
&\Gamma \vdash \mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i =_{F\sigma} \mathsf{return}\,v\,,\\
&\Gamma \vdash \mathsf{try}\,\mathsf{raise}_{\mathsf{exc}_j}\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i =_{F\sigma} t_j\,,\\
&\Gamma \vdash \mathsf{try}\,\mathsf{raise}_{\mathsf{exc}}\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i =_{F\sigma} \mathsf{raise}_{\mathsf{exc}} \quad (\mathsf{exc} \notin \{\mathsf{exc}_i\}_i)\,.
\end{aligned}$$

Semantically, for any $\gamma \in [\![\Gamma]\!]$, the family of computation terms $\{t_i\}_i$ provides a model M_γ of the effect theory $\mathcal{T}_{\mathrm{eff}}$ on the set $[\![\sigma]\!] + E$. For all i, the operation symbol $\mathsf{raise}_{\mathsf{exc}_i}$ is interpreted by $[\![t_i]\!](\gamma)$, while other operation symbols $\mathsf{raise}_{\mathsf{exc}}$ are interpreted as in the free model.

Then, the above equations respectively state that the handling construct extends the inclusion of values, and that it acts homomorphically on exceptions. Hence, the interpretation of the handling construct is given by

$$[\![\Gamma \vdash \mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i]\!] =_{\mathrm{def}} \gamma \mapsto (U\theta_\gamma)([\![t]\!](\gamma))\,,$$

where θ_γ is the unique induced homomorphism $F[\![\sigma]\!] \to M_\gamma$, for which the following diagram commutes.

$$\begin{tikzcd}
{[\![\sigma]\!]} \arrow[d, "\eta_{[\![\sigma]\!]}"'] \arrow[dr, "\eta_{[\![\sigma]\!]}"] & \\
UF[\![\sigma]\!] \arrow[r, "U\theta_\gamma"'] & UM_\gamma = UF[\![\sigma]\!]
\end{tikzcd}$$

Benton and Kennedy [BK01] generalised the handling construct to one of the form (for reasons discussed in Section 7.3, we use a different syntax)

$$\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t'\,,$$

typed as

$$\frac{\Gamma \vdash t{:}F\sigma \qquad \Gamma \vdash t_i{:}\underline{\tau} \quad (\text{for all } i) \qquad \Gamma, x{:}\sigma \vdash t'{:}\underline{\tau}}{\Gamma \vdash \mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t'{:}\underline{\tau}}\,.$$

Here, an exception exc_i may be handled by a computation term t_i of any given type $\underline{\tau}$, while returned values are "handled" with the computation term t'. As remarked by the authors, this handling construct allows a more concise programming style, program optimisations, and a stack-free small-step operational semantics [BK01].

The behaviour of the extended handling construct is described by:

$$\Gamma \vdash \mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t' =_{\underline{\tau}} t'[v/x]\,,$$

$$\Gamma \vdash \mathsf{try}\,\mathsf{raise}_{\mathsf{exc}_j}\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t' =_{\underline{\tau}} t_j\,,$$

$$\Gamma \vdash \mathsf{try}\,\mathsf{raise}_{\mathsf{exc}}\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t' =_{\underline{\tau}} \mathsf{raise}_{\mathsf{exc}} \quad (\mathsf{exc} \notin \{\mathsf{exc}_i\}_i)\,.$$

Semantically, for any $\boldsymbol{\gamma} \in \llbracket\Gamma\rrbracket$, the family of computation terms $\{\mathsf{exc}_i = t_i\}_i$ provides a model $M_{\boldsymbol{\gamma}}$ of the effect theory as before, except that its carrier is now $U\llbracket\underline{\tau}\rrbracket$. Then, the above two equations again state that the handling construct acts homomorphically on exceptions, but now extends the map $\llbracket t'\rrbracket(\boldsymbol{\gamma},-)\colon \llbracket\sigma\rrbracket \to M_{\boldsymbol{\gamma}}$. Hence, the interpretation of the handling construct is given by

$$\llbracket\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t'\rrbracket =_{\mathrm{def}} \boldsymbol{\gamma} \mapsto (U\theta_{\boldsymbol{\gamma}})(\llbracket t\rrbracket(\boldsymbol{\gamma}))\,,$$

where θ_γ is the unique induced homomorphism $F\llbracket\sigma\rrbracket \to M_\gamma$, for which the following diagram commutes.

$$\begin{tikzcd}
\llbracket\sigma\rrbracket \arrow[d, "\eta_{\llbracket\sigma\rrbracket}"'] \arrow[dr, "\llbracket t'\rrbracket(\boldsymbol{\gamma},-)"] & \\
UF\llbracket\sigma\rrbracket \arrow[r, "U\theta_{\boldsymbol{\gamma}}"'] & UM_{\boldsymbol{\gamma}} = U\llbracket\underline{\tau}\rrbracket
\end{tikzcd}$$

The universal property of the free model $F\llbracket\sigma\rrbracket$ states that *each* homomorphism $F\llbracket\sigma\rrbracket \to M$ is induced by a map $\llbracket\sigma\rrbracket \to UM$, hence Benton and Kennedy's approach to the handling construct is the most general one possible from the algebraic point of view.

We can now see how to give handlers of other algebraic effects. An exception handler on a type $\underline{\tau}$ is given by a computation term $t_i : \underline{\tau}$ for each exception $\mathsf{exc}_i \in E$ we wish to handle. Likewise, a generalised handler is given by a computation term $y_1 : \underline{\tau}, \ldots, y_n : \underline{\tau} \vdash t_{\mathsf{op}} : \underline{\tau}$ for each operation symbol $\mathsf{op} : n \in \Sigma_{\mathrm{eff}}$ we wish to handle (we are still considering the standard single-sorted equational theory at this point). As the effect signature is finite, the handler can contain all the operation symbols. For those that we do not wish to handle, we set $t_{\mathsf{op}} = \mathsf{op}(y_1, \ldots, y_n)$.

The behaviour of the handling construct is determined by two equations:

$$\Gamma \vdash \mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\,\{\mathsf{op}(y_i)_i = t_{\mathsf{op}}\}_{\mathsf{op}:n\in\Sigma_{\mathrm{eff}}} =_{F\sigma} \mathsf{return}\,v\,,$$

$$\Gamma \vdash \mathsf{try}\,\mathsf{op}(t_j)_j\,\mathsf{with}\,\{\mathsf{op}(y_i)_i = t_{\mathsf{op}}\}_{\mathsf{op}:n\in\Sigma_{\mathrm{eff}}}$$

$$=_{F\sigma} t_{\mathsf{op}}[\mathsf{try}\,t_j\,\mathsf{with}\{\mathsf{op}(y_i)_i = t_{\mathsf{op}}\}_{\mathsf{op}:n\in\Sigma_{\mathrm{eff}}}/y_j]_j\,,$$

and we could analogously describe the extended handling construct of Benton and Kennedy.

However, in order to interpret handlers with models as before, the replacement maps on $U[\![\underline{\tau}]\!]$ have to satisfy the equations of the effect theory. We say that a handler has to be *correct*, a notion we shall define more precisely after we give the calculus.

Exception handlers are usually described and used within the same language: for each exception, we give a replacement computation term, which can contain further exception handlers. This is possible because the effect theory for exceptions is trivial, hence every exception handler is correct. The same holds for time, or interactive input and output. But for arbitrary algebraic effects, this causes a complex interdependence between the typing relation and the equational logic, which guarantees that all well-typed handlers have a sound interpretation.

Even worse, determining whether a given assignment of computation terms to operation symbols yields a model of the effect theory is in general undecidable [PP09]. The proof of this fact is long, technical, and not too relevant to the development in the rest of the thesis, hence we shall not pursue it.

Instead of equipping the calculus with a mechanism that ensures the correctness of handlers, we are going to provide two languages: one to describe handlers, and another one to use them. In this way the selection of correct handlers is delegated to the meta-level. This approach is similar to the one taken in HASKELL, where a programmer is given access to the effects only through the use of built-in monads, which had their monadic laws checked by the language designers.

The two languages will be very similar and will both build on the term language of the logic. To describe handlers, we are going to extend the term language with type variables; this will allow handlers to be polymorphic. To use handlers, we are going to extend the term language with the handling construct. When referring to the first language, we are going to use the "handler" qualifier, for example *handler value types*, while for the second language, we shall use the "program " qualifier. Furthermore, we shall use the same meta-variables for their types and terms, with any ambiguities resolved from the context.

7.2 The handler language

Definition 7.1 Take a countably infinite set of *type variables* X. The sets of *handler value types* σ, *handler value terms* v, and *handler computation terms* t are given by the same grammar as their counterparts in term language of the logic, given in Section 4.2, while *handler computation types* $\underline{\tau}$ are given by the grammar for computation types, extended by

$$\underline{\tau} ::= X \mid \cdots .$$

Contexts and typing judgements are given exactly as in the term language of the logic. Note that for any assignment of computation types to type variables and for any handler type or term, we get a counterpart in the term language. For example, given a handler computation term t, we get a computation term $t[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ by substituting each type variable X_i by a computation type $\underline{\tau}_i$.

A handler is given by a handling term for each operation, which may further depend on additional parameters, passed to the handler by the handling construct. This can also be used to pass additional handling constructs to a handler.

Definition 7.2 The set of *handlers* h is given by the following grammar:

$$h ::= (\boldsymbol{x}_p : \boldsymbol{\sigma}_p ; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}} .$$

The handlers are typed as $\vdash h : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\ \textbf{handler}$ by the following rule:

$$\frac{\boldsymbol{x}_p : \boldsymbol{\sigma}_p, \boldsymbol{x} : \boldsymbol{\beta};\ \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p, (y_i : \boldsymbol{\alpha}_i \to \underline{\tau})_i \vdash t_{\mathsf{op}} : \underline{\tau} \quad (\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in \Sigma_{\mathsf{eff}})}{\vdash (\boldsymbol{x}_p : \boldsymbol{\sigma}_p ; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}} : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\ \textbf{handler}} .$$

When $\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}$ is omitted, we assume that $t_{\mathsf{op}} = \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\ y_i(\boldsymbol{x}_i))_i$, so that op is not handled, so to speak. We omit the semicolon in handlers when either $\boldsymbol{\sigma}_p$ or $\underline{\boldsymbol{\tau}}_p$ is empty, or write $h : \underline{\tau}\ \textbf{handler}$ when both are empty.

A handler may be polymorphic because type variables may occur in $\boldsymbol{\sigma}_p$, $\underline{\boldsymbol{\tau}}_p$ or $\underline{\tau}$. We say that a handler $\vdash h : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\ \textbf{handler}$ is *uniform* when $\underline{\tau} = X$, and *parametrically uniform* when $\underline{\tau} = \sigma \to X$ for some type variable X.

For each model of the logic (a model of the base theory that maps arity types to countable sets), and each assignment ρ of models $\rho(X)$ to type variables X, handler value types σ are interpreted by sets $[\![\sigma]\!]_\rho$, given by $[\![U\underline{\tau}]\!]_\rho = U[\![\underline{\tau}]\!]_\rho$ and in the obvious way for the other handler value types, while handler computation types $\underline{\tau}$ are interpreted by models $[\![\underline{\tau}]\!]_\rho$, given by

$$[\![X]\!]_\rho = \rho(X) \qquad [\![F\sigma]\!]_\rho = F[\![\sigma]\!]_\rho \qquad [\![\underline{\mathbf{1}}]\!]_\rho = \mathbf{1}$$
$$[\![\underline{\tau}_1 \times \underline{\tau}_2]\!]_\rho = [\![\underline{\tau}_1]\!]_\rho \times [\![\underline{\tau}_2]\!]_\rho \qquad [\![\sigma \to \underline{\tau}]\!]_\rho = [\![\underline{\tau}]\!]_\rho^{[\![\sigma]\!]_\rho}\,.$$

Then, contexts and terms are interpreted as in the term language of the logic, while a handler $h : (\boldsymbol{\sigma}_p, \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}$ **handler** is interpreted by a parameterised family $[\![h]\!]_\rho(\boldsymbol{\gamma}_p, \boldsymbol{\delta}_p)$ of interpretations of Σ_{eff}, where $\boldsymbol{\gamma}_p \in [\![\boldsymbol{\sigma}]\!]_\rho$ and $\boldsymbol{\delta}_p \in U[\![\underline{\boldsymbol{\tau}}]\!]_\rho$. Each such interpretation gives a model of the effect signature Σ_{eff} with a carrier $U[\![\underline{\tau}]\!]_\rho$, and operation symbols $\mathsf{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$ interpreted with maps

$$\mathsf{op}_h(\boldsymbol{\gamma}, \boldsymbol{\delta}) =_{\mathrm{def}} [\![t_{\mathsf{op}}]\!](\boldsymbol{\gamma}_p, \boldsymbol{\gamma}, \boldsymbol{\delta}_p, \boldsymbol{\delta})\,.$$

We say that h is *correct (with respect to a given model of* $\mathcal{T}_{\mathrm{eff}}$*)* if for all assignments ρ, and for all $\boldsymbol{\gamma}_p \in [\![\boldsymbol{\sigma}]\!]_\rho$ and $\boldsymbol{\delta}_p \in U[\![\underline{\boldsymbol{\tau}}]\!]_\rho$, the interpretation $[\![h]\!]_\rho(\boldsymbol{\gamma}_p, \boldsymbol{\delta}_p)$ defines a model of the effect theory $\mathcal{T}_{\mathrm{eff}}$ on $U[\![\underline{\tau}]\!]_\rho$.

7.3 The program language

Now, assume a *handler signature* Σ_{hand} of *handler symbols*

$$H : (\boldsymbol{\sigma}_p; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\ \textbf{handler}\,,$$

each of which will be interpreted by a correct handler. The sets of *program value types* σ, *program computation types* $\underline{\tau}$, and *program value terms* v are given by the same grammar as their counterparts in term language of the logic, given in Section 4.2, while *handler computation terms* t are given by the grammar for computation types, extended by the handling construct

$$t ::= \mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p; \boldsymbol{t}_p)\, \mathsf{as}\, x : \sigma\, \mathsf{in}\, t' \mid \cdots\,.$$

Similar as in sequencing, the handled term t does not have a unique type, hence the variable x has to have an explicit type (although we often omit it) in order to ensure the uniqueness of the typing derivation.

When the full handling construct is not necessary, we write $\mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p; \boldsymbol{t}_p)$ instead of $\mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p; \boldsymbol{t}_p)\, \mathsf{as}\, x : \sigma\, \mathsf{in}\, \mathsf{return}\, x$.

Note that the syntax of the handling construct differs somewhat from the one introduced by Benton and Kennedy [BK01], which is of the form

$$\mathsf{try}\, x \Leftarrow t\, \mathsf{in}\, t'\, \mathsf{unless}\, \{\mathsf{exc}_i \Rightarrow t_i\}_i\,.$$

As noted by the authors themselves, this syntax is confusing when used in programming: it does not make it obvious that t is handled whereas t' is not. This is particularly confusing when t' is large, which is the usual case in programming. An alternative they propose is

$$\mathsf{try}\, x \Leftarrow t\, \mathsf{unless}\, \{\mathsf{exc}_i \Rightarrow t_i\}_i\, \mathsf{in}\, t' \;,$$

but then it is not obvious that x is bound in t', but not in the handler. The syntax of our construct addresses those issues and clarifies the order of evaluation: after t is handled with H, the results are bound to x and used in t'.

Given an assignment of program computation types $\underline{\tau}_i$ to type variables X_i, the handling construct for a handler symbol $H:(\boldsymbol{\sigma}_p;\underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\, \mathbf{handler} \in \Sigma_{\text{hand}}$ is typed by

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash t{:}F\sigma \\ \Gamma;\Delta \vdash \boldsymbol{v}_p:\boldsymbol{\sigma}_p[\underline{\tau}_i/X_i]_i \qquad \Gamma;\Delta \vdash \boldsymbol{t}_p:\underline{\boldsymbol{\tau}}_p[\underline{\tau}_i/X_i]_i \qquad \Gamma, x{:}\sigma;\, \Delta \vdash t':\underline{\tau}[\underline{\tau}_i/X_i]_i\end{array}}{\Gamma;\Delta \vdash \mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p;\boldsymbol{t}_p)\, \mathsf{as}\, x{:}\sigma\, \mathsf{in}\, t':\underline{\tau}[\underline{\tau}_i/X_i]_i}\;.$$

Note that since the types are given inductively, we can show by induction on the structure of $\underline{\tau}$ that $\underline{\tau}[\underline{\tau}_i/X_i]_i = \underline{\tau}[\underline{\tau}'_i/X_i]_i$ implies $\underline{\tau}_i = \underline{\tau}'_i$ for all variables X_i that occur in $\underline{\tau}$.

To interpret the handling construct, we assume given a *handler definition* $\mathcal{H}$, mapping each handler symbol $H:(\boldsymbol{\sigma}_p;\underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\, \mathbf{handler} \in \Sigma_{\text{hand}}$ to a correct handler $\vdash \mathcal{H}(H):(\boldsymbol{\sigma}_p;\underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\, \mathbf{handler}$. Then, the handling construct is interpreted as follows.

Take $\boldsymbol{\gamma} \in [\![\Gamma]\!]$ and $\boldsymbol{\delta} \in [\![\Delta]\!]$ and let ρ be an assignment that maps X_i to $[\![\underline{\tau}_i]\!]$. Since each handler $\mathcal{H}(H)$ is correct, the Σ_{eff} interpretation

$$[\![\mathcal{H}(H)]\!]_\rho([\![\boldsymbol{v}_p]\!]_\rho(\boldsymbol{\gamma},\boldsymbol{\delta}), [\![\boldsymbol{t}_p]\!]_\rho(\boldsymbol{\gamma},\boldsymbol{\delta}))$$

gives a model M of the effect theory $\mathcal{T}_{\text{eff}}$ with carrier $U[\![\underline{\tau}]\!]_\rho$. By the universality of the free model $F[\![\sigma]\!]$, there is a unique homomorphism $\theta_{\boldsymbol{\gamma},\boldsymbol{\delta}}: F[\![\sigma]\!] \to M$ extending $[\![t']\!](\boldsymbol{\gamma},-,\boldsymbol{\delta})$, in the sense that the following diagram commutes:

$$\begin{tikzcd}
{[\![\sigma]\!]} \arrow[d, "\eta_{[\![\sigma]\!]}"'] \arrow[dr, "{[\![t']\!](\boldsymbol{\gamma},-,\boldsymbol{\delta})}"] & \\
UF[\![\sigma]\!] \arrow[r, "U\theta_{\boldsymbol{\gamma},\boldsymbol{\delta}}"] & UM
\end{tikzcd}$$

The handling construct $\Gamma;\Delta \vdash \mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p;\boldsymbol{t}_p)\, \mathsf{as}\, x\!:\!\sigma\, \mathsf{in}\, t' : \underline{\tau}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ is then interpreted by the map

$$\langle \boldsymbol{\gamma}, \boldsymbol{\delta} \rangle \mapsto \theta_{\boldsymbol{\gamma},\boldsymbol{\delta}}([\![t]\!](\boldsymbol{\gamma},\boldsymbol{\delta}))\colon [\![\Gamma]\!] \times [\![\Delta]\!] \to U[\![\underline{\tau}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]]\!] ,$$

where $[\![\underline{\tau}]\!]_\rho$ and $[\![\underline{\tau}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]]\!]$ are equal by the definition of ρ.

Note that in a single term, multiple instances of a single handler symbol can be used with different assignments of computation types to type variables. To do so, we only have to start applying the substitutions to the innermost handling constructs.

7.4 Examples

7.4.1 Exceptions

The standard uniform exception handler

$$H_{\mathsf{exc}}\!:\!(\mathbf{exc} \to X) \to X\ \mathbf{handler}$$

is given by

$$(y\!:\!\mathbf{exc} \to X).\ \{\mathsf{raise}_e = ye\}\!:\!(\mathbf{exc} \to X) \to X\ \mathbf{handler}\,.$$

Since the effect theory for exceptions is trivial, it is correct.

Benton and Kennedy's construct $\mathsf{try}\, x \Leftarrow t\, \mathsf{in}\, t'\, \mathsf{unless}\, \{e_1 \Rightarrow t_1 \mid \cdots \mid e_n \Rightarrow t_n\}$ can be written as $\mathsf{try}\, t\, \mathsf{with}\, H_{\mathsf{exc}}(t_{\mathsf{exc}})\, \mathsf{as}\, x\!:\!\sigma\, \mathsf{in}\, t'$ for suitable σ and $t_{\mathsf{exc}}\!:\!\mathbf{exc} \to \underline{\tau}$. Our construct is actually a bit more general because E may be infinite and because we are in a call-by-push-value framework rather than a call-by-value one.

7.4.2 Stream redirection

Shell processes in UNIX-like operating systems communicate with the user using input and output streams, usually connected to a keyboard and a terminal window. Such streams can be redirected to other processes so that simple commands can be combined into more powerful ones.

One case is the redirection proc > outfile, which takes the output stream of a process proc, and writes it to to a file outfile for later, whereas proc > /dev/null writes it to the *null device*, which acts as a black hole. This effectively discards the standard output stream, for example when the user does not want to fill his terminal window with unnecessary output.

Another case is the pipe proc1|proc2, where the output of proc1 is fed to the input of proc2. This is crucial in implementing the UNIX philosophy of "writing programs that do one thing and do it well," because it allows the chaining of multiple simple processes into a more complex one. For example

```
latex thesis.tex|grep full
```

runs LaTeX on the file thesis.tex, and passes its output messages to the command grep, which selects only the lines that contain full. Hence, the user is not presented with the whole LaTeX output, but only with the warnings about overfull and underfull boxes.

A simpler example of a pipe is yes|proc. The command yes outputs an infinite stream made of a predetermined character (the default one being y). Such pipe then gives a way of routinely confirming a series of actions, for example deleting a large number of files. This is not always the best way, since commands usually provide a safer means of doing the same thing, but is often useful when they do not.

If we represent interactive input and output as in Example 4.11, then for a computation t, we can express yes|t > /dev/null by $\mathsf{try}\, t\, \mathsf{with}\, H_{\mathrm{red}}(\mathsf{y})$, where the handler symbol $H_{\mathrm{red}} : (\mathbf{char}) \to X\ \mathbf{handler}$ is defined to be

$$(a : \mathbf{char}).\ \{\mathsf{out}_c(y) = y, \mathsf{in}(y) = y(a)\}\,.$$

This again gives a correct handler because the effect theory for interactive input and output is trivial.

7.4.3 CCS renaming and hiding

The representation of CCS processes [Mil80] from Section 6.5 treats only processes, given by action prefix and sum. However, process renaming and hiding can be represented by handlers. This makes sense, as operations construct the effects, while renaming and hiding are deconstructive operations.

The renaming $t[b/a]$ can be written as $\mathsf{try}\, t\, \mathsf{with}\, H_{\mathrm{ren}}(a,b)$, where the handler symbol $H_{\mathrm{ren}} : (\mathbf{act}, \mathbf{act}) \to F\mathbf{0}\ \mathbf{handler}$ is defined by:

$$H_{\mathrm{ren}} = (a : \mathbf{act}, b : \mathbf{act}).\ \{\mathsf{act}_{a'}(y) = \mathsf{if}\ a' = a\ \mathsf{then}\ \mathsf{act}_b(y)\ \mathsf{else}\ \mathsf{act}_{a'}(y)\}\,,$$

while hiding $t \backslash \{a\}$ can be written as $\mathsf{try}\, t\, \mathsf{with}\, H_{\text{hid}}(a)$, where the handler symbol $H_{\text{hid}} : (\mathbf{act}) \to F\mathbf{0}\,\mathbf{handler}$ is defined by:

$$H_{\text{hid}} = (a : \mathbf{act}).\ \{\mathsf{act}_{a'}(y) = \mathsf{if}\, a' = a\, \mathsf{then}\, \mathsf{nil}\, \mathsf{else}\, \mathsf{act}_{a'}(y)\}\,.$$

Note that handling terms for nil and or are omitted, hence the two operations are handled by themselves. The equations of the effect theory for CCS refer only to nil and or, hence both handlers are correct.

However, the implementation of parallel composition in our setting remains an open problem. To observe the difficulties of using handlers in describing it, let us first take a look at the simpler case of a synchronisation operator $\|$ [vGP]. This is defined by:

$$\begin{aligned} \mathsf{nil} \parallel y &= \mathsf{nil}\,, \\ \mathsf{or}(x_1, x_2) \parallel y &= \mathsf{or}(x_1 \parallel y, x_2 \parallel y)\,, \\ \mathsf{act}_a(x) \parallel y &= x \parallel^a y\,, \end{aligned}$$

where $\|^a$ is defined by:

$$\begin{aligned} x \parallel^a \mathsf{nil} &= \mathsf{nil}\,, \\ x \parallel^a \mathsf{or}(y_1, y_2) &= \mathsf{or}(x \parallel^a y_1, x \parallel^a y_2)\,, \\ x \parallel^a \mathsf{act}_b(y) &= \begin{cases} \mathsf{act}_a(x \parallel y) & \text{if } b = a\ \ , \\ \mathsf{nil} & \text{otherwise.} \end{cases} \end{aligned}$$

Now, the difficulty is not in the fact that we have two recursive definitions, the second one being parametric; this can be resolved using mutually defined and parameter-passing handlers, given in Section 7.4.6. The difficulty lies in the fact that the operator is defined recursively on the structure of *both* its arguments, and this is not primitively recursive.

The problems with expressing the parallel composition are very much the same, except that we need to add complementary and silent actions and that its definition is a bit more complex because the parallel composition allows unsynchronised actions as well.

7.4.4 Explicit nondeterminism

The evaluation of a nondeterministic computation usually takes only one of all the possible paths. But in logic programming [CM87], we do an exhaustive search for all solutions that satisfy given constraints in the order given by the solver implementation. Such nondeterminism is represented slightly differently. We take an effect signature, consisting of operation symbols $\mathsf{fail}:0$ and $\mathsf{pick}:2$, with the effect theory consisting of the following equations, which state that the operations form a monoid:

$$\begin{aligned} z &\vdash \mathsf{pick}(z,\mathsf{fail}) = z\ , \\ z &\vdash \mathsf{pick}(\mathsf{fail},z) = z\ , \\ z_1,z_2,z_3 &\vdash \mathsf{pick}(z_1,\mathsf{pick}(z_2,z_3)) = \mathsf{pick}(\mathsf{pick}(z_1,z_2),z_3)\ . \end{aligned}$$

The free-model monad maps a set to the set of all finite sequences of its elements, which is HASKELL's nondeterminism monad [PJ03].

A user is usually presented with a way of browsing through the solutions, for example extracting all the solutions into a list. Since our calculus has no polymorphic lists (although it could easily be extended with them), we take base types α and $\mathbf{list}_\alpha$, function symbols

$$\begin{aligned} &\mathsf{nil}:\mathbf{list}_\alpha\ , && \mathsf{cons}:(\alpha,\mathbf{list}_\alpha)\to\mathbf{list}_\alpha\ , \\ &\mathsf{head}:(\mathbf{list}_\alpha)\to\alpha\ , && \mathsf{tail}:(\mathbf{list}_\alpha)\to\mathbf{list}_\alpha\ , \\ &\mathsf{append}:(\mathbf{list}_\alpha,\mathbf{list}_\alpha)\to\mathbf{list}_\alpha\ , \end{aligned}$$

and an appropriate base theory. Then, all the results of a computation of type $F\alpha$ can be extracted into a returned value of type $F\mathbf{list}_\alpha$ using the handler

$$\begin{aligned} &\{\mathsf{fail} = \mathsf{return}\,\mathsf{nil}\ , \\ &\ \mathsf{pick}(y_1,y_2) = y_1\,\mathsf{to}\,x_1\!:\!\mathbf{list}_\alpha.\ y_2\,\mathsf{to}\,x_2\!:\!\mathbf{list}_\alpha.\ \mathsf{return}\,\mathsf{append}(x_1,x_2)\}\ . \end{aligned}$$

Since append is associative and we have $\mathsf{append}(\mathsf{nil},x) = x = \mathsf{append}(x,\mathsf{nil})$, it is easy to check that the handler is correct.

7.4.5 Optimal result

Another possibility with the ordinary nondeterminism, when the datatype permits it, is to select the result, optimal to some criterion. For example, we could choose the greatest integer returned by a computation using a handler

$$\{\mathsf{or}(y_1,y_2) = y_1\,\mathsf{to}\,x_1.\ y_2\,\mathsf{to}\,x_2.\ \mathsf{return}\,\mathsf{max}(x_1,x_2)\,.\}$$

Since both max and sequencing (for nondeterminism) are associative, commutative, and idempotent, the handler is correct.

Although this handler gives an elegant and concise way of finding an optimum, it is not particularly efficient as it has to traverse all possible returned values. Even more: if we consider the usual interpretation of nondeterminism, it is not even possible to implement such a handler. A similar remark applies to the handler in Section 7.4.4. This also serves to show that equipping a calculus with a mechanism to check the correctness is not enough, as some computational limitations cannot be captured with equations, and an intervention on the meta-level is necessary.

7.4.6 Parameter-passing handlers

Sometimes, we wish to handle different instances of the same operation differently, for example to suppress output after a certain number of outputted characters. Although a handler prescribes a fixed replacement for each operation, we can use handlers on a function type $\sigma \to \underline{\tau}$ to simulate handlers on $\underline{\tau}$ that pass around a parameter of type σ.

Instead of

$$(\boldsymbol{x}_p : \boldsymbol{\sigma}_p ; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\, \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = \lambda x{:}\sigma.\, t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}} : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to (\sigma \to \underline{\tau})\, \mathbf{handler}\,.$$

where all the occurrences of y_i it t_{op} are applied to some $v : \sigma$, the changed parameter, we write

$$(\boldsymbol{x}_p : \boldsymbol{\sigma}_p ; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\, \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) @ x{:}\sigma = t'_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}} : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau} @ \sigma\, \mathbf{handler}\,,$$

where t'_{op} results from substituting $y_i @ v$ for $y_i v$ in t_{op}. We also write

$$\mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p ; \boldsymbol{t}_p) @ v\, \mathsf{as}\, x' : \sigma' @ x : \sigma\, \mathsf{in}\, t'$$

instead of

$$(\mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p ; \boldsymbol{t}_p)\, \mathsf{as}\, x' : \sigma'\, \mathsf{in}\, \lambda x{:}\sigma.\, t')v\,.$$

We could similarly simulate mutually defined handlers by handlers on product types, however there are so far no interesting examples of their use.

7.4.7 Timeout

When the evaluation of a computation takes too long, we may want to abort it and provide a predefined result instead, a behaviour called *timeout*. Take the base theory of integers, together with the relation symbol $>:(\mathbf{int},\mathbf{int})$; and an effect theory for time, described in Section 2.4.

Then, timeout can be described by a handler which passes around a parameter $T:\mathbf{int}$, which represents the amount of time we are willing to wait before we abort the evaluation and return y_p. The handler is defined by:

$$(y_p:X).\ \{\mathsf{tick}(y)@T:\mathbf{int} = \mathsf{tick}(\mathsf{if}\, T > 0\, \mathsf{then}\, y@(T-1)\, \mathsf{else}\, y_p)\}\,.$$

The handling term is wrapped in tick in order to preserve the time spent during the evaluation of the handled computation. Since the effect theory for time is trivial, we could also give a correct handler without tick wrapped around it. But then applying the handling construct to a computation, which takes a certain amount of time, results in a computation that takes no time at all, and if tick represents the actual passing of time, such handler has no known implementation. This is similar to nondeterminism as presented in Section 7.4.5, where equations fail to describe the physical limitations of the computational model.

7.4.8 Input redirection

With parameter passing, we can implement the redirection proc < infile, which feeds the contents of infile to the standard input of proc. If we extend the base theory for interactive input and output with the base type $\mathbf{list_{char}}$ and appropriate function symbols from Section 7.4.4, then a handler $H_{\mathrm{in}}:X\,@\mathbf{list_{char}}\ \mathbf{handler}$, which passes a string to the input stream of a process, is defined by

$$\{\mathsf{input}(y)@\ell:\mathbf{list_{char}} = \mathsf{if}\ \ell = \mathsf{nil}\ \mathsf{then}\ \mathsf{input}(a.\ y(a)@\mathsf{nil})\ \mathsf{else}\ y(\mathsf{head}(\ell))@\mathsf{tail}(\ell)\}\,.$$

Because the effect theory for interactive input and output is trivial, the handler is correct.

Unfortunately, there is no obvious way of implementing the pipe $t_1|t_2$: the difficulty is very much like that with the CCS parallel combinator. A possible approach, used in so called *pseudo-pipes* in DOS is to write the output of t_1 to a temporary string and then pass that string to the input of t_2. But unlike in piping, the two processes do not ran in parallel, and if the first one outputs an infinite stream of characters (like yes, for example), the evaluation never terminates.

7.4.9 Read-only state

When we have a restricted access to a database, we can describe the state operations by taking a signature, consisting of a single operation symbol $\mathsf{lookup}\colon \mathbf{loc}; \mathbf{dat}$, and the effect theory, generated by the following equations:

$$\begin{aligned}\mathsf{lookup}_\ell(d.\,\mathsf{lookup}_\ell(d'.\,z(d,d'))) &= \mathsf{lookup}_\ell(d.\,z(d,d))\,,\\ \mathsf{lookup}_\ell(d.\,z) &= z\,,\\ \mathsf{lookup}_\ell(d.\,\mathsf{lookup}_{\ell'}(d'.\,z(d,d'))) &= \mathsf{lookup}_{\ell'}(d'.\,\mathsf{lookup}_\ell(d.\,z(d,d'))) \qquad (\ell \neq \ell')\,.\end{aligned}$$

Now, even though we cannot change the state, we may run a computation, intercept all the lookups, and provide a replacement value instead. Assuming a single memory location ℓ, this may be done with a handler, defined as

$$(d\colon\mathbf{dat}).\,\{\mathsf{lookup}_\ell(y) = y(d)\}\,.$$

We are going to postpone the proof of correctness of the handler to Example 8.3, where we are going to construct equations that correspond to the ones of the effect theory and prove them in the logic. Then, the correctness will follow from the soundness of the interpretation.

7.4.10 Rollback

When a computation raises an exception while modifying the memory, for example, when a connection drops half-way through a database transaction, we want to revert all the modifications made. This behaviour is termed *rollback*.

We take the base and the effect signatures for exceptions as in Example 4.9 and state as in Example 4.13, and the effect theory for state, together with the equation $\mathsf{update}_{\ell,d}(\mathsf{raise}_{\mathsf{exc}}) = \mathsf{raise}_{\mathsf{exc}}$ for each exception $\mathsf{exc} \in E$ [HPP06]. Then, the exception handler, defined in Section 7.4.1, is no longer correct. For example, if we fix different $d_1, d_2\colon\mathbf{dat}$ and set

$$t_0 =_{\mathrm{def}} \lambda e\colon\mathbf{exc}.\,\mathsf{lookup}_\ell(d.\ \mathsf{if}\, d = d_1\, \mathsf{then}\, \mathsf{return}\, d_1\, \mathsf{else}\, \mathsf{return}\, d_2)\,,$$

then

$$\mathsf{update}_{\ell,d_1}(\mathsf{try}\, \mathsf{update}_{\ell,d_2}(\mathsf{raise}_{\mathsf{exc}})\, \mathsf{with}\, t_0)$$

evaluates to

$$\mathsf{update}_{\ell,d_2}(\mathsf{return}\, d_2)\,,$$

while

$$\mathsf{update}_{\ell,d_1}(\mathsf{try}\,\mathsf{raise}_{\mathsf{exc}}\,\mathsf{with}\, t_0)$$

evaluates to

$$\mathsf{update}_{\ell,d_1}(\mathsf{return}\, d_1)\,.$$

Instead, we use a handler that resets the state to a predefined value, for example the one before the evaluation, if an exception occurs. In the case of a single location ℓ, the handler $H_{\text{rollback}} : (\mathbf{dat}; \mathbf{exc} \to X) \to X$ **handler** is defined by

$$(x_p : \mathbf{dat}; y_p : \mathbf{exc} \to X).\, \{\mathsf{raise}_e = \mathsf{update}_{\ell,x_p}(y_p\, e)\}\,.$$

Then, we may store the initial state and use the handler on $t{:}F\sigma$ by

$$!\ell\,\mathsf{to}\, d_p.\ \mathsf{try}\, t\, \mathsf{with}\, H_{\text{rollback}}(d_p; t_p)$$

for some $t_p : \mathbf{exc} \to F\sigma$.

An alternative is to use a parametrically uniform handler, which does not modify the memory, but keeps track of all the changes to the location ℓ in the parameter, and commits them only after a computation has returned a value, meaning that no exceptions have been raised. This handler is:

$$\begin{aligned} &(y_p : \mathbf{exc} \to X).\, \{ \\ &\qquad\qquad \mathsf{lookup}(y)@d = y\, d@d \\ &\qquad\qquad \mathsf{update}_{d'}(y)@d = y@d' \\ &\qquad\qquad \mathsf{raise}_{\mathsf{exc}}()@d = y_p\, \mathsf{exc} \\ &\qquad \} \end{aligned}$$

and is used on programs t by

$$!\ell\,\mathsf{to}\, d_p.\ \mathsf{try}\, t\, \mathsf{with}\, H(t_p)@d_p\, \mathsf{as}\, x@d\, \mathsf{in}\, \mathsf{update}_d(\mathsf{return}\, x)\,.$$

The second handler can be generalised to the case of multiple locations. Then, the parameter is a list of all the modified locations, together with their current state. Computationally, this is preferable to passing around the entire state. However, such handler is not correct and behaves as expected only when its initial parameter is an empty list. A similar failure occurs with a handler whose passed parameter is a function that rolls back the state in case of an exception [PP09].

As in the case of read-only state, we are going to postpone the proof of correctness of the rollback handlers to Examples 8.4 and 8.5.

Chapter 8

Extending the logic with handlers

Finally, we bring the studies of logic for algebraic effects and handlers of algebraic effects together, and adapt the logic to account for handlers. This is relatively straightforward: we interpret handlers with models of the effect theory and the handling construct with induced homomorphisms, and those notions are already present in the logic.

Just like we introduced two calculi for handlers, one to describe them and one to use them, we give two separate logics, one to reason about handlers, and one to reason about computations that use them.

For the first logic, referred to as *the handler logic*, we take the logic for algebraic effects, except that instead of value and computation types and terms, we take their handler counterparts. That is, the only difference from the logic for algebraic effects is that the types of the handler logic may contain type variables. Note that this polymorphism serves only as a parametrisation of the logic, and adds no new reasoning rules.

We write $\Gamma;\Delta;\Pi \mid \Psi \vdash_H \varphi$ when the judgement $\Gamma;\Delta;\Pi \mid \Psi \vdash \varphi$ is derivable in the handler logic.

Proposition 8.1 *Assume that*

$$\Gamma;\Delta;\Pi \mid \Psi \vdash_H \varphi$$

holds. Then, for any assignment of computation types $\underline{\boldsymbol{\tau}}$ to type variables $\boldsymbol{X}$, we have

$$\Gamma[\underline{\boldsymbol{\tau}}/\boldsymbol{X}];\Delta[\underline{\boldsymbol{\tau}}/\boldsymbol{X}];\Pi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}] \mid \Psi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}] \vdash_L \varphi[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]\,,$$

where the substitution $[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$ in contexts and propositions is defined in the obvious way.

Proof We proceed by an induction on the derivation of $\Gamma;\Delta;\Pi \mid \Psi \vdash_H \varphi$. Since the handler logic contains no additional reasoning rules, the proof is straightforward. □

For the second logic, referred to as *the program logic*, we also take an extension of logic for algebraic effects, parametric with regard to the same base and effect theories as the handler logic. In addition, we parametrise it with a handler signature Σ_{hand} and a handler definition $\mathcal{H}$. Then, we extend computation terms with the handling construct, which we describe with two equational schemas: the first one states that the interpretation of the handling construct extends a given map, while the second one states that it is homomorphic. Finally, we add an axiom that internalises the correctness of handlers in the logic. We write $\Gamma;\Delta;\Pi \mid \Psi \vdash_P \varphi$ for judgements, derivable in the program logic.

For any given handler symbol $H : (\boldsymbol{\sigma}_p; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}$ **handler**, handler definition

$$\mathcal{H}(H) = (\boldsymbol{x}_p : \boldsymbol{\sigma}_p; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}},$$

assignment of program computation types $\underline{\boldsymbol{\tau}}$ to type variables $\boldsymbol{X}$, program value terms $\Gamma;\Delta \vdash \boldsymbol{v}_p : \boldsymbol{\sigma}_p[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$, and program computation terms $\Gamma;\Delta \vdash \boldsymbol{t}_p : \underline{\boldsymbol{\tau}}_p[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]$, the additional schemas are:

- β-equivalence for the handling construct:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_P \mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x{:}\sigma\,\mathsf{in}\,t =_{\underline{\tau}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]} t[v/x]},$$

- homomorphic nature of the handling construct:

$$\frac{}{\begin{gathered}\Gamma;\Delta;\Pi \mid \Psi \vdash_P \mathsf{try}\,\mathsf{op}_{\boldsymbol{v}}(\boldsymbol{x}_i.\,t_i)_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x{:}\sigma\,\mathsf{in}\,t \\ =_{\underline{\tau}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]} t_{\mathsf{op}}[\boldsymbol{v}_p/\boldsymbol{x}_p, \boldsymbol{v}/\boldsymbol{x}, \boldsymbol{t}_p/\boldsymbol{y}_p, (\lambda \boldsymbol{x}_i : \boldsymbol{\alpha}_i.\ \mathsf{try}\,t_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x{:}\sigma\,\mathsf{in}\,t/y_i)_i][\underline{\boldsymbol{\tau}}/\boldsymbol{X}]\end{gathered}},$$

- correctness of handlers:

$$\frac{\mathcal{H}(H) = (\boldsymbol{x}_p : \boldsymbol{\sigma}_p; \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}}}{\Gamma, \boldsymbol{x}_p : \boldsymbol{\sigma}_p;\ \Delta, \boldsymbol{y}_p : \underline{\boldsymbol{\tau}}_p;\ \Pi \mid \Psi \vdash_P \{t_{\mathsf{op}}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}}\,\mathsf{models}\,\mathcal{T}_{\mathsf{eff}}}.$$

As the handler definition $\mathcal{H}$ maps each handler symbol to a correct handler, the three schemas are sound. It seems as if the last schema could be a consequence of inheritance from the effect theory and the homomorphic nature of the handling construct. However, this is not the case.

For example, take the effect theory for nondeterminism and a handler H, defined by $\mathcal{H}(H) = t_{\mathsf{or}}$ for some $y_1{:}F\sigma, y_2{:}F\sigma \vdash t_{\mathsf{or}}{:}F\sigma$. Inheriting the idempotency equation $z \vdash \mathsf{or}(z,z) = z$ gives us an equation

$$\Gamma;\Delta, y{:}F\sigma;\Pi \mid \Psi \vdash_P \mathsf{or}(y,y) =_{F\sigma} y\,.$$

By applying a handler to both sides of the equation, we get

$$\Gamma;\Delta, y{:}F\sigma;\Pi \mid \Psi \vdash_P t_{\mathsf{or}}[\mathsf{try}\,y\,\mathsf{with}\,H/y_1, \mathsf{try}\,y\,\mathsf{with}\,H/y_2] =_{F\sigma} \mathsf{try}\,y\,\mathsf{with}\,H\,.$$

Unfortunately, this does not seem sufficient to prove

$$\Gamma;\Delta, y{:}F\sigma;\Pi \mid \Psi \vdash_P t_{\mathsf{or}}[y/y_1, y/y_2] =_{F\sigma} y\,,$$

and we have to assume such equations as axioms. It would be interesting to see if some proof of such equations is possible within the program logic.

8.1 The handler logic

The main use of the handler logic is in showing that a given handler is correct. In particular, if for a given collection of handler computation terms t_{op}, we show that $\{t_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\,\mathsf{models}\,\mathcal{T}_{\mathsf{eff}}$ holds, we may use the soundness of the interpretation to prove that such collection gives a correct handler.

Proposition 8.2 *Take a collection of handler computation terms*

$$\{\boldsymbol{x}_p{:}\boldsymbol{\sigma}_p, \boldsymbol{x}{:}\boldsymbol{\beta}; \boldsymbol{y}_p{:}\underline{\boldsymbol{\tau}}_p, (y_i{:}\boldsymbol{\alpha}_i \to \underline{\tau})_i \vdash t_{\mathsf{op}}{:}\underline{\tau}\}_{\mathsf{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n\in\Sigma_{\mathsf{eff}}}$$

and assume that $\boldsymbol{x}_p{:}\boldsymbol{\sigma}_p; \boldsymbol{y}_p{:}\underline{\boldsymbol{\tau}}_p \vdash_H \{t_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\,\mathsf{models}\,\mathcal{T}_{\mathsf{eff}}$ *holds.*

Then, the handler $(\boldsymbol{x}_p{:}\boldsymbol{\sigma}_p; \boldsymbol{y}_p{:}\underline{\boldsymbol{\tau}}_p).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}$ *is correct.*

Example 8.3 The handler for read-only state, given in Section 7.4.9, is defined by

$$(d_p{:}\mathbf{dat}).\ \{\mathsf{lookup}_\ell(y) = y\,d_p\}\,.$$

Following the procedure described in Section 5.4, we obtain the following corresponding equations:

$$(\lambda d.\,(\lambda d'.\,y\,d\,d')\,d_p)\,d_p =_X (\lambda d.\,y\,d\,d)\,d_p$$

$$(\lambda d.\,y)\,d_p =_X y\,.$$

The two equations obviously hold, hence $\vdash_H \{t_{\mathsf{lookup}}{:}X\}\,\mathsf{models}\,\mathcal{T}_{\mathsf{eff}}$ is derivable and the read-only handler is correct.

Example 8.4 Similarly, we can prove the correctness of the rollback handler, given in Section 7.4.10 and defined by

$$(d_p\!:\!\mathbf{dat}; y_p\!:\!\mathbf{exc} \to X).\ \{\mathsf{raise}_e = \mathsf{update}_{d_p}(y_p\, e)\}\,.$$

Operation symbols $\mathsf{lookup}:\mathbf{dat}$ and $\mathsf{update}:\mathbf{dat};1$ are not handled, hence the corresponding handler terms for lookup, update, and raise, are:

$$\begin{gathered} d_p\!:\!\mathbf{dat}; y_p\!:\!\mathbf{exc} \to X, y\!:\!\mathbf{dat} \to X \vdash \mathsf{lookup}(d.\, y\, d) \\ d_p\!:\!\mathbf{dat}, d\!:\!\mathbf{dat}; y_p\!:\!\mathbf{exc} \to X, y\!:\!X \vdash \mathsf{update}_d(y) \\ d_p\!:\!\mathbf{dat}, e\!:\!\mathbf{exc}; y_p\!:\!\mathbf{exc} \to X \vdash \mathsf{update}_{d_p}(y\, e)\,, \end{gathered}$$

while the equations corresponding to the equations of the effect theory are:

$$\begin{gathered} \mathsf{lookup}(d_1.(\lambda d_2.\, \mathsf{lookup}(d_3.\, (\lambda d_4.\, y\, d_2\, d_4)\, d_3))\, d_1) =_X \mathsf{lookup}(d_5.\, (\lambda d_6.\, y\, d_6\, d_6)\, d_5)\,, \\ \mathsf{lookup}(d_1.\, (\lambda d_2.\, \mathsf{update}_{d_2}(y))\, d_1) =_X y\,, \\ \mathsf{update}_{d_1}(\mathsf{lookup}(d_2.\, (\lambda d_3.\, y\, d_3)\, d_2)) =_X \mathsf{update}_{d_4}(y\, d_4)\,, \\ \mathsf{update}_{d_1}(\mathsf{update}_{d_2}(y)) =_X \mathsf{update}_{d_2}(y)\,, \\ \mathsf{update}_{d_1}(\mathsf{update}_{d_p}(y_p\, e)) =_X \mathsf{update}_{d_p}(y_p\, e)\,. \end{gathered}$$

After β-reducing the 'administrative' λ-abstractions, the first four equations exactly correspond to the equations inherited from the effect theory, while the last equation is an instance of the fourth one. Hence, we have $\vdash_H \{t_{\mathsf{op}}\!:\!X\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}} \vDash \mathcal{T}_{\mathsf{eff}}$ and the rollback handler is correct.

Example 8.5 For the second rollback handler, the handler terms for lookup, update and raise are

$$\begin{gathered} y_p\!:\!\mathbf{exc} \to X, y\!:\!\mathbf{dat} \to (\mathbf{dat} \to X) \vdash \lambda d\!:\!\mathbf{dat}.\, (y\, d)\, d\!:\!\mathbf{dat} \to X \\ d'\!:\!\mathbf{dat}; y_p\!:\!\mathbf{exc} \to X, y\!:\!\mathbf{dat} \to X \vdash \lambda d\!:\!\mathbf{dat}.\, y\, d'\!:\!\mathbf{dat} \to X \\ \mathsf{exc}\!:\!\mathbf{exc}; y_p\!:\!\mathbf{exc} \to X \vdash \lambda d\!:\!\mathbf{dat}.\, y_p\, \mathsf{exc}\!:\!\mathbf{dat} \to X\,. \end{gathered}$$

The equations corresponding to the equations of the effect theory are:

$$\begin{gathered} \lambda d_1.\, ((\lambda d_2.\, \lambda d_3.\, ((\lambda d_4.\, y\, d_2\, d_4)\, d_3)\, d_3)\, d_1)\, d_1 =_{\mathbf{dat}\to X} \lambda d_5.\, ((\lambda d_6.\, y\, d_6\, d_6)\, d_5)\, d_5\,, \\ \lambda d_1.\, ((\lambda d_2.\, \lambda d_3.\, y\, d_2)\, d_1)\, d_1 =_{\mathbf{dat}\to X} y\,, \\ \lambda d_1.\, (\lambda d_2.\, ((\lambda d_3.\, y d_3)\, d_2)\, d_2)\, d =_{\mathbf{dat}\to X} \lambda d_4.\, (y d)\, d\,, \\ \lambda d_1.\, (\lambda d_2.\, y\, d')\, d =_{\mathbf{dat}\to X} \lambda d_3.\, y\, d'\,, \\ \lambda d_1.\, (\lambda d_2.\, y_p\, \mathsf{exc})\, d =_{\mathbf{dat}\to X} \lambda d_3.\, y_p\, \mathsf{exc}\,, \end{gathered}$$

where we have omitted the contexts and types when writing abstractions.

All five equations can be proved using β- and η-equivalence, hence the second variant of rollback handler is also correct.

8.2 The program logic

In the program logic, we have two slightly different means of describing the induced homomorphism from the free model: the free model principle and the handling construct. The free model principle provides on the fly construction of both models and homomorphisms inside the logic, while the handling construct requires a fixed handler definition and provides a term constructor to obtain homomorphisms. However, the two describe the same concept and are related through the logic.

Proposition 8.6 *Take a handler symbol $H : \underline{\tau}$* **handler** *and a handler definition $\mathcal{H}$, which maps it to a correct handler*

$$\mathcal{H}(H) = \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}} ,$$

and a computation term $\Gamma; \Delta \vdash t' : \sigma \to \underline{\tau}$.

Then, we have

$$\Gamma; \Delta; \Pi \mid \Psi \vdash_P \exists \hat{y} : UF\sigma \to \underline{\tau}. \\ \hat{y}\,\mathsf{extends}\,t' \land \hat{y}\,\mathsf{homomorphism} \land \mathsf{try}\,t\,\mathsf{with}\,H\,\mathsf{as}\,x\,\mathsf{in}\,t'x =_{\underline{\tau}} \hat{y}(\mathsf{thunk}\,t) .$$

Proof From the correctness of handler H, we get

$$\Gamma; \Delta; \Pi \mid \Psi \vdash_P \{t_{\mathsf{op}}[\underline{\boldsymbol{\tau}}/\boldsymbol{X}]\}_{\mathsf{op} \in \Sigma_{\mathsf{eff}}}\,\mathsf{models}\,\mathcal{T}_{\mathsf{eff}}$$

and the free model principle entails the existence of a computation variable $\hat{y} : UF\sigma \to \underline{\tau}$ such that

$$\hat{y}\,\mathsf{extends}\,t' \land \hat{y}\,\mathsf{homomorphism}$$

holds. We proceed by the principle of computational induction. Define the predicate π as

$$(y' : F\sigma).\ \mathsf{try}\,y'\,\mathsf{with}\,H\,\mathsf{as}\,x\,\mathsf{in}\,t'x =_{\underline{\tau}} \hat{y}(\mathsf{thunk}\,y') .$$

Then, for the base case $\pi(\mathsf{return}\,x')$, we have

$$\begin{aligned}\Gamma;\Delta;\Pi \mid \Psi \vdash_P{} & \mathsf{try}\,\mathsf{return}\,x'\,\mathsf{with}\,H\,\mathsf{as}\,x\,\mathsf{in}\,t'x \\ &= t'x' && \text{(by β-equivalence)} \\ &= \hat{y}(\mathsf{thunk}\,\mathsf{return}\,x') && \text{(because $\hat{y}\,\mathsf{extends}\,t'$)}\ .\end{aligned}$$

For the step case, take an operation symbol $\mathsf{op}\colon \boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n$. Then, we have to show $\pi(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i)$, assuming $\pi(y_i\boldsymbol{x}_i)$ for all i and $\boldsymbol{x}_i$. We get

$$\begin{aligned}\Gamma;\Delta;\Pi \mid \Psi \vdash_P{} & \mathsf{try}\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i\,\mathsf{with}\,H\,\mathsf{as}\,x\,\mathsf{in}\,t'x \\ &=_{F\sigma} t_{\mathsf{op}}[\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H\,\mathsf{as}\,x\,\mathsf{in}\,t'x/y_i]_i && \text{(by definition of H)} \\ &=_{F\sigma} t_{\mathsf{op}}[\hat{y}(\mathsf{thunk}\,y_i\boldsymbol{x}_i)/y_i]_i && \text{(by the induction hypotheses)} \\ &=_{F\sigma} \hat{y}(\mathsf{thunk}(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i)) && \text{(because $\hat{y}\,\mathsf{homomorphism\ holds}$)}\ ,\end{aligned}$$

and by the induction principle, we get $\Gamma;\Delta;\Pi \mid \Psi \vdash_P \pi(t)$ for all program computation terms t. □

We could also write a (slightly more complex) variant of the above proposition for handlers with parameters.

As a basis for studying further properties of the handling construct, we take axioms for exception handlers, suggested by Levy [Lev06b] (we transcribe the axioms to our syntax):

$$\begin{aligned}&\mathsf{try}\,\mathsf{return}\,v\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t' = t'[v/x] \\ &\mathsf{try}\,\mathsf{exc}_j\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t' = t_j \\ &\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = \mathsf{exc}_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{return}\,x = t \\ &\mathsf{try}(\mathsf{try}\,t_1\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x_1\,\mathsf{in}\,t_2)\mathsf{with}\{\mathsf{exc}_j = t'_j\}_j\,\mathsf{as}\,x_2\,\mathsf{in}\,t = \\ &\quad \mathsf{try}\,t_1\,\mathsf{with}\{\mathsf{exc}_i = \mathsf{try}\,t_i\,\mathsf{with}\{\mathsf{exc}_j = t'_j\}_j\,\mathsf{as}\,x_2\,\mathsf{in}\,t\}_i\,\mathsf{as}\,x_1\,\mathsf{in} \\ &\quad \mathsf{try}\,t_2\,\mathsf{with}\{\mathsf{exc}_i = t'_i\}_i\,\mathsf{as}\,x_2\,\mathsf{in}\,t\,,\end{aligned}$$

and “commuting conversions”, suggested by Benton and Kennedy [BK01] (also transcribed to our syntax):

$$\begin{aligned}&\mathsf{fst}(\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t') = \mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = \mathsf{fst}\,t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{fst}\,t'\ , \\ &\mathsf{snd}(\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t') = \mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = \mathsf{snd}\,t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{snd}\,t'\ , \\ &(\mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_i = t_i\}_i\,\mathsf{as}\,x\,\mathsf{in}\,t')v = \mathsf{try}\,t\,\mathsf{with}\{\mathsf{exc}_j = t_jv\}_j\,\mathsf{as}\,x\,\mathsf{in}\,t'v\ .\end{aligned}$$

Benton and Kennedy suggested an additional conversion, which corresponds to Levy's last (associativity) equation, and two more conversions that describe the relationship between pattern matching and the handling construct. Since Benton and Kennedy worked in a call-by-value setting, the proper translation of their equations involves an explicit sequencing. Then, all the equations translate to an instance of the associativity equation. However, unlike the conversions for pattern matching, the above three equations remain valid and serve as a motivation even if we directly transcribe them in the call-by-push-value framework.

Levy's first two equations are β-equivalence equations, and are instances of the two axiom schemas that describe the handling construct in our logic. The third equation is η-equivalence, and can be generalised to

$$\Gamma;\Delta \vdash_P \mathsf{try}\, t\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, \mathsf{return}\, x =_{\underline{\tau}} t \quad (\mathcal{H}(H) = \{\})\,.$$

This follows from Proposition 8.7 and η-equivalence for sequencing.

Proposition 8.7 *If $\mathcal{H}(H) = \{\}$, we have*

$$\Gamma;\Delta \vdash_P \mathsf{try}\, t\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, t' =_{\underline{\tau}} t\, \mathsf{to}\, x.\, t'\,.$$

Proof We proceed by the principle of computational induction. Define the predicate π to be

$$(y{:}F\sigma).\ \mathsf{try}\, y\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, t' =_{\underline{\tau}} y\, \mathsf{to}\, x.\, t'\,.$$

The base case $\Gamma, x'{:}\sigma;\, \Delta \vdash_P \pi(\mathsf{return}\, x')$ is equivalent to

$$\Gamma, x'{:}\sigma;\, \Delta \vdash_P \mathsf{try}\, \mathsf{return}\, x'\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, t' =_{\underline{\tau}} t'[x'/x] =_{\underline{\tau}} \mathsf{return}\, x'\, \mathsf{to}\, x.\, t'\,,$$

which is derivable by the β-equivalence for the handling construct and sequencing.

For the step case, take an operation $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n$. Then, we have to show $\pi(\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i)_i)$, assuming $\pi(y_i\boldsymbol{x}_i)$ for all i and $\boldsymbol{x}_i$. Then, we get

$$\begin{aligned}
&\vdash_P \mathsf{try}\, \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i)_i\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, t' && \\
&=_{\underline{\tau}} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, \mathsf{try}\, y_i\boldsymbol{x}_i\, \mathsf{with}\, H\, \mathsf{as}\, x\, \mathsf{in}\, t') && \text{(by definition of } H\text{)} \\
&=_{\underline{\tau}} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i\, \mathsf{to}\, x.\, t') && \text{(by the induction hypotheses)} \\
&=_{\underline{\tau}} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\, y_i\boldsymbol{x}_i)\, \mathsf{to}\, x.\, t' && \text{(by algebraicity of operations)}\ .
\end{aligned}$$

By the induction principle, we get $\Gamma;\Delta \vdash_P \pi(t)$ for all computation terms t. □

Levy's fourth equation, which states associativity of the exception handling construct, unfortunately has no valid generalisation of the form

$$\begin{aligned}&\mathsf{try}\,(\mathsf{try}\,t_1\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,t_2)\,\mathsf{with}\,H'(\boldsymbol{v}'_p;\boldsymbol{t}'_p)\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t\\ &=\mathsf{try}\,t_1\,\mathsf{with}\,H''(\boldsymbol{t}''_p;\boldsymbol{v}''_p)\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,(\mathsf{try}\,t_2\,\mathsf{with}\,H'(\boldsymbol{v}'_p;\boldsymbol{t}'_p)\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t)\,,\end{aligned}$$

for some $H''\!:\!(\boldsymbol{\sigma}''_p;\underline{\boldsymbol{\tau}}''_p)\to\underline{\tau}\,\mathbf{handler}$. First, since the handler definition $\mathcal{H}$ is fixed, an appropriate handler symbol H'' might not exist. Even worse, there may even be no possible model for it to denote.

Example 8.8 For a counterexample, take: the base theory of integers; an effect signature with a single unary operation symbol f; the empty effect theory; a handler signature of symbols H, H', and H'', all of type $F\mathbf{int}\,\mathbf{handler}$; an arbitrary handler computation term $y\!:\!F\mathbf{int}\vdash t_f\!:\!F\sigma$; and a handler definition $\mathcal{H}$, such that

$$\begin{aligned}\mathcal{H}(H)&=\{f(y)=y\,\mathsf{to}\,x.\ \mathsf{if}\,x<2\,\mathsf{then}\,\mathsf{return}(x+1)\,\mathsf{else}\,\mathsf{return}\,0\}\,,\\ \mathcal{H}(H')&=\{\}\,,\\ \mathcal{H}(H'')&=\{f(y)=t_f\}\,.\end{aligned}$$

Next, take t_2 to be $\mathsf{return}\,x_1$, and t to be $\mathsf{return}(x \bmod 2)$, where mod is the modulo operator. Then, the above equation simplifies to

$$(\mathsf{try}\,t_1\,\mathsf{with}\,H)\,\mathsf{to}\,x.\ \mathsf{return}(x \bmod 2)=_{F\mathbf{int}}\mathsf{try}\,t_1\,\mathsf{with}\,H''\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{return}(x \bmod 2)\,.$$

If $t_1=f(\mathsf{return}\,0)$, we have

$$\mathsf{return}\,1=_{F\mathbf{int}}t_f[\mathsf{return}\,0/y]\,,$$

but if we take $t_1=f(\mathsf{return}\,2)$, we have

$$\mathsf{return}\,0=_{F\mathbf{int}}t_f[\mathsf{return}\,0/y]\,.$$

Hence, there is no such t_f because the effect theory is empty and as such consistent.

However, the associativity equation for the exception handler, just like the three conversions suggested by Benton and Kennedy, turns out to be an instance of Proposition 8.10, which describes the commutativity between the handling construct and certain homomorphisms.

Definition 8.9 Take handler symbols

$$H : (\boldsymbol{\sigma}_p ; \underline{\boldsymbol{\tau}}_p) \to \underline{\tau}\ \textbf{handler}\,,$$
$$H' : (\boldsymbol{\sigma}'_p ; \underline{\boldsymbol{\tau}}'_p) \to \underline{\tau}'\ \textbf{handler}\,;$$

a handler definition $\mathcal{H}$, given by

$$\mathcal{H}(H) = \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathrm{eff}}}\,,$$
$$\mathcal{H}(H') = \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t'_{\mathsf{op}}\}_{\mathsf{op} \in \Sigma_{\mathrm{eff}}}\,;$$

handler value terms

$$\Gamma ; \Delta \vdash \boldsymbol{v}_p : \boldsymbol{\sigma}_p \qquad \text{and} \qquad \Gamma ; \Delta \vdash \boldsymbol{v}_p : \boldsymbol{\sigma}'_p\,;$$

and handler computation terms

$$\Gamma ; \Delta \vdash \boldsymbol{t}_p : \underline{\boldsymbol{\tau}}_p \qquad \text{and} \qquad \Gamma ; \Delta \vdash \boldsymbol{t}'_p : \underline{\boldsymbol{\tau}}'_p\,.$$

We say that a handler computation term $\Gamma ; \Delta, y : \underline{\tau} \vdash t_h : \underline{\tau}'$ is a *homomorphism between $H(\boldsymbol{v}_p ; \boldsymbol{t}_p)$ and $H'(\boldsymbol{v}'_p ; \boldsymbol{t}'_p)$*, if

$$\Gamma ; \Delta \vdash_P t_h[t_{\mathsf{op}}[\boldsymbol{v}_p/\boldsymbol{x}_p ; \boldsymbol{t}_p/\boldsymbol{y}_p]/y] =_{\underline{\tau}'} t'_{\mathsf{op}}[\boldsymbol{v}'_p/\boldsymbol{x}_p ; \boldsymbol{t}'_p/\boldsymbol{y}_p, (\lambda \boldsymbol{x}_i : \boldsymbol{\alpha}_i .\, t_h[y_i \boldsymbol{x}_i/y]/y_i)_i]$$

holds for all operation symbols $\mathsf{op} : \boldsymbol{\beta} ; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n$.

Proposition 8.10 *Take handler symbols, handler definition, program value and computation terms as in the statement of Definition 8.9, and let φ_h state that*

$$\Gamma ; \Delta, y : \underline{\tau} \vdash t_h : \underline{\tau}'$$

is a homomorphism between $H(\boldsymbol{v}_p ; \boldsymbol{t}_p)$ and $H'(\boldsymbol{v}'_p ; \boldsymbol{t}'_p)$. Then, we have

$$\Gamma ; \Delta ; \Pi \mid \Psi, \varphi_h \vdash_P t_h[\mathsf{try}\, t\, \mathsf{with}\, H(\boldsymbol{v}_p ; \boldsymbol{t}_p)\, \mathsf{as}\, x\, \mathsf{in}\, t'/y] =_{\underline{\tau}'} \mathsf{try}\, t\, \mathsf{with}\, H'(\boldsymbol{v}'_p ; \boldsymbol{t}'_p)\, \mathsf{as}\, x\, \mathsf{in}\, t_h[t'/y]\,.$$

Proof We proceed by the principle of computational induction on the handled term. For the base case, where the handled term is a returned value, we have

$$\begin{aligned}
&\vdash_P t_h[\mathsf{try}\, \mathsf{return}\, v\, \mathsf{with}\, H(\boldsymbol{v}_p ; \boldsymbol{t}_p)\, \mathsf{as}\, x\, \mathsf{in}\, t'/y] && \\
&=_{\underline{\tau}'} t_h[t'[v/x]/y] && \text{(by β-equivalence)} \\
&=_{\underline{\tau}'} (t_h[t'/y])[v/x] && \text{(because x is not free in t_h)} \\
&=_{\underline{\tau}'} \mathsf{try}\, \mathsf{return}\, v\, \mathsf{with}\, H'(\boldsymbol{v}'_p ; \boldsymbol{t}'_p)\, \mathsf{as}\, x\, \mathsf{in}\, t_h[t'/y] && \text{(by β-equivalence)}\,,
\end{aligned}$$

while for a step case for an operation symbol $\mathsf{op}{:}\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n$, we have

$\vdash_P t_h[\mathsf{try}\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t'/y]$

$= t_h[t_{\mathsf{op}}[\boldsymbol{v}_p/\boldsymbol{x}_p;\boldsymbol{t}_p/\boldsymbol{y}_p,(\lambda\boldsymbol{x}_i.\,\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t'/y_i)_i]/y]$

(by definition of H)

$= (t_h[t_{\mathsf{op}}[\boldsymbol{v}_p/\boldsymbol{x}_p;\boldsymbol{t}_p/\boldsymbol{y}_p]/y])[\lambda\boldsymbol{x}_i.\,\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t'/y_i]_i$

(because y_i are not free in t_h)

$= t'_{\mathsf{op}}[\boldsymbol{v}'_p/\boldsymbol{x}_p;\boldsymbol{t}'_p/\boldsymbol{y}_p;(\lambda\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\,t_h[y_i\boldsymbol{x}_i/y]/y_i)_i][\lambda\boldsymbol{x}_i.\,\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t'/y_i]_i$

(because t_h is a homomorphism)

$= t'_{\mathsf{op}}[\boldsymbol{v}'_p/\boldsymbol{x}_p;\boldsymbol{t}'_p/\boldsymbol{y}_p;\lambda\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\,t_h[\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t'/y]/y_i]_i$

(by substitution and because y_i are not free in $\boldsymbol{v}'_p$, $\boldsymbol{t}'_p$, and t_h)

$= t'_{\mathsf{op}}[\boldsymbol{v}'_p/\boldsymbol{x}_p;\boldsymbol{t}'_p/\boldsymbol{y}_p;\lambda\boldsymbol{x}_i{:}\boldsymbol{\alpha}_i.\,\mathsf{try}\,y_i\boldsymbol{x}_i\,\mathsf{with}\,H'(\boldsymbol{v}'_p;\boldsymbol{t}'_p)\,\mathsf{as}\,x\,\mathsf{in}\,t_h[t'/y]/y_i]_i$

(by induction hypothesis)

$= \mathsf{try}\,\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_i.\,y_i\boldsymbol{x}_i)_i\,\mathsf{with}\,H'(\boldsymbol{v}'_p;\boldsymbol{t}'_p)\,\mathsf{as}\,x\,\mathsf{in}\,t_h[t'/y]$

(by definition of H'.)

□

Corollary 8.11 *Take handler symbols*

$$H{:}(\boldsymbol{\sigma}_p;\underline{\boldsymbol{\tau}}_p)\to F\sigma_2\ \mathbf{handler}\,,$$
$$H'{:}(\boldsymbol{\sigma}'_p;\underline{\boldsymbol{\tau}}'_p)\to \underline{\tau}\ \mathbf{handler}\,,$$
$$H''{:}(\boldsymbol{\sigma}''_p;\underline{\boldsymbol{\tau}}''_p)\to \underline{\tau}\ \mathbf{handler}\,;$$

a handler definition $\mathcal{H}$, given by

$$\mathcal{H}(H)=\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y})=t_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\,,$$
$$\mathcal{H}(H')=\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y})=t'_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\,,$$
$$\mathcal{H}(H'')=\{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y})=t''_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\,;$$

program value terms $\boldsymbol{v}_p{:}\boldsymbol{\sigma}_p$, $\boldsymbol{v}'_p{:}\boldsymbol{\sigma}'_p$, and $\boldsymbol{v}''_p{:}\boldsymbol{\sigma}''_p$; and program computation terms $\boldsymbol{t}_p{:}\underline{\boldsymbol{\tau}}_p$, $\boldsymbol{t}'_p{:}\underline{\boldsymbol{\tau}}'_p$, and $\boldsymbol{t}''_p{:}\underline{\boldsymbol{\tau}}''_p$.

Furthermore, let φ_h state that

$$\Gamma;\Delta,y{:}F\sigma_2\vdash \mathsf{try}\,y\,\mathsf{with}\,H'(\boldsymbol{v}'_p;\boldsymbol{t}'_p)\,\mathsf{as}\,x_2{:}\sigma_2\,\mathsf{in}\,t{:}\underline{\tau}$$

is a homomorphism between $H(\boldsymbol{v}_p;\boldsymbol{t}_p)$ and $H''(\boldsymbol{v}''_p;\boldsymbol{t}''_p)$. Then, the associativity equation

$$\begin{aligned}\Gamma;\Delta \mid \varphi_h \vdash_P\ &\mathsf{try}\,(\mathsf{try}\,t_1\,\mathsf{with}\,H(\boldsymbol{v};\boldsymbol{t})\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,t_2)\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t\\ &=_{\underline{\tau}} \mathsf{try}\,t_1\,\mathsf{with}\,H''(\boldsymbol{t}'';\boldsymbol{v}'')\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,(\mathsf{try}\,t_2\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t)\,,\end{aligned}$$

holds.

Corollary 8.12 *Take the effect theory for exceptions and a handler symbol H_{exc}, defined as in Section 7.4.1. Then, we have*

$$\begin{aligned}\Gamma;\Delta \vdash_P\ &\mathsf{try}\,(\mathsf{try}\,t_1\,\mathsf{with}\,H_{\mathsf{exc}}(t_{\mathsf{exc}})\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\,t_2)\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t\\ &=_{\underline{\tau}} \mathsf{try}\,t_1\,\mathsf{with}\,H_{\mathsf{exc}}(\lambda e\!:\!\mathbf{exc}.\ \mathsf{try}\,t_{\mathsf{exc}}\,e\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t)\,\mathsf{as}\,x_1\!:\!\sigma_1\,\mathsf{in}\\ &\qquad \mathsf{try}\,t_2\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t\end{aligned}$$

for any $\Gamma;\Delta \vdash t_{\mathsf{exc}}\!:\!\mathbf{exc} \to \underline{\tau}$.

Proof Since

$$\mathcal{H}(H_{\mathsf{exc}}) = (y\!:\!\mathbf{exc} \to X).\ \{\mathsf{raise}_e = y\,e\}\!:\!(\mathbf{exc} \to X) \to X\ \mathbf{handler}\,,$$

we immediately have

$$\mathsf{try}\,y_{\mathsf{exc}}\,e\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t = (\lambda e\!:\!\mathbf{exc}.\ \mathsf{try}\,y_{\mathsf{exc}}\,e\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t)\,e\,.$$

Thus,

$$y\!:\!F\sigma_2 \vdash \mathsf{try}\,y\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t$$

is a homomorphism between $H_{\mathsf{exc}}(y_{\mathsf{exc}})$ and

$$H_{\mathsf{exc}}(\lambda e\!:\!\mathbf{exc}.\ \mathsf{try}\,y_{\mathsf{exc}}\,e\,\mathsf{with}\,H'(\boldsymbol{v}';\boldsymbol{t}')\,\mathsf{as}\,x_2\!:\!\sigma_2\,\mathsf{in}\,t)\,.$$

The associativity follows from Proposition 8.10. □

Unlike associativity, the three conversions suggested by Benton and Kennedy hold not only for exception handlers, but for a wider class of handlers, defined by effect terms.

Definition 8.13 A uniform handler

$$(\boldsymbol{x}_p\!:\!\boldsymbol{\beta}_p;(y_{pi}\!:\!\boldsymbol{\alpha}_{pi} \to X)_i).\ \{\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{y}) = t_{\mathsf{op}}\}_{\mathsf{op}\in\Sigma_{\mathsf{eff}}}\!:\!(\boldsymbol{\beta}_p;(\boldsymbol{\alpha}_{pi} \to X)_i) \to X\ \mathbf{handler}$$

is *simple*, if for each $\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$, the handling term t_{op} is obtained by instantiating an effect term

$$\boldsymbol{x}_p\colon\boldsymbol{\beta}_p,\boldsymbol{x}\colon\boldsymbol{\beta};\ (z_{pi}\colon(\boldsymbol{\alpha}_{pi}))_i,(z_j\colon(\boldsymbol{\alpha}_j))_j \vdash e_{\mathsf{op}}\,.$$

In particular, we replace each $z\colon(\boldsymbol{\alpha})$ in e_{op} by $(\boldsymbol{x}_i).\ y\boldsymbol{x}_i$ for some $y\colon\boldsymbol{\alpha}\to X$.

Example 8.14 Exception handler and stream redirection handler, as defined in Section 7.4.2, are both simple. CCS hiding and renaming are not simple because their definition includes conditionals. However, one could straightforwardly extend effect terms with conditionals and the following results adapt routinely.

Proposition 8.15 *Assume that $\mathcal{H}(H)$ is a simple handler and take a computation term $\Gamma;\Delta,y\colon\underline{\tau}\vdash t_h\colon\underline{\tau}'$. Next, for all operation symbols $\mathsf{op}\colon\boldsymbol{\beta};\boldsymbol{\alpha}_1,\dots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}$, define φ_{op} to be*

$$t_h[\mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_j.\ y_j\boldsymbol{x}_j)_j/y] =_{\underline{\tau}'} \mathsf{op}_{\boldsymbol{x}}(\boldsymbol{x}_j.\ t_h[y_j\boldsymbol{x}_j/y])_j$$

and let φ_h state that t_h is a homomorphism between handlers $H(\boldsymbol{x}_p;\boldsymbol{y}_p)$ and $H(\boldsymbol{x}_p;(\lambda\boldsymbol{x}_{pi}\colon\boldsymbol{\alpha}_{pi}.\ t_h[y_{pi}\boldsymbol{x}_{pi}/y])_i)$. Then, we have

$$\Gamma,\boldsymbol{x}\colon\boldsymbol{\beta};\ \Delta,(y_j\colon\boldsymbol{\alpha}_j\to\underline{\tau})_j \mid \Psi,\ \bigwedge_{\mathsf{op}\in\Sigma_{\mathrm{eff}}} \varphi_{\mathsf{op}} \vdash_P \varphi_h\,.$$

Proof First, note that

$$\boldsymbol{x}\colon\boldsymbol{\beta};\ (y_{pi}\colon\boldsymbol{\alpha}_{pi}\to\underline{\tau})_i \vdash_P t_h[e[(\boldsymbol{x}_i).\ y_i\boldsymbol{x}_i/z_i]_i/y] =_{\underline{\tau}'} e[(\boldsymbol{x}_i).\ t_h[y_i\boldsymbol{x}_i/y]/z_i]_i$$

holds for all effect terms $\boldsymbol{x}\colon\boldsymbol{\beta};\ (y_{pi}\colon(\boldsymbol{\alpha}_{pi}))_i \vdash e$, in particular the ones that define H. The proof proceeds by induction on the structure of e: the base case is immediate and the step case follows from the assumption of the proposition.

Then, we have:

$$\begin{aligned}
&\vdash_P t_h[t_{\mathsf{op}}/y]\\
&=_{\mathrm{def}} t_h[e_{\mathsf{op}}[(\boldsymbol{x}_i).\ y_i\boldsymbol{x}_i/z_i]_i/y]\\
&=_{\underline{\tau}'} e_{\mathsf{op}}[(\boldsymbol{x}_i).\ t_h[y_i\boldsymbol{x}_i/y]/z_i]_i\\
&\text{(by assumption)}\\
&=_{\underline{\tau}'} e_{\mathsf{op}}[(\boldsymbol{x}_i).\ (\lambda\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i.\ t_h[y_i\boldsymbol{x}_i/y])\boldsymbol{x}_i/z_i]_i\\
&\text{(by η-equivalence)}\\
&=_{\underline{\tau}'} e_{\mathsf{op}}[(\boldsymbol{x}_i).\ y_i\boldsymbol{x}_i/z_i]_i[\lambda\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i.\ t_h[y_i(\boldsymbol{x}_i)/y]/y_i]_i\\
&\text{(by substitution)}\\
&=_{\mathrm{def}} t_{\mathsf{op}}[\lambda\boldsymbol{x}_i\colon\boldsymbol{\alpha}_i.\ t_h[y_i(\boldsymbol{x}_i)/y]/y_i]_i\,,
\end{aligned}$$

where y_i ranges over both arguments of the operation and parameters of the handler. Thus, t_h is a homomorphism between $H(\boldsymbol{x}_p;\boldsymbol{y}_p)$ and

$$H(\boldsymbol{x}_p;(\lambda\boldsymbol{x}_{pi}\!:\!\boldsymbol{\alpha}_{pi}.\, t_h[y_{pi}\boldsymbol{x}_{pi}/y])_i)\,.$$

□

It is easy to check that the condition is satisfied if we define t_h to be $\mathsf{fst}\,y$, $\mathsf{snd}\,y$, or yv, hence simple handlers satisfy generalisations of equations suggested by Benton and Kennedy.

Corollary 8.16 *Let $\mathcal{H}(H)$ be a simple handler, given as above. Then, we have*

$$\Gamma;\Delta\vdash_P \mathsf{fst}(\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t') =_{\underline{\tau}_1} \mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;(\mathsf{fst}\,t_{pi})_i)\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{fst}\,t'\,,$$

$$\Gamma;\Delta\vdash_P \mathsf{snd}(\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t') =_{\underline{\tau}_2} \mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;(\mathsf{snd}\,t_{pi})_i)\,\mathsf{as}\,x\,\mathsf{in}\,\mathsf{snd}\,t'\,,$$

$$\Gamma;\Delta\vdash_P (\mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;\boldsymbol{t}_p)\,\mathsf{as}\,x\,\mathsf{in}\,t')v =_{\underline{\tau}} \mathsf{try}\,t\,\mathsf{with}\,H(\boldsymbol{v}_p;(t_{pi})_i v)\,\mathsf{as}\,x\,\mathsf{in}\,t'v\,.$$

Chapter 9

Recursion

Now that we have developed our approach, we extend it with recursion. For that reason, we turn from the category **Set** of sets to the category ω-**Cpo** of countable-chain complete partial orders (ω-cpos) and continuous maps between them (see Example 2.36 and [GHK$^+$03]). We first extend the representation of the underlying system, then introduce a least fixed point constructor to computation terms, and observe the impact that the recursion has on the reasoning rules of the logic. Finally, we note the differences in the presence of handlers.

9.1 Base and effect theories

To adapt the base theory to recursion, we first extend its formulae with *inequations* $v_1 \leqslant_\beta v_2$, typed as

$$\frac{\Gamma \vdash v_1 : \beta \qquad \Gamma \vdash v_2 : \beta}{\Gamma \vdash v_1 \leqslant_\beta v_2 : \mathbf{form}}\,.$$

Then, we demand that the base theory is closed under the following additional rules:

- reflexivity, transitivity, and antisymmetry of inequality:

$$\frac{}{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v \leqslant_\beta v}\,, \qquad \frac{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_1 \leqslant_\beta v_2 \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_2 \leqslant_\beta v_3}{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_1 \leqslant_\beta v_3}\,,$$

$$\frac{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_1 \leqslant_\beta v_2 \qquad \Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_2 \leqslant_\beta v_1}{\Gamma \mid \Psi \vdash_{\mathcal{T}_{\text{base}}} v_1 =_\beta v_2}\,.$$

Finally, we select a subset of *computable* relation symbols and for each of the non-computable relation symbols $\mathsf{rel} : (\boldsymbol{\beta})$, we specify a subset of its arguments β_i as *admissible*. We take all arguments of computable relation symbols to be admissible.

Definition 9.1 An *interpretation* $\mathcal{I}$ of the base signature Σ_{base} in ω-**Cpo** is given by:

- an ω-cpo $[\![\beta]\!]_{\mathcal{I}}$ for each base type β, such that $[\![\alpha]\!]_{\mathcal{I}}$ is a countable set (ω-cpo, equipped with the discrete partial order) for each arity type α;
- a continuous map

  $$[\![\mathsf{f}]\!]_{\mathcal{I}} : [\![\beta_1]\!]_{\mathcal{I}} \times \cdots \times [\![\beta_n]\!]_{\mathcal{I}} \to [\![\beta]\!]_{\mathcal{I}}$$

  for each function symbol $\mathsf{f} : (\beta_1, \ldots, \beta_n) \to \beta \in \Sigma_{\text{base}}$;
- a subset

  $$[\![\mathsf{rel}]\!]_{\mathcal{I}} \subseteq [\![\beta_1]\!]_{\mathcal{I}} \times \cdots \times [\![\beta_n]\!]_{\mathcal{I}}$$

  for each non-computable relation symbol $\mathsf{rel} : (\beta_1, \ldots, \beta_n) \in \Sigma_{\text{base}}$, for which $[\![\mathsf{rel}]\!]_{\mathcal{I}}$ is a sub-cpo when one fixes all of its non-admissible arguments;
- a subset

  $$[\![\mathsf{rel}]\!]_{\mathcal{I}} \subseteq [\![\beta_1]\!]_{\mathcal{I}} \times \cdots \times [\![\beta_n]\!]_{\mathcal{I}}$$

  for each computable relation symbol $\mathsf{rel} : (\beta_1, \ldots, \beta_n) \in \Sigma_{\text{base}}$, such that there exists a continuous map

  $$\chi_{[\![\mathsf{rel}]\!]_{\mathcal{I}}} : [\![\beta_1]\!]_{\mathcal{I}} \times \cdots \times [\![\beta_n]\!]_{\mathcal{I}} \to \mathbf{1} + \mathbf{1}\,,$$

  such that $\chi_{[\![\mathsf{rel}]\!]_{\mathcal{I}}}(\boldsymbol{\gamma}) = \mathsf{inj}_1(\star)$ if and only if $\boldsymbol{\gamma} \in [\![\mathsf{rel}]\!]_{\mathcal{I}}$.

We extend the effect theory with *conditional inequations* $\Gamma; Z \vdash e_1 \leqslant e_2\ (\varphi)$, where $\Gamma; Z \vdash e_1$ and $\Gamma; Z \vdash e_2$ are well-typed effect terms, and $\Gamma \vdash \varphi : \mathbf{form}$ is a well-typed base formula.

In addition, we assume that the effect signature Σ_{eff} contains an operation symbol $\Omega : 0$, and that the effect theory $\mathcal{T}_{\text{eff}}$ contains an inequation $z \vdash \Omega \leqslant z$, saying that Ω is the least element [HPP06].

Example 9.2 The effect theory given by the equations for a semi-lattice describes the convex powerdomain in the presence of recursion. To describe other types of powerdomains, the effect theory has to be extended with

$$z_1, z_2 \vdash z_1 \leqslant \mathsf{or}(z_1, z_2)\,,$$

for the lower powerdomain, or with

$$z_1, z_2 \vdash \mathsf{or}(z_1, z_2) \leqslant z_1$$

to get the upper powerdomain [HPP06].

The effect theory gives rise to a countable discrete Lawvere ω-**Cpo**-theory $\mathcal{L}$ and an adjunction $F \dashv U \colon \mathrm{Mod}_{\mathcal{L}}(\omega\text{-}\mathbf{Cpo}) \to \omega\text{-}\mathbf{Cpo}$ in a standard way [HP06].

9.2 Terms and predicates

In the term language, we keep the same value and computation types, except that we limit relation symbols in conditionals to computable ones. We add recursion with a least fixed point construct $\mu y{:}\underline{\tau}.\ t{:}\underline{\tau}$, which is typed as

$$\frac{\Gamma; \Delta, y{:}\underline{\tau} \vdash t{:}\underline{\tau}}{\Gamma; \Delta \vdash \mu y{:}\underline{\tau}.\ t{:}\underline{\tau}}\,.$$

Note that the fixed point construct is not limited to functions, and can be used on arbitrary computation terms.

Example 9.3 A computation term

$$\mu y{:}\boldsymbol{F}\mathbf{1}.\ \mathsf{or}(\mathsf{return}\,\star, \mathsf{out}_a(y)){:}\boldsymbol{F}\mathbf{1}$$

represents a computation that nondeterministically chooses between returning $\star$, or outputting the character a and then recursively calling itself. Thus, it can output a finite number of characters and then return $\star$, or diverge, outputting an infinite number of characters.

We extend the propositions of the logic with two additional atomic propositions: $v_1 \leqslant_\sigma v_2$ and $t_1 \leqslant_{\underline{\tau}} t_2$, which are typed as

$$\frac{\Gamma; \Delta \vdash v_1{:}\sigma \qquad \Gamma; \Delta \vdash v_2{:}\sigma}{\Gamma; \Delta; \Pi \vdash v_1 \leqslant_\sigma v_2{:}\mathbf{prop}}\,, \qquad \frac{\Gamma; \Delta \vdash t_1{:}\underline{\tau} \qquad \Gamma; \Delta \vdash t_2{:}\underline{\tau}}{\Gamma; \Delta; \Pi \vdash t_1 \leqslant_{\underline{\tau}} t_2{:}\mathbf{prop}}\,.$$

We additionally specify a subset of predicate variables as admissible.

Definition 9.4 A value variable x is *admissible* in a proposition φ:**prop** if:

- φ does not contain x;
- φ is of the form $\pi(\boldsymbol{v};\boldsymbol{t})$ and π is an admissible predicate;
- φ is of the form $\mathsf{rel}(\boldsymbol{v})$ and all terms v_i that contain x are admissible arguments of rel;
- φ is of the form $v_1 =_\sigma v_2$, $t_1 =_{\underline{\tau}} t_2$, $v_1 \leqslant_\sigma v_2$ or $t_1 \leqslant_{\underline{\tau}} t_2$;
- φ is of the form $\top$, $\bot$, $\varphi_1 \wedge \varphi_2$, or $\varphi_1 \vee \varphi_2$, and x is admissible in φ_1 and φ_2;
- φ is of the form $\varphi_1 \Rightarrow \varphi_2$, and x does not appear in φ_1 and is admissible in φ_2.
- φ is of the form $\forall x' {:} \sigma.\, \varphi'$ for $x \neq x'$ or $\forall y {:} \underline{\tau}.\, \varphi'$, and x is admissible in φ'.

For computation variables, the definition is analogous. A predicate π:**prop**$(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$ is admissible, if:

- π is an admissible predicate variable;
- π is of the form $(\boldsymbol{x}{:}\boldsymbol{\sigma};\boldsymbol{y}{:}\underline{\boldsymbol{\tau}}).\, \varphi$, and all the variables $\boldsymbol{x}$ are admissible in φ.
- π is of the form $\nu P{:}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}}).\, \pi'$, where P is an admissible predicate variable and π' is an admissible predicate.

As before, we extend the interpretation to value types, which are interpreted by ω-cpos, and to computation types, which are, due to the presence of the inequation $z \vdash \Omega \leqslant z$ in the effect theory, interpreted by ω-cppos, that is ω-cpos with a least element $\bot$.

It is routine to show that for each computation term $\Gamma;\Delta, y{:}\underline{\tau} \vdash t{:}\underline{\tau}$, the interpretation $[\![t]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to U[\![\underline{\tau}]\!]$ yields a continuous map, hence we can interpret the computation fixed point construct as a least fixed point construction, which also yields a continuous map. As before, propositions $\Gamma;\Delta;\Pi \vdash \varphi$:**prop** are interpreted by subsets

$$[\![\varphi]\!] \subseteq [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$$

while predicates $\Gamma;\Delta;\Pi \vdash \pi$:**prop**$(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$ are interpreted by maps

$$[\![\pi]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!] \to \mathcal{P}([\![\boldsymbol{\sigma}]\!] \times U[\![\underline{\boldsymbol{\tau}}]\!])\,.$$

Proposition 9.5 *Take any proposition* $\Gamma;\Delta;\Pi \vdash \varphi : \mathbf{prop}$*, any* $\boldsymbol{a} \in [\![\Gamma]\!]$*, and any* $\boldsymbol{U} \in [\![\Pi]\!]$ *such that* U_i *is a sub-cpo of* $[\![\boldsymbol{\sigma}_i]\!] \times U[\![\underline{\boldsymbol{\tau}}_i]\!]$ *for any admissible predicate variable* $P_i : \mathbf{prop}(\boldsymbol{\sigma}_i;\underline{\boldsymbol{\tau}}_i) \in \Pi$*. Then, the subset*

$$\{\langle \boldsymbol{\gamma},\boldsymbol{\delta}\rangle \in [\![\Gamma]\!] \times [\![\Delta]\!] \mid \langle \boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\varphi]\!] \text{ and}$$
$$\text{for all } i \in \{1,\ldots,n\}, \text{ either } \gamma_i = a_i \text{ or } x_i \text{ is admissible in } \varphi\}$$

is a sub-cpo of $[\![\Gamma]\!] \times [\![\Delta]\!]$*. Above, we effectively fix all value variables except for some admissible ones.*

Take any predicate $\Gamma;\Delta;\Pi \vdash \pi : \mathbf{prop}(\boldsymbol{\sigma};\underline{\boldsymbol{\tau}})$*, and any* $\langle \boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U}\rangle \in [\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\Pi]\!]$ *such that* U_i *is a sub-cpo of* $[\![\boldsymbol{\sigma}_i]\!] \times U[\![\underline{\boldsymbol{\tau}}_i]\!]$ *for any admissible predicate variable* $P_i : \mathbf{prop}(\boldsymbol{\sigma}_i;\underline{\boldsymbol{\tau}}_i) \in \Pi$*. Then, the subset* $[\![\pi]\!](\boldsymbol{\gamma},\boldsymbol{\delta},\boldsymbol{U})$ *is a sub-cpo of* $[\![\boldsymbol{\sigma}]\!] \times U[\![\underline{\boldsymbol{\tau}}]\!]$

Proof We proceed by structural induction. All the cases proceed routinely. □

Example 9.6 Existential quantifiers cannot be used when constructing admissible propositions and predicates. For example, take a base type $\mathbf{nat}_\infty$, together with the base theory of integers, and an additional element $\infty : \mathbf{nat}_\infty$ such that $n \leqslant \infty$ for all integers n. Then, take the admissible predicate

$$\pi =_{\mathrm{def}} (x_1 : \mathbf{nat}_\infty).\, x_1 \leqslant_{\mathbf{nat}_\infty} x_2 \wedge x_2 \neq \infty\,.$$

For any assignment of an integer n to x_2, we get a finite sub-cpo $\{0,1,\ldots,n\}$. However, the interpretation of the predicate

$$(x_1 : \mathbf{nat}_\infty).\, \exists x_2 : \mathbf{nat}_\infty.\, \pi(x_1)$$

is the subset of all integers, and is not a sub-cpo.

Example 9.7 We also cannot use least fixed points of predicates to construct admissible predicates. Again, take the base signature and theory as in Example 9.6, and take the admissible predicate

$$(x : \mathbf{nat}_\infty).\, P(\mathsf{succ}(x)) \wedge P(0)\,,$$

where P is an admissible predicate variable. The interpretation of

$$\mu P : (\mathbf{nat}_\infty).\, (x : \mathbf{nat}_\infty).\, P(\mathsf{succ}(x)) \wedge P(0)$$

is again the subset of all integers and as such not a sub-cpo.

9.3 Reasoning rules

To the logic, we add the reasoning rules for the two inequalities and for the computation fixed point construct:

- reflexivity, transitivity, and antisymmetry of value inequality:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v \leqslant_\sigma v}\,, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v_1 \leqslant_\sigma v_2 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L v_2 \leqslant_\sigma v_3}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v_1 \leqslant_\sigma v_3}\,,$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v_1 \leqslant_\sigma v_2 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L v_2 \leqslant_\sigma v_1}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L v_1 =_\sigma v_2}\,,$$

- reflexivity, transitivity, and antisymmetry of computation inequality:

$$\frac{}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t \leqslant_{\underline{\tau}} t}\,, \qquad \frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t_1 \leqslant_{\underline{\tau}} t_2 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L t_2 \leqslant_{\underline{\tau}} t_3}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t_1 \leqslant_{\underline{\tau}} t_3}\,,$$

$$\frac{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t_1 \leqslant_{\underline{\tau}} t_2 \qquad \Gamma;\Delta;\Pi \mid \Psi \vdash_L t_2 \leqslant_{\underline{\tau}} t_1}{\Gamma;\Delta;\Pi \mid \Psi \vdash_L t_1 =_{\underline{\tau}} t_2}\,,$$

- least pre-fixed point of a computation:

$$\frac{}{\Gamma;\Delta;\Pi \vdash_L t[\mu y{:}\underline{\tau}.\ t/y] \leqslant \mu y{:}\underline{\tau}.\ t}\,,$$

- principle of Scott induction:

$$\frac{}{\Gamma;\Delta;\Pi \mid \pi(\Omega), \forall y{:}\underline{\tau}.\ \pi(y) \Rightarrow \pi(t) \vdash_L \pi(\mu y{:}\underline{\tau}.\ t)}\,.$$

In the principle of Scott induction, π is restricted to be admissible. Other rules of the logic remain the same, except that the predicate in the principle of induction over computations is also restricted to be admissible.

The proof of soundness of the modified rules is straightforward for the reflexivity, transitivity, and antisymmetry of the inequalities. Then, the rule for the least pre-fixed point construct of a computation is sound by the definition of $\mu y{:}\underline{\tau}.\ t$. The principle of Scott induction is sound because π is admissible, hence $[\![\pi]\!]$ yields a sub-cpo. For the principle of computational induction, we show that

$[\![\pi]\!]$ is a sub-cpo because π is an admissible predicate. Then, we get a continuous map $j\colon [\![\pi]\!] \to UF[\![\sigma]\!]$ and proceed as in Proposition 5.4.

All the uses of the principle of computational induction in Chapter 6 were for admissible predicates, hence those results continue to hold in the presence of recursion. This, however, does not give the translation or guarantee its soundness when recursion is already featured in the source logic, for example in Hennessy-Milner logic for CCS with recursively defined processes.

9.4 Handlers

Handlers can be extended with recursion along the same lines as the logic. All of the uses of the principle of computational induction in Chapter 8 were again only for admissible predicates, hence all the obtained results continue to hold. Note that correct handlers cannot redefine Ω because of the inequation $\Omega \leqslant z$, hence the handling constructs are interpreted by strict homomorphisms.

Chapter 10

Conclusions

10.1 Accomplishments

We started with the a-calculus, a minimal equational logic with the purpose of emphasising the distinguishing features of effectful computations. This purpose was fulfilled as almost all observations in the logic were made in the a-calculus already: the fundamental nature of sequencing; algebraicity of operations; the principle of computational induction, though described in terms of normal forms; derivability of η-equivalence and associativity for sequencing; and commutativity of sequencing for commutative effect theories. In addition, the a-calculus is complete with regard to a fairly general denotational semantics.

The first aim of the thesis was to propose a powerful logic of algebraic effects. The proposed logic builds on Levy's versatile call-by-push-value approach, which describes both call-by-name and call-by-value, and extends it with a comprehensive set of logical rules, including two algebraic principles which capture the essence of free models. The logic has a sound semantics in terms of sets, and can easily be adapted to incorporate recursion. The resulting logic is indeed powerful, as seen in its use in generalising the results of the a-calculus and in translating three different program logics: Moggi's computational λ-calculus, Hennessy-Milner logic, and Moggi's variant of Pitts's evaluation logic, restricted to effects that admit a representation with balanced theories.

The second aim of the thesis was to give an algebraic treatment of exception handlers. We achieved this aim by interpreting exception handlers as models of the theory for exceptions and the handling construct using the induced homomorphism from the free model. This approach leads naturally to handlers of other algebraic effects, which is a completely novel programming concept. We identified the handler correctness issue that arises with this generalisation, and proposed a reasonable solution, which introduces two languages: one to define handlers and one to use them. Then, we presented the flexibility of generalised handlers by using them to describe many unrelated computational concepts from both theory and practice. Finally, we extended the logic for algebraic effect with handlers. Doing this was straightforward, which highlights both the modular structure of the logic and the naturalness of the concept of handlers.

10.2 Future work

On the topic of the α-calculus, there is little to be done, except giving an operational semantics. Another option would be to formalise it in one of the theorem provers such as *Coq* [Tea], *Isabelle* [NPW02], or *Twelf* [PS99]. This would also serve as a base for the formalisation of the logic.

Instead, it would be interesting to develop the term language of the logic into an independent equational logic. To do so, one would first have to state the effect theory in terms of a conditional equational theory, closed under a suitable set of rules [Plo06], rather than a collection of conditional equations. Then, we could also extend semantics to a more general category and seek a possible proof of completeness.

The presented logic seems hard to grasp due to its size. This comes as no surprise as we wanted a powerful and comprehensive logic. We could make some simplifications, however. For example, we could eliminate computation variables in favour of value variables over thunks, or allow arity and parameter types of operations to be first-order types. This would allow us to express almost all of the logic without the use of sequences, which are all too prevalent in the current presentation. Another possibility is to study a fragment of the logic such as the equational logic mentioned above, or a modal fragment without first-order connectives and with modalities taken as primitives.

Although the successful translation of Moggi's computational λ-calculus, of Hennessy-Milner logic, and of Moggi's variant of Pitts's evaluation logic provides some evidence of the expressiveness and versatility of the logic, there are many other logics still to be embraced. On one hand, we have various logics based on evaluation logic, for example global evaluation logic [GSM06] or dynamic logic [SM04]. And on the other hand, we have state logics as the Hoare logic [Hoa69] and separation logic, whose translation would probably require the logic to be adapted to a coalgebraic treatment of effects, which seems more natural for state [PS04].

Finally, the logic has been developed only over the categories of sets and of ω-cpos. There is an open question of what the logic should be over a general category, and a more immediate question on how to give a logic over the category of pre-sheaves or sheaves. This would allow the inclusion of new names [GP01] and local variables [PP02], which should, together with the above mentioned treatment of state, give way to a translation of separation logic [Rey02].

The concept of handlers of algebraic effects is novel and relatively unexplored, so there are still many open questions. For example, the language lacks a general operational semantics, which should generalise the one, given by Benton and Kennedy [BK01]. Then, the work done on combinations of effects should be extended to combinations of handlers.

However, the most important open problem is how to simultaneously handle more than one computation. This would allow an algebraic treatment of the CCS parallel operator or the UNIX pipe combinator. Unfortunately, immediate ideas such as bi-homomorphisms or thunked computations in parameters fail.

The separation between the language that describes and the language that uses handlers forms the core of our approach. This has advantages, as it simplifies reasoning and gives the language designer a better control over the handlers allowed. Still, it would be interesting to explore the option of a single language. A possible solution would be to give a single language with a suitable type-theory that limits handlers to uniform or simple ones, and by that ensures that all well-typed handlers are correct.

Finally, handlers are a programming concept, and their true value can be seen only after they have been implemented. As a first step, a toy language *eff*, is being developed by the author in co-operation with Andrej Bauer at the University of Ljubljana. As the next step, we could extend HASKELL [PJ03] in two ways: by enriching the built-in effects with operations and handlers, or by giving programmers a way to write their own handlers with no direct access to effects, which can be used to program in an extension of the monadic style.

Bibliography

[BHM00] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *APPSEM 2000*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122, 2000.

[BK01] Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, 2001.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra*. Cambridge University Press, 1994.

[CM87] William F. Clocksin and Chris Mellish. *Programming in Prolog*. Springer, 3rd edition, 1987.

[End00] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2000.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[GHK^{+}03] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana Stewart Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003.

[GP01] Murdoch Jamie Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[Grä79] George A. Grätzer. *Universal Algebra*. Springer, 2nd edition, 1979.

[GSM06] Sergey Goncharov, Lutz Schröder, and Till Mossakowski. Completeness of global evaluation logic. In *31st Symposium on Mathematical Foundations of Computer Science*, volume 4162 of *Lecture Notes in Computer Science*, pages 447–458, 2006.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HP06] Martin Hyland and Anthony John Power. Discrete Lawvere theories and computational effects. *Theoretical Computer Science*, 366(1-2):144–162, 2006.

[HPP06] Martin Hyland, Gordon David Plotkin, and Anthony John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2nd edition, 2004.

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[Lev06a] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

[Lev06b] Paul Blain Levy. Monads and adjunctions for global exceptions. *Electronic Notes in Theoretical Computer Science*, 158:261–287, 2006.

[Man98] Ernie G. Manes. Implementing collection classes with monads. *Mathematical Structures in Computer Science*, 8(3):231–276, 1998.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[ML71] Saunders Mac Lane. *Categories for the working mathematician*. Springer-Verlag New York, 1971.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *4th Symposium on Logic in Computer Science*, pages 14–23, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information And Computation*, 93(1):55–92, 1991.

[Mog94] Eugenio Moggi. A general semantics for evaluation logic. In *9th Symposium on Logic in Computer Science*, pages 353–362, 1994.

[Mog95] Eugenio Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22(1/2):117–152, 1995.

[MS09] Rasmus Ejlers Møgelberg and Alex Simpson. Relational parametricity for computational effects. *Logical Methods in Computer Science*, 5(3), 2009.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Pit91] Andrew M. Pitts. Evaluation logic. In *4th Higher Order Workshop*, pages 162–189, 1991.

[PJ03] Simon L. Peyton Jones. Haskell 98. *Journal of Functional Programming*, 13(1):0–255, 2003.

[Plo06] Gordon David Plotkin. Some varieties of equational logic. In *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 150–156, 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pow06] Anthony John Power. Countable Lawvere theories and computational effects. *Electronic Notes in Theoretical Computer Science*, 161:59–71, 2006.

[PP01] Gordon David Plotkin and Anthony John Power. Adequacy for algebraic effects. In *4th International Conference on Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24, 2001.

[PP02] Gordon David Plotkin and Anthony John Power. Notions of computation determine monads. In *5th International Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356, 2002.

[PP03] Gordon David Plotkin and Anthony John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[PP04] Gordon David Plotkin and Anthony John Power. Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163, 2004.

[PP08] Gordon David Plotkin and Matija Pretnar. A logic for algebraic effects. In *23rd Symposium on Logic in Computer Science*, pages 118–129, 2008.

[PP09] Gordon David Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94, 2009.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, 1999.

[PS04] Anthony John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. *Electronic Notes in Theoretical Computer Science*, 106:297–314, 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Symposium on Logic in Computer Science*, pages 55–74, 2002.

[Sco93] Dana Stewart Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, 1993.

[SM04] Lutz Schröder and Till Mossakowski. Monad-independent dynamic logic in HasCasl. *Journal of Logic and Computation*, 14(4):571–619, 2004.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Tea] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt. Available at `http://coq.inria.fr/coq/distrib/current/refman/`.

[vGP] Rob van Glabbeek and Gordon David Plotkin. CSP and the algebraic theory of effects. Available at `http://homepages.inf.ed.ac.uk/gdp/publications/`.

Index