Foundations for Programming and Implementing Effect Handlers

Daniel Hillerström

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

The University of Edinburgh

2021

Abstract

First-class control operators provide programmers with an expressive and efficient means for manipulating control through reification of the current control state as a first-class object, enabling programmers to implement their own computational effects and control idioms as shareable libraries. Effect handlers provide a particularly structured approach to programming with first-class control by naming control reifying operations and separating them from their handling.

This thesis is composed of three strands of work in which I develop operational foundations for programming and implementing effect handlers as well as exploring the expressive power of effect handlers.

The first strand develops a fine-grain call-by-value core calculus of a statically typed programming language with a *structural* notion of effect types, as opposed to the *nominal* notion of effect types that dominates the literature. With the structural approach, effects need not be declared before use. The usual safety properties of statically typed programming are retained by making crucial use of *row polymorphism* to build and track effect signatures. The calculus features three forms of handlers: deep, shallow, and parameterised. They each offer a different approach to manipulate the control state of programs. Traditional deep handlers are defined by folds over computation trees, and are the original con-struct proposed by Plotkin and Pretnar. Shallow handlers are defined by case splits (rather than folds) over computation trees. Parameterised handlers are deep handlers extended with a state value that is threaded through the folds over computation trees. To demonstrate the usefulness of effects and handlers as a practical programming abstraction I implement the essence of a small UNIX-style operating system complete with multi-user environment, time-sharing, and file I/O.

The second strand studies *continuation passing style* (CPS) and *abstract machine semantics*, which are foundational techniques that admit a unified basis for implementing deep, shallow, and parameterised effect handlers in the same environment. The CPS translation is obtained through a series of refinements of a basic first-order CPS translation for a fine-grain call-by-value language into an untyped language. Each refinement moves toward a more intensional representation of continuations eventually arriving at the notion of *generalised continuation*, which admit simultaneous support for deep, shallow, and parameterised handlers. The initial refinement adds support for deep handlers by representing stacks of continuations and handlers as a curried sequence of arguments. The image of the resulting translation is not *properly tail-recursive*, mean-

ing some function application terms do not appear in tail position. To rectify this the CPS translation is refined once more to obtain an uncurried representation of stacks of continuations and handlers. Finally, the translation is made higher-order in order to contract administrative redexes at translation time. The generalised continuation representation is used to construct an abstract machine that provide simultaneous support for deep, shallow, and parameterised effect handlers. kinds of effect handlers.

The third strand explores the expressiveness of effect handlers. First, I show that deep, shallow, and parameterised notions of handlers are interdefinable by way of *typed macro-expressiveness*, which provides a syntactic notion of expressiveness that affirms the existence of encodings between handlers, but it provides no information about the computational content of the encodings. Second, using the semantic notion of expressiveness I show that for a class of programs a programming language with first-class control (e.g. effect handlers) admits asymptotically faster implementations than possible in a language without first-class control.

Lay summary

Computer programs interact with the real world, e.g. to send and retrieve e-mails, stream videos, transferal of data from or onto some pluggable data storage medium, and so forth. This interaction is governed by the operating system, which is responsible for running programs and providing them with the vocabulary to interact with the world. Programs use words from this vocabulary with a preconceived idea of their meaning, however, importantly, words are just mere syntax. The semantics of each word is determined by the operating system (typically such that it aligns with the intent of the program).

This separation of syntax and semantics makes it possible for programs and operating systems to evolve independently, because any program can be run by any operating system whose vocabulary conforms to the expectations of the program. It has proven to be a remarkably successful model for building and maintaining computer programs.

Conventionally, an operating system has been a complex and monolithic single global entity in a computer system. However, *effect handlers* are a novel programming abstraction, which enables programs to be decomposed into syntax and semantics internally, by localising the notion of operating systems. In essence, an effect handler is a tiny programmable operating system, that a program may use internally to determine the meaning of its subprograms. The key property of effect handlers is that they compose seamlessly, and as a result the semantics of a program can be compartmentalised into several fine-grained and comprehensible components. The ability to seamlessly swap out one component for another component provides a promising basis for modular construction and reconfiguration of computer programs.

In this dissertation I develop the foundations for programming with effect handlers. Specifically, I present a practical design for programming with effect handlers as well as applications, I develop two universal implementation strategies for effect handlers, and I give a precise mathematical characterisation of the inherent computational efficiency of effect handlers.

Acknowledgements

Firstly, I want to thank Sam Lindley for his guidance, advice, and encouragement throughout my studies. He has been an enthusiastic supervisor, and he has always been generous with his time. I am fortunate to have been supervised by him. Secondly, I want to extend my gratitude to John Longley, who has been an excellent second supervisor and has always shown enthusiasm about my work. Thirdly, I want to thank my academic brother Simon Fowler, who has always been a good and inspirational friend. Regardless of academic triumphs and failures, we have always had fun.

I am extremely grateful to KC Sivaramakrishnan, who took a genuine interest in my research early on and invited me to come spend some time at OCaml Labs in Cambridge. My initial visit to Cambridge sparked the beginning of a long-standing and productive collaboration. Also, thanks to Gemma Gordon, who I have had the pleasure of sharing an office with during one of my stints at OCaml Labs.

I have been fortunate to work with Robert Atkey, who has been a continuous source of inspiration and interesting research ideas. Our work is clearly reflected in this dissertation. I also want to thank to my other collaborators: Andreas Rossberg, Anil Madhavapeddy, Leo White, Stephen Dolan, and Jeremy Yallop.

I have had the pleasure of working in LFCS at the same time as James McKinna. James has always taken a genuine interest in my work and challenged me with intellectually stimulating questions. I appreciate our many conversations even though I spent days, weeks, sometimes months, and in some instances years to come up with adequate answers. I also want to extend my thanks to other former and present members of Informatics: Brian Campbell, Christophe Dubach, James Cheney, J. Garrett Morris, Gordon Plotkin, Mary Cryan, Murray Cole, Michel Steuwer, and Philip Wadler.

My time as a student in Informatics Forum has been enjoyable in large part thanks to my friends: Amna Shahab, Chris Vasiladiotis, Craig McLaughlin, Danel Ahman, Daniel Mills, Frank Emrich, Emanuel Martinov, Floyd Chitalu, Jack Williams, Jakub Zalewski, Larisa Stoltzfus, Maria Gorinova, Marcin Szymczak, Paul Piho, Philip Ginsbach, Radu Ciobanu, Rajkarn Singh, Rosinda Fuentes Pineda, Rudi Horn, Shayan Najd, Stan Manilov, and Vanya Yaneva-Cormack.

Thanks to Ohad Kammar and Stephen Gilmore for agreeing to serve as the internal examiners for my dissertation. As for external examiners, I am truly humbled and thankful for Andrew Kennedy and Edwin Brady agreeing to examine my dissertation.

Throughout my studies I have received funding from the School of Informatics

at The University of Edinburgh, as well as an EPSRC grant EP/L01503X/1 (EPSRC Centre for Doctoral Training in Pervasive Parallelism), and by ERC Consolidator Grant Skye (grant number 682315). I finished this dissertation whilst being employed on the UKRI Future Leaders Fellowship “Effect Handler Oriented Programming” (reference number MR/T043830/1).

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except 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.

The following previously published work of mine features prominently within this dissertation. Each chapter details the relevant relations to my previous work.

- Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016
- Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *FSCD*, volume 84 of *LIPIcs*, pages 18:1–18:19, 2017
- Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018
- Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020
- Daniel Hillerström, Sam Lindley, and John Longley. Effects for efficiency: Asymptotic speedup with first-class control. *Proc. ACM Program. Lang.*, 4(ICFP): 100:1–100:29, 2020

(*Daniel Hillerström, Edinburgh, Scotland, 2021)*

Bara du sätter gränserna

Contents

Chapter 1

Introduction

Plotkin and Pretnar's *effect handlers* provide a promising modular basis for effectful programming [143, 227, 228]. The basic tenet of programming with effect handlers is that programs are written with respect to an interface of effectful operations they expect to be offered by their environment. An effect handler is an environment that implements an effect interface (also known as a computational effect). Programs can run under any effect handler whose implementation conforms to the expected effect interface.

In this regard, the *doing* and *being* of effects are kept separate [137, 174], which is a necessary condition for modular abstraction [212]. A key property of effect handlers is that they provide modular instantiation of effect interfaces through seamless composition, meaning the programmer can compose any number of complementary handlers to obtain a full implementation of some interface [119]. The ability to seamless compose handlers gives rise to a new programming paradigm which we shall call *effect handler oriented programming* in which the meaning of effectful programs may be decomposed into a collection of fine-grained effect handlers.

The key enabler for seamlessly composition is *first-class control*, which provides a mechanism for reifying the program control state as a first-class data object known as a continuation [104]. Through structured manipulation of continuations control gets transferred between programs and their handlers.

In this dissertation I present a practical design for programming languages with support for effect handler oriented programming, I develop two foundational implementation techniques for effect handlers, and I study their inherent computational expressiveness and efficiency.

1.1 Why first-class control matters

First things first, let us settle on the meaning of the qualifier ‘first-class’. A programming language entity (or citizen) is regarded as being first-class if it can be used on an equal footing with other entities. A familiar example is functions as first-class values. A first-class function may be treated like any other primitive value, i.e. passed as an argument to other functions, returned from functions, stored in data structures, or let-bound.

First-class control makes the control state of the program available as a first-class value known as a continuation object at any point during evaluation [104]. This object comes equipped with at least one operation for restoring the control state. As such the control flow of the program becomes a first-class entity that the programmer may manipulate to implement interesting control phenomena.

From the perspective of programmers first-class control is a valuable programming feature because it enables them to implement their own control idioms, such as async/await [259], as if they were native to the programming language. More important, with first-class control programmer-defined control idioms are local phenomena which can be encapsulated in a library such that the rest of the program does not need to be made aware of their existence. Conversely, without first-class control some control idioms can only be implemented using global program restructuring techniques such as continuation passing style.

From the perspective of compiler engineers first-class control is valuable because it unifies several control-related constructs under one single construct. First-class control can even be beneficial for implementing programming languages which have no notion of first-class control in source language. A runtime with support for first-class control can considerably simplify and ease maintainability of an implementation of a programming language with various distinct second-class control idioms such as async/await [259], coroutines [69], etc, because compiler engineers need only implement and maintain a single control mechanism rather than having to implement and maintain individual runtime support for each control idiom of the source language.

The idea of first-class control is old. It was conceived already during the design of the programming language Algol [13] (one of the early high-level programming languages along with Fortran [12] and Lisp [192]) when Landin [161] sought to model unrestricted goto-style jumps using an extended λ-calculus. Since then a wide variety of first-class control operators have appeared. We can coarsely categorise them into two groups: *undelimited* and *delimited* (in Chapter A we will perform a finer analysis

of first-class control). Undelimited control operators are global phenomena that let programmers capture the entire control state of their programs, whereas delimited control operators are local phenomena that provide programmers with fine-grain control over which parts of the control state to capture. Thus there are good reasons for preferring delimited control over undelimited control for practical programming.

1.1.1 Why effect handlers matter

The problem with traditional delimited control operators such as Danvy and Filinski's shift/reset [62] or Felleisen's control/prompt [81] is that they hard-wire an implementation for the *control effect* interface, which provides only a single operation for reifying the control state. In itself this interface does not limit what effects are expressible as the control effect is in a particular sense 'the universal effect' because it can simulate any other computational effect [88].

The problem, meanwhile, is that the universality of the control effect hinders modular programming as the control effect is inherently unstructured. In essence, programming with traditional delimited control to simulate effects is analogous to programming with the universal type [176] in statically typed programming languages, and having to program with the universal type is usually a telltale that the programming abstraction is inadequate for the intended purpose.

In contrast, effect handlers provide a structured form of delimited control, where programmers can give distinct names to control reifying operations and separate them from their handling. Throughout this dissertation we will see numerous examples of how effect handlers makes programming with delimited structured (c.f. the following section, Chapter 2, and Chapter A.).

1.2 State of effectful programming

Functional programmers tend to view programs as impenetrable black boxes, whose outputs are determined entirely by their inputs [129, 132]. This is a compelling view which admits a canonical mathematical model of computation [45, 46]. Alas, this view does not capture the reality of practical programs, which interact with their environment. Functional programming prominently features two distinct, but related, approaches to effectful programming, which Filinski [88] succinctly characterises as *effects as data* and *effects as behaviour*. The former uses data abstraction to encapsulate effects [203,

267] which is compelling because it recovers some of benefits of the black box view for effectful programs, though, at the expense of a change of programming style [137]. The latter retains the usual direct style of programming either by hard-wiring the semantics of the effects into the language or by more flexible means via first-class control.

In this section I will provide a brief perspective on different approaches to programming with effects along with an informal introduction to the related concepts. We will look at each approach through the lens of global mutable state — the "hello world" of effectful programming.

1.2.1 Direct-style state

We can realise stateful behaviour by either using language-supported state primitives, globally structure our program to follow a certain style, or using first-class control in the form of delimited control to simulate state. We do not consider undelimited control, because it is insufficient to express mutable state [103].

Builtin mutable state

It is common to find mutable state builtin into the semantics of mainstream programming languages. However, different languages vary in their approach to mutable state. For instance, state mutation underpins the foundations of imperative programming languages belonging to the C family of languages. They typically do not distinguish between mutable and immutable values at the level of types. On the contrary, programming languages belonging to the ML family of languages use types to differentiate between mutable and immutable values. They reflect mutable values in types by using a special unary type constructor $\mathsf{Ref}^{\mathsf{Type}\to\mathsf{Type}}$. Furthermore, ML languages equip the Ref constructor with three operations.

$$\mathbf{ref} : S \to \mathsf{Ref}\ S \qquad ! : \mathsf{Ref}\ S \to S \qquad := : \mathsf{Ref} \to S \to \langle\rangle$$

The first operation *initialises* a new mutable state cell of type S; the second operation *gets* the value of a given state cell; and the third operation *puts* a new value into a given state cell. It is important to note that getting the value of a state cell does not alter its contents, whilst putting a value into a state cell overrides the previous contents.

The following function illustrates a use of the get and put primitives to manipulate

the contents of some global state cell *st*.

$$\begin{aligned}&\mathsf{incrEven} : 1 \to \mathsf{Bool}\\ &\mathsf{incrEven}\,\langle\rangle \stackrel{\text{def}}{=} \mathbf{let}\ v \leftarrow !st\ \mathbf{in}\ st := 1 + v; \mathsf{even}\ v\end{aligned}$$

The type signature is oblivious to the fact that the function internally makes use of the state effect to compute its return value. The body of the function first retrieves the current value of the state cell and binds it to *st*. Subsequently, it destructively increments the value of the state cell. Finally, it applies the predicate $\mathsf{even} : \mathsf{Int} \to \mathsf{Bool}$ to the original state value to test whether its parity is even (this example function is a slight variation of an example by Gibbons [107]). We can run this computation as a subcomputation in the context of global state cell *st*.

$$\mathbf{let}\ st \leftarrow \mathbf{ref}\ 4\ \mathbf{in}\ \langle \mathsf{incrEven}\,\langle\rangle; !st\rangle \rightsquigarrow^{+} \langle \mathsf{true}; 5\rangle : \mathsf{Bool} \times \mathsf{Int}$$

Operationally, the whole computation initialises the state cell *st* to contain the integer value 4. Subsequently it runs the incrEven computation, which returns the boolean value true and as a side-effect increments the value of *st* to be 5. The whole computation returns the boolean value paired with the final value of the state cell.

Transparent state-passing purely functionally

It is possible to implement stateful behaviour in a language without any computational effects, e.g. simply typed λ-calculus, by following a particular design pattern known as *state-passing*. The principal idea is to parameterise stateful functions by the current state and make them return whatever result they compute along with the updated state value. More precisely, in order to endow some n-ary function with argument types A_i and return type R with state of type S, we transform the function signature as follows.

$$[\![A_1 \to \cdots \to A_n \to R]\!]_S \stackrel{\text{def}}{=} A_1 \to \cdots \to A_n \to S \to R \times S$$

By convention we always insert the state parameter at the tail end of the parameter list. We may read the suffix $S \to R \times S$ as a sort of effect annotation indicating that a particular function utilises state. The downside of state-passing is that it is a global technique which requires us to rewrite the signatures (and their implementations) of all functions that makes use of state.

We can reimplement the incrEven in state-passing style as follows.

$$\begin{aligned}&\mathsf{incrEven} : 1 \to \mathsf{Int} \to \mathsf{Bool} \times \mathsf{Int}\\ &\mathsf{incrEven}\,\langle\rangle \stackrel{\text{def}}{=} \lambda st.\langle \mathsf{even}\ st; 1 + st\rangle\end{aligned}$$

State initialisation is simply function application.

$$\mathsf{incrEven}\ \langle\rangle\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5\rangle : \mathsf{Bool} \times \mathsf{Int}$$

Programming in state-passing style is laborious and no fun as it is anti-modular, because for effect-free higher-order functions to work with stateful functions they too must be transformed or at the very least be duplicated to be compatible with stateful function arguments. Nevertheless, state-passing is an important technique as it is the secret sauce that enables us to simulate mutable state with other programming techniques.

Opaque state-passing with delimited control

Delimited control appears during the late 80s in different forms [61, 251]. There are several different forms of delimited control. The particular form of delimited control that I will use here is due to Danvy and Filinski [61]. Nevertheless, the secret sauce of all forms of delimited control is that a delimited control operator makes it possible to pry open function boundaries as control may transfer out of an arbitrary evaluation context, leaving behind a hole that can later be filled by some value supplied externally.

Danvy and Filinski's formulation of delimited control introduces two primitives.

$$\langle - \rangle : (1 \to R) \to R \qquad \textbf{shift} : ((A \to R) \to R) \to A$$

The first primitive $\langle - \rangle$ (pronounced 'reset') is a control delimiter. Operationally, reset evaluates a given thunk in an empty evaluation context and returns the final result of that evaluation. The second primitive **shift** is a control reifier. An application **shift** reifies and erases the control state up to (but not including) the nearest enclosing reset. The reified control state represents the continuation of the invocation of **shift** (up to the innermost reset); it gets passed as a function to the argument of **shift**.

We define both primitives over some fixed return type R (an actual practical implementation would use polymorphism to make them more flexible). By instantiating $R = S \to A \times S$, where S is the type of state and A is the type of return values, then we can use shift and reset to simulate mutable state using state-passing in way that is opaque to the rest of the program. Let us first define operations for accessing and modifying the state cell.

$$\begin{array}{ll} \mathsf{get} : 1 \to S & \mathsf{put} : S \to 1 \\ \mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \textbf{shift}\,(\lambda k.\lambda st.k\ st\ st) & \mathsf{put}\ st \stackrel{\text{def}}{=} \textbf{shift}\,(\lambda k.\lambda st'.k\ \langle\rangle\ st) \end{array}$$

The body of get applies **shift** to capture the current continuation, which gets supplied to the anonymous function $(\lambda k.\lambda st.k\ st\ st)$. The continuation parameter k has type

$S \to S \to A \times S$. The continuation is applied to two instances of the current state value st. The first instance is the value returned to the caller of get, whilst the second instance is the state value available during the next invocation of either get or put. This aligns with the intuition that accessing a state cell does not modify its contents. The implementation of put is similar, except that the first argument to k is the unit value, because the caller of put expects a unit in return. Also, it ignores the current state value st' and instead passes the state argument st onto the activation of the next state operation. Again, this aligns with the intuition that modifying a state cell destroys its previous contents.

Using these two operations we can implement a version of incrEven that takes advantage of delimited control to simulate global state.

$$\begin{aligned}&\mathsf{incrEven} : 1 \to \mathsf{Bool}\\&\mathsf{incrEven}\,\langle\rangle \stackrel{\text{def}}{=} \mathbf{let}\ st \leftarrow \mathsf{get}\,\langle\rangle\ \mathbf{in}\ \mathsf{put}\,(1+st);\ \mathsf{even}\ st\end{aligned}$$

Modulo naming of operations, this version is similar to the version that uses builtin state. The type signature of the function is even the same. Before we can apply this function we must first implement a state initialiser.

$$\begin{aligned}&\mathsf{runState} : (1 \to A) \to S \to A \times S\\&\mathsf{runState}\ m\ st_0 \stackrel{\text{def}}{=} \langle \lambda\langle\rangle.\mathbf{let}\ x \leftarrow m\,\langle\rangle\ \mathbf{in}\ \lambda st.\langle x; st\rangle\rangle\ st_0\end{aligned}$$

The function runState acts as both the state cell initialiser and runner of the stateful computation. The first parameter m is a thunk that may perform stateful operations and the second parameter st_0 is the initial value of the state cell. The implementation wraps an instance of reset around the application of m in order to delimit the extent of applications of **shift** within m. It is important to note that each invocation of get and put gives rise to a state-accepting function, thus when m is applied a chain of state-accepting functions gets constructed lazily. The chain ends in the state-accepting function returned by the reset instance. The application of the reset instance to st_0 effectively causes evaluation of each function in this chain to start.

After instantiating $A = \mathsf{Bool}$ and $S = \mathsf{Int}$ we can use the runState function to apply the incrEven function.

$$\mathsf{runState}\ \mathsf{incrEven}\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5\rangle : \mathsf{Bool} \times \mathsf{Int}$$

1.2.2 Monadic state

During the late 80s and early 90s monads rose to prominence as a practical programming idiom for structuring effectful programming [137, 138, 201, 203, 267–269]. The

concept of monad has its origins in category theory and its mathematical nature is well-understood [31, 187]. The emergence of monads as a programming abstraction began when Moggi [201, 203] proposed to use monads as the mathematical foundation for modelling computational effects in denotational semantics. Moggi's view was that *monads determine computational effects*. The key property of this view is that pure values of type A are distinguished from effectful computations of type $T\ A$, where T is the monad representing the effect(s) of the computation. This view was put into practice by Wadler [267, 269], who popularised monadic programming in functional programming by demonstrating how monads increase the ease at which programs may be retrofitted with computational effects. In practical programming terms, monads may be thought of as constituting a family of design patterns, where each pattern gives rise to a distinct effect with its own collection of operations. Part of the appeal of monads is that they provide a structured interface for programming with effects such as state, exceptions, nondeterminism, interactive input and output, and so forth, whilst preserving the equational style of reasoning about pure functional programs [107, 108].

The presentation of monads here is inspired by Wadler's presentation of monads for functional programming [267], and it ought to be familiar to users of Haskell [138].

Definition 1.1. A monad is a triple $(T^{\mathrm{Type}\to\mathrm{Type}}, \mathbf{return}, \gg\!=)$ where T is some unary type constructor, **return** is an operation that lifts an arbitrary value into the monad (sometimes this operation is called 'the unit operation'), and $\gg\!=$ is the application operator of the monad (this operator is pronounced 'bind'). Adequate implementations of **return** and $\gg\!=$ must conform to the following interface.

$$\mathbf{return} : A \to T\ A \qquad\qquad \gg\!= : T\ A \to (A \to T\ B) \to T\ B$$

Interactions between **return** and $\gg\!=$ are governed by the monad laws.

Left identity	$\mathbf{return}\ x \gg\!= k \ = \ k\ x$
Right identity	$m \gg\!= \mathbf{return} \ = \ m$
Associativity	$(m \gg\!= k) \gg\!= f \ = \ m \gg\!= (\lambda x.k\ x \gg\!= f)$

We may understand the type $T\ A$ as inhabiting computations that compute a *tainted* value of type A. In this regard, we may understand T as denoting the taint involved in computing A, i.e. we can think of T as sort of effect annotation which informs us about which effectful operations the computation may perform to produce A. The monad interface may be instantiated in different ways to realise different computational effects.

In the following subsections we will see three different instantiations with which we will implement global mutable state.

Monadic programming is a top-down approach to effectful programming, where the concrete monad structure is taken as a primitive which controls interactions between effectful operations. The monad laws ensure that monads have some algebraic structure, which programmers can use when reasoning about their monadic programs. Similarly, optimising compilers may take advantage of the structure to emit more efficient code.

The success of monads as a programming idiom is difficult to understate as monads have given rise to several popular control-oriented programming abstractions including the asynchronous programming idiom async/await [47, 172, 259].

State monad

The state monad is an instantiation of the monad interface that encapsulates mutable state by using the state-passing technique internally. In addition it equips the monad with two operations for manipulating the state cell.

Definition 1.2. The state monad is defined over some fixed state type S.

$$T\ A \stackrel{\text{def}}{=} S \to A \times S$$

$$\begin{array}{ll} \mathbf{return} : A \to T\ A & \gg\!= : T\ A \to (A \to T\ B) \to T\ B \\ \mathbf{return}\ x \stackrel{\text{def}}{=} \lambda st.\langle x; st\rangle & \gg\!= \stackrel{\text{def}}{=} \lambda m.\lambda k.\lambda st.\mathbf{let}\ \langle x; st'\rangle = m\ st\ \mathbf{in}\ (k\ x)\ st' \end{array}$$

The **return** of the monad is a state-accepting function of type $S \to A \times S$ that returns its first argument paired with the current state. The bind operator also produces a state-accepting function of type $S \to A \times S$. The bind operator first supplies the current state st to the monad argument m. This application yields a value result of type A and an updated state st'. The result is supplied to the continuation k, which produces another state accepting function that gets applied to the previously computed state value st'.

The state monad is equipped with two dual operations for accessing and modifying the state encapsulated within the monad.

$$\begin{array}{ll} \mathsf{get} : 1 \to T\ S & \mathsf{put} : S \to T\ 1 \\ \mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \lambda st.\langle st; st\rangle & \mathsf{put}\ st \stackrel{\text{def}}{=} \lambda st'.\langle\langle\rangle; st\rangle \end{array}$$

Interactions between the two operations satisfy the following equations [107].

$$\begin{array}{lrcl} \mathsf{Get\text{-}get} & \mathsf{get}\,\langle\rangle \gg\!= (\lambda st.\mathsf{get}\,\langle\rangle \gg\!= (\lambda st'.k\ st\ st')) & = & \mathsf{get} \gg\!= \lambda st.k\ st\ st \\ \mathsf{Get\text{-}put} & \mathsf{get}\,\langle\rangle \gg\!= (\lambda st.\mathsf{put}\ st) & = & \mathbf{return}\ \langle\rangle \\ \mathsf{Put\text{-}get} & \mathsf{put}\ st \gg\!= (\lambda\langle\rangle.\mathsf{get}\,\langle\rangle \gg\!= (\lambda st.k\ st') & = & \mathsf{put}\ st \gg\!= (\lambda\langle\rangle.k\ st) \\ \mathsf{Put\text{-}put} & \mathsf{put}\ st \gg\!= (\lambda st.\mathsf{put}\ st') & = & \mathsf{put}\ st' \gg\!= (\lambda\langle\rangle.k\ st) \end{array}$$

The first equation states that performing one get after another get is redundant. The second equation captures the intuition that getting a value and then putting has no observable effect on the state cell. The third equation states that performing a get immediately after putting a value is equivalent to returning that value. The fourth equation states that only the latter of two consecutive puts is observable.

The literature often uses the presentation (or a similar one) with the four equations above, even though, there exists a smaller presentation in which the first equation is redundant as it is derivable from the second and third equations (c.f. Appendix B).

We can implement a monadic variation of the incrEven function that uses the state monad to emulate manipulations of the state cell as follows.

$$T\ A \stackrel{\text{def}}{=} \mathsf{Int} \to A \times \mathsf{Int}$$

$$\mathsf{incrEven} : 1 \to T\ \mathsf{Bool}$$

$$\mathsf{incrEven}\ \langle\rangle \stackrel{\text{def}}{=} \mathsf{get}\ \langle\rangle \gg\!= (\lambda st.\mathsf{put}\ (1 + st) \gg\!= \lambda\langle\rangle.\mathbf{return}\ (\mathsf{even}\ st)))$$

We fix the state type of our monad to be the integer type. The type signature of the function incrEven may be read as describing a thunk that returns a boolean value, and whilst computing this boolean value the function may perform any effectful operations given by the monad T [203, 267], i.e. get and put. Operationally, the function retrieves the current value of the state cell via the invocation of get. The bind operator passes this value onto the continuation, which increments the value and invokes put. The continuation applies a predicate the even predicate to the original state value. The structure of the monad means that the result of running this computation gives us a pair consisting of boolean value indicating whether the initial state was even and the final state value.

The state initialiser and monad runner is simply thunk forcing and function application combined.

$$\mathsf{runState} : (1 \to T\ A) \to S \to A \to S$$

$$\mathsf{runState}\ m\ st_0 \stackrel{\text{def}}{=} m\ \langle\rangle\ st_0$$

By instantiating $S = \mathsf{Int}$ and $A = \mathsf{Bool}$ we can obtain the same result as before.

$$\mathsf{runState}\ \mathsf{incrEven}\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5\rangle : \mathsf{Bool} \times \mathsf{Int}$$

We can instantiate the monad structure in a similar way to simulate other computational effects such as exceptions, nondeterminism, concurrency, and so forth [203, 267].

Continuation monad

As in J.R.R. Tolkien's fictitious Middle-earth [264] there exists one monad to rule them all, one monad to realise them, one monad to subsume them all, and in the term language bind them. This powerful monad is the *continuation monad*.

The continuation monad may be regarded as 'the universal monad' as it can embed any other monad, and thereby simulate any computational effect [89]. It derives its name from its connection to continuation passing style [267], which is a particular style of programming where each function is parameterised by the current continuation (we will discuss continuation passing style in detail in Chapter 4). The continuation monad is powerful exactly because each of its operations has access to the current continuation.

Definition 1.3. The continuation monad is defined over some fixed return type R [267].

$$T\ A \stackrel{\text{def}}{=} (A \to R) \to R$$

$$\textbf{return} : A \to T\ A \qquad\qquad \gg\!= : T\ A \to (A \to T\ B) \to T\ B$$

$$\textbf{return}\ x \stackrel{\text{def}}{=} \lambda k.k\ x \qquad\qquad \gg\!= \stackrel{\text{def}}{=} \lambda m.\lambda k.\lambda c.m\,(\lambda x.k\ x\ c)$$

The **return** operation lifts a value into the monad by using it as an argument to the continuation k. The bind operator binds the current continuation to c. In the body it applies the monad m to an anonymous continuation function of type $A \to T\ B$. Internally, the monad m will apply this continuation when it is on the form **return**. Thus the parameter x gets bound to the **return** value of the monad. This parameter gets supplied as an argument to the next monadic action k alongside the current continuation c.

If we instantiate $R = S \to A \times S$ for some type S then we can implement the state monad inside the continuation monad.

$$\mathsf{get} : 1 \to T\ S \qquad\qquad \mathsf{put} : S \to T\ 1$$

$$\mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \lambda k.\lambda st.k\ st\ st \qquad\qquad \mathsf{put}\ st \stackrel{\text{def}}{=} \lambda k.\lambda st'.k\ \langle\rangle\ st$$

The get operation takes as input a (binary) continuation k of type $S \to S \to A \times S$ and produces a state-accepting function that applies the continuation to the given state st. The first occurrence of st is accessible to the caller of get, whilst the second occurrence passes the value st onto the next operation invocation on the monad. The operation put works in the same way. The primary difference is that put does not return the value of the state cell; instead it returns simply the unit value $\langle\rangle$. One can show that this implementation of get and put abides by the same equations as the implementation given in Definition 1.2.

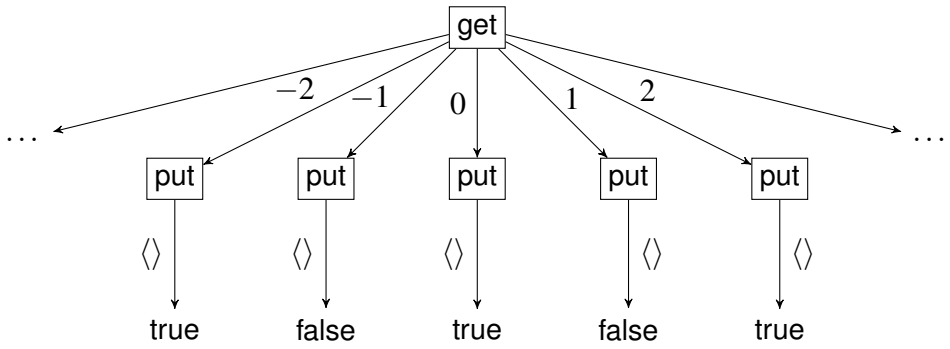


Figure 1.1: Computation tree for incrEven.

The state initialiser and runner for the monad supplies the initial continuation.

$$\begin{aligned}&\mathsf{runState} : (1 \to T\ A) \to S \to A \times S\\&\mathsf{runState}\ m\ st_0 \stackrel{\text{def}}{=} m\ \langle\rangle\ (\lambda x.\lambda st.\langle x; st\rangle)\ st_0\end{aligned}$$

The initial continuation $(\lambda x.\lambda st.\langle x; st\rangle)$ corresponds to the **return** of the state monad. By fixing $S = \mathsf{Int}$ and $A = \mathsf{Bool}$, we can use the continuation monad to interpret incrEven.

$$\mathsf{runState}\ \mathsf{incrEven}\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5\rangle : \mathsf{Bool} \times \mathsf{Int}$$

The continuation monad gives us a succinct framework for implementing and programming with computational effects, however, it comes at the expense of extensibility and modularity. Adding a new operation to the monad may require modifying its internal structure, which entails a complete reimplementation of any existing operations.

Free monad

The state monad and the continuation monad offer little flexibility with regards to the concrete interpretation of state as in both cases the respective monad hard-wires a particular interpretation. An alternative is the *free monad* which decouples the structure of the monad from its interpretation. Just like other monads the free monad satisfies the monad laws, however, unlike other monads the free monad does not perform any computation *per se*. Instead the free monad builds an abstract representation of the computation in form of a computation tree, whose interior nodes correspond to an invocation of some operation on the monad, where each outgoing edge correspond to a possible continuation of the operation; the leaves correspond to return values. Figure 1.1 depicts the computation tree for the incrEven function. This particular computation tree has infinite width, because the operation get has infinitely many possible continuations (we take

the denotation of Int to be $\mathbb{Z}$). Conversely, each put node has only one outgoing edge, because put has only a single possible continuation, namely, the trivial continuation $\langle\rangle$.

The meaning of a free monadic computation is ascribed by a separate function, or interpreter, that traverses the computation tree. The shape of computation trees is captured by the following generic type definition.

$$\mathsf{Free}\ F\ A \stackrel{\text{def}}{=} [\mathsf{Return} : A | \mathsf{Op} : F\ (\mathsf{Free}\ F\ A)]$$

The type constructor Free takes two type arguments. The first parameter F is itself a type constructor of kind Type $\to$ Type. The second parameter is the usual type of values computed by the monad. The **return** tag creates a leaf of the computation tree, whilst the Op tag creates an interior node. In the type signature for Op the type variable F is applied to the Free type. The idea is that $F\ K$ computes an enumeration of the signatures of the possible operations on the monad, where K is the type of continuation for each operation. Thus the continuation of an operation is another computation tree node.

Definition 1.4. The free monad is a triple $(F^{\text{Type}\to\text{Type}}, \mathbf{return}, \gg\!=)$ which forms a monad with respect to F. In addition an adequate instance of F must supply a map, $\mathsf{fmap} : (A \to B) \to F\ A \to F\ B$, over its structure (in more precise technical terms: F must be a *functor* [31]).

$$T\ A \stackrel{\text{def}}{=} \mathsf{Free}\ F\ A$$

$$\mathbf{return} : A \to T\ A \qquad \gg\!= : T\ A \to (A \to T\ B) \to T\ B$$

$$\mathbf{return}\ x \stackrel{\text{def}}{=} \mathsf{Return}\ x \qquad \gg\!= \stackrel{\text{def}}{=} \lambda m.\lambda k.\mathbf{case}\ m\ \{\mathsf{Return}\ x \mapsto k\ x;\ \mathsf{Op}\ y \mapsto \mathsf{Op}\ (\mathsf{fmap}\ (\lambda m'.m' \gg\!= k)\ y)\}$$

The **return** operation simplify reflects itself by injecting the value x into the computation tree as a leaf node. The bind operator threads the continuation k through the computation tree. Upon encounter a leaf node the continuation gets applied to the value of the node. Note how this is reminiscent of the **return** of the continuation monad. The bind operator works in tandem with the fmap of F to advance past Op nodes. The fmap function is responsible for applying its functional argument to the next computation tree node which is embedded inside y. We define an auxiliary function to alleviate some of the boilerplate involved with performing operations on the monad.

$$\mathsf{do} : F\ A \to \mathsf{Free}\ F\ A$$
$$\mathsf{do}\ op \stackrel{\text{def}}{=} \mathsf{Op}\ (\mathsf{fmap}\ (\lambda x.\mathsf{Return}\ x)\ op)$$

This function injects some operation op into the computation tree as an operation node.

In order to implement state with the free monad we must first declare a signature of its operations and implement the required fmap for the signature.

$$\begin{aligned}
&\mathsf{FreeState}\ S\ R \stackrel{\text{def}}{=} [\mathsf{Get} : S \to R | \mathsf{Put} : S \times (1 \to R)] \\
&\mathsf{fmap} : (A \to B) \to \mathsf{FreeState}\ S\ A \to \mathsf{FreeState}\ S\ B \\
&\mathsf{fmap}\ f\ op \stackrel{\text{def}}{=} \mathbf{case}\ op\ \{\mathsf{Get}\ k \quad \mapsto \mathsf{Get}\ (\lambda st.f\ (k\ st)); \\
&\qquad\qquad\qquad\qquad \mathsf{Put}\ \langle st'; k\rangle \mapsto \mathsf{Put}\ \langle st'; \lambda\langle\rangle.f\ (k\ \langle\rangle)\rangle\}
\end{aligned}$$

The signature FreeState declares the two stateful operations Get and Put over state type S and continuation type R. The Get tag is parameterised a continuation function of type $S \to R$. The idea is that an application of this function provides access to the current state, whilst computing the next node of the computation tree. The Put operation is parameterised by the new state value and a thunk, which computes the next computation tree node. The fmap instance applies the function f to the continuation k of each operation. By instantiating $F = \mathsf{FreeState}\,S$ and using the do function we can give the get and put operations a familiar look and feel.

$$\begin{aligned}
&\mathsf{get} : 1 \to T\ S && \mathsf{put} : S \to T\ 1 \\
&\mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \mathsf{do}\ (\mathsf{Get}\ (\lambda st.st)) && \mathsf{put}\ st \stackrel{\text{def}}{=} \mathsf{do}\ (\mathsf{Put}\ \langle st; \lambda\langle\rangle.\langle\rangle\rangle)
\end{aligned}$$

Both operations are performed with the identity function as their respective continuation function. We do not have much choice in this regard as for instance in the case of get we must ultimately return a computation of type $T\ S$, and the only value of type S we have access to in this context is the one supplied externally to the continuation function.

The state initialiser and runner for the monad is an interpreter. As the programmers, we are free to choose whatever interpretation of state we desire. For example, the following interprets the stateful operations using the state-passing technique.

$$\begin{aligned}
&\mathsf{runState} : (1 \to \mathsf{Free}\ (\mathsf{FreeState}\ S)\ R) \to S \to R \times S \\
&\mathsf{runState}\ m\ st \stackrel{\text{def}}{=} \mathbf{case}\ m\ \langle\rangle\ \{\mathsf{Return}\ x \quad \mapsto (x, st); \\
&\qquad\qquad\qquad\qquad \mathsf{Op}\ (\mathsf{Get}\ k) \quad \mapsto \mathsf{runState}\ (\lambda\langle\rangle.k\ st)\ st; \\
&\qquad\qquad\qquad\qquad \mathsf{Op}\ (\mathsf{Put}\ \langle st'; k\rangle) \mapsto \mathsf{runState}\ k\ st'\}
\end{aligned}$$

The interpreter implements a *fold* over the computation tree by pattern matching on the shape of the tree (or equivalently monad) [194]. In the case of a **return** node the interpreter returns the payload x along with the final state value st. If the current node is a Get operation, then the interpreter recursively calls itself with the same state value st and a thunked application of the continuation k to the current state st. The recursive

activation of runState will force the thunk in order to compute the next computation tree node. In the case of a Put operation the interpreter calls itself recursively with new state value st' and the continuation k (which is a thunk). One may prove that this interpretation of get and put satisfies the equations of Definition 1.2.

By instantiating $S = \mathsf{Int}$ and $R = \mathsf{Bool}$ we can use this interpreter to run incrEven.

$$\mathsf{runState\ incrEven\ 4} \rightsquigarrow^{+} \langle \mathsf{true}; 5 \rangle : \mathsf{Bool} \times \mathsf{Int}$$

The free monad brings us close to the essence of programming with effect handlers.

1.2.3 Back to direct-style

Monads do not freely compose, because monads must satisfy a distributive property in order to combine [145]. Alas, not every monad has a distributive property. The lack of composition is to an extent remedied by monad transformers, which provide a programmatic abstraction for stacking one monad on top of another [77]. The problem with monad transformers is that they enforce an ordering on effects that affects the program semantics (c.f. my MSc dissertation for a concrete example of this [125]).

However, a more fundamental problem with monads is that they break the basic doctrine of modular abstraction, which says we should program against an abstract interface, not an implementation. Effectful programming using monads fixates on the concrete structure first, and adds effect operations second. As a result monadic effect operations are intimately tied to the concrete structure of their monad.

Before moving onto direct-style alternatives, it is worth mentioning McBride and Paterson's idioms (known as applicative functors in Haskell) as an alternative to monadic programming [191]. Idioms provide an applicative style for programming with effects. Even though idioms are computationally weaker than monads, they are still capable of encapsulating a wide range of computational effects whose realisation do not require the full monad structure (consult Yallop [278] for a technical analysis of idioms and monads). Another thing worth pointing out is that it is possible to have a direct-style interface for effectful programming in the source language, which the compiler can translate into monadic binds and returns automatically. For a concrete example of this see the work of Vazou and Leijen [265].

Let us wrap up this crash course in effectful programming by looking at two approaches for programming in direct-style with effects that make structured use of delimited control, before finishing with a brief discussion of effect tracking.

Monadic reflection on state

Monadic reflection is a technique due to Filinski [87, 88, 89, 90] which makes use of delimited control to perform a local switch from monadic style into direct-style and vice versa. The key insight is that a control reifier provides an escape hatch that makes it possible for computation to locally jump out of the monad, as it were. The scope of this escape hatch is restricted by the control delimiter, which forces computation back into the monad. Monadic reflection introduces two operators, which are defined over some monad T and some fixed result type R.

$$\begin{array}{ll} \downarrow : (1 \to R) \to T\ R & \uparrow : T\ A \to A \\ \downarrow m \stackrel{\text{def}}{=} \langle \lambda\langle\rangle.\mathbf{return}\,(m\,\langle\rangle)\rangle & \uparrow m \stackrel{\text{def}}{=} \mathbf{shift}\,(\lambda k.m \gg\!= k) \end{array}$$

The first operator $\downarrow$ (pronounced 'reify') performs *monadic reification*. Semantically it makes the effect corresponding to T transparent. The implementation installs a reset instance to delimit control effects of m. The result of forcing m gets lifted into the monad T. The second operator $\uparrow$ (pronounced 'reflect') performs *monadic reflection*. It makes the effect corresponding to T opaque. The implementation applies **shift** to capture the current continuation (up to the nearest instance of reset). Subsequently, it evaluates the monadic computation m and passes the result of this evaluation to the continuation k, which effectively performs the jump out of the monad.

Suppose we instantiate $T = \mathsf{State}\ S$ for some type S, then we can realise direct-style versions of the state operations get and put, whose internal implementations make use of the monadic state operations.

$$\begin{array}{ll} \mathsf{get} : 1 \to S & \mathsf{put} : S \to 1 \\ \mathsf{get}\,\langle\rangle \stackrel{\text{def}}{=} \uparrow (T.\mathsf{get}\,\langle\rangle) & \mathsf{put}\ st \stackrel{\text{def}}{=} \uparrow (T.\mathsf{put}\,st) \end{array}$$

I am slightly abusing notation here as I use component selection notation on the constructor type T in order to disambiguate the reflected operation names and monadic operation names. Nevertheless, the implementations of get and put simply reflect their monadic counterparts. Note that the type signatures are the same as the signatures for operations that we implemented using shift/reset in Section 1.2.1.

The initialiser and runner for some reflected stateful computation is defined in terms of the state monad runner.

$$\begin{array}{l} \mathsf{runState} : (1 \to R) \to S \to R \times S \\ \mathsf{runState}\ m\ st_0 \stackrel{\text{def}}{=} T.\mathsf{runState}\ (\lambda\langle\rangle.\downarrow m)\ st_0 \end{array}$$

The runner reifies the computation m to obtain an instance of the state monad, which it then runs using the state monad implementation of runState.

Since this state interface is the same as shift/reset-based interface, we can simply take a carbon copy of the shift/reset-based implementation of incrEven and run it after instantiating $R = \mathsf{Bool}$ and $S = \mathsf{Int}$.

$$\mathsf{runState}\ \mathsf{incrEven}\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5 \rangle : \mathsf{Bool} \times \mathsf{Int}$$

Handling state

At the start of the 00s decade Plotkin and Power [224, 225, 226] introduced algebraic theories of computational effects, or simply *algebraic effects*, which inverts Moggi's view of effects such that *computational effects determine monads*. In their view a computational effect is described by an algebraic effect, which consists of a signature of abstract operations and a collection of equations that govern their behaviour, together they generate a free monad rather than the other way around. Algebraic effects provide a bottom-up approach to effectful programming in which abstract effect operations are taken as primitive. Using these operations we may build up concrete structures. In practical programming terms, we may understand an algebraic effect as an abstract interface, whose operations build the underlying free monad.

Definition 1.5. An algebraic effect is given by a pair $\mathcal{T} = (\Sigma, \mathsf{E})$ consisting of an effect signature $\Sigma = \{(\ell_i : A \twoheadrightarrow B)_i\}_i$ of typed operation symbols ℓ_i, whose interactions are govern by set of equations E. We will not concern ourselves with the mathematical definition of equation, as in this dissertation we will always fix $\mathsf{E} = \emptyset$, meaning that the interactive patterns of operations are unrestricted. As a consequence we will regard an operation symbol as a syntactic entity subject only to a static semantics. The type $A \twoheadrightarrow B$ denotes the space of operations whose payload has type A and whose interpretation yields a value of type B.

As with the free monad, the meaning of an algebraic effect operation is conferred by some separate interpreter. In the algebraic theory of computational effects such interpreters are known as handlers for algebraic effects, or simply *effect handlers*. They were introduced by Plotkin and Pretnar [227, 228] by the end of the decade. A crucial difference between effect handlers and interpreters of free monads is that effect handlers use delimited control to realise the behaviour of computational effects. Practical programming with effect handlers was popularised by Kammar et al. [143], who advocated algebraic effects and their handlers as a modular basis for effectful programming.

Effect handlers introduce two dual control constructs.

$$\mathbf{do}\ \ell^{A \twoheadrightarrow B}\ V^A : B \qquad\qquad \mathbf{handle}\ M^C\ \mathbf{with}\ H^{C \Rightarrow D} : D$$

$$H ::= \{\mathbf{return}\ x^C \mapsto N^D\} \mid \{\langle\!\langle \ell^{A \twoheadrightarrow B}\ p^A \twoheadrightarrow k^{B \to D} \rangle\!\rangle \mapsto N^D\} \uplus H^{C \Rightarrow D}$$

The **do** construct reifies the control state up to a suitable handler and packages it up with the operation symbol ℓ and its payload V before transferring control to the suitable handler. As control is transferred a hole is left in the evaluation context that must be filled before evaluation can continue. The **handle** construct delimits **do** invocations within the computation M according to the handler definition H. Handler definitions consist of the union of a single **return**-clause and the disjoint union of zero or more operation clauses. The **return**-clause specifies what to do with the return value of a computation. An operation clause $\langle\!\langle \ell\ p \twoheadrightarrow k \rangle\!\rangle$ matches on an operation symbol and binds its payload to p and its continuation k. Note that the domain type of the continuation agrees with the codomain type of the operation symbol, and the codomain type of the continuation agrees with the codomain type of the handler definition. Continuation application fills the hole left by the **do** construct, thus providing a value interpretation of the invocation. The continuation returns inside the handler once the **return**-clause computation has finished. Operationally, effect handlers may be regarded as an extension of Benton and Kennedy [18] style exception handlers.

We can implement mutable state with effect handlers as follows.

$$\Sigma \stackrel{\text{def}}{=} \{\mathsf{Get} : 1 \twoheadrightarrow S; \mathsf{Put} : S \twoheadrightarrow 1\}$$

$$\mathsf{get} : 1 \to S \qquad\qquad \mathsf{put} : S \to 1$$

$$\mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Get}\ \langle\rangle \qquad\qquad \mathsf{put}\ st \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Put}\ st$$

As with the free monad, we are completely free to pick whatever interpretation of state we desire. If we want an interpretation that is compatible with the usual equations for state, then we can simply use the state-passing technique again.

$$\begin{aligned}
&\mathsf{runState} : (1 \to A) \to S \to A \times S \\
&\mathsf{runState}\ m\ st_0 \stackrel{\text{def}}{=} \mathbf{let}\ f \leftarrow \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad \{\mathbf{return}\ x \qquad\quad \mapsto \lambda st.\langle x; st\rangle; \\
&\qquad\qquad\ \langle\!\langle \mathsf{Get}\ \langle\rangle \twoheadrightarrow k \rangle\!\rangle \mapsto \lambda st.k\ st\ st; \\
&\qquad\qquad\ \langle\!\langle \mathsf{Put}\ st' \twoheadrightarrow k \rangle\!\rangle \mapsto \lambda st.k\ \langle\rangle\ st'\} \\
&\qquad\qquad \mathbf{in}\ f\ st_0
\end{aligned}$$

Note the similarity with the implementation of the interpreter for the free state monad. Save for the syntactic differences, the main difference between this implementation and

the free state monad interpreter is that here the continuation k implicitly reinstalls the handler, whereas in the free state monad interpreter we explicitly reinstalled the handler via a recursive application. By fixing $S = \mathsf{Int}$ and $A = \mathsf{Bool}$ we can use the above effect handler to run the delimited control variant of incrEven.

$$\mathsf{runState}\ \mathsf{incrEven}\ 4 \rightsquigarrow^{+} \langle \mathsf{true}; 5 \rangle : \mathsf{Bool} \times \mathsf{Int}$$

Effect handlers come into their own when multiple effects are combined. Throughout the dissertation we will see multiple examples of handlers in action (e.g. Chapter 2).

Effect tracking

A benefit of using monads for effectful programming is that we get effect tracking 'for free' (some might object to this statement and claim we paid for it by having to program in monadic style). Effect tracking is a useful tool for making programming with effects less prone to error in much the same way a static type system is useful for detecting a wide range of potential runtime errors at compile time.

Effect systems provide suitable typing discipline for statically tracking the observable effects of programs [207]. The notion of effect system was developed around the same time as monads rose to prominence, though, its development was independent of monads. Nevertheless, Wadler and Thiemann [270] have shown that effect systems and monads are formally related, providing effect systems with some formal validity. Subsequently, Kammar [141] has contributed to the formal understanding of effect systems through development of a general algebraic theory of effect systems. Lucassen and Gifford [183] developed the original effect system as a means for lightweight static analyses of functional programs with imperative features. For instance, Lucassen [182] made crucial use of an effect system to statically distinguish between safe and unsafe terms for parallel execution.

The principal idea of a Lucassen and Gifford style effect system is to annotate computation types with the collection of effects that their inhabitants are allowed to perform, e.g. the type $A \to B!E$ is inhabited by functions that accept a value of type A as input and ultimately return a value of type B. As an inhabitant computes the B value it is allowed to perform the effect operations mentioned by the effect signature E.

This typing discipline fits nicely with the effect handlers-style of programming. The **do** construct provides a mechanism for injecting an operation into the effect signature, whilst the **handle** construct provides a way to eliminate an effect operation from the signature [14, 119]. If we instantiate $A = 1$, $B = \mathsf{Bool}$, and $E = \Sigma$, then we obtain a

type-and-effect signature for the handler version of incrEven.

$$\mathsf{incrEven} : 1 \to \mathsf{Bool}!\{\mathsf{Get} : 1 \twoheadrightarrow \mathsf{Int}; \mathsf{Put} : \mathsf{Int} \twoheadrightarrow 1\}$$

Now, the signature of incrEven communicates precisely what it expects from the ambient context. It is clear that we must run this function under a handler that interprets at least Get and Put.

Some form of polymorphism is necessary to make an effect system extensible and useful in practice. Otherwise effect annotations end up pervading the entire program in a similar fashion as monads do. In Chapter 3 we will develop an extensible effect system based on row polymorphism.

1.3 Scope

Summarised in one sentence this dissertation is about practical programming language designs for programming with effect handlers, their foundational implementation techniques, and implications for the expressive power of their host language.

Numerous variations and extensions of effect handlers have been proposed since their inceptions. In this dissertation I restrict my attention to Plotkin and Pretnar's deep handlers, their shallow variation, and parameterised handlers which are a slight variation of deep handlers. In particular I work with free algebraic theories, which is to say my designs do not incorporate equational theories for effects. Furthermore, I frame my study in terms of simply-typed and polymorphic λ-calculi for which I give computational interpretations in terms of contextual operational semantics and realise using two foundational operational techniques: continuation passing style and abstract machine semantics. When it comes to expressiveness there are multiple possible dimensions to investigate and multiple different notions of expressivity available. I focus on two questions: 'are deep, shallow, and parameterised handlers interdefinable?' which I investigate via a syntactic notion of expressiveness due Felleisen [82]. And, 'does effect handlers admit any essential computational efficiency?' which I investigate using a semantic notion of expressiveness due to Longley and Normann [180].

1.3.1 Scope extrusion

The literature on effect handlers is rich, and my dissertation is but one of many on topics related to effect handlers. In this section I provide a few pointers to related work involving effect handlers that I will not otherwise discuss in this dissertation.

Readers interested in the mathematical foundations and original development of effect handlers should consult Pretnar's PhD dissertation [230].

Most programming language treatments of algebraic effects and their handlers sideline equational theories, despite equational theories being an important part of the original treatment of effect handlers. Lukšič's PhD dissertation brings equations back onto the pitch as Lukšič [184] develops a core calculus with a novel local notion of equational theories for algebraic effects.

To get a grasp of the reasoning principles for effect handlers, interested readers should consult McLaughlin's PhD dissertation, which contains a development of relational reasoning techniques for shallow multi-handlers [193]. McLaughlin's techniques draw inspiration from the logical relation reasoning techniques for deep handlers due to Biernacki et al. [26].

Ahman's PhD dissertation is relevant for readers interested in the integration of computational effects into dependent type theories [5]. Ahman develops an intensional Martin-Löf [189] style dependent type theory equipped with a novel computational dependent type, which makes it possible to treat type-dependency in the sequential composition of effectful computations uniformly.

Lexical effect handlers are a variation on Plotkin and Pretnar's deep handlers, which provide a form of lexical scoping for effect operations, thus statically binding them to their handlers. Geron's PhD dissertation develops the mathematical theory of scoped effect operations, whilst Biernacki et al. [28] study them in conjunction with ordinary handlers from a programming perspective.

Functional programmers were early adopters of effect handlers. They either added language-level support for handlers [15, 26, 38, 44, 71, 125, 165, 174] or embedded them in libraries [35, 40, 143, 147, 151, 154, 276]. Thus functional perspectives on effect handlers are plentiful in the literature. Some notable examples of perspectives on effect handlers outside functional programming are: Brachthäuser's PhD dissertation, which contains an object-oriented perspective on effect handlers in Java [34]; Saleh's PhD dissertation offers a logic programming perspective via an effect handlers extension to Prolog; and Leijen [167] has an imperative take on effect handlers in C.

1.4 Contributions

The key contributions of this dissertation are spread across the three main parts. The following listing summarises the contributions of each part.

Programming

- A practical design for a programming language equipped with a structural effect system and deep, shallow, and parameterised effect handlers.
- A case study in effect handler oriented programming demonstrating how to compose the essence of an UNIX-style operating system with user session management, task parallelism, and file I/O using standard effects and handlers.

Implementation

- A novel generalisation of the notion of continuation known as *generalised continuation*, which provides a succinct foundation for implementing deep, shallow, and parameterised handlers.
- A higher-order continuation passing style translation based on generalised continuations, which yields a universal implementation strategy for effect handlers.
- An abstract machine semantics based on generalised continuations, which characterises the low-level stack manipulations admitted by effect handlers at runtime.

Expressiveness

- A formal proof that deep, shallow, and parameterised handlers are equi-expressible in the sense of typed macro-expressiveness.
- A robust mathematical characterisation of the computational efficiency of effect handlers, which shows that effect handlers can improve the asymptotic runtime of certain classes of programs.

Another contribution worth noting is the continuation literature review in Appendix A, which provides a comprehensive operational characterisation of various notions of continuations and first-class control phenomena.

1.5 Structure of this dissertation

The following is a summary of the chapters belonging to each part of this dissertation.

Programming

- Chapter 2 showcases effect handler oriented programming in practice by implementing a small operating system dubbed Tiny UNIX based on Ritchie and Thompson's original UNIX. The implementation starts from a basic notion of file i/o, which evolves into a feature-rich operating system with full file i/o, multiple user environments, multi-tasking, and more, by composing ever more effect handlers.

- Chapter 3 introduces a polymorphic fine-grain call-by-value core calculus, λ_{b}, which makes key use of Rémy-style row polymorphism to implement polymorphic variants, structural records, and a structural effect system. The calculus distils the essence of the core of the Links programming language. The chapter also presents three extensions of λ_{b}, which are λ_{h} that adds deep handlers, $\lambda_{\mathsf{h}^\dagger}$ that adds shallow handlers, and $\lambda_{\mathsf{h}^\ddagger}$ that adds parameterised handlers.

Implementation

- Chapter 4 develops a higher-order continuation passing style translation for effect handlers through a series of step-wise refinements of an initial standard continuation passing style translation for λ_{b}. Each refinement slightly modifies the notion of continuation employed by the translation. The development ultimately leads to the key invention of generalised continuation, which is used to give a continuation passing style semantics to deep, shallow, and parameterised handlers.

- Chapter 5 demonstrates an application of generalised continuations to abstract machine as we plug generalised continuations into Felleisen and Friedman's CEK machine to obtain an adequate abstract runtime with simultaneous support for deep, shallow, and parameterised handlers.

Expressiveness

- Chapter 6 shows that deep, shallow, and parameterised notions of handlers can simulate one another up to specific notions of administrative reduction.

- Chapter 7 studies the fundamental efficiency of effect handlers. In this chapter, we show that effect handlers enable an asymptotic improvement in runtime complexity for a certain class of functions. Specifically, we consider the *generic*

count problem using a pure PCF-like base language $\lambda_{\mathsf{b}}^{\rightarrow}$ (a simply typed variation of λ_{b}) and its extension with effect handlers $\lambda_{\mathsf{h}}^{\rightarrow}$. We show that $\lambda_{\mathsf{h}}^{\rightarrow}$ admits an asymptotically more efficient implementation of generic count than any $\lambda_{\mathsf{b}}^{\rightarrow}$ implementation.

Conclusions

- Chapter 8 concludes and discusses future work.

Appendices

- Appendix A presents a literature survey of continuations and first-class control. I classify continuations according to their operational behaviour and provide an overview of the various first-class sequential control operators that appear in the literature. The application spectrum of continuations is discussed as well as implementation strategies for first-class control.
- Appendix B presents a small proof for the claim made in Section 1.2.2, that the state equation "Get after get" is redundant.
- Appendix C contains the proof details for the proof of Theorem 7.11.
- Appendix D discusses the Berger count program, which is briefly mentioned in Section 7.4, in more detail.

Part I

Programming

Chapter 2

Composing UNIX with effect handlers

There are several analogies for understanding effect handlers as a programming abstraction, e.g. as interpreters for effects, folds over computation trees (as in Section 1.2), resumable exceptions. A particularly compelling programmatic analogy is *effect handlers as composable operating systems*. Effect handlers and operating systems share operational characteristics: an operating system interprets a set of system commands performed via system calls, in a similar way to how an effect handler interprets a set of abstract operations performed via operation invocations (this analogy was suggested to me by James McKinna; personal communication, 2017). The compelling aspect of this analogy is that we can understand a monolithic and complex operating system like UNIX [241] as a collection of effect handlers, or alternatively, a collection of tiny operating systems, that when composed yield a semantics for UNIX.

In this section we will take this reading of effect handlers literally, and demonstrate how we can harness the power of (deep) effect handlers to implement a UNIX-style operating system with multiple user sessions, time-sharing, and file i/o. We dub the system Tiny UNIX. It is a case study that demonstrates the versatility of effect handlers, and shows how standard computational effects such as *exceptions*, *dynamic binding*, *nondeterminism*, and *state* make up the essence of an operating system. These effects are standard in the sense that they appear frequently in 101 tutorials on effects.

For the sake of clarity, we will occasionally make some blatant simplifications, nevertheless the resulting implementation will capture the essence of a UNIX-like operating system. The implementation will be composed of several small modular effect handlers, that each handles a particular set of system commands. In this respect, we will truly realise Tiny UNIX in the spirit of the UNIX philosophy [235, Section 1.6].

The source calculus underpinning the language used to implement Tiny UNIX will

be introduced informally on-the-fly. The formal syntax and semantics will be introduced in Chapter 3

Terminology In the remainder of this dissertation, I will make a slight change of terminology to disambiguate programmatic continuations, i.e. continuations exposed to the programmer, from continuations in continuation passing style (Chapter 4) and continuations in abstract machines (Chapter 5). I will refer to programmatic continuations as 'resumptions', and reserve the term 'continuation' for continuations concerning implementation details.

Chapter outline Sections 2.1–2.5 use *deep handlers* to model file i/o, user environments, process termination, process duplication, and process interruption.

Section 2.1 implements a basic mechanism which facilities writing to a global file. This mechanism is used as a stepping stone for building a more feature rich model.

Section 2.2 This section demonstrates a use of effect handlers as exception handlers, as we use *exceptions* to implement process termination.

Section 2.3 exemplifies user-specific environments as an instance of *dynamic binding*, as we use the environment, or reader, effect to implement user environments. The section also contains an instance of a *tail-resumptive* handler as well as demonstrates an application of dynamic overloading of interpretation of residual effectful operations in resumptions.

Section 2.4 models UNIX's process duplicating primitive 'fork' by making use of the *nondeterminism* effect. Furthermore, in this section we also implement a simple time-sharing facility that is capable of interleaving user processes.

Section 2.5 replaces the basic i/o model of Section 2.1 by a full-fledged i/o model supporting file creation, opening, reading, writing, and linking. This model is realised by making use of the *state* effect.

Section 2.6 demonstrates an application of *shallow handlers* to implement parts of the UNIX programming environment by simulating composable UNIX-style *pipes* for data stream processing.

Section 2.7 implements a variation of the time-sharing system from Section 2.4 with process synchronisation by taking advantage of the ability to internalise computation state with *parameterised handlers*.

Relation to prior work At the time of examination the work in this chapter was novel in the sense that it had not been published elsewhere. Since the examination of this dissertation, but prior to submission of its final version, I presented parts of this work at the ML Family Workshop.

- Daniel Hillerström. Composing UNIX with Effect Handlers: A Case Study in Effect Handler Oriented Programming (extended abstract). ML Workshop, 2021

Excerpts of Sections 2.1–2.4 have appeared in (mostly verbatim) in the report of Dagstuhl Seminar 21292. Those excerpts are to be considered a copy of the work described in this chapter rather than the other way around.

- Danel Ahman, Amal Ahmed, Sam Lindley, and Andreas Rossberg. Scalable Handling of Effects (Dagstuhl Seminar 21292). *Dagstuhl Reports*, 11(6):54–81, 2021

2.1 Basic i/o

The file system is a cornerstone of UNIX as the notion of *file* in UNIX provides a unified abstraction for storing text, interprocess communication, and access to devices such as terminals, printers, network, etc. Initially, we shall take a rather basic view of the file system. In fact, our initial system will only contain a single file, and moreover, the system will only support writing operations. This system hardly qualifies as a UNIX file system. Nevertheless, it serves a crucial role for development of Tiny UNIX, because it provides the only means for us to be able to observe the effects of processes. We defer development of a more advanced file system to Section 2.5.

Much like UNIX we shall model a file as a list of characters, that is $\mathsf{File} \stackrel{\text{def}}{=} \mathsf{List\ Char}$. For convenience we will use the same model for strings, $\mathsf{String} \stackrel{\text{def}}{=} \mathsf{List\ Char}$, such that we can use string literal notation to denote the `"contents of a file"`. The signature of the basic file system will consist of a single operation Write for writing a list of characters to the file.

$$\mathsf{BIO} \stackrel{\text{def}}{=} \{\mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String} \rangle \twoheadrightarrow 1\}$$

The operation is parameterised by a FileDescr and a character sequence. We will leave the FileDescr type abstract until Section 2.5, however, we shall assume the existence of a term stdout : FileDescr such that we can perform invocations of Write. Let us define a suitable handler for this operation.

$$
\begin{aligned}
&\mathsf{basicIO} : (1 \to \alpha!\mathsf{BIO}) \to \langle \alpha; \mathsf{File} \rangle \\
&\mathsf{basicIO}\ m \stackrel{\text{def}}{=} \mathbf{handle}\ m\,\langle\rangle\ \mathbf{with} \\
&\qquad \mathbf{return}\ \mathit{res} \qquad\qquad\qquad\qquad \mapsto \langle \mathit{res}; [] \rangle \\
&\qquad \langle\!\langle \mathsf{Write}\ \langle \_; \mathit{cs} \rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathbf{let}\ \langle \mathit{res}; \mathit{file} \rangle = \mathit{resume}\,\langle\rangle\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \langle \mathit{res}; \mathit{cs} +\!\!+ \mathit{file} \rangle
\end{aligned}
$$

The type signature $(1 \to \alpha!\mathsf{BIO}) \to \langle \alpha; \mathsf{File} \rangle$ classifies a second-order function that takes as input a computation that produces some value α, and in doing so may perform the BIO effect. As we program in a call-by-value setting the input computation is represented as a suspended computation, or thunk. Thus the type $1 \to \alpha!\mathsf{BIO}$ classifies a thunk, that when forced may perform the BIO effect, i.e. the Write operation, to produce a value of type α. The second-order function ultimately returns a pair consisting of the return value α and the final state of the file. The return type of the second-order function is 'pure' in the sense that the function does not perform any effectful operations itself to produce its result.

In the implementation the input computation is bound by the name *m*, which is applied to a $\langle\rangle$ (pronounced 'unit'; as we will see in Chapter 3 it happens to be the empty record) under a **handle** $\cdots$ **with** $\cdots$ construct which is the term syntax for effect handler application. Occurrences of the Write operation inside *m* are handled with respect to the given handler, whose definition consists of two cases: a **return**-case and a Write-case. The **return**-case runs when $m\,\langle\rangle$ reduces to a value. The resulting value gets bound to the name *res*. The body of the **return**-case pairs the result *res* with the empty file [] which models the scenario where the computation *m* performed no Write-operations, e.g. $\mathsf{basicIO}\,(\lambda\langle\rangle.\langle\rangle) \rightsquigarrow^{+} \langle\langle\rangle; \text{""}\rangle$. The Write-case runs when Write is invoked inside of *m*. The payload of the operation along with its resumption gets bound on the left hand side of $\mapsto$. We use deep pattern matching to ignore the first element of the payload and to bind the second element of the payload to *cs*. The resumption gets bound to the name *resume*. It is worth noting that at this point, the type of *resume* is (*morally*) $1 \to \langle \alpha; \mathsf{File} \rangle$, where the domain type matches the codomain type of the operation Write and the codomain type matches the expected type of the current context. (The actual type is $1 \to \langle \alpha; \mathsf{File} \rangle ! \{\mathsf{Write} : \theta; \varepsilon\}$, where θ is a presence variable denoting

that Write is polymorphic in whether it is present, i.e. the ambient context which *resume* gets invoked in is allowed to perform another invocation of Write, and ε is an effect variable, which can be instantiated with additional operation labels to allow *resume* to be used in a greater context. In many instances we can ignore presence polymorphim and effect polymorphism as described in Section 3.2.5, hence we omit these annotation whenever possible.) This latter fact is due to the deep semantics of **handle**-construct, which means that an invocation of *resume* implicitly installs another **handle**-construct with the same handler around the residual evaluation context embodied in *resume*. The body of the Write-case extends the file by first invoking the resumption, which returns a pair containing the result of m and the file state. This pair is deconstructed using the **let**-pair deconstruction construct, which projects and binds the result component to *res* and the file state component to *file*. The file gets extended with the character sequence cs before it is returned along with the original result of m. Intuitively, we may think of this implementation of Write as a peculiar instance of buffered writing, where we temporarily store the contents of each Write operation on call stack and commit them to the file as we unwind the stack after the computation m finishes.

Let us define an auxiliary function that writes a string to the stdout file.

$$\begin{aligned}&\mathsf{echo} : \mathsf{String} \to 1\,!\,\mathsf{BIO}\\&\mathsf{echo}\ cs \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Write}\,\langle \mathsf{stdout}; cs\rangle\end{aligned}$$

The **do**-construct is the invocation construct, or introduction form, for effectful operations. The function echo is a simple wrapper around an invocation of Write. We can now write some contents to the file and observe the effects.

$$\begin{aligned}&\mathsf{basicIO}\,(\lambda\langle\rangle.\mathsf{echo}\ \texttt{"Hello"}; \mathsf{echo}\ \texttt{"World"})\\\rightsquigarrow^{+}\ &\langle\langle\rangle; \texttt{"HelloWorld"}\rangle : \langle 1; \mathsf{File}\rangle\end{aligned}$$

2.2 Exceptions: process termination

A process may terminate successfully by running to completion, or it may terminate with success or failure in the middle of some computation by performing an *exit* system call. The exit system call is typically parameterised by an integer value intended to indicate whether the exit was due to success or failure. By convention, UNIX interprets the integer zero as success and any nonzero integer as failure, where the specific value is supposed to correspond to some known error code.

We can model the exit system call by way of a single operation Exit.

$$\mathsf{Status} \stackrel{\text{def}}{=} \{\mathsf{Exit} : \mathsf{Int} \twoheadrightarrow 0\}$$

The operation is parameterised by an integer value, however, an invocation of Exit can never return, because the type 0 is uninhabited. Thus Exit acts like an exception. It is convenient to abstract invocations of Exit to make it possible to invoke the operation in any context.

$$\begin{aligned}&\mathsf{exit} : \mathsf{Int} \to \alpha!\mathsf{Status}\\ &\mathsf{exit}\ n \stackrel{\text{def}}{=} \mathbf{absurd}\ (\mathbf{do}\ \mathsf{Exit}\ n)\end{aligned}$$

The **absurd** computation term is used to coerce the return type 0 of Fail into α. This coercion is safe, because 0 is an uninhabited type. An interpretation of Exit amounts to implementing an exception handler.

$$\begin{aligned}&\mathsf{status} : (1 \to \alpha!\mathsf{Status}) \to \mathsf{Int}\\ &\mathsf{status}\ m \stackrel{\text{def}}{=}\ \mathbf{handle}\ m\,\langle\rangle\ \mathbf{with}\\ &\qquad\qquad\mathbf{return}\ \_ \mapsto 0\\ &\qquad\qquad\langle\!\langle \mathsf{Exit}\ n \rangle\!\rangle\ \mapsto n\end{aligned}$$

Following the UNIX convention, the **return**-case interprets a successful completion of m as the integer 0. The operation case returns whatever payload the Exit operation was carrying. As a consequence, outside of status, an invocation of Exit 0 in m is indistinguishable from m returning normally, e.g. $\mathsf{status}\,(\lambda\langle\rangle.\mathsf{exit}\ 0) = \mathsf{status}\,(\lambda\langle\rangle.\langle\rangle)$. Technically, the Exit-case provides access to the resumption of Exit in m, however, we do not write this resumption here, because it is useless as its type is $0 \to \mathsf{Int}$. It expects as argument an element of the empty type, which is of course impossible to provide, hence we can never invoke it. In general, an operation clause may drop the provided resumption even if the resumption is usable.

To illustrate status and exit in action consider the following example, where the computation gets terminated mid-way.

$$\begin{aligned}&\mathsf{basicIO}\,(\lambda\langle\rangle.\mathsf{status}\,(\lambda\langle\rangle.\mathsf{echo}\ \texttt{"dead"}; \mathsf{exit}\ 1; \mathsf{echo}\ \texttt{"code"}))\\ \rightsquigarrow^{+}\ &\langle 1; \texttt{"dead"}\rangle : \langle \mathsf{Int}; \mathsf{File}\rangle\end{aligned}$$

The (delimited) continuation of exit 1 is effectively dead code. Here, we have a choice as to how we compose the handlers. Swapping the order of handlers would cause the whole computation to return just $1 : \mathsf{Int}$, because the status handler discards the return value of its computation. Thus with the alternative layering of handlers the system would throw away the file state after the computation finishes. However, in this particular instance the semantics the (local) behaviour of the operations Write and Exit would be unaffected if the handlers were swapped. In general the behaviour of operations may be affected

by the order of handlers. The canonical example of this phenomenon is the composition of nondeterminism and state, which we will discuss in Section 2.5.

2.3 Dynamic binding: user-specific environments

When a process is run in UNIX, the operating system makes available to the process a collection of name-value pairs called the *environment*. The name of a name-value pair is known as an *environment variable*. During execution the process may perform a system call to ask the operating system for the value of some environment variable. The value of environment variables may change throughout process execution, moreover, the value of some environment variables may vary according to which user asks the environment. For example, an environment may contain the environment variable `USER` that is bound to the name of the enquiring user.

An environment variable can be viewed as an instance of dynamic binding. The idea of dynamic binding as a binding form in programming dates back as far as the original implementation of Lisp [192], and still remains an integral feature in successors such as Emacs Lisp [171]. It is well-known that dynamic binding can be encoded as a computational effect by using delimited control [153]. Unsurprisingly, we will use this insight to simulate user-specific environments using effect handlers.

For simplicity we fix the users of the operating system to be root, Alice, and Bob.

$$\mathsf{User} \stackrel{\text{def}}{=} [\mathsf{Alice}; \mathsf{Bob}; \mathsf{Root}]$$

Our environment will only support a single environment variable intended to store the name of the current user. The value of this variable can be accessed via an operation $\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}$. Using this operation we can readily implement the *whoami* utility from the GNU coreutils [186, Section 20.3], which returns the name of the current user.

$$\begin{aligned} &\mathsf{whoami} : 1 \to \mathsf{String}!\{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}\} \\ &\mathsf{whoami}\ \langle\rangle \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Ask}\ \langle\rangle \end{aligned}$$

The following handler implements the environment.

$$\begin{aligned} &\mathsf{env} : \langle \mathsf{User}; 1 \to \alpha!\{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}\}\rangle \to \alpha \\ &\mathsf{env}\ \langle user; m\rangle \stackrel{\text{def}}{=} \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\ &\qquad \mathbf{return}\ res \qquad\qquad \mapsto res \\ &\qquad \langle\!\langle \mathsf{Ask}\ \langle\rangle \twoheadrightarrow resume\rangle\!\rangle \mapsto \mathbf{case}\ user\ \{\mathsf{Alice} \mapsto resume\ \texttt{"alice"} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Bob} \mapsto resume\ \texttt{"bob"} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Root} \mapsto resume\ \texttt{"root"}\} \end{aligned}$$

The handler takes as input the current *user* and a computation that may perform the Ask operation. When an invocation of Ask occurs the handler pattern matches on the *user* parameter and resumes with a string representation of the user. This handler is an instance of a so-called *tail-resumptive* handler [167, 277] as each resumption invocation appears in tail position within in the operation clause. With this implementation we can interpret an application of whoami.

$$\mathsf{env}\ \langle \mathsf{Root}; \mathsf{whoami} \rangle \rightsquigarrow^{+} \texttt{"root"} : \mathsf{String}$$

It is not difficult to extend this basic environment model to support an arbitrary number of variables. This can be done by parameterising the Ask operation by some name representation (e.g. a string), which the environment handler can use to index into a list of string values. In case the name is not bound in the environment, the handler can embrace the laissez-faire attitude of UNIX and resume with the empty string.

User session management It is somewhat pointless to have multiple user-specific environments, if the system does not support some mechanism for user session handling, such as signing in as a different user. In UNIX the command *substitute user* (su) enables the invoker to impersonate another user account, provided the invoker has sufficient privileges. We will implement su as an operation $\mathsf{Su} : \mathsf{User} \twoheadrightarrow 1$ which is parameterised by the user to be impersonated. To model the security aspects of su, we will use the weakest possible security model: unconditional trust. Put differently, we will not bother with security at all to keep things relatively simple. Consequently, anyone can impersonate anyone else.

The session signature consists of two operations, Ask, which we used above, and Su, for switching user.

$$\mathsf{Session} \stackrel{\text{def}}{=} \{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}; \mathsf{Su} : \mathsf{User} \twoheadrightarrow 1\}$$

As usual, we define a small wrapper around invocations of Su.

$$\begin{aligned} &\mathsf{su} : \mathsf{User} \to 1 ! \{\mathsf{Su} : \mathsf{User} \twoheadrightarrow 1\} \\ &\mathsf{su}\ \mathit{user} \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Su}\ \mathit{user} \end{aligned}$$

The intended operational behaviour of an invocation of Su *user* is to load the environment belonging to *user* and continue the continuation under this environment. We can achieve this behaviour by defining a handler for Su that invokes the provided

resumption under a fresh instance of the env handler.

$$\mathsf{sessionmgr} : \langle \mathsf{User}; 1 \to \alpha!\mathsf{Session} \rangle \to \alpha$$
$$\mathsf{sessionmgr}\,\langle \mathit{user}; m \rangle \stackrel{\text{def}}{=} \mathsf{env}\langle \mathit{user}; (\lambda\langle\rangle.\mathbf{handle}\ m\,\langle\rangle\ \mathbf{with}$$
$$\mathbf{return}\ \mathit{res} \mapsto \mathit{res}$$
$$\langle\!\langle \mathsf{Su}\ \mathit{user}' \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathsf{env}\langle \mathit{user}'; \mathit{resume} \rangle)\rangle$$

The function sessionmgr manages a user session. It takes two arguments: the initial user (*user*) and the computation (*m*) to run in the current session. An initial instance of env is installed with *user* as argument. The computation argument is a handler for Su enclosing the computation *m*. The Su-case installs a new instance of env, which is the environment belonging to user', and runs the resumption *resume* under this instance. The new instance of env shadows the initial instance, and therefore it will intercept and handle any subsequent invocations of Ask arising from running the resumption. The next invocation of Su will install another environment instance, which will shadow both the previously installed instance and the initial instance. This ability to dynamically overload residual operations is a key difference between Plotkin and Pretnar's effect handlers (as this thesis is about) and *lexical* effect handlers, as the latter bind operations to the first suitable handler instance [37, 277].

As an illustrative example of the dynamic overloading in action, let us plug together the all components of our system we have defined thus far.

$$\begin{array}{l}\mathsf{basicIO}\,(\lambda\langle\rangle.\\ \quad \mathsf{sessionmgr}\,\langle \mathsf{Root}; \lambda\langle\rangle.\\ \quad\quad \mathsf{status}\,(\lambda\langle\rangle.\mathsf{su\ Alice};\ \mathsf{echo}\,(\mathsf{whoami}\,\langle\rangle);\ \mathsf{echo}\ \texttt{" "};\\ \quad\quad\quad\quad \mathsf{su\ Bob};\ \ \mathsf{echo}\,(\mathsf{whoami}\,\langle\rangle);\ \mathsf{echo}\ \texttt{" "};\\ \quad\quad\quad\quad \mathsf{su\ Root};\ \mathsf{echo}\,(\mathsf{whoami}\,\langle\rangle)))\rangle)\\ \rightsquigarrow^{+} \langle 0; \texttt{"alice bob root"}\rangle : \langle \mathsf{Int}; \mathsf{File}\rangle\end{array}$$

The session manager (sessionmgr) is installed in between the basic IO handler (basicIO) and the process status handler (status). The initial user is Root, and thus the initial environment is the environment that belongs to the root user. Main computation signs in as Alice and writes the result of the system call whoami to the global file, and then repeats these steps for Bob and Root. Ultimately, the computation terminates successfully (as indicated by 0 in the first component of the result) with global file containing the three user names.

The above example demonstrates that we now have the basic building blocks to build a multi-user system.

2.4 Nondeterminism: time sharing

Time sharing is a mechanism that enables multiple processes to run concurrently, and hence, multiple users to work concurrently. Thus far in our system there is exactly one process. In UNIX there exists only a single process whilst the system is bootstrapping itself into operation. After bootstrapping is complete the system duplicates the initial process to start running user managed processes, which may duplicate themselves to create further processes. The process duplication primitive in UNIX is called *fork* [241]. The fork-invoking process is typically referred to as the parent process, whilst its clone is referred to as the child process. Following an invocation of fork, the parent process is provided with a nonzero identifier for the child process and the child process is provided with the zero identifier. This enables processes to determine their respective role in the parent-child relationship, e.g.

$$\begin{aligned}&\textbf{let } i \leftarrow \mathit{fork}\ \langle\rangle \textbf{ in}\\ &\textbf{if } i = 0 \textbf{ then } \ \mathit{child's\ code}\\ &\textbf{else } \ \mathit{parent's\ code}\end{aligned}$$

In our system, we can model fork as an effectful operation, that returns a boolean to indicate the process role; by convention we will interpret the return value true to mean that the process assumes the role of parent.

$$\begin{aligned}&\mathsf{fork} : 1 \to \mathsf{Bool}!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}\\ &\mathsf{fork}\ \langle\rangle \stackrel{\text{def}}{=} \textbf{do}\ \mathsf{Fork}\ \langle\rangle\end{aligned}$$

In UNIX the parent process *continues* execution after the fork point, and the child process *begins* its execution after the fork point. Thus, operationally, we may understand fork as returning twice to its invocation site. We can implement this behaviour by invoking the resumption arising from an invocation of Fork twice: first with true to continue the parent process, and subsequently with false to start the child process (or the other way around if we feel inclined). The following handler implements this behaviour.

$$\begin{aligned}&\mathsf{nondet} : (1 \to \alpha!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}) \to \mathsf{List}\ \alpha\\ &\mathsf{nondet}\ m \stackrel{\text{def}}{=} \textbf{handle}\ m\ \langle\rangle\ \textbf{with}\\ &\qquad\qquad \textbf{return}\ \mathit{res} \qquad\qquad \mapsto [\mathit{res}]\\ &\qquad\qquad \langle\!\langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathit{resume}\ \mathsf{true} +\!\!+ \mathit{resume}\ \mathsf{false}\end{aligned}$$

The **return**-case returns a singleton list containing a result of running m. The Fork-case invokes the provided resumption *resume* twice. Each invocation of *resume* effectively

copies m and runs each copy to completion. Each copy returns through the **return**-case, hence each invocation of *resume* returns a list of the possible results obtained by interpreting Fork first as true and subsequently as false. The results are joined by list concatenation ($+\!\!+$). Thus the handler returns a list of all the possible results of m. This handler is an instance of a *multi-shot* handler, because it contains at least one operation clause which invokes the provided resumption multiple times.

Let us consider nondet together with the previously defined handlers. But first, let us define two computations.

```
ritchie, hamlet : 1 → 1!{Write : ⟨FileDescr; String⟩ ↠ 1}
ritchie ⟨⟩ ≝ echo "UNIX is basically ";
             echo "a simple operating system, ";
             echo "but ";
             echo "you have to be a genius
                   to understand the simplicity.\n"
hamlet ⟨⟩ ≝ echo "To be, or not to be, ";
            echo "that is the question:\n";
            echo "Whether 'tis nobler in the mind to suffer\n"
```

The computation ritchie writes a quote by Dennis Ritchie to the file, whilst the computation hamlet writes a few lines of William Shakespeare's *The Tragedy of Hamlet, Prince of Denmark*, Act III, Scene I [246] to the file. Using nondet and fork together with the previously defined infrastructure, we can fork the initial process such that both of the above computations are run concurrently.

```
basicIO (λ⟨⟩.
    nondet (λ⟨⟩.
        sessionmgr ⟨Root; λ⟨⟩.
            status (λ⟨⟩.if fork ⟨⟩ then su Alice; ritchie ⟨⟩
                                   else su Bob; hamlet ⟨⟩))))
⇝⁺ ⟨[0,0]; "UNIX is basically a simple operating system, but
            you have to be a genius to understand the simplicity.\n
            To be, or not to be, that is the question:\n
            Whether 'tis nobler in the mind to suffer\n"⟩ : ⟨List Int; File⟩
```

The computation running under the status handler immediately performs an invocation of fork, causing nondet to explore both the **then**-branch and the **else**-branch. In the

former, Alice signs in and quotes Ritchie, whilst in the latter Bob signs in and quotes a Hamlet. Looking at the output there is supposedly no interleaving of computation, since the individual writes have not been interleaved. From the stack of handlers, we *know* that there has been no interleaving of computation, because no handler in the stack handles interleaving. Thus, our system only supports time sharing in the extreme sense: we know from the nondet handler that every effect of the parent process will be performed and handled before the child process gets to run. In order to be able to share time properly amongst processes, we must be able to interrupt them.

Interleaving computation We need an operation for interruptions and corresponding handler to handle interrupts in order for the system to support interleaving of processes.

$$\begin{aligned}&\mathsf{interrupt} : 1 \to 1!\{\mathsf{Interrupt} : 1 \twoheadrightarrow 1\}\\&\mathsf{interrupt}\ \langle\rangle \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Interrupt}\ \langle\rangle\end{aligned}$$

The intended behaviour of an invocation of Interrupt is to suspend the invoking computation in order to yield time for another computation to run. We can achieve this behaviour by reifying the process state. For the purpose of interleaving processes via interruptions it suffices to view a process as being in either of two states: 1) it is done, that is it has run to completion, or 2) it is paused, meaning it has yielded to provide room for another process to run. We can model the state using a recursive variant type parameterised by some return value α, a set of effects ε that the process may perform, and a present variable to denote the presence of Interrupt.

$$\begin{aligned}\mathsf{Pstate}\ \alpha\ \varepsilon\ \theta \stackrel{\text{def}}{=} [&\mathsf{Done} : \alpha;\\&\mathsf{Paused} : 1 \to \mathsf{Pstate}\ \alpha\ \varepsilon\ \theta!\{\mathsf{Interrupt} : \theta; \varepsilon\}]\end{aligned}$$

This data type definition is an instance of the *resumption monad* [211]. The Done-tag simply carries the return value of type α. The Paused-tag carries a suspended computation, which returns another instance of Pstate, and may or may not perform any further invocations of Interrupt. Payload type of Paused is precisely the type of a resumption originating from a handler that handles only the operation Interrupt such as the following handler.

$$\begin{aligned}&\mathsf{reifyProcess} : (1 \to \alpha!\{\mathsf{Interrupt} : 1 \twoheadrightarrow 1; \varepsilon\}) \to \mathsf{Pstate}\ \alpha\ \varepsilon\ \theta\\&\mathsf{reifyProcess}\ m \stackrel{\text{def}}{=} \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}\\&\qquad\qquad \mathbf{return}\ \mathit{res} \mapsto \mathsf{Done}\ \mathit{res}\\&\qquad\qquad \langle\!\langle \mathsf{Interrupt}\ \langle\rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathsf{Paused}\ \mathit{resume}\end{aligned}$$

This handler tags and returns values with Done. It also tags and returns the resumption provided by the Interrupt-case with Paused. It is worth noting that the actual type of *resume* is $1 \to \mathsf{Pstate}\ \alpha\ \varepsilon\ \theta!\{\mathsf{Interrupt} : \theta; \varepsilon\}$, which shows us that the variables ε and θ threaded through Pstate come from the ambient context. This particular implementation amounts to a handler-based variation of Harrison's non-reactive resumption monad [112]. If we compose this handler with the nondeterminism handler, then we obtain a term with the following type.

$$\mathsf{nondet}\,(\lambda\langle\rangle.\mathsf{reifyProcess}\ m) : \mathsf{List}\ (\mathsf{Pstate}\ \alpha\ \{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}; \varepsilon\})$$

for some $m : 1 \to \{\mathsf{Proc}; \varepsilon\}$ where $\mathsf{Proc} \stackrel{\text{def}}{=} \{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}; \mathsf{Interrupt} : 1 \twoheadrightarrow 1\}$. The composition yields a list of process states, some of which may be in suspended state. In particular, the suspended computations may have unhandled instances of Fork as signified by it being present in the effect row. The reason for this is that in the above composition when reifyProcess produces a Paused-tagged resumption, it immediately returns through the **return**-case of nondet, meaning that the resumption escapes the nondet. Recall that a resumption is a delimited continuation that captures the extent from the operation invocation up to and including the nearest enclosing suitable handler. In this particular instance, it means that the nondet handler is part of the extent. We ultimately want to return just a list of αs to ensure every process has run to completion. To achieve this, we need a function that keeps track of the state of every process, and in particular it must run each Paused-tagged computation under the nondet handler to produce another list of process state, which must be handled recursively.

$$\begin{array}{l}
\mathsf{schedule} : \mathsf{List}\ (\mathsf{Pstate}\ \alpha\ \{\mathsf{Fork} : \mathsf{Bool}; \varepsilon\}\ \theta) \to \mathsf{List}\ \alpha!\varepsilon \\
\mathsf{schedule}\ ps \stackrel{\text{def}}{=} \mathbf{let}\ run \leftarrow \mathbf{rec}\ sched\,\langle ps; done\rangle. \\
\qquad\qquad \mathbf{case}\ ps\ \{\qquad\qquad\qquad [] \mapsto done \\
\qquad\qquad\qquad\qquad (\mathsf{Done}\ res) :: ps' \mapsto sched\,\langle ps'; res :: done\rangle \\
\qquad\qquad\qquad\qquad (\mathsf{Paused}\ m) :: ps' \mapsto sched\,\langle ps' +\!\!+ (\mathsf{nondet}\ m); done\rangle\} \\
\qquad\quad \mathbf{in}\ run\,\langle ps; []\rangle
\end{array}$$

The function schedule implements a process scheduler. It takes as input a list of process states, where Paused-tagged computations may perform the Fork operation. Locally it defines a recursive function *sched* which carries a list of active processes *ps* and the results of completed processes *done*. The function inspects the process list *ps* to test whether it is empty or nonempty. If it is empty it returns the list of results *done*. Otherwise, if the head is Done-tagged value, then the function is recursively invoked

with tail of processes ps' and the list *done* augmented with the value *res*. If the head is a Paused-tagged computation m, then *sched* is recursively invoked with the process list ps' concatenated with the result of running m under the nondet handler.

Using the above machinery, we can define a function which adds time-sharing capabilities to the system.

$$\begin{aligned}&\mathsf{timeshare} : (1 \to \alpha!\mathsf{Proc}) \to \mathsf{List}\ \alpha\\&\mathsf{timeshare}\ m \stackrel{\text{def}}{=} \mathsf{schedule}\,[\mathsf{Paused}\,(\lambda\langle\rangle.\mathsf{reifyProcess}\ m)]\end{aligned}$$

The function timeshare handles the invocations of Fork and Interrupt in some computation m by starting it in suspended state under the reifyProcess handler. The schedule actually starts the computation, when it runs the computation under the nondet handler.

The question remains how to inject invocations of Interrupt such that computation gets interleaved.

Interruption via interception To implement process preemption operating systems typically to rely on the underlying hardware to asynchronously generate some kind of interruption signals. These signals can be caught by the operating system's process scheduler, which can then decide to which processes to suspend and continue. If our core calculi had an integrated notion of asynchrony and effects along the lines of Ahman and Pretnar's core calculus $\lambda_{æ}$ [6], then we could potentially treat interruption signals as asynchronous effectful operations, which can occur spontaneously and, as suggested by Dolan et al. [72] and realised by Poulson [229], be handled by a user-definable handler.

In the absence of asynchronous effects we have to inject synchronous interruptions ourselves. One extreme approach is to trust the user to perform invocations of Interrupt periodically. Another approach is based on the fact that every effect (except for divergence) occurs via some operation invocation, and every-so-often the user is likely to perform computational effect, thus the basic idea is to bundle Interrupt with invocations of other operations. For example, we can insert an instance of Interrupt in some of the wrapper functions for operation invocations that we have defined so conscientiously thus far. The problem with this approach is that it requires a change of type signatures. To exemplify this problem consider type of the echo function if we were to bundle an invocation of Interrupt along side Write.

$$\begin{aligned}&\mathsf{echo}' : \mathsf{String} \to 1!\{\mathsf{Interrupt} : 1 \twoheadrightarrow 1; \mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String}\rangle \twoheadrightarrow 1\}\\&\mathsf{echo}'\ cs \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Interrupt}\,\langle\rangle; \mathbf{do}\ \mathsf{Write}\,\langle \mathsf{stdout}; cs\rangle\end{aligned}$$

In addition to Write the effect row must now necessarily mention the Interrupt operation. As a consequence this approach is not backwards compatible, since the original definition of echo can be used in a context that prohibits occurrences of Interrupt. Clearly, this alternative definition cannot be applied in such a context.

There is backwards-compatible way to bundle the two operations together. We can implement a handler that *intercepts* invocations of Write and handles them by performing an interrupt and, crucially, reperforming the intercepted write operation.

$$
\begin{aligned}
&\mathsf{interruptWrite} : \quad (1 \to \alpha!\{\mathsf{Interrupt} : 1 \twoheadrightarrow 1; \mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String}\rangle \twoheadrightarrow 1\}) \\
&\qquad\qquad \to \alpha!\{\mathsf{Interrupt} : 1 \twoheadrightarrow 1; \mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String}\rangle \twoheadrightarrow 1\} \\
&\mathsf{interruptWrite}\ m \stackrel{\text{def}}{=} \ \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad \mathbf{return}\ \mathit{res} \qquad\qquad\qquad\qquad \mapsto \mathit{res} \\
&\qquad\qquad \langle\!\langle \mathsf{Write}\ \langle \mathit{fd}; \mathit{cs}\rangle \twoheadrightarrow \mathit{resume}\rangle\!\rangle \mapsto \mathsf{interrupt}\,\langle\rangle; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{resume}\,(\mathbf{do}\ \mathsf{Write}\ \langle \mathit{fd}; \mathit{cs}\rangle)
\end{aligned}
$$

This handler is not 'self-contained' as the other handlers we have defined previously. It gives in some sense a 'partial' interpretation of Write as it leaves open the semantics of Interrupt and Write, i.e. this handler must be run in a suitable context of other handlers.

Let us plug this handler into the previous example to see what happens.

```
basicIO (λ⟨⟩.
    timeshare (λ⟨⟩.
        interruptWrite (λ⟨⟩.
            sessionmgr ⟨Root; λ⟨⟩.
                status (λ⟨⟩.if fork ⟨⟩ then su Alice; ritchie ⟨⟩
                           else su Bob; hamlet ⟨⟩)))))
⇝+ ⟨[0,0]; "UNIX is basically To be, or not to be,\n
           a simple operating system, that is the question:\n
           but Whether 'tis nobler in the mind to suffer\n
           you have to be a genius to understand the simplicity.\n"⟩
  : ⟨List Int; File⟩
```

Evidently, each write operation has been interleaved, resulting in a mishmash poetry of Shakespeare and UNIX. I will leave it to the reader to be the judge of whether this new poetry belongs under the category of either classic arts vandalism or novel contemporary reinterpretations. As the saying goes: *art is in the eye of the beholder*.

2.5 State: file i/o

Thus far the system supports limited I/O, abnormal process termination, multiple user sessions, and multi-tasking via concurrent processes. At this stage we have most of core features in place. We still have to complete the I/O model. The current I/O model provides an incomplete file system consisting of a single write-only file. In this section we will implement a UNIX-like file system that supports file creation, opening, truncation, read and write operations, and file linking.

To implement a file system we will need to use state. State can readily be implemented with an effect handler [143]. It is a deliberate choice to leave state for last, because once you have state it is tempting to use it excessively — to the extent it becomes a cliche. As demonstrated in the previous sections, it is possible to achieve many things that have a stateful flavour without explicit state by harnessing the implicit state provided by the program stack.

In the following subsection, I will provide an interface for stateful operations and their implementation in terms of a handler. The stateful operations will be put to use in the subsequent subsection to implement a basic sequential file system.

Handling state

As we have already seen in Section 1.2, the interface for accessing and updating a state cell consists of two operations.

$$\mathsf{State}\ \beta \stackrel{\text{def}}{=} \{\mathsf{Get} : 1 \twoheadrightarrow \beta; \mathsf{Put} : \beta \twoheadrightarrow 1\}$$

The intended operational behaviour of Get operation is to read the value of type β of the state cell, whilst the Put operation is intended to replace the current value held by the state cell with another value of type β. As per usual business, the following functions abstract the invocation of the operations.

$$\mathsf{get} : 1 \to \beta!\{\mathsf{Get} : 1 \twoheadrightarrow \beta\} \qquad \mathsf{put} : 1 \to \beta!\{\mathsf{Put} : \beta \twoheadrightarrow 1\}$$
$$\mathsf{get}\ \langle\rangle \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Get}\ \langle\rangle \qquad \mathsf{put}\ st \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Put}\ st$$

The following handler interprets the operations.

$$
\begin{aligned}
&\mathsf{runState} : \langle \beta; 1 \to \alpha!\mathsf{State}\ \beta \rangle \to \langle \alpha; \beta \rangle \\
&\mathsf{runState}\ \langle st_0; m \rangle \stackrel{\text{def}}{=} \mathbf{let}\ run \leftarrow \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad\qquad\qquad \mathbf{return}\ res \mapsto \lambda st.\langle res; st \rangle \\
&\qquad\qquad\qquad\qquad \langle\!\langle \mathsf{Get}\ \langle\rangle \twoheadrightarrow resume \rangle\!\rangle \mapsto \lambda st.resume\ st\ st \\
&\qquad\qquad\qquad\qquad \langle\!\langle \mathsf{Put}\ st' \twoheadrightarrow resume \rangle\!\rangle \mapsto \lambda st.resume\ \langle\rangle\ st' \\
&\qquad\qquad \mathbf{in}\ run\ st_0
\end{aligned}
$$

The runState handler provides a generic way to interpret any stateful computation. It takes as its first parameter the initial value of the state cell. The second parameter is a potentially stateful computation. Ultimately, the handler returns the value of the input computation along with the current value of the state cell.

Each case returns a state-accepting function. The **return**-case returns a function that produces a pair consisting of return value of m and the final state st. The Get-case returns a function that applies the resumption *resume* to the current state st. Recall that return type of a resumption is the same as its handler's return type, so since the handler returns a function, it follows that $resume : \beta \to \beta \to \langle \alpha, \beta \rangle$. In other words, the invocation of *resume* produces another state-accepting function. This function arises from the next activation of the handler either by way of a subsequent operation invocation in m or the completion of m to invoke the **return**-case. Since Get does not modify the value of the state cell it passes st unmodified to the next handler activation. In the Put-case the resumption must also produce a state-accepting function of the same type, however, the type of the resumption is slightly different $resume : 1 \to \beta \to \langle \alpha, \beta \rangle$. The unit type is the expected return type of Put. The state-accepting function arising from $resume\ \langle\rangle$ is supplied with the new state value st'. This application effectively discards the current state value st.

The first operation invocation in m, or if it completes without invoking Get or Put, the handler returns a function that accepts the initial state. The function gets bound to *run* which is subsequently applied to the provided initial state st_0 which causes evaluation of the stateful fragment of m to continue.

Backtrackable state vs non-backtrackable state The meaning of stateful operations may depend on whether the ambient environment is nondeterministic. Post-composing nondeterminism with state gives rise to the *backtrackable state* phenomenon, where state modifications are local to each strand of nondeterminism, that is each strand maintains its own copy of the state [108]. The state is backtrackable, because returning back

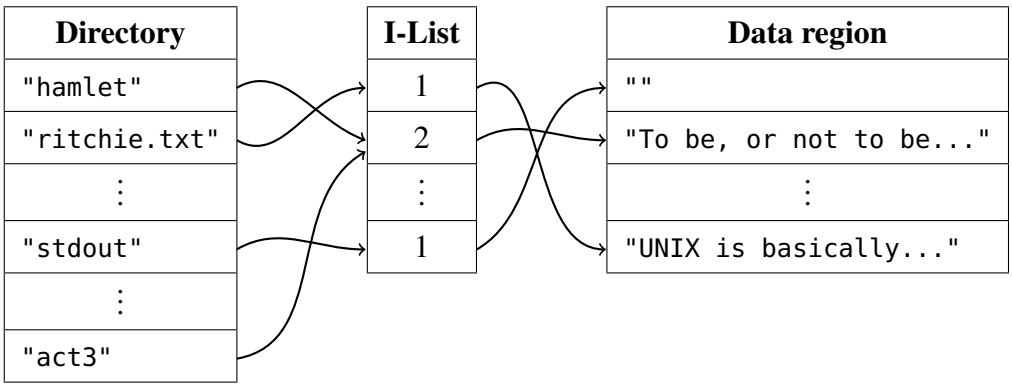


Figure 2.1: UNIX directory, i-list, and data region mappings.

---

to a previous branch point restores the state as it were prior to the branch. In contrast, post-composing state with nondeterminism results in the *non-backtrackable state* interpretation, where the state is shared across every strand of nondeterminism, meaning that backtracking to a previous branch point does not restore the original state at the time of the branch, but rather keeps the current state as is.

For modelling the file system we opt for the non-backtrackable state interpretation such that changes made to file system are visible to all processes. The backtrackable state interpretation could prove useful if we were to model a virtual file system per process such that each process would have its own unique standard out file.

The two state phenomena are inter-encodable. Pauwels et al. [213] give a systematic behaviour-preserving transformation for nondeterminism with backtrackable state into nondeterminism with non-backtrackable state and vice versa.

Basic serial file system

A file system provide an abstraction over storage media in a computer system by organising the storage space into a collection of files. This abstraction facilities typical file operations: allocation, deletion, reading, and writing. UNIX dogmatises the notion of file to the point where *everything is a file*. A typical UNIX-style file system differentiates between ordinary files, directory files, and special files [241]. An ordinary file is a sequence of characters. A directory file is a container for all kinds of files. A special file is an interface for interacting with an i/o device.

We will implement a *basic serial file system*, which we dub BSFS. It will be basic in the sense that it models the bare minimum to pass as a file system, that is we will implement support for the four basic operations: file allocation, file deletion, file reading,

and file writing. The read and write operations will be serial, meaning every file is read in order from its first character to its last character, and every file is written to by appending the new content. BSFS will only contain ordinary files, and as a result the file hierarchy will be entirely flat. Although, the system can readily be extended to be hierarchical, it comes at the expense of extra complexity, that blurs rather than illuminates the model.

Directory, i-list, and data region A storage medium is an array of bytes. An UNIX file system is implemented on top of this array by interpreting certain intervals of the array differently. These intervals provide the space for the essential administrative structures for file organisation.

1. The *directory* is a collection of human-readable names for files. In general, a file may have multiple names. Each name is stored along with a pointer into the i-list.

2. The *i-list* is a collection of i-nodes. Each i-node contains the meta data for a file along with a pointer into the data region.

3. The *data region* contains the actual file contents.

These structures make up the BSFS. Figure 2.1 depicts an example with the three structures and a mapping between them. The only file meta data tracked by BSFS is the number of names for a file. The three structures and their mappings can be implemented using association lists. Although, a better practical choice may be a functional map or functional array [210], association lists have the advantage of having a simple, straightforward implementation.

$$\mathsf{Directory} \stackrel{\text{def}}{=} \mathsf{List}\,\langle \mathsf{String}; \mathsf{Int}\rangle \qquad \mathsf{DataRegion} \stackrel{\text{def}}{=} \mathsf{List}\,\langle \mathsf{Int}; \mathsf{File}\rangle$$

$$\mathsf{INode} \stackrel{\text{def}}{=} \langle \mathit{lno} : \mathsf{Int}; \mathit{loc} : \mathsf{Int}\rangle \qquad \mathsf{IList} \stackrel{\text{def}}{=} \mathsf{List}\,\langle \mathsf{Int}; \mathsf{INode}\rangle$$

Mathematically, we may think the type Directory as denoting a partial function $\mathbb{C}^* \rightharpoonup \mathbb{Z}$, where $\mathbb{C}$ is a suitable alphabet. The function produces an index into the i-list. Similarly, the type IList denotes a partial function $\mathbb{Z} \rightharpoonup \mathbb{Z} \times \mathbb{Z}$, where the codomain is the denotation of INode. The first component of the pair is the number of names linked to the i-node, and as such $\mathbb{Z}$ is really an overapproximation as an i-node cannot have a negative number of names. The second component is an index into the data region. The denotation of the type DataRegion is another partial function $\mathbb{Z} \rightharpoonup \mathbb{C}^*$.

We define the type of the file system to be a record of the three association lists along with two counters for the next available index into the data region and i-list, respectively.

$$\mathsf{FileSystem} \stackrel{\text{def}}{=} \langle \mathit{dir} : \mathsf{Directory}; \mathit{ilist} : \mathsf{IList}; \mathit{dreg} : \mathsf{DataRegion}; \\ \mathit{dnext} : \mathsf{Int}; \mathit{inext} : \mathsf{Int} \rangle$$

We can then give an implementation of the initial state of the file system.

$$\mathsf{fs}_0 \stackrel{\text{def}}{=} \langle \mathit{dir} = [\langle \texttt{"stdout"}; 0 \rangle]; \mathit{ilist} = [\langle 0; \langle \mathit{lno} = 1; \mathit{loc} = 0 \rangle \rangle]; \mathit{dreg} = [\langle 0; \texttt{""} \rangle]; \\ \mathit{dnext} = 1; \mathit{inext} = 1 \rangle$$

Initially the file system contains a single, empty file with the name `stdout`. Next we will implement the basic operations on the file system separately.

We have made a gross simplification here, as a typical file system would provide some *file descriptor* abstraction for managing access open files. In BSFS we will operate directly on i-nodes, meaning we define $\mathsf{FileDescr} \stackrel{\text{def}}{=} \mathsf{Int}$, meaning the file open operation will return an i-node identifier. As consequence it does not matter whether a file is closed after use as file closing would be a no-op (closing a file does not change the state of its i-node). Therefore BSFS will not provide a close operation. As a further consequence the file system will have no resource leakage.

File reading and writing Let us begin by giving a semantics to file reading and writing. We need an abstract operation for each file operation.

$$\mathsf{FileRW} \stackrel{\text{def}}{=} \{\mathsf{Read} : \mathsf{Int} \twoheadrightarrow \mathsf{Option\ String}; \mathsf{Write} : \langle \mathsf{Int}; \mathsf{String} \rangle \twoheadrightarrow 1\}$$

The operation Read is parameterised by an i-node number (i.e. index into the i-list) and possibly returns the contents of the file pointed to by the i-node. The operation may fail if it is provided with a stale i-node number. Thus the option type is used to signal failure or success to the caller. The Write operation is parameterised by an i-node number and some strings to be appended onto the file pointed to by the i-node. The operation returns unit, and thus the operation does not signal to its caller whether it failed or succeed. Before we implement a handler for the operations, we will implement primitive read and write operations that operate directly on the file system. We will use the primitive operations to implement the semantics for Read and Write. To implement the primitive the operations we will need two basic functions on association lists. I will

only their signatures here.

$$\mathsf{lookup} : \langle \alpha; \mathsf{List}\,\langle \alpha; \beta \rangle \rangle \to \beta!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\}$$
$$\mathsf{modify} : \langle \alpha; \beta; \mathsf{List}\,\langle \alpha; \beta \rangle \rangle \to \langle \alpha; \beta \rangle$$

Given a key of type α the lookup function returns the corresponding value of type β in the given association list. If the key does not exists, then the function invokes the Fail operation to signal failure. The modify function takes a key and a value. If the key exists in the provided association list, then it replaces the value bound by the key with the provided value. Using these functions we can implement the primitive read and write operations.

$$\begin{aligned}&\mathsf{fread} : \langle \mathsf{Int}; \mathsf{FileSystem} \rangle \to \mathsf{String}!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\}\\ &\mathsf{fread}\,\langle ino; fs \rangle \stackrel{\text{def}}{=} \textbf{let}\ inode \leftarrow \mathsf{lookup}\,\langle ino; fs.ilist \rangle\ \textbf{in}\\ &\qquad\qquad \mathsf{lookup}\,\langle inode.loc; fs.dreg \rangle\end{aligned}$$

The function fread takes as input the i-node number for the file to be read and a file system. First it looks up the i-node structure in the i-list, and then it uses the location in the i-node to look up the file contents in the data region. Since fread performs no exception handling it will fail if either look up fails. The implementation of the primitive write operation is similar.

$$\begin{aligned}&\mathsf{fwrite} : \langle \mathsf{Int}; \mathsf{String}; \mathsf{FileSystem} \rangle \to \mathsf{FileSystem}!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\}\\ &\mathsf{fwrite}\,\langle ino; cs; fs \rangle \stackrel{\text{def}}{=} \textbf{let}\ inode \leftarrow \mathsf{lookup}\,\langle ino; fs.ilist \rangle\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ file \leftarrow \mathsf{lookup}\,\langle inode.loc; fs.dreg \rangle\ \textbf{in}\\ &\qquad\qquad \langle fs\ \textbf{with}\ dreg = \mathsf{modify}\,\langle inode.loc; file +\!\!+ cs; fs \rangle \rangle\end{aligned}$$

The first two lines grab hold of the file, whilst the last line updates the data region in file system by appending the string *cs* onto the file. Before we can implement the handler, we need an exception handling mechanism. The following exception handler interprets Fail as some default value.

$$\begin{aligned}&\mathsf{withDefault} : \langle \alpha; 1 \to \alpha!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\} \rangle \to \alpha\\ &\mathsf{withDefault}\,\langle default; m \rangle \stackrel{\text{def}}{=}\ \textbf{handle}\ m\ \langle\rangle\ \textbf{with}\\ &\qquad\qquad\qquad\qquad \textbf{return}\ x \qquad\quad \mapsto x\\ &\qquad\qquad\qquad\qquad \langle\!\langle \mathsf{Fail}\ \langle\rangle \twoheadrightarrow \_ \rangle\!\rangle \mapsto default\end{aligned}$$

The Fail-case is simply the default value, whilst the **return**-case is the identity. Now we can use all the above pieces to implement a handler for the Read and Write operations.

$$
\begin{array}{l}
\mathsf{fileRW} : (1 \to \alpha!\mathsf{FileRW}) \to \alpha!\mathsf{State\ FileSystem} \\
\mathsf{fileRW}\ m \stackrel{\text{def}}{=} \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
\qquad \begin{array}{ll}
\mathbf{return}\ res & \mapsto res \\
\langle\!\langle \mathsf{Read}\ ino \twoheadrightarrow resume \rangle\!\rangle & \mapsto \mathbf{let}\ cs \leftarrow \mathsf{withDefault}\ \langle \mathsf{None}; \lambda\langle\rangle. \\
 & \qquad \mathsf{Some}\ (\mathsf{fread}\ \langle ino; \mathsf{get}\ \langle\rangle\rangle)\rangle \\
 & \quad \mathbf{in}\ resume\ cs \\
\langle\!\langle \mathsf{Write}\ \langle ino; cs\rangle \twoheadrightarrow resume \rangle\!\rangle & \mapsto \mathsf{withDefault}\ \langle\langle\rangle; \lambda\langle\rangle. \\
 & \qquad \mathbf{let}\ fs \leftarrow \mathsf{fwrite}\ \langle ino; cs; \mathsf{get}\ \langle\rangle\rangle \\
 & \qquad \mathbf{in}\ \mathsf{put}\ fs\rangle;\ resume\ \langle\rangle
\end{array}
\end{array}
$$

The Read-case uses the fread function to implement reading a file. The file system state is retrieved using the state operation get. The possible failure of fread is dealt with by the withDefault handler by interpreting failure as None. The Write-case makes use of the fwrite function to implement writing to a file. Again the file system state is retrieved using get. The put operation is used to update the file system state with the state produced by the successful invocation of fwrite. Failure is interpreted as unit, meaning that from the caller's perspective the operation fails silently.

File creation and opening The signature of file creation and opening is unsurprisingly comprised of two operations.

$$\mathsf{FileCO} \stackrel{\text{def}}{=} \{\mathsf{Create} : \mathsf{String} \twoheadrightarrow \mathsf{Option\ Int}; \mathsf{Open} : \mathsf{String} \twoheadrightarrow \mathsf{Option\ Int}\}$$

The implementation of file creation and opening follows the same pattern as the implementation of reading and writing. As before, we implement a primitive routine for each operation that interacts directly with the file system structure. We first implement the primitive file opening function as the file creation function depends on this function.

$$
\begin{array}{l}
\mathsf{fopen} : \langle \mathsf{String}; \mathsf{FileSystem}\rangle \to \mathsf{Int}!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\} \\
\mathsf{fopen}\ \langle fname; fs\rangle \stackrel{\text{def}}{=} \mathsf{lookup}\ \langle fname; fs.dir\rangle
\end{array}
$$

Opening a file in the file system simply corresponds to returning the i-node index associated with the filename in the directory table.

The UNIX file create command does one of two things depending on the state of the file system. If the create command is provided with the name of a file that is

already present in the directory, then the system truncates the file, and returns the file descriptor for the file. Otherwise the system allocates a new empty file and returns its file descriptor [241]. To check whether a file already exists in the directory we need a function has that given a filename and the file system state returns whether there exists a file with the given name. This function can be built completely generically from the functions we already have at our disposal.

$$\begin{aligned}&\mathsf{has} : \langle \alpha; \mathsf{List}\,\langle \alpha; \beta\rangle\rangle \to \mathsf{Bool}\\ &\mathsf{has}\,\langle k; xs\rangle \stackrel{\text{def}}{=} \mathsf{withDefault}\,\langle \mathsf{false}; (\lambda\langle\rangle.\mathsf{lookup}\,\langle k; xs\rangle; \mathsf{true})\rangle\end{aligned}$$

The function has applies lookup under the failure handler with default value false. If lookup returns successfully then its result is ignored, and the computation returns true, otherwise the computation returns the default value false. With this function we can implement the semantics of create.

$$\begin{aligned}&\mathsf{fcreate} : \langle \mathsf{String}; \mathsf{FileSystem}\rangle \to \langle \mathsf{Int}; \mathsf{FileSystem}\rangle\,!\,\{\mathsf{Fail} : 1 \twoheadrightarrow 0\}\\ &\mathsf{fcreate}\,\langle \mathit{fname}; \mathit{fs}\rangle \stackrel{\text{def}}{=} \textbf{if}\ \mathsf{has}\,\langle \mathit{fname}; \mathit{fs}.\mathit{dir}\rangle\ \textbf{then}\\ &\qquad\qquad \textbf{let}\ \mathit{ino} \leftarrow \mathsf{fopen}\,\langle \mathit{fname}; \mathit{fs}\rangle\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{inode} \leftarrow \mathsf{lookup}\,\langle \mathit{ino}; \mathit{fs}\rangle\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{dreg}' \leftarrow \mathsf{modify}\,\langle \mathit{inode}.\mathit{loc}; \text{""}; \mathit{fs}.\mathit{dreg}\rangle\ \textbf{in}\\ &\qquad\qquad \langle \mathit{ino}; \langle \mathit{fs}\ \textbf{with}\ \mathit{dreg} = \mathit{dreg}'\rangle\rangle\\ &\qquad\quad \textbf{else}\\ &\qquad\qquad \textbf{let}\ \mathit{loc} \leftarrow \mathit{fs}.\mathit{lnext}\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{dreg} \leftarrow \langle \mathit{loc}; \text{""}\rangle :: \mathit{fs}.\mathit{dreg}\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{ino} \leftarrow \mathit{fs}.\mathit{inext}\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{inode} \leftarrow \langle \mathit{loc} = \mathit{loc}; \mathit{lno} = 1\rangle\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{ilist} \leftarrow \langle \mathit{ino}; \mathit{inode}\rangle :: \mathit{fs}.\mathit{ilist}\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ \mathit{dir} \leftarrow \langle \mathit{fname}; \mathit{ino}\rangle :: \mathit{fs}.\mathit{dir}\ \textbf{in}\\ &\qquad\qquad \langle \mathit{ino}; \langle \mathit{dir} = \mathit{dir}; \mathit{ilist} = \mathit{ilist}; \mathit{dreg} = \mathit{dreg};\\ &\qquad\qquad\qquad \mathit{lnext} = \mathit{loc} + 1; \mathit{inext} = \mathit{ino} + 1\rangle\rangle\end{aligned}$$

The **then**-branch accounts for the case where the filename *fname* already exists in the directory. First we retrieve the i-node for the file to obtain its location in the data region such that we can truncate the file contents. The branch returns the i-node index along with the modified file system. The **else**-branch allocates a new empty file. First we allocate a location in the data region by copying the value of *fs.lnext* and consing the location and empty string onto *fs.dreg*. The next three lines allocates the i-node for the file in a similar fashion. The second to last line associates the filename with the

new i-node. The last line returns the identifier for the i-node along with the modified file system, where the next location (*lnext*) and next i-node identifier (*inext*) have been incremented. It is worth noting that the effect signature of fcreate mentions Fail even though it will never fail. It is present in the effect row due to the use of fopen and lookup in the **then**-branch. Either application can only fail if the file system is in an inconsistent state, where the index *ino* has become stale. The f-family of functions have been carefully engineered to always leave the file system in a consistent state.

Now we can implement the semantics for the Create and Open effectful operations. The implementation is similar to the implementation of fileRW.

$$
\begin{array}{l}
\mathsf{fileCO} : (1 \to \alpha!\mathsf{FileCO}) \to \alpha!\mathsf{State\ FileSystem} \\
\mathsf{fileCO}\ m \stackrel{\text{def}}{=} \mathbf{handle}\ m\,\langle\rangle\ \mathbf{with} \\
\qquad \begin{array}{ll}
\mathbf{return}\ \mathit{res} & \mapsto \mathit{res} \\
\langle\!\langle \mathsf{Create}\ \mathit{fname} \twoheadrightarrow \mathit{resume} \rangle\!\rangle & \mapsto \mathbf{let}\ \mathit{ino} \leftarrow \mathsf{withDefault}\,\langle \mathsf{None}; \lambda\langle\rangle . \\
 & \qquad \mathbf{let}\ \langle \mathit{ino}; \mathit{fs} \rangle = \mathsf{fcreate}\,\langle \mathit{fname}; \mathsf{get}\,\langle\rangle \rangle \\
 & \qquad \mathbf{in}\ \mathsf{put}\ \mathit{fs};\ \mathsf{Some}\ \mathit{ino} \rangle \\
 & \quad \mathbf{in}\ \mathit{resume}\ \mathit{ino} \\
\langle\!\langle \mathsf{Open}\ \mathit{fname} \twoheadrightarrow \mathit{resume} \rangle\!\rangle & \mapsto \mathbf{let}\ \mathit{ino} \leftarrow \mathsf{withDefault}\ \langle \mathsf{None}; \lambda\langle\rangle . \\
 & \qquad \mathsf{Some}\,(\mathsf{fopen}\,\langle \mathit{fname}; \mathsf{get}\,\langle\rangle \rangle) \rangle \\
 & \quad \mathbf{in}\ \mathit{resume}\ \mathit{ino}
\end{array}
\end{array}
$$

Stream redirection The processes we have defined so far use the echo utility to write to the stdout file. The target file stdout is hardwired into the definition of echo (Section 2.1). To take advantage of the capabilities of the new file system we could choose to modify the definition of echo such that it is parameterised by the target file. However, such a modification is a breaking change. Instead we can define a *stream redirection* operator that allow us to redefine the target of Write operations locally.

$$
\begin{array}{l}
> : \langle 1 \to \alpha!\{\mathsf{Write} : \langle \mathsf{Int}; \mathsf{String} \rangle \twoheadrightarrow 1\}; \mathsf{String} \rangle \\
\quad \to \alpha!\{\mathsf{Create} : \mathsf{String} \twoheadrightarrow \mathsf{Option\ Int}; \mathsf{Exit} : \mathsf{Int} \twoheadrightarrow 0; \mathsf{Write} : \langle \mathsf{Int}; \mathsf{String} \rangle \twoheadrightarrow 1\} \\
m > \mathit{fname} \stackrel{\text{def}}{=} \mathbf{let}\ \mathit{ino} \leftarrow \mathbf{case\ do}\ \mathsf{Create}\ \mathit{fname}\ \{\ \mathsf{None} \mapsto \mathsf{exit}\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Some}\ \mathit{ino} \mapsto \mathit{ino}\} \\
\qquad\qquad \mathbf{in\ handle}\ m\,\langle\rangle\ \mathbf{with} \\
\qquad\qquad\ \mathbf{return}\ \mathit{res} \qquad\qquad\qquad \mapsto \mathit{res} \\
\qquad\qquad\ \langle\!\langle \mathsf{Write}\ \langle \_; \mathit{cs} \rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathit{resume}\,(\mathbf{do}\ \mathsf{Write}\,\langle \mathit{ino}; \mathit{cs} \rangle)
\end{array}
$$

The operator > first attempts to create a new target file with name *fname*. If it fails it simply exits with code 1. Otherwise it continues with the i-node reference *ino*. The

handler overloads the definition of Write inside the provided computation m. The new definition drops the i-node reference of the initial target file and replaces it by the reference to new target file.

This stream redirection operator is slightly more general than the original redirection operator in the original UNIX environment. As the UNIX redirection operator only redirects writes targeted at the *stdout* file [241], whereas the above operator redirects writes regardless of their initial target. It is straightforward to implement this original UNIX behaviour by inspecting the first argument of Write in the operation clause before committing to performing the redirecting Write operation. Modern UNIX environments typically provide more fine-grained control over redirects, for example by allowing the user to specify on a per file basis which writes should be redirected. Again, we can implement this behaviour by comparing the provided file descriptor with the descriptor in the payload of Write.

We can plug everything together to observe the new file system in action.

```
runState ⟨fs0; fileRW (λ⟨⟩.
  fileCO (λ⟨⟩.
    timeshare (λ⟨⟩.
      interruptWrite (λ⟨⟩.
        sessionmgr ⟨Root; λ⟨⟩.
          status (λ⟨⟩.if fork ⟨⟩ then su Alice; ritchie > "ritchie.txt"
                                  else su Bob; hamlet > "hamlet")))))⟩
⇝+ ⟨[0,0];
   ⟨dir = [⟨"hamlet";2⟩, ⟨"ritchie.txt";1⟩, ⟨"stdout";0⟩];
    ilist = [⟨2; ⟨lno = 1; loc = 2⟩⟩, ⟨1; ⟨lno = 1; loc = 1⟩⟩, ⟨0; ⟨lno = 1; loc = 0⟩⟩];
    dreg = [⟨2; "To be, or not to be,\nthat is the question:\n
                 Whether 'tis nobler in the mind to suffer\n"⟩,
            ⟨1; "UNIX is basically a simple operating system,
                 but you have to be a genius to understand the simplicity.\n"⟩,
            ⟨0; ""⟩]; lnext = 3; inext = 3⟩⟩
   : ⟨List Int; FileSystem⟩
```

The writes of the processes ritchie and hamlet are now being redirected to designated files `ritchie.txt` and `hamlet`, respectively. The operating system returns the completion status of all the processes along with the current state of the file system such that it can be used as the initial file system state on the next start of the operating system.

File linking and unlinking

At this point the implementation of BSFS is almost feature complete. However, we have yet to implement two dual file operations: linking and unlinking. The former enables us to associate a new filename with an existing i-node, thus providing a mechanism for making soft copies of files (i.e. the file contents are shared). The latter lets us dissociate a filename from an i-node, thus providing a means for removing files. The interface of linking and unlinking is given below.

$$\mathsf{FileLU} \stackrel{\text{def}}{=} \{\mathsf{Link} : \langle \mathsf{String}; \mathsf{String} \rangle \twoheadrightarrow 1; \mathsf{Unlink} : \mathsf{String} \twoheadrightarrow 1\}$$

The Link operation is parameterised by two strings. The first string is the name of the *source* file and the second string is the *destination* name (i.e. the new name). The Unlink operation takes a single string argument, which is the name of the file to be removed.

As before, we bundle the low level operations on the file system state into their own functions. We start with file linking.

$$\begin{aligned}
&\mathsf{flink} : \langle \mathsf{String}; \mathsf{String}; \mathsf{FileSystem} \rangle \to \mathsf{FileSystem}!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\} \\
&\mathsf{flink}\,\langle src; dest; fs \rangle \stackrel{\text{def}}{=} \mathbf{if}\ \mathsf{has}\,\langle dest; fs.dir \rangle\ \mathbf{then\ absurd\ do}\ \mathsf{Fail}\,\langle\rangle \\
&\qquad\qquad \mathbf{else\ let}\ ino \leftarrow \mathsf{lookup}\,\langle src; fs.dir \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ dir' \leftarrow \langle dest; ino \rangle :: fs.dir\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ inode \leftarrow \mathsf{lookup}\,\langle ino; fs.ilist \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ inode' \leftarrow \langle inode\ \mathbf{with}\ lno = inode.lno + 1 \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ ilist' \leftarrow \mathsf{modify}\,\langle ino; inode'; fs.ilist \rangle\ \mathbf{in} \\
&\qquad\qquad\quad \langle fs\ \mathbf{with}\ dir = dir'; ilist = ilist' \rangle
\end{aligned}$$

The function flink checks whether the destination filename, *dest*, already exists in the directory. If it exists then the function raises the Fail exception. Otherwise it looks up the index of the i-node, *ino*, associated with the source file, *src*. Next, the directory is extended with the destination filename, which gets associated with this index, meaning *src* and *dest* both share the same i-node. Finally, the link count of the i-node at index *ino* gets incremented, and the function returns the updated file system state.

The semantics of file unlinking is slightly more complicated as an i-node may become unlinked, meaning that it needs to garbage collected along with its file contents in the data region. To implement file removal we make use of another standard operation on association lists.

$$\mathsf{remove} : \langle \alpha; \langle \alpha; \beta \rangle \rangle \to \langle \alpha; \beta \rangle$$

The first parameter to remove is the key associated with the entry to be removed from the association list, which is given as the second parameter. If the association list does not have an entry for the given key, then the function behaves as the identity. The behaviour of the function in case of multiple entries for a single key does not matter as our system is carefully set up to ensure that each key has an unique entry.

$$
\begin{aligned}
&\mathsf{funlink} : \langle \mathsf{String}; \mathsf{FileSystem} \rangle \to \mathsf{FileSystem}!\{\mathsf{Fail} : 1 \twoheadrightarrow 0\} \\
&\mathsf{funlink}\,\langle \mathit{fname}; \mathit{fs} \rangle \stackrel{\text{def}}{=} \textbf{if}\ \mathsf{has}\,\langle \mathit{fname}; \mathit{fs}.\mathit{dir} \rangle\ \textbf{then} \\
&\qquad\qquad \textbf{let}\ \mathit{ino} \leftarrow \mathsf{lookup}\,\langle \mathit{fname}; \mathit{fs}.\mathit{dir} \rangle\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ \mathit{dir}' \leftarrow \mathsf{remove}\,\langle \mathit{fname}; \mathit{fs}.\mathit{dir} \rangle\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ \mathit{inode} \leftarrow \mathsf{lookup}\,\langle \mathit{ino}; \mathit{fs}.\mathit{ilist} \rangle\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ \langle \mathit{ilist}'; \mathit{dreg}' \rangle \leftarrow \textbf{if}\ \mathit{inode}.\mathit{lno} > 1\ \textbf{then} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ \mathit{inode}' \leftarrow \langle \mathit{inode}\ \textbf{with} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{lno} = \mathit{inode}.\mathit{lno} - 1 \rangle \\
&\qquad\qquad\qquad\qquad \textbf{in}\ \langle \mathsf{modify}\,\langle \mathit{ino}; \mathit{inode}'; \mathit{fs}.\mathit{ilist} \rangle; \mathit{fs}.\mathit{dreg} \rangle \\
&\qquad\qquad\qquad\quad \textbf{else}\ \langle \mathsf{remove}\,\langle \mathit{ino}; \mathit{fs}.\mathit{ilist} \rangle; \\
&\qquad\qquad\qquad\qquad\quad \mathsf{remove}\,\langle \mathit{inode}.\mathit{loc}; \mathit{fs}.\mathit{dreg} \rangle \rangle \\
&\qquad\qquad \textbf{in}\ \langle \mathit{fs}\ \textbf{with}\ \mathit{dir} = \mathit{dir}'; \mathit{ilist} = \mathit{ilist}'; \mathit{dreg} = \mathit{dreg}' \rangle \\
&\qquad\quad \textbf{else absurd do}\ \mathsf{Fail}\,\langle\rangle
\end{aligned}
$$

The funlink function checks whether the given filename *fname* exists in the directory. If it does not, then it raises the Fail exceptions. However, if it does exist then the function proceeds to lookup the index of the i-node for the file, which gets bound to *ino*, and subsequently remove the filename from the directory. Afterwards it looks up the i-node with index *ino*. Now one of two things happen depending on the current link count of the i-node. If the count is greater than one, then we need only decrement the link count by one, thus we modify the i-node structure. If the link count is 1, then i-node is about to become stale, thus we must garbage collect it by removing both the i-node from the i-list and the contents from the data region. Either branch returns the new state of i-list and data region. Finally, the function returns the new file system state.

With the flink and funlink functions, we can implement the semantics for Link and

Unlink operations following the same patterns as for the other file system operations.

```
fileLU : (1 → α!FileLU) → α!State FileSystem
fileLU m ≝ handle m ⟨⟩ with
    return res                          ↦ res
    ⟪Link ⟨src; dest⟩ ↠ resume⟫         ↦ withDefault ⟨⟩; λ⟨⟩.
                                            let fs = flink ⟨src; dest; get ⟨⟩⟩
                                            in put fs⟩; resume ⟨⟩
    ⟪Unlink fname ↠ resume⟫             ↦ withDefault ⟨⟩; λ⟨⟩.
                                            let fs = funlink ⟨fname; get ⟨⟩⟩
                                            in put fs⟩; resume ⟨⟩
```

The composition of fileRW, fileCO, and fileLU complete the implementation of BSFS.

```
FileIO ≝ {FileRW; FileCO; FileLU}

fileIO : (1 → α!FileIO) → α!State FileSystem
fileIO m ≝ fileRW (λ⟨⟩.fileCO (λ⟨⟩.fileLU m))
```

The three handlers may as well be implemented as a single monolithic handler, since they implement different operations, return the same value, and make use of the same state cell. In practice a monolithic handler may have better performance. However, a sufficiently clever compiler would be able to take advantage of the fusion laws of deep handlers to fuse the three handlers into one (e.g. using the technique of Wu and Schrijvers [275]), and thus allow modular composition without composition.

We now have the building blocks to implement a file copying utility. We will implement the utility such that it takes an argument to decide whether it should make a soft copy such that the source file and destination file are linked, or it should make a hard copy such that a new i-node is allocated and the bytes in the data regions gets duplicated.

```
cp : ⟨Bool; String; String⟩ → 1!{FileIO; Exit : Int ↠ 0}
cp ⟨link; src; dest⟩ ≝ if link then do Link ⟨src; dest⟩
                       else case do Open src
                            { None     ↦ exit 1
                              Some ino ↦
                                case do Read ino { None    ↦ exit 1
                                                   Some cs ↦ echo cs > dest}}
```

If the *link* parameter is true, then the utility makes a soft copy by performing the operation Link to link the source file and destination file. Otherwise the utility makes a hard copy by first opening the source file. If Open returns the None (i.e. the open failed) then the utility exits with code 1. If the open succeeds then the entire file contents are read. If the read operation fails then we again just exit, however, in the event that it succeeds we apply the echo to the file contents and redirects the output to the file *dest*.

The logic for file removal is part of the semantics for Unlink. Therefore the implementation of a file removal utility is simply an application of the operation Unlink.

$$\begin{aligned}&\mathsf{rm} : \mathsf{String} \to 1!\{\mathsf{Unlink} : \mathsf{String} \twoheadrightarrow 1\}\\&\mathsf{rm}\ \mathit{fname} \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Unlink}\ \mathit{fname}\end{aligned}$$

We can now plug it all together.

```
runState ⟨fs0; fileIO (λ⟨⟩.
  timeshare (λ⟨⟩.
    interruptWrite (λ⟨⟩.
      sessionmgr ⟨Root; λ⟨⟩.
        status (λ⟨⟩.if fork ⟨⟩
                  then su Alice; ritchie > "ritchie.txt";
                       cp ⟨false; "ritchie.txt"; "ritchie"⟩;
                       rm "ritchie.txt"
                  else su Bob; hamlet > "hamlet";
                       cp ⟨true; "hamlet"; "act3"⟩)))))⟩
⇝+ ⟨[0,0];
   ⟨dir = [⟨"ritchie";3⟩, ⟨"act3";2⟩, ⟨"hamlet";2⟩, ⟨"stdout";0⟩];
    ilist = [⟨3; ⟨lno = 1; loc = 3⟩⟩, ⟨2; ⟨lno = 2; loc = 2⟩⟩, ⟨0; ⟨lno = 1; loc = 0⟩⟩];
    dreg = [⟨3; "UNIX is basically a simple operating system,
                 but you have to be a genius
                 to understand the simplicity.\n"⟩,
            ⟨2; "To be, or not to be,\nthat is the question:\n
                 Whether 'tis nobler in the mind to suffer\n"⟩,
            ⟨0; ""⟩]; lnext = 4; inext = 4⟩⟩
   : ⟨List Int; FileSystem⟩
```

Alice copies the file `ritchie.txt` as `ritchie`, and subsequently removes the original file, which effectively amounts to a roundabout way of renaming a file. It is evident

from the file system state that the file is a hard copy as the contents of `ritchie.txt` now reside in location 3 rather than location 1 in the data region. Bob makes a soft copy of the file `hamlet` as `act3`, which is evident by looking at the directory where the two filenames point to the same i-node (with index 2), whose link counter has value 2.

Summary Throughout the preceding sections we have used effect handlers to give a semantics to a UNIX-style operating system by treating system calls as effectful operations, whose semantics are given by handlers, acting as composable micro-kernels. Starting from a simple bare minimum file I/O model we seen how the modularity of effect handlers enable us to develop a feature-rich operating system in an incremental way by composing several handlers to implement a basic file system, multi-user environments, and multi-tasking support. Each incremental change to the system has been backwards compatible with previous changes in the sense that we have not modified any previously defined interfaces in order to support a new feature. It serves as a testament to demonstrate the versatility of effect handlers, and it suggests that handlers can be a viable option to use with legacy code bases to retrofit functionality. The operating system makes use of fourteen operations, which are being handled by twelve handlers, some of which are used multiple times, e.g. the env and > handlers.

2.6 UNIX-style pipes

In this section we will implement UNIX *pipes* to replicate the UNIX programming experience. A UNIX pipe is an abstraction for streaming communication between two processes. Technically, a pipe works by connecting the standard out file descriptor of the first process to the standard in file descriptor of the second process. The second process can then handle the output of the first process by reading its own standard in file [241].

We could implement pipes using the file system, however, it would require us to implement a substantial amount of bookkeeping as we would have to generate and garbage collect a standard out file and a standard in file per process. Instead we can represent the files as effectful operations and connect them via handlers. The principal idea is to implement an abstraction similar to Ganz et al.'s seesaw trampoline, where two processes take turn to run [105]. We will have a *consumer* process that *awaits* input, and a *producer* process that *yields* output. However, implementing this sort of abstraction with deep handlers is irksome, because deep handlers hard-wire the inter-

pretation of operations in the computation and therefore do not let us readily change the interpretation of operations. By contrast, *shallow handlers* offer more flexibility as they let us change the handler after each operation invocation. The technical reason being that resumptions provided a shallow handler do not implicitly include the handler as well, thus an invocation of a resumption originating from a shallow handler must be explicitly run under another handler by the programmer. To illustrate shallow handlers in action, let us consider how one might implement a demand-driven UNIX pipeline operator as two mutually recursive handlers.

$$
\begin{aligned}
&\mathsf{pipe} : \langle 1 \to \alpha!\{\mathsf{Yield} : \beta \twoheadrightarrow 1\}; 1 \to \alpha!\{\mathsf{Await} : 1 \twoheadrightarrow \beta\}\rangle \to \alpha \\
&\mathsf{pipe}\,\langle p; c\rangle \stackrel{\text{def}}{=} \mathbf{handle}^{\dagger}\ c\,\langle\rangle\ \mathbf{with} \\
&\qquad\qquad \mathbf{return}\ x \qquad\qquad \mapsto x \\
&\qquad\qquad \langle\!\langle \mathsf{Await}\ \langle\rangle \twoheadrightarrow \mathit{resume}\rangle\!\rangle \mapsto \mathsf{copipe}\,\langle \mathit{resume}; p\rangle \\
&\mathsf{copipe} : \langle \beta \to \alpha!\{\mathsf{Await} : 1 \twoheadrightarrow \beta\}; 1 \to \alpha!\{\mathsf{Yield} : \beta \twoheadrightarrow 1\}\rangle \to \alpha \\
&\mathsf{copipe}\,\langle c; p\rangle \stackrel{\text{def}}{=} \mathbf{handle}^{\dagger}\ p\,\langle\rangle\ \mathbf{with} \\
&\qquad\qquad \mathbf{return}\ x \qquad\qquad \mapsto x \\
&\qquad\qquad \langle\!\langle \mathsf{Yield}\ y \twoheadrightarrow \mathit{resume}\rangle\!\rangle \mapsto \mathsf{pipe}\,\langle \mathit{resume}; \lambda\langle\rangle.c\,y\rangle
\end{aligned}
$$

A pipe takes two suspended computations, a producer p and a consumer c. Each of the computations returns a value of type α. The producer can perform the Yield operation, which yields a value of type β and the consumer can perform the Await operation, which correspondingly awaits a value of type β. The Yield operation corresponds to writing to standard out, whilst Await corresponds to reading from standard in. The pipe runs the consumer under a **handle**†-construct, which is the term syntax for shallow handler application. If the consumer terminates with a value, then the **return** clause is executed and returns that value as is. If the consumer performs the Await operation, then the copipe handler is invoked with the resumption of the consumer (*resume*) and the producer (p) as arguments. This models the effect of blocking the consumer process until the producer process provides some data. The type of *resume* in this context is $\beta \to \alpha!\{\mathsf{Await} : 1 \twoheadrightarrow \beta\}$, that is the Await operation is present in the effect row of the *resume*, which is the type system telling us that a bare application of *resume* is unguarded as in order to safely apply the resumption, we must apply it in a context which handles Await. This is the key difference between a shallow resumption and a deep resumption.

The copipe function runs the producer to get a value to feed to the waiting consumer. If the producer performs the Yield operation, then pipe is invoked with the

resumption of the producer along with a thunk that applies the consumer's resumption to the yielded value. For aesthetics, we define a right-associative infix alias for pipe: $p \mid c \stackrel{\text{def}}{=} \lambda\langle\rangle.\mathsf{pipe}\,\langle p; c\rangle$.

Let us put the pipe operator to use by performing a simple string frequency analysis on a file. We will implement the analysis as a collection of small single-purpose utilities which we connect by way of pipes. We will build a collection of small utilities. We will make use of two standard list iteration functions.

$$\begin{aligned}\mathsf{map} &: \langle \alpha \to \beta; \mathsf{List}\ \alpha\rangle \to \mathsf{List}\ \beta \\ \mathsf{iter} &: \langle \alpha \to \beta; \mathsf{List}\ \alpha\rangle \to 1\end{aligned}$$

The function map applies its function argument to each element of the provided list in left-to-right order and returns the resulting list. The function iter is simply map where the resulting list is ignored. Our first utility is a simplified version of the GNU coreutil utility *cat*, which copies the contents of files to standard out [186, Section 3.1]. Our version will open a single file and stream its contents one character at a time.

```
cat : String → 1!{FileIO; Yield : Char ↠ 1; Exit : Int ↠ 0}
cat fname ≝ case do Open fname {
              None     ↦ exit 1
              Some ino ↦ case do Read ino {
                           None    ↦ exit 1
                           Some cs ↦ iter ⟨λc.do Yield c; cs⟩; do Yield '\0'}}
```

The last line is the interesting line of code. The contents of the file gets bound to *cs*, which is supplied as an argument to the list iteration function iter. The function argument yields each character. Each invocation of Yield suspends the iteration until the next character is awaited. This is an example of inversion of control as iteration function iter has effectively been turned into a generator, whose elements are computed on demand. We use the character \0 to identify the end of a stream. It is essentially a character interpretation of the empty list (file) [].

The cat utility processes the entire contents of a given file. However, we may only be interested in some parts. The GNU coreutil *head* provides a way to process only a fixed amount of lines and ignore subsequent lines [186, Section 5.1]. We will implement a simplified version of this utility which lets us keep the first n lines of a stream and discard the remainder. This process will act as a *filter*, which is an intermediary process

in a pipeline that both awaits and yields data.

$$
\begin{aligned}
&\mathsf{head} : \mathsf{Int} \to 1!\{\mathsf{Await} : 1 \twoheadrightarrow \mathsf{Char}; \mathsf{Yield} : \mathsf{Char} \twoheadrightarrow 1\} \\
&\mathsf{head}\ n \stackrel{\text{def}}{=} \textbf{if } n = 0 \textbf{ then do } \mathsf{Yield}\ \texttt{'\textbackslash 0'} \\
&\qquad\qquad \textbf{else let } c \leftarrow \textbf{do } \mathsf{Await}\ \langle\rangle \textbf{ in} \\
&\qquad\qquad\qquad \textbf{do } \mathsf{Yield}\ c; \\
&\qquad\qquad\qquad \textbf{if } c = \texttt{'\textbackslash 0'} \textbf{ then } \langle\rangle \\
&\qquad\qquad\qquad \textbf{else if } c = \texttt{'\textbackslash n'} \textbf{ then } \mathsf{head}\ (n-1) \\
&\qquad\qquad\qquad \textbf{else } \mathsf{head}\ n
\end{aligned}
$$

The function first checks whether more lines need to be processed. If n is zero, then it yields the nil character to signify the end of stream. This has the effect of ignoring any future instances of Yield in the input stream. Otherwise it awaits a character. Once a character has been received the function yields the character in order to include it in the output stream. After the yield, it checks whether the character was nil in which case the process terminates. Alternatively, if the character was a newline the function applies itself recursively with n decremented by one. Otherwise it applies itself recursively with the original n.

The head filter does not transform the shape of its data stream. It both awaits and yields a character. However, the awaits and yields need not operate on the same type within the same filter, meaning we can implement a filter that transforms the shape of the data. Let us implement a variation of the GNU coreutil *paste* which merges lines of files [186, Section 8.2]. Our implementation will join characters in its input stream into strings separated by spaces and newlines such that the string frequency analysis utility need not operate on the low level of characters.

$$
\begin{aligned}
&\mathsf{paste} : 1 \to 1!\{\mathsf{Await} : 1 \twoheadrightarrow \mathsf{Char}; \mathsf{Yield} : \mathsf{String} \twoheadrightarrow 1\} \\
&\mathsf{paste}\ \langle\rangle \stackrel{\text{def}}{=} \mathit{pst}\ \langle \textbf{do } \mathsf{Await}\ \langle\rangle; \texttt{""}\rangle \\
&\qquad \textbf{where } \mathit{pst}\ \langle \texttt{'\textbackslash 0'}; \mathit{str}\rangle \stackrel{\text{def}}{=} \textbf{do } \mathsf{Yield}\ \mathit{str}; \textbf{do } \mathsf{Yield}\ \texttt{"\textbackslash 0"} \\
&\qquad\qquad \mathit{pst}\ \langle \texttt{'\textbackslash n'}; \mathit{str}\rangle \stackrel{\text{def}}{=} \textbf{do } \mathsf{Yield}\ \mathit{str}; \textbf{do } \mathsf{Yield}\ \texttt{"\textbackslash n"}; \mathit{pst}\ \langle \textbf{do } \mathsf{Await}\ \langle\rangle; \texttt{""}\rangle \\
&\qquad\qquad \mathit{pst}\ \langle \texttt{' '}; \mathit{str}\rangle \stackrel{\text{def}}{=} \textbf{do } \mathsf{Yield}\ \mathit{str}; \mathit{pst}\ \langle \textbf{do } \mathsf{Await}\ \langle\rangle; \texttt{""}\rangle \\
&\qquad\qquad \mathit{pst}\ \langle c; \mathit{str}\rangle \stackrel{\text{def}}{=} \mathit{pst}\ \langle \textbf{do } \mathsf{Await}\ \langle\rangle; \mathit{str} +\!\!+ [c]\rangle
\end{aligned}
$$

The heavy-lifting is delegated to the recursive function *pst* which accepts two parameters: 1) the next character in the input stream, and 2) a string buffer for building the output string. The function is initially applied to the first character from the stream (returned by the invocation of Await) and the empty string buffer. The function *pst*

is defined by pattern matching on the character parameter. The first three definitions handle the special cases when the received character is nil, newline, and space, respectively. If the character is nil, then the function yields the contents of the string buffer followed by a string with containing only the nil character. If the character is a newline, then the function yields the string buffer followed by a string containing the newline character. Afterwards the function applies itself recursively with the next character from the input stream and an empty string buffer. The case when the character is a space is similar to the previous case except that it does not yield a newline string. The final definition simply concatenates the character onto the string buffer and recurses.

Another useful filter is the GNU stream editor abbreviated *sed* [218]. It is an advanced text processing editor, whose complete functionality we will not attempt to replicate here. We will just implement the ability to replace a string by another. This will be useful for normalising the input stream to the frequency analysis utility, e.g. decapitalise words, remove unwanted characters, etc.

$$
\begin{aligned}
&\mathsf{sed} : \langle \mathsf{String}; \mathsf{String} \rangle \to 1!\{\mathsf{Await} : 1 \twoheadrightarrow \mathsf{String}; \mathsf{Yield} : \mathsf{String} \twoheadrightarrow 1\} \\
&\mathsf{sed}\,\langle \mathit{target}; \mathit{str}' \rangle \stackrel{\text{def}}{=} \mathbf{let}\ \mathit{str} \leftarrow \mathbf{do}\ \mathsf{Await}\ \langle\rangle\ \mathbf{in} \\
&\qquad\qquad \mathbf{if}\ \mathit{str} = \mathit{target}\ \mathbf{then\ do}\ \mathsf{Yield}\ \mathit{str}'; \mathsf{sed}\,\langle \mathit{target}; \mathit{str}' \rangle \\
&\qquad\qquad \mathbf{else\ do}\ \mathsf{Yield}\ \mathit{str}; \mathsf{sed}\,\langle \mathit{target}; \mathit{str}' \rangle
\end{aligned}
$$

The function sed takes two string arguments. The first argument is the string to be replaced in the input stream, and the second argument is the replacement. The function first awaits the next string from the input stream, then it checks whether the received string is the same as *target* in which case it yields the replacement str' and recurses. Otherwise it yields the received string and recurses.

Now let us implement the string frequency analysis utility. It work on strings and count the occurrences of each string in the input stream.

$$
\begin{aligned}
&\mathsf{freq} : 1 \to 1!\{\mathsf{Await} : 1 \twoheadrightarrow \mathsf{String}; \mathsf{Yield} : \mathsf{List}\,\langle \mathsf{String}; \mathsf{Int} \rangle \twoheadrightarrow 1\} \\
&\mathsf{freq}\,\langle\rangle \stackrel{\text{def}}{=} \mathit{freq}'\,\langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle; [] \rangle \\
&\qquad \mathbf{where}\ \mathit{freq}'\,\langle \texttt{"\textbackslash 0"}; \mathit{tbl} \rangle \stackrel{\text{def}}{=} \mathbf{do}\ \mathsf{Yield}\ \mathit{tbl} \\
&\qquad\qquad \mathit{freq}'\,\langle \mathit{str}; \mathit{tbl} \rangle \stackrel{\text{def}}{=} \mathbf{let}\ \mathit{tbl}' \leftarrow \mathsf{withDefault}\,\langle \langle \mathit{str}; 1 \rangle :: \mathit{tbl}; \lambda\langle\rangle . \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \mathit{sum} \leftarrow \mathsf{lookup}\,\langle \mathit{str}; \mathit{tbl} \rangle\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{modify}\,\langle \mathit{str}; \mathit{sum} + 1; \mathit{tbl} \rangle \rangle \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{freq}'\,\langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle; \mathit{tbl}' \rangle
\end{aligned}
$$

The auxiliary recursive function freq' implements the analysis. It takes two arguments: 1) the next string from the input stream, and 2) a table to keep track of how many times

each string has occurred. The table is implemented as an association list indexed by strings. The function is initially applied to the first string from the input stream and the empty list. The function is defined by pattern matching on the string argument. The first definition handles the case when the input stream has been exhausted in which case the function yields the table. The other case is responsible for updating the entry associated with the string *str* in the table *tbl*. There are two subcases to consider: 1) the string has not been seen before, thus a new entry will have to created; or 2) the string already has an entry in the table, thus the entry will have to be updated. We handle both cases simultaneously by making use of the handler withDefault, where the default value accounts for the first subcase, and the computation accounts for the second. The computation attempts to lookup the entry associated with *str* in *tbl*, if the lookup fails then withDefault returns the default value, which is the original table augmented with an entry for *str*. If an entry already exists it gets incremented by one. The resulting table tbl' is supplied to a recursive application of $freq'$.

We need one more building block to complete the pipeline. The utility freq returns a value of type List ⟨String; Int⟩, we need a utility to render the value as a string in order to write it to a file.

$$\begin{aligned}&\mathsf{renderTable} : 1 \to 1!\{\mathsf{Await} : 1 \twoheadrightarrow \mathsf{List}\,\langle \mathsf{String}; \mathsf{Int}\rangle\}\\&\mathsf{renderTable}\,\langle\rangle \stackrel{\text{def}}{=} \mathsf{map}\,\langle \lambda\langle s; i\rangle.s +\!\!+ \text{":"} +\!\!+ \mathsf{intToString}\,i +\!\!+ \text{";"}; \mathbf{do}\ \mathsf{Await}\,\langle\rangle\rangle\end{aligned}$$

The function performs one invocation of Await to receive the table, and then performs a map over the table. The function argument to map builds a string from the provided string-integer pair. Here we make use of an auxiliary function, intToString : Int → String, that turns an integer into a string. The definition of this function is omitted here for brevity.

We now have all the building blocks to construct a pipeline for performing string frequency analysis on a file. The following performs the analysis on the two first lines

of Hamlet quote.

$$
\begin{array}{l}
\mathsf{runState}\,\langle \mathsf{fs}_0; \mathsf{fileIO}\,(\lambda\langle\rangle. \\
\quad \mathsf{timeshare}\,(\lambda\langle\rangle. \\
\quad\quad \mathsf{interruptWrite}\,(\lambda\langle\rangle. \\
\quad\quad\quad \mathsf{sessionmgr}\,\langle \mathsf{Root}; \lambda\langle\rangle. \\
\quad\quad\quad\quad \mathsf{status}\,(\lambda\langle\rangle.\mathsf{hamlet} > \texttt{"hamlet"}; \\
\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ p \leftarrow\ (\lambda\langle\rangle.\mathsf{cat}\ \texttt{"hamlet"}) \mid (\lambda\langle\rangle.\mathsf{head}\ 2) \mid \mathsf{paste} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid (\lambda\langle\rangle.\mathsf{sed}\,\langle\texttt{"be,"};\texttt{"be"}\rangle) \mid (\lambda\langle\rangle.\mathsf{sed}\,\langle\texttt{"To"};\texttt{"to"}\rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid (\lambda\langle\rangle.\mathsf{sed}\,\langle\texttt{"question:"};\texttt{"question"}\rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{freq} \mid \mathsf{renderTable} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ (\lambda\langle\rangle.\mathsf{echo}\ (p\,\langle\rangle)) > \texttt{"analysis"})))))\rangle \\
\rightsquigarrow^{+} \langle [0]; \\
\quad \langle dir = [\langle\texttt{"analysis"};2\rangle, \langle\texttt{"hamlet"};1\rangle, \langle\texttt{"stdout"};0\rangle]; \\
\quad\ ilist = [\langle 2; \langle lno = 1; loc = 2\rangle\rangle, \langle 1; \langle lno = 1; loc = 1\rangle\rangle, \langle 0; \langle lno = 1; loc = 0\rangle\rangle]; \\
\quad\ dreg = [\langle 2; \texttt{"to:2;be:2;or:1;not:1;\textbackslash n:2;that:1;is:1} \\
\quad\quad\quad\quad \texttt{the:1;question:1;"}\rangle, \\
\quad\quad\quad \langle 1; \texttt{"To be, or not to be,\textbackslash nthat is the question:\textbackslash n} \\
\quad\quad\quad\quad \texttt{Whether 'tis nobler in the mind to suffer\textbackslash n"}\rangle, \\
\quad\quad\quad \langle 0; \texttt{""}\rangle]; lnext = 3; inext = 3\rangle\rangle \\
\quad : \langle \mathsf{List\ Int}; \mathsf{FileSystem}\rangle
\end{array}
$$

The pipeline gets bound to the variable p. The pipeline starts with call to cat which streams the contents of the file `"hamlet"` to the process head applied to 2, meaning it will only forward the first two lines of the file to its successor. The third process paste receives the first two lines one character at a time and joins the characters into strings delimited by whitespace. The next three instances of sed perform some string normalisation. The first instance removes the trailing comma from the string `"be,"`; the second normalises the capitalisation of the word “to”; and the third removes the trailing colon from the string `"question:"`. The seventh process performs the frequency analysis and outputs a table, which is being rendered as a string by the eighth process. The output of the pipeline is supplied to the echo utility whose output is being redirected to a file named `"analysis"`. Contents of the file reside in location 2 in the data region. Here we can see that the analysis has found that the words “to”, “be”, and the newline character “\n” appear two times each, whilst the other words appear once each.

2.7 Process synchronisation

In Section 2.4 we implemented a time-sharing system on top of a simple process model. However, the model lacks a process synchronisation facility. It is somewhat difficult to cleanly add support for synchronisation to the implementation as it is in Section 2.4. Firstly, because the interface of $\mathsf{Fork} : 1 \to \mathsf{Bool}$ only gives us two possible process identifiers: true and false, meaning at any point we can only identify two processes. Secondly, and more importantly, some state is necessary to implement synchronisation, but the current implementation of process scheduling is split amongst two handlers and one auxiliary function, all of which need to coordinate their access and manipulation of the state cell. One option is to use some global state via the interface from Section 2.5, which has the advantage of making the state manipulation within the scheduler modular, but it also has the disadvantage of exposing the state as an implementation detail — and it comes with all the caveats of programming with global state. *Parameterised handlers* provide an elegant solution, which lets us internalise the state within the scheduler. Essentially, a parameterised handler is an ordinary deep handler equipped with some state. This state is accessible only internally in the handler and can be updated upon each application of a parameterised resumption. A parameterised resumption is represented as a binary function which in addition to the interpretation of its operation also take updated handler state as input.

We will see how a parameterised handler enables us to implement a richer process model supporting synchronisation with ease. The effect signature of process concurrency is as follows.

$$\mathsf{Co} \stackrel{\text{def}}{=} \{\mathsf{UFork} : 1 \twoheadrightarrow \mathsf{Int}; \mathsf{Wait} : \mathsf{Int} \twoheadrightarrow 1; \mathsf{Interrupt} : 1 \twoheadrightarrow 1\}$$

The operation UFork models UNIX *fork* [241]. It is generalisation of the Fork operation from Section 2.4. The operation is intended to return twice: once to the parent process with a unique process identifier for the child process, and a second time to the child process with the zero identifier. The Wait operation takes a process identifier as argument and then blocks the invoking process until the process associated with the provided identifier has completed. The $\mathsf{Interrupt}$ operation is the same as in Section 2.4; it temporarily suspends the invoking process in order to let another process run.

The main idea is to use the state cell of a parameterised handler to manage the process queue and to keep track of the return values of completed processes. The scheduler will return an association list of process identifiers mapped to the return value of their respective process when there are no more processes to be run. The process

queue will consist of reified processes, which we will represent using parameterised resumptions. To make the type signatures understandable we will make use of three mutually recursive type aliases.

$$\begin{aligned}
&\mathsf{Proc}\quad \alpha\ \varepsilon \stackrel{\text{def}}{=} \mathsf{Sstate}\ \alpha\ \varepsilon \to \mathsf{List}\ \langle \mathsf{Int}; \alpha \rangle\, !\, \varepsilon \\
&\mathsf{Pstate}\ \alpha\ \varepsilon \stackrel{\text{def}}{=} [\mathsf{Ready} : \mathsf{Proc}\ \alpha\ \varepsilon; \mathsf{Blocked} : \langle \mathsf{Int}; \mathsf{Proc}\ \alpha\ \varepsilon \rangle] \\
&\mathsf{Sstate}\ \alpha\ \varepsilon \stackrel{\text{def}}{=} \langle q : \mathsf{List}\ \langle \mathsf{Int}; \mathsf{Pstate}\ \alpha\ \varepsilon \rangle; \mathit{done} : \mathsf{List}\ \alpha; \mathit{pid} : \mathsf{Int}; \mathit{pnext} : \mathsf{Int} \rangle
\end{aligned}$$

The Proc alias is the type of reified processes. It is defined as a function that takes the current scheduler state and returns an association list of αs indexed by integers. This is almost the type of a parameterised resumption as the only thing missing is the a component for the interpretation of an operation. The second alias Pstate enumerates the possible process states. Either a process is *ready* to be run or it is *blocked* on some other process. The payload of the Ready tag is the process to run. The Blocked tag is parameterised by a pair, where the first component is the identifier of the process that is being waited on and the second component is the process to be continued when the other process has completed. The third alias Sstate is the type of scheduler state. It is a quadruple, where the first label q is the process queue. It is implemented as an association list indexed by process identifiers. The second label *done* is used to store the return values of completed processes. The third label *pid* is used to remember the identifier of currently executing process, and the fourth label *pnext* is used to compute a unique identifier for new processes.

We will abstract some of the scheduling logic into an auxiliary function runNext, which is responsible for dequeuing and running the next process from the queue.

$$\begin{aligned}
&\mathsf{runNext} : \mathsf{Sstate}\ \alpha\ \varepsilon \to \mathsf{List}\ \alpha\, !\, \varepsilon \\
&\mathsf{runNext}\ \mathit{st} \stackrel{\text{def}}{=} \mathbf{case}\ \mathit{st}.q\ \{ \\
&\qquad [] \mapsto \mathit{st}.\mathit{done} \\
&\qquad \langle \mathit{pid}; \mathsf{Blocked}\ \langle \mathit{pid}'; \mathit{resume} \rangle \rangle :: q' \mapsto \\
&\qquad\quad \mathbf{let}\ \mathit{st}' \leftarrow \langle \mathit{st}\ \mathbf{with}\ q = q' +\!\!+ [\langle \mathit{pid}; \mathsf{Blocked}\ \langle \mathit{pid}'; \mathit{resume} \rangle \rangle] \rangle\ \mathbf{in} \\
&\qquad\quad \mathsf{runNext}\ \mathit{st}' \\
&\qquad \langle \mathit{pid}; \mathsf{Ready}\ \mathit{resume} \rangle :: q' \mapsto \\
&\qquad\quad \mathbf{let}\ \mathit{st}' \leftarrow \langle \mathit{st}\ \mathbf{with}\ q = q'; \mathit{pid} = \mathit{pid} \rangle\ \mathbf{in} \\
&\qquad\quad \mathit{resume}\ \mathit{st}'\ \}
\end{aligned}$$

The function operates on the scheduler state. It first performs a case split on the process queue. There are three cases to consider.

1. The queue is empty. Then the function returns the list *done*, which is the list of process return values.

2. The next process is blocked. Then the process is appended on to the end of the queue, and runNext is applied recursively to the scheduler state st' with the updated queue.

3. The next process is ready. Then the q and *pid* fields within the scheduler state are updated accordingly. The reified process *resume* is applied to the updated scheduler state st'.

Evidently, this function may enter an infinite loop if every process is in blocked state. This may happen if we deadlock any two processes by having them wait on one another. Using this function we can define a handler that implements a process scheduler.

$$
\begin{array}{l}
\mathsf{scheduler} : \langle \alpha ! \{\mathsf{Co}; \varepsilon\}; \mathsf{Sstate}\ \alpha\ \varepsilon\rangle \Rightarrow^{\ddagger} \mathsf{List}\,\langle \mathsf{Int}; \alpha\rangle ! \varepsilon \\
\mathsf{scheduler} \stackrel{\text{def}}{=} st. \\
\qquad \textbf{return}\ x \mapsto \\
\qquad\quad \textbf{let}\ done' \leftarrow \langle st.pid; x\rangle :: st.done\ \textbf{in} \\
\qquad\quad \mathsf{runNext}\,\langle st\ \textbf{with}\ done = done'\rangle \\
\qquad \langle\!\langle \mathsf{UFork}\ \langle\rangle \twoheadrightarrow resume\rangle\!\rangle \mapsto \\
\qquad\quad \textbf{let}\ resume' \leftarrow \lambda st.resume\,\langle 0; st\rangle\ \textbf{in} \\
\qquad\quad \textbf{let}\ pid \leftarrow st.pnext\ \textbf{in} \\
\qquad\quad \textbf{let}\ q' \leftarrow st.q +\!\!+ [\langle pid; \mathsf{Ready}\ resume'\rangle]\ \textbf{in} \\
\qquad\quad \textbf{let}\ st' \leftarrow \langle st\ \textbf{with}\ q = q'; pnext = pid + 1\rangle\ \textbf{in} \\
\qquad\quad resume\,\langle pid; st'\rangle \\
\qquad \langle\!\langle \mathsf{Wait}\ pid \twoheadrightarrow resume\rangle\!\rangle \mapsto \\
\qquad\quad \textbf{let}\ resume' \leftarrow \lambda st.resume\ \langle\langle\rangle; st\rangle\ \textbf{in} \\
\qquad\quad \textbf{let}\ q' \leftarrow \textbf{if}\ \mathsf{has}\,\langle pid; st.q\rangle \\
\qquad\qquad\qquad\quad \textbf{then}\ st.q +\!\!+ [\langle st.pid; \mathsf{Blocked}\,\langle pid; resume'\rangle\rangle] \\
\qquad\qquad\qquad\quad \textbf{else}\ st.q +\!\!+ [\langle st.pid; \mathsf{Ready}\ resume'\rangle] \\
\qquad\quad \textbf{in}\ \mathsf{runNext}\,\langle st\ \textbf{with}\ q = q'\rangle \\
\qquad \langle\!\langle \mathsf{Interrupt}\ \langle\rangle \twoheadrightarrow resume\rangle\!\rangle \mapsto \\
\qquad\quad \textbf{let}\ resume' \leftarrow \lambda st.resume\,\langle\langle\rangle; st\rangle\ \textbf{in} \\
\qquad\quad \textbf{let}\ q' \leftarrow st.q +\!\!+ [\langle st.pid; \mathsf{Ready}\ resume'\rangle]\ \textbf{in} \\
\qquad\quad \mathsf{runNext}\ \langle st\ \textbf{with}\ q = q'\rangle
\end{array}
$$

The handler definition scheduler takes as input a computation that computes a value of type α whilst making use of the concurrency operations from the Co signature. In

addition it takes the initial scheduler state as input. Ultimately, the handler returns a computation that computes a list of αs, where all the Co-operations have been handled. In the definition the scheduler state is bound by the name *st*.

The **return** case is invoked when a process completes. The return value x is paired with the identifier of the currently executing process and consed onto the list *done*. Subsequently, the function runNext is invoked in order to the next ready process.

The UFork case implements the semantics for process forking. First the child process is constructed by abstracting the parameterised resumption *resume* such that it becomes an unary state-accepting function, which can be ascribed type Proc α ε. The parameterised resumption applied to the process identifier 0, which lets the receiver know that it assumes the role of child in the parent-child relationship amongst the processes. The next line retrieves the unique process identifier for the child. Afterwards, the child process is pushed on to the queue in ready state. The next line updates the scheduler state with the new queue and a new unique identifier for the next process. Finally, the parameterised resumption is applied to the child process identifier and the updated scheduler state.

The Wait case implements the synchronisation operation. The parameter *pid* is the identifier of the process that the invoking process wants to wait on. First we construct an unary state-accepting function. Then we check whether there exists a process with identifier *pid* in the queue. If there is one, then we enqueue the current process in blocked state. If no such process exists (e.g. it may already have finished), then we enqueue the current process in ready state. Finally, we invoke runNext with the scheduler state updated with the new process queue in order to run the next ready process.

The Interrupt case suspends the current process by enqueuing it in ready state, and dequeuing the next ready process.

Using this handler we can implement version 2 of the time-sharing system.

$$\begin{aligned}&\mathsf{timeshare2} : (1 \to \alpha!\mathsf{Co}) \to \mathsf{List}\,\langle \mathsf{Int}; \alpha\rangle\\&\mathsf{timeshare2}\ m \stackrel{\text{def}}{=} \mathbf{let}\ st_0 \leftarrow \langle q = []; done = []; pid = 1; pnext = 2\rangle\ \mathbf{in}\\&\qquad\qquad \mathbf{handle}^{\ddagger}\ m\,\langle\rangle\ \mathbf{with}\ \mathsf{scheduler}\ st_0\end{aligned}$$

The computation m, which may perform any of the concurrency operations, is handled by the parameterised handler scheduler. The parameterised handler definition is applied to the initial scheduler state, which has an empty process queue, an empty done list, and it assigns the first process the identifier 1, and sets up the identifier for the next process to be 2.

With UFork and Wait we can implement the *init* process, which is the initial startup process in UNIX [241]. This process remains alive until the operating system is shutdown. It is the ancestor of every process created by the operating system.

$$
\begin{array}{l}
\mathsf{init} : (1 \to 1!\varepsilon) \to 1!\{\mathsf{Co};\varepsilon\} \\
\mathsf{init}\ \mathit{main} \stackrel{\text{def}}{=} \mathbf{let}\ \mathit{pid} \leftarrow \mathbf{do}\ \mathsf{UFork}\ \langle\rangle\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{pid} = 0 \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{main}\ \langle\rangle \\
\qquad\qquad\qquad\qquad \mathbf{else\ do}\ \mathsf{Wait}\ \mathit{pid}
\end{array}
$$

We implement init as a higher-order function. It takes a main routine that will be applied when the system has been started. The function first performs UFork to duplicate itself. The child branch executes the *main* routine, whilst the parent branch waits on the child.

Now we can plug everything together.

$$
\begin{array}{l}
\mathsf{runState}\,\langle \mathsf{fs}_0; \mathsf{fileIO}\,(\lambda\langle\rangle. \\
\quad \mathsf{timeshare2}\,(\lambda\langle\rangle. \\
\quad\quad \mathsf{interruptWrite}\,(\lambda\langle\rangle. \\
\quad\quad\quad \mathsf{sessionmgr}\,\langle \mathsf{Root}; \lambda\langle\rangle. \\
\quad\quad\quad\quad \mathsf{status}\,(\lambda\langle\rangle.\mathsf{init}\,(\lambda\langle\rangle.\mathbf{let}\ \mathit{pid} \leftarrow \mathbf{do}\ \mathsf{UFork}\ \langle\rangle\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{pid} = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathsf{su}\ \mathsf{Alice}; \mathsf{ritchie}\ \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{su}\ \mathsf{Bob}; \mathbf{do}\ \mathsf{Wait}\ \mathit{pid}; \mathsf{hamlet}\ \langle\rangle)))))\rangle \\
\rightsquigarrow^{+} \langle[\langle 1;0\rangle;\langle 2;0\rangle;\langle 3;0\rangle]; \\
\quad \langle \mathit{dir} = [\langle \texttt{"stdout"};0\rangle]; \\
\quad\ \mathit{ilist} = [\langle 0;\langle \mathit{lno} = 1; \mathit{loc} = 0\rangle\rangle]; \\
\quad\ \mathit{dreg} = [\langle 0; \texttt{"UNIX is basically a simple operating system, but} \\
\qquad\qquad\qquad\quad \texttt{you have to be a genius to understand the simplicity.\textbackslash n} \\
\qquad\qquad\qquad\quad \texttt{To be, or not to be,\textbackslash nthat is the question:\textbackslash n} \\
\qquad\qquad\qquad\quad \texttt{Whether 'tis nobler in the mind to suffer\textbackslash n"}\rangle] \\
\qquad\qquad \mathit{lnext} = 1; \mathit{inext} = 1\rangle\rangle \\
\quad : \langle \mathsf{List}\,\langle \mathsf{Int}; \mathsf{Int}\rangle; \mathsf{FileSystem}\rangle
\end{array}
$$

Process number 1 is init, which forks itself to run its argument. The argument runs as process 2, which also forks itself, thus creating a process 3. Process 3 executes the child branch, which switches user to Alice and invokes the ritchie process which writes to standard out. Process 2 executes the parent branch, which switches user to Bob and

waits for the child process to complete before it invokes the routine hamlet which also writes to standard out. It is evident from looking at the file system state that the writes to standard out has not been interleaved as the contents of `"stdout"` appear in order. We can also see from the process completion list that Alice's process (pid 3) is the first to complete with status 0, and the second to complete is Bob's process (pid 2) with status 0, whilst the last process to complete is the init process (pid 1) with status 0.

Retrofitting fork In the previous program we replaced the original implementation of timeshare (Section 2.4), which handles invocations of $\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}$, by timeshare2, which handles the more general operation $\mathsf{UFork} : 1 \twoheadrightarrow \mathsf{Int}$. In practice, we may be unable to dispense of the old interface so easily, meaning we have to retain support for, say, legacy reasons. As we have seen previously we can interpret an operation in terms of another operation. Thus to retain support for Fork we simply have to insert a handler under timeshare2 which interprets Fork in terms of UFork. The operation case of this handler would be akin to the following.

$$\langle\!\langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \textbf{let}\ \mathit{pid} \leftarrow \textbf{do}\ \mathsf{UFork}\ \langle\rangle\ \textbf{in}\ \mathit{resume}\,(\mathit{pid} \neq 0)$$

The interpretation of Fork inspects the process identifier returned by the UFork to determine the role of the current process in the parent-child relationship. If the identifier is nonzero, then the process is a parent, hence Fork should return true to its caller, and otherwise it should return false, thus preserving the functionality of the legacy code.

2.8 Related work

Effect-driven concurrency In their tutorial of the Eff programming language Bauer and Pretnar [15] implement a simple lightweight thread scheduler. It is different from the schedulers presented in this section as their scheduler only uses resumptions linearly. This is achieved by making the fork operation *higher-order* such that the operation is parameterised by a computation. The computation is run under a fresh instance of the handler. On one hand this approach has the benefit of making threads cheap as it is no stack copying is necessary at runtime. On the other hand it does not guarantee that every operation is handled uniformly (when in the setting of deep handlers) as every handler in between the fork operation invocation site and the scheduler handler needs to be manually reinstalled when the computation argument is run. Nevertheless,

this is the approach to concurrency that Dolan et al. [71] have adopted for Multicore OCaml [71]. In my MSc(R) dissertation I used a similar approach to implement a cooperative version of the actor concurrency model of Links as a user-definable Links library [126]. This library was used by a prototype compiler for Links to make the runtime as lean as possible (this compiler hooked directly into the backend of the Multicore OCaml compiler in order to produce native code for effect handlers [119]). This line of work was further explored by Convent [51], who implemented various cooperative actor-based concurrency abstractions using effect handlers in the Frank programming language. Poulson [229] expanded upon this work by investigating ways to handle preemptive concurrency.

Fowler et al. [101] uses effect handlers in the setting of linearly typed fault-tolerant distributed programming. They use effect handlers to codify an exception handling mechanism, which automatically consumes linear resources. Exceptions are implemented as operations, that are handled by *cancelling* their resumptions. Cancellation is a runtime primitive that gathers and closes active resources in the computation represented by some resumption.

Dolan et al. [72] and Leijen [166] gave two widely different implementations of the async/await idiom using effect handlers. Dolan et al.'s implementation is based on higher-order operations with linearly used resumptions, whereas Leijen's implementation is based on first-order operations with multi-shot resumptions, and thus, it is close in the spirit to the schedulers we have considered in this chapter.

Continuations and operating systems The idea of using continuations to implement various facets of operating systems is not new. However, most work has focused on implementing some form of multi-tasking mechanism. Wand [271] implements a small multi-tasking kernel with support for mutual exclusion and data protection using undelimited continuations in the style of the catch operator of Scheme. Dybvig and Hieb [74] implements *engines* using call/cc in Scheme — an engine is a kind of process abstraction which support preemption. An engine runs a computation on some time budget. If computation exceeds the allotted time budget, then it is interrupted. They represent engines as reified continuations and use the macro system of Scheme to insert clock ticks at appropriate places in the code. Kiselyov and Shan [149] develop a small fault-tolerant operating system with multi-tasking support and a file system using delimited continuations. Their file system is considerably more sophisticated than the one we implemented in this chapter as it supports transactional storage, meaning user

processes can roll back actions such as file deletion and file update.

Resumption monad The resumption monad is both a semantic and programmatic abstraction for interleaving computation. Papspyrou [211] applies a resumption monad transformer to construct semantic models of concurrent computation. A resumption monad transformer, i.e. a monad T that transforms an arbitrary monad M to a new monad $T\ M$ with commands for interrupting computation. Harrison [112] demonstrates the resumption monad as a practical programming abstraction by implementing a small multi-tasking kernel. Harrison implements two variations of the resumption monad: basic and reactive. The basic resumption monad is a closed environment for interleaving different strands of computations. It is closed in the sense that strands of computation cannot interact with the ambient context of their environment. The reactive resumption monad makes the environment open by essentially registering a callback with an interruption action. This provides a way to model system calls.

The origins of the (semantic) resumption monad can be traced back to at least Moggi [202], who described a monad for modelling the interleaving semantics of Milner's *calculus of communicating systems* [197].

The usage of *resumption* in the name has a slightly different meaning than the term 'resumption' we have been using throughout this chapter. We have used 'resumption' to mean delimited continuation. In the setting of the resumption monad it has a precise domain-theoretic meaning. It is derived from Plotkin's domain of resumptions, which in turn is derived from Milner's domain of processes [197, 221].

Chapter 3

Calculi for effect handler oriented programming

In this chapter we formalise the language that was introduced informally in the previous chapter. In fact, we will formally introduce the language as several core calculi: a base calculus λ_{b}, which does not have effect handlers; an extension of λ_{b} with deep handlers λ_{h}; another extension with shallow handlers $\lambda_{\mathsf{h}^\dagger}$; and a final extension with parameterised handlers $\lambda_{\mathsf{h}^\ddagger}$. The calculi are based on CoreLinks by Lindley and Cheney [173], which distils the essence of the functional multi-tier web-programming language Links [53]. Links belongs to the ML-family [200] of programming languages as it features typical characteristics of ML languages such as a static type system supporting parametric polymorphism with type inference (in fact Links supports first-class polymorphism), and its evaluation semantics is strict. However, Links differentiates itself from the rest of the ML-family by making crucial use of *row polymorphism* to support extensible records, variants, and tracking of computational effects. Thus Links has a rather strong emphasis on structural types rather than nominal types.

CoreLinks captures all of these properties of Links. Our calculus λ_{b} differs in several aspects from CoreLinks. For example, the underlying formalism of CoreLinks is call-by-value, whilst the formalism of λ_{b} is *fine-grain call-by-value* [170], which shares similarities with A-normal form (ANF) [91] as it syntactically distinguishes between value and computation terms by mandating every intermediate computation being named. However unlike ANF, fine-grain call-by-value remains closed under β-reduction. The reason for choosing fine-grain call-by-value as our formalism is entirely due to convenience. Fine-grain call-by-value is a convenient formalism for working with continuations. Another point of difference between CoreLinks and λ_{b} is that the

former models the integrated database query sublanguage of Links. We do not consider the query sublanguage at all, and instead our three extensions λ_{h}, $\lambda_{\mathsf{h}^\dagger}$, and $\lambda_{\mathsf{h}^\ddagger}$ focus entirely on modelling the interaction and programming with computational effects.

Chapter outline

Section 3.1 introduces the base calculus λ_{b}, which makes crucial use of *rows* to support variant, record, and effect polymorphism.

Section 3.2 extends λ_{b} with *deep effect handlers*, resulting in the calculus λ_{h}.

Section 3.3 adds *shallow effect handlers* to base calculus, resulting in the calculus $\lambda_{\mathsf{h}^\dagger}$.

Section 3.4 presents the final variation of the base calculus, as in this section we extend the base calculus with *parameterised effect handlers*, yielding the calculus $\lambda_{\mathsf{h}^\ddagger}$.

Section 3.5 discusses related work.

Relation to prior work The deep and shallow handler calculi that are introduced in Section 3.2, Section 3.3, and Section 3.4 are adapted with minor syntactic changes from the following work.

i Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016

ii Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018

iii Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020

3.1 A language based on rows

At glance λ_{b} is an intrinsically typed language supporting type and effect polymorphism. The key characteristic of λ_{b} is that it uses Rémy-style *rows* to support variant, record, and effect polymorphism at the same time [237].

We begin by presenting the syntax of kinds, types, and terms in Section 3.1.1. Afterwards we present the static semantics in Section 3.1.2, before we present the dynamic semantics in Section 3.1.3. As a convention, we always work up to α-conversion [45]

Value types	$A, B \in \mathrm{VType} ::= A \to C \mid \forall \alpha^K.C \mid \langle R \rangle \mid [R] \mid \alpha$
Computation types	$C, D \in \mathrm{CType} ::= A!E$
Effect types	$E \in \mathrm{Effect} ::= \{R\}$
Row types	$R \in \mathrm{Row} ::= \ell : P; R \mid \rho \mid \cdot$
Presence types	$P \in \mathrm{Presence} ::= \mathsf{Pre}(A) \mid \mathsf{Abs} \mid \theta$
Types	$T \in \mathrm{Type} ::= A \mid C \mid E \mid R \mid P$
Kinds	$K \in \mathrm{Kind} ::= \mathsf{Type} \mid \mathsf{Comp} \mid \mathsf{Effect} \mid \mathsf{Row}_{\mathcal{L}} \mid \mathsf{Presence}$
Label sets	$\mathcal{L} \in \mathrm{Label} ::= \emptyset \mid \{\ell\} \uplus \mathcal{L}$
Type environments	$\Gamma \in \mathrm{TyEnv} ::= \cdot \mid \Gamma, x : A$
Kind environments	$\Delta \in \mathrm{KindEnv} ::= \cdot \mid \Delta, \alpha : K$

Figure 3.1: Syntax of types, kinds, and their environments.

---

of types and terms. Following Pierce [215] we omit cases in definitions that deal only with the bureaucracy of renaming. For any transformation $[\![-]\!]$ on a term M, or type, we write $[\![M]\!] \overset{\text{α-def}}{\simeq} M'$ to mean that M' is the result of transforming M where implicit renaming may have occurred.

3.1.1 Syntax of kinds, types, and terms

The syntax and semantics of types in λ_{b} are based on those of System F [109], whilst the term syntax is a variation of ML term syntax in a-normal form. We use the notation $A[B/\alpha]$ to mean the capture-avoiding substitution of the type B for the type variable α in the type A. Similarly, we write $M[V/x]$ to mean the capture-avoiding substitution of the value V for the term variable x in the computation term M.

Types and their kinds

The types are divided into several distinct syntactic categories which are given in Figure 3.1 along with the syntax of kinds and environments.

Value types We distinguish between values and computations at the level of types. Value types comprise the function type $A \to C$, which maps values of type A to compu-

tations of type C; the polymorphic type $\forall\alpha^K.C$ is parameterised by a type variable α of kind K; and the record type $\langle R\rangle$ represents records with fields constrained by row R. Dually, the variant type $[R]$ represents tagged sums constrained by row R.

Computation types and effect types The computation type $A!E$ is given by a value type A and an effect type E, which specifies the effectful operations a computation inhabiting this type may perform. An effect type $E = \{R\}$ is constrained by row R.

Row types Row types play a pivotal role in our type system as effect, record, and variant types are uniformly given by row types. A *row type* describes a collection of distinct labels, each annotated by a presence type. A presence type indicates whether a label is *present* with type A ($\mathsf{Pre}(A)$), *absent* (Abs) or *polymorphic* in its presence (θ). For example, the effect row $\{\mathsf{Read} : \mathsf{Pre}(\mathsf{Int}), \mathsf{Write} : \mathsf{Abs}, \cdot\}$ denotes a read-only context in which the operation label Read may occur to access some integer value, whilst the operation label Write cannot appear.

Row types are either *closed* or *open*. A closed row type ends in $\cdot$, whilst an open row type ends with a *row variable* ρ (in an effect row we usually use ε rather than ρ and refer to it as an *effect variable*). The example effect row above is closed, an open variation of it ends in an effect variable ε, i.e. $\{\mathsf{Read} : \mathsf{Pre}(\mathsf{Int}), \mathsf{Write} : \mathsf{Abs}, \varepsilon\}$. The row variable in an open row type can be instantiated with additional labels subject to the restriction that each label may only occur at most once (we enforce this restriction at the level of kinds). We identify rows up to the reordering of labels as follows.

$$\textsf{R-Closed}\ \frac{}{\cdot \equiv_{\text{row}} \cdot} \qquad \textsf{R-Open}\ \frac{}{\rho \equiv_{\text{row}} \rho'} \qquad \textsf{R-Head}\ \frac{R \equiv_{\text{row}} R'}{\ell : P; R \equiv_{\text{row}} \ell : P; R'}$$

$$\textsf{R-Swap}\ \frac{R \equiv_{\text{row}} R'}{\ell : P; \ell' : P'; R \equiv_{\text{row}} \ell' : P'; \ell : P; R'}$$

The R-Closed rule states that the closed marker $\cdot$ is equivalent to itself, similarly the R-Open rule states that any two row variables are equivalent if and only if they have the same syntactic name. The R-Head rule compares the head of two given rows and inductively compares their tails. The R-Swap rule permits reordering of labels. We assume structural equality on labels. The R-Head rule

$$\textsf{K-TyVar}\ \frac{}{\Delta, \alpha : K \vdash \alpha : K} \qquad \textsf{K-Comp}\ \frac{\Delta \vdash A : \textsf{Type} \qquad \Delta \vdash E : \textsf{Effect}}{\Delta \vdash A!E : \textsf{Comp}}$$

$$\textsf{K-Fun}\ \frac{\Delta \vdash A : \textsf{Type} \qquad \Delta \vdash C : \textsf{Comp}}{\Delta \vdash A \to C : \textsf{Type}} \qquad \textsf{K-Forall}\ \frac{\Delta, \alpha : K \vdash C : \textsf{Comp}}{\Delta \vdash \forall \alpha^K . C : \textsf{Type}} \qquad \textsf{K-Record}\ \frac{\Delta \vdash R : \textsf{Row}_\emptyset}{\Delta \vdash \langle R \rangle : \textsf{Type}}$$

$$\textsf{K-Variant}\ \frac{\Delta \vdash R : \textsf{Row}_\emptyset}{\Delta \vdash [R] : \textsf{Type}} \qquad \textsf{K-Effect}\ \frac{\Delta \vdash R : \textsf{Row}_\emptyset}{\Delta \vdash \{R\} : \textsf{Effect}} \qquad \textsf{K-Present}\ \frac{\Delta \vdash A : \textsf{Type}}{\Delta \vdash \textsf{Pre}(A) : \textsf{Presence}}$$

$$\textsf{K-Absent}\ \frac{}{\Delta \vdash \textsf{Abs} : \textsf{Presence}} \qquad \textsf{K-EmptyRow}\ \frac{}{\Delta \vdash \cdot : \textsf{Row}_{\mathcal{L}}} \qquad \textsf{K-ExtendRow}\ \frac{\Delta \vdash P : \textsf{Presence} \qquad \Delta \vdash R : \textsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : P; R : \textsf{Row}_{\mathcal{L}}}$$

Figure 3.2: Kinding rules

The standard zero and unit types are definable using rows. We define the zero type as the empty, closed variant $0 \stackrel{\text{def}}{=} [\cdot]$. Dually, the unit type is defined as the empty, closed record type, i.e. $1 \stackrel{\text{def}}{=} \langle \cdot \rangle$.

For brevity, we shall often write $\ell : A$ to mean $\ell : \textsf{Pre}(A)$.

Kinds The kinds classify the different categories of types. The Type kind classifies value types, Presence classifies presence annotations, Comp classifies computation types, Effect classifies effect types, and lastly $\textsf{Row}_{\mathcal{L}}$ classifies rows. The formation rules for kinds are given in Figure 3.2. The kinding judgement $\Delta \vdash T : K$ states that type T has kind K in kind environment Δ. The row kind is annotated by a set of labels $\mathcal{L}$. We use this set to track the labels of a given row type to ensure uniqueness amongst labels in each row type. For example, the kinding rule K-ExtendRow uses this set to constrain which labels may be mentioned in the tail of R.

Environments Kind and type environments are right-extended sequences of bindings. A kind environment binds type variables to their kinds, whilst a type environment binds term variables to their types.

Types and their inhabitants We now have the basic vocabulary to construct types in λ_b. For instance, the signature of the standard polymorphic identity function is

$$\forall \alpha^{\mathsf{Type}}.\alpha \to \alpha!\emptyset.$$

Modulo the empty effect signature, this type is akin to the type one would give for the identity function in System F [109, 238], and thus we can use standard techniques from parametricity [266] to reason about inhabitants of this signature. However, in our system we can give an even more general type to the identity function:

$$\forall \alpha^{\mathsf{Type}}, \varepsilon^{\mathsf{Row}_\emptyset}.\alpha \to \alpha!\{\varepsilon\}.$$

This type is polymorphic in its effect signature as signified by the singleton open effect row $\{\varepsilon\}$, meaning it may be used in an effectful context. By contrast, the former type may only be used in a strictly pure context, i.e. the effect-free context.

We can use the effect system to give precise types to effectful computations. For example, we can give the signature of some polymorphic computation that may only be run in a read-only context

$$\forall \alpha^{\mathsf{Type}}, \varepsilon^{\mathsf{Row}_{\{\mathsf{Read},\mathsf{Write}\}}}.\alpha!\{\mathsf{Read} : \mathsf{Int}; \mathsf{Write} : \mathsf{Abs}; \varepsilon\}.$$

The effect row comprise a nullary Read operation returning some integer and an absent operation Write. The absence of Write means that the computation cannot run in a context that admits a present Write. It can, however, run in a context that admits a presence polymorphic $\mathsf{Write} : \theta$ as the presence variable θ may instantiated to Abs. An inhabitant of this type may be run in larger effect contexts, i.e. contexts that admit more operations, because the row ends in an effect variable.

The type and effect system is also precise about how a higher-order function may use its function arguments. For example consider the signature of a map-operation over some datatype such as $\mathsf{Option}\ \alpha^{\mathsf{Type}} \stackrel{\text{def}}{=} [\mathsf{None}; \mathsf{Some} : \alpha; \cdot]$

$$\forall \alpha^{\mathsf{Type}}, \beta^{\mathsf{Type}}, \varepsilon^{\mathsf{Row}_\emptyset}.\langle \alpha \to \beta!\{\varepsilon\}; \mathsf{Option}\ \alpha; \cdot\rangle \to \mathsf{Option}\ \beta!\{\varepsilon\}.$$

The first argument is the function that will be applied to the data carried by second argument. Note that the two effect rows are identical and share the same effect variable ε, it is thus evident that an inhabitant of this type can only perform whatever effects its first argument is allowed to perform.

Higher-order functions may also transform their function arguments, e.g. modify their effect rows. The following is the signature of a higher-order function which restricts its argument's effect context

$$\forall \alpha^{\mathsf{Type}}, \varepsilon^{\mathsf{Row}_{\{\mathsf{Read}\}}}, \varepsilon'^{\mathsf{Row}_\emptyset}.(1 \to \alpha!\{\mathsf{Read} : \mathsf{Int}; \varepsilon\}) \to (1 \to \alpha!\{\mathsf{Read} : \mathsf{Abs}; \varepsilon\})!\{\varepsilon'\}.$$

$$
\begin{array}{lll}
\textsf{Variables} & & x \in \mathrm{Var} \\
\textsf{Values} & V, W \in \mathrm{Val} ::= & x \mid \lambda x^{A}.M \mid \mathbf{rec}\, f^{A \to C}\, x.M \mid \Lambda \alpha^{K}.M \\
& & \mid \ \langle\rangle \mid \langle \ell = V; W \rangle \mid (\ell\ V)^{R} \\
\\
\textsf{Computations} & M, N \in \mathrm{Comp} ::= & V\, W \mid V\, T \\
& & \mid \ \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \\
& & \mid \ \mathbf{case}\ V \{ \ell\ x \mapsto M; y \mapsto N \} \mid \mathbf{absurd}^{C}\ V \\
& & \mid \ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\
\textsf{Terms} & t \in \mathrm{Term} ::= & x \mid V \mid M
\end{array}
$$

Figure 3.3: Term syntax of λ_{b}.

The function argument is allowed to perform a Read operation, whilst the returned function cannot. Moreover, the two functions share the same effect variable ε. Like the option-map signature above, an inhabitant of this type performs no effects of its own as the (right-most) effect row is a singleton row containing a distinct effect variable ε'.

Syntactic sugar Explicitly writing down all of the kinding and type annotations is a bit on the heavy side. In order to simplify the notation of our future examples we are going to adopt a few conventions. First, we shall not write kind annotations, when the kinds can unambiguously be inferred from context. Second, we do not write quantifiers in prenex position. Type variables that appear unbound in a signature are implicitly understood be bound at the outermost level of the type (this convention is commonly used by practical programming language, e.g. SML [200] and Haskell [138]). Third, we shall adopt the convention that the row types for closed records and variants are implicitly understood to end in a $\cdot$, whereas for effect rows we shall adopt the opposite convention that an effect row is implicitly understood to be open and ending in a fresh ε unless it ends in an explicit $\cdot$. In Section 3.2.5 we will elaborate more on the syntactic sugar for effects. The rationale for these conventions is that they align with a ML programmer's intuition for monomorphic record and variant types, and in this dissertation records and variants will often be monomorphic. Conversely, effect rows will most often be open.

Terms

The syntax for terms is given in Figure 3.3. We assume a countably infinite set of names Var from which we draw fresh variable names. We shall typically denote term variables by x, y, or z. The syntax partitions terms into values and computations. Value terms comprise variables (x), lambda abstraction ($\lambda x^A.M$), recursive abstraction ($\mathbf{rec}\, f^{A\to C}\, x.M$), type abstraction ($\Lambda\alpha^K.M$), and the introduction forms for records and variants. Records are introduced using the empty record ($\langle\rangle$) and record extension ($\langle \ell = V; W\rangle$), whilst variants are introduced using injection ($(\ell\ V)^R$), which injects a field with label ℓ and value V into a row whose type is R. We include the row type annotation in to support bottom-up type reconstruction.

All elimination forms are computation terms. Abstraction and type abstraction are eliminated using application ($V\,W$) and type application ($V\,A$) respectively. The record eliminator ($\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N$) splits a record V into x, the value associated with ℓ, and y, the rest of the record. Non-empty variants are eliminated using the case construct ($\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}$), which evaluates the computation M if the tag of V matches ℓ. Otherwise it falls through to y and evaluates N. The elimination form for empty variants is ($\mathbf{absurd}^C\ V$). There is one computation introduction form, namely, the trivial computation ($\mathbf{return}\ V$) which returns value V. Its elimination form is the expression ($\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N$) which evaluates M and binds the result value to x in N.

As our calculus is intrinsically typed, we annotate terms with type or kind information (term abstraction, type abstraction, injection, operations, and empty cases). However, we shall omit these annotations whenever they are clear from context.

Tail recursion In practice implementations of functional programming languages tend to be tail-recursive in order to enable unbounded iteration. Otherwise nested (repeated) function calls would quickly run out of stack space on a conventional computer. Intuitively, tail-recursion permits an already allocated stack frame for some on-going function call to be reused by a nested function call, provided that this nested call is the last thing to occur before returning from the on-going function call. A special case is when the nested function call is a fresh invocation of the on-going function call, i.e. a self-reference. In this case the nested function call is known as a *tail recursive call*, otherwise it is simply known as a *tail call*. Thus the qualifier "tail-recursive" may be somewhat confusing as for an implementation to be tail-recursive it must support recycling of stack frames for tail calls; it is not sufficient to support tail recursive calls.

Any decent implementation of Standard ML [200], OCaml [169], or Scheme [253]

will be tail-recursive. I deliberately say implementation rather than specification, because it is often the case that the specification or the user manual do not explicitly require a suitable implementation to be tail-recursive; in fact of the three languages just mentioned only Scheme explicitly mandates an implementation to be tail-recursive [253].

Tail calls will become important in Chapter 4 when we will discuss continuation passing style as an implementation technique for effect handlers, as tail calls happen to be ubiquitous in continuation passing style. Therefore let us formally characterise tail calls. For our purposes, the most robust characterisation is a syntactic characterisation, as opposed to a semantic characterisation, because in the presence of control effects (which we will add in Section 3.2) it surprisingly tricky to describe tail calls in terms of control flow such as "the last thing to occur before returning from the enclosing function" as a function may return multiple times. In particular, the effects of a function may be replayed several times.

For this reason we will adapt a syntactic characterisation of tail calls due to Clinger [48]. First, we define what it means for a computation to syntactically *appear in tail position*.

Definition 3.1 (Tail position)**.** Tail position is defined for computation terms as follows.

- The body M of a λ-abstraction $(\lambda x.M)$ appears in tail position.
- The body M of a Λ-abstraction $(\Lambda\alpha.M)$ appears in tail position.
- If **case** $V\ \{\ell\ x \mapsto M; y \mapsto N\}$ appears in tail position, then both M and N appear in tail positions.
- If **let** $\langle \ell = x; y \rangle = V$ **in** N appears in tail position, then N is in tail position.
- If **let** $x \leftarrow M$ **in** N appears in tail position, then N appear in tail position.
- Nothing else appears in tail position.

Definition 3.2 (Tail call)**.** An application term $V\,W$ is said to be a tail call if it appears in tail position.

3.1.2 Typing rules

The typing rules are divided into value and computation term typing rules.

Values

T-Var
$$\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash x : A}$$

T-Lam
$$\frac{\Delta;\Gamma, x : A \vdash M : C}{\Delta;\Gamma \vdash \lambda x^{A}.M : A \to C}$$

T-Rec
$$\frac{\Delta;\Gamma, f : A \to C, x : A \vdash M : C}{\Delta;\Gamma \vdash (\mathbf{rec}\, f^{A \to C}\, x.M) : A \to C}$$

T-PolyLam
$$\frac{\Delta, \alpha : K;\Gamma \vdash M : C \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Delta;\Gamma \vdash \Lambda\alpha^{K}.M : \forall\alpha^{K}.C}$$

T-Unit
$$\frac{}{\Delta;\Gamma \vdash \langle\rangle : 1}$$

T-Extend
$$\frac{\Delta;\Gamma \vdash V : A \qquad \Delta;\Gamma \vdash W : \langle \ell : \mathsf{Abs}; R\rangle}{\Delta;\Gamma \vdash \langle \ell = V; W\rangle : \langle \ell : \mathsf{Pre}(A); R\rangle}$$

T-Inject
$$\frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash (\ell\ V)^{R} : [\ell : \mathsf{Pre}(A); R]}$$

Computations

T-App
$$\frac{\Delta;\Gamma \vdash V : A \to C \qquad \Delta;\Gamma \vdash W : A}{\Delta;\Gamma \vdash V\,W : C}$$

T-PolyApp
$$\frac{\Delta;\Gamma \vdash V : \forall\alpha^{K}.C \qquad \Delta \vdash A : K}{\Delta;\Gamma \vdash VA : C[A/\alpha]}$$

T-Split
$$\frac{\Delta;\Gamma \vdash V : \langle \ell : \mathsf{Pre}(A); R\rangle \qquad \Delta;\Gamma, x : A, y : \langle \ell : \mathsf{Abs}; R\rangle \vdash N : C}{\Delta;\Gamma \vdash \mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N : C}$$

T-Case
$$\frac{\Delta;\Gamma \vdash V : [\ell : \mathsf{Pre}(A); R] \qquad \Delta;\Gamma, x : A \vdash M : C \qquad \Delta;\Gamma, y : [\ell : \mathsf{Abs}; R] \vdash N : C}{\Delta;\Gamma \vdash \mathbf{case}\ V\{\ell\, x \mapsto M; y \mapsto N\} : C}$$

T-Absurd
$$\frac{\Delta;\Gamma \vdash V : []}{\Delta;\Gamma \vdash \mathbf{absurd}^{C}\ V : C}$$

T-Return
$$\frac{\Delta;\Gamma \vdash V : A}{\Delta;\Gamma \vdash \mathbf{return}\ V : A!E}$$

T-Let
$$\frac{\Delta;\Gamma \vdash M : A!E \qquad \Delta;\Gamma, x : A \vdash N : B!E}{\Delta;\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : B!E}$$

Figure 3.4: Typing rules

Typing values The rule T-Var states that a term variable x has type A if x is mapped to the type A in the environment Γ. The rules T-Lam and T-Rec both concern term abstraction, where the latter subsumes the former as it states that a recursive term abstraction has type $A \to C$ if the body has type C assuming the environment Γ is extended with the binder $f : A \to C$ and the parameter $x : A$. The T-Lam rule works similarly except for the self binder f. The next rule T-PolyLam states that a type abstraction $(\Lambda\alpha.M)$ has type $\forall\alpha.C$ if the computation M has type C assuming $\alpha : K$ and α does not appear in the free type variables (FTV) of current type environment Γ. The T-Unit rule provides the basis for all records as it simply states that the empty record has type unit. The T-Extend rule handles record extension. Supposing we wish to extend some record $\langle W \rangle$ with $\ell = V$, that is $\langle \ell = V; W \rangle$. This extension has type $\langle \ell : \mathsf{Pre}(A); R \rangle$ if and only if V is well-typed and we can ascribe $W : \langle \ell : \mathsf{Abs}; R \rangle$. Since $\langle \ell : \mathsf{Abs}; R \rangle$ must be well-kinded with respect to Δ, the label ℓ cannot be mentioned in W, thus ℓ cannot occur more than once in the record. Similarly, the dual rule T-Inject states that the injection $(\ell\ V)^R$ has type $[\ell : \mathsf{Pre}(A); R]$ if the payload V is well-typed. The implicit well-kindedness condition on R ensures that ℓ cannot be injected twice. To illustrate how the kinding system prevents duplicated labels consider the following ill-typed example

$$(\mathsf{S}\ \langle\rangle)^{\mathsf{S}:1} : [\mathsf{S} : 1; \mathsf{S} : 1].$$

Typing fails because the resulting row type is ill-kinded by the K-ExtendRow-rule:

$$\dfrac{\dfrac{\dfrac{\vdash \mathsf{Pre}(1) : \mathsf{Presence} \qquad \vdash \cdot : \mathsf{Row}_{\{\mathsf{S}\} \uplus \{\mathsf{S}\}}}{\vdash \mathsf{S} : \mathsf{Pre}(1); \cdot : \mathsf{Row}_{\emptyset \uplus \{\mathsf{S}\}}}\ \text{K-ExtendRow} \qquad \vdash \mathsf{Pre}(1) : \mathsf{Presence}}{\vdash \mathsf{S} : \mathsf{Pre}(1); \mathsf{S} : \mathsf{Pre}(1); \cdot : \mathsf{Row}_{\emptyset}}\ \text{K-ExtendRow}}{\vdash [\mathsf{S} : \mathsf{Pre}(1); \mathsf{S} : \mathsf{Pre}(1); \cdot] : \mathsf{Type}}\ \text{K-Variant}$$

The two sets $\{\mathsf{S}\}$ and $\{\mathsf{S}\}$ are clearly not disjoint, thus the second premise of the last application of K-ExtendRow cannot be satisfied.

Typing computations The T-App rule states that an application $V\,W$ has computation type C if the function-term V has type $A \to C$ and the argument term W has type A, that is both the argument type and the domain type of the abstractor agree. The type application rule T-PolyApp tells us that a type application $V\,A$ is well-typed whenever the abstractor term V has the polymorphic type $\forall\alpha^K.C$ and the type A has kind K. This rule makes use of type substitution. The T-Split rule handles typing of record deconstructing. When splitting a record term V on some label ℓ binding it to x and

$$
\begin{array}{lrcll}
\textsf{S-App} & (\lambda x^{A}.M)V & \rightsquigarrow & M[V/x] & \\
\textsf{S-Rec} & (\mathbf{rec}\, f^{A\to C}\, x.M)\, V & \rightsquigarrow & M[(\mathbf{rec}\, f^{A\to C}\, x.M)/f, V/x] & \\
\textsf{S-TyApp} & (\Lambda\alpha^{K}.M)A & \rightsquigarrow & M[A/\alpha] & \\
\textsf{S-Split} & \mathbf{let}\ \langle \ell = x; y\rangle = \langle \ell = V; W\rangle\ \mathbf{in}\ N & \rightsquigarrow & N[V/x, W/y] & \\
\textsf{S-Case}_1 & \mathbf{case}\ (\ell\ V)^{R}\{\ell\, x \mapsto M; y \mapsto N\} & \rightsquigarrow & M[V/x] & \\
\textsf{S-Case}_2 & \mathbf{case}\ (\ell\ V)^{R}\{\ell'\, x \mapsto M; y \mapsto N\} & \rightsquigarrow & N[(\ell\ V)^{R}/y], & \text{if } \ell \neq \ell' \\
\textsf{S-Let} & \mathbf{let}\, x \leftarrow \mathbf{return}\, V\ \mathbf{in}\, N & \rightsquigarrow & N[V/x] & \\
\textsf{S-Lift} & \mathcal{E}[M] & \rightsquigarrow & \mathcal{E}[N], & \text{if } M \rightsquigarrow N
\end{array}
$$

$$
\textsf{Evaluation contexts} \quad \mathcal{E} \in \mathrm{Cont} ::= [\,] \mid \mathbf{let}\, x \leftarrow \mathcal{E}\ \mathbf{in}\, N
$$

Figure 3.5: Contextual small-step semantics

---

the remainder to y. The label we wish to split on must be present with some type A, hence we require that $V : \langle \ell : \mathsf{Pre}(A); R\rangle$. This restriction prohibits us for splitting on an absent or presence polymorphic label. The continuation of the splitting, N, must then have some computation type C subject to the following restriction: $N : C$ must be well-typed under the additional assumptions $x : A$ and $y : \langle \ell : \mathsf{Abs}; R\rangle$, statically ensuring that it is not possible to split on ℓ again in the continuation N. The T-Case rule is similar, but has two possible continuations: the success continuation, M, and the fall-through continuation, N. The label being matched must be present with some type A in the type of the scrutinee, thus we require $V : [\ell : \mathsf{Pre}(A); R]$. The success continuation has some computation C under the assumption that the binder $x : A$, whilst the fall-through continuation also has type C it is subject to the restriction that the binder $y : [\ell : \mathsf{Abs}; R]$ which statically enforces that no subsequent case split in N can match on ℓ. The T-Absurd states that we can ascribe any computation type to the term **absurd** V if V has the empty type $[]$. The trivial computation term is typed by the T-Return rule, which says that **return** V has computation type $A!E$ if the value V has type A. The T-Let rule types let bindings. The computation being bound, M, must have computation type $A!E$, whilst the continuation, N, must have computation C subject to the additional assumption that the binder $x : A$.

3.1.3 Dynamic semantics

The dynamic semantics of λ_{b} (and its extensions) use a Felleisen [79]-style contextual small-step semantics, since in conjunction with fine-grain call-by-value (FGCBV), it yields a considerably simpler semantics than the traditional structural operational semantics (SOS) [223]. The reason being that only the rule for let bindings admits a continuation wheres in ordinary call-by-value SOS each congruence rule admits a continuation. The simpler semantics comes at the expense of a more verbose syntax, which is not a concern as one can readily convert between fine-grain call-by-value and ordinary call-by-value.

Reduction semantics The reduction relation $\rightsquigarrow \subseteq \mathrm{Comp} \times \mathrm{Comp}$ relates a computation term to another if the former can reduce to the latter in a single step. Figure 3.5 depicts the reduction rules. The application rules S-App and S-TyApp eliminates a lambda and type abstraction, respectively, by substituting the argument for the parameter in their body computation M. Record splitting is handled by the S-Split rule: splitting on some label ℓ binds the payload V to x and the remainder W to y in the continuation N. Disjunctive case splitting is handled by the two rules $\mathsf{S\text{-}Case}_1$ and $\mathsf{S\text{-}Case}_2$. The former rule handles the success case, when the scrutinee's tag ℓ matches the tag of the success clause, thus binds the payload V to x and proceeds to evaluate the continuation M. The latter rule handles the fall-through case, here the scrutinee gets bounds to y and evaluation proceeds with the continuation N. The S-Let rule eliminates a trivial computation term **return** V by substituting V for x in the continuation N.

Evaluation contexts Recall from Section 3.1.1, Figure 3.3 that the syntax of let bindings allows a general computation term M to occur on the right hand side of the binding, i.e. **let** $x \leftarrow M$ **in** N. Thus we are seemingly stuck in the general case, as the S-Let rule only applies if the right hand side is a trivial computation. However, it is at this stage we make use of the notion of *evaluation contexts* due to Felleisen [79]. An evaluation context is syntactic construction which decompose the dynamic semantics into a set of base rules (i.e. the rules presented thus far) and an inductive rule, which enables us to focus on a particular computation term, M, in some larger context, $\mathcal{E}$, and reduce it in the said context to another computation N if M reduces outside out the context to that particular N. In our formalism, we call this rule S-Lift. Evaluation contexts are generated from the empty context ($[\,]$) and let expressions (**let** $x \leftarrow \mathcal{E}$ **in** N).

The choices of using fine-grain call-by-value and evaluation contexts may seem

odd, if not arbitrary at this point; the reader may wonder with good reason why we elect to use fine-grain call-by-value over ordinary call-by-value. In Sections 3.2–3.4 we will reap the benefits from our design choices, as we shall see that the combination of fine-grain call-by-value and evaluation contexts provide the basis for a convenient, simple semantic framework for working with continuations.

Syntactic sugar We will adopt a few conventions to make the notation more convenient for writing out examples. First, we elide type annotations when they are clear from the context. We will often write code in direct-style assuming the standard left-to-right call-by-value elaboration into fine-grain call-by-value [91, 203]. For example, the expression $f\,(h\,w) + g\,\langle\rangle$ is syntactic sugar for:

$$\textbf{let}\ x \leftarrow h\,w\ \textbf{in let}\ y \leftarrow f\,x\ \textbf{in let}\ z \leftarrow g\,\langle\rangle\ \textbf{in}\ y + z$$

We define sequencing of computations in the standard way.

$$M; N \stackrel{\text{def}}{=} \textbf{let}\ x \leftarrow M\ \textbf{in}\ N, \quad \text{where } x \notin FV(N)$$

We make use of standard syntactic sugar for pattern matching. For instance, we write

$$\lambda\langle\rangle.M \stackrel{\text{def}}{=} \lambda x^{1}.M, \quad \text{where } x \notin FV(M)$$

for suspended computations. We encode booleans using variants:

$$\mathsf{Bool} \stackrel{\text{def}}{=} [\mathsf{True} : 1; \mathsf{False} : 1] \qquad \mathsf{true} \stackrel{\text{def}}{=} \mathsf{True}\,\langle\rangle \qquad \mathsf{false} \stackrel{\text{def}}{=} \mathsf{False}\,\langle\rangle$$

$$\textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N \stackrel{\text{def}}{=} \textbf{case}\ V\ \{\mathsf{True}\ \langle\rangle \mapsto M; \mathsf{False}\ \langle\rangle \mapsto N\}$$

3.1.4 Metatheoretic properties of λ_{b}

Thus far we have defined the syntax, static semantics, and dynamic semantics of λ_{b}. In this section, we state and prove some customary metatheoretic properties about λ_{b}.

The reduction semantics satisfy a *unique decomposition* property, which guarantees the existence and uniqueness of complete decomposition for arbitrary computation terms into evaluation contexts.

Lemma 3.3 (Unique decomposition)**.** *For any computation $M \in Comp$ it holds that M is either stuck or there exists a unique evaluation context $\mathcal{E} \in Cont$ and a redex $N \in Comp$ such that $M = \mathcal{E}[N]$.*

Proof. By structural induction on M. □

The calculus satisfies the standard *progress* property, which states that *every* closed computation term either reduces to a trivial computation term **return** V for some value V, or there exists some N such that $M \rightsquigarrow N$.

Definition 3.4 (Computation normal form)**.** A computation $M \in \text{Comp}$ is said to be *normal* if it is of the form **return** V for some value $V \in \text{Val}$.

We write $\rightsquigarrow^*$ for the reflexive and transitive closure of the reduction relation $\rightsquigarrow$.

Theorem 3.5 (Progress)**.** *Suppose* $\vdash M : C$*, then* M *is normal or there exists* $\vdash N : C$ *such that* $M \rightsquigarrow^* N$.

Proof. By induction on the typing derivations. □

The calculus also satisfies the *subject reduction* property, which states that if some computation M is well typed and reduces to some other computation M', then M' is also well typed.

Theorem 3.6 (Subject reduction)**.** *Suppose* $\Delta; \Gamma \vdash M : C$ *and* $M \rightsquigarrow N$*, then* $\Delta; \Gamma \vdash N : C$.

Proof. By induction on the typing derivations. □

3.2 Deep handling of effects

Deep effect handlers are the 'classic' effect handlers in the sense they were the kind of handlers originally introduced by Plotkin and Pretnar [227]. A point that is worthwhile to make is that deep handlers do not depend on the existence of an explicit recursion operator, rather, they come with their own structured recursion scheme. In fact, deep handlers [227, 230] are defined by folds (specifically *catamorphisms* [194]) over computation trees, meaning they provide a uniform semantics to the handled operations of a given computation. In contrast, shallow handlers are defined as case-splits over computation trees, and thus, allow a nonuniform semantics to be given to operations. We will discuss this point in more detail in Section 3.3.

In this section we augment λ_{b} with deep handlers to yield the core calculus λ_{h}.

3.2.1 Performing effectful operations

An effectful operation is a purely syntactic construction, which has no predefined dynamic semantics. In our calculus effectful operations are a computational phenomenon,

and thus, their introduction form is a computation term. To type operation we augment the syntactic category of value types with a new arrow.

$$\begin{array}{ll} \text{Value types} & A, B \in \text{VType} ::= \cdots \mid A \twoheadrightarrow B \\ \text{Computations} & M, N \in \text{Comp} ::= \cdots \mid (\mathbf{do}\ \ell\ V)^E \end{array}$$

The operation arrow, $\twoheadrightarrow$, denotes the operation space. The operation space arrow is similar to the function space arrow in that the type A denotes the domain type of the operation, i.e. the type of the operation payload, and the codomain type B denotes the return type of the operation. Contrary to the function space constructor, $\to$, the operation space constructor does not have an associated effect row. As we will see later, the reason that the operation space constructor does not have an effect row is that the effects of an operation is conferred by its handler.

The intended behaviour of the new computation term $(\mathbf{do}\ \ell\ V)^E$ is that it performs some operation ℓ with value argument V. Thus the **do**-construct is similar to the typical exception-signalling **throw** or **raise** constructs found in programming languages with support for exceptions. In fact operationally, an effectful operation may be thought of as an exception which is resumable [165]. The term is annotated with an effect row E, providing a way to make the current effect context accessible during typing.

$$\textsf{T-Do}\quad \frac{\Delta \vdash E \qquad E = \{\ell : A \twoheadrightarrow B; R\} \qquad \Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash (\mathbf{do}\ \ell\ V)^E : B ! E}$$

An operation invocation is only well-typed if the effect row E is well-kinded and mentions the operation with a present type, or put differently: the current effect context must permit an instance of the operation to occur. The argument type A must be the same as the domain type of the operation. The type of the whole term is the (value) return type of the operation paired with the current effect context.

We have the basic machinery for writing effectful programs, albeit we cannot evaluate those programs without handlers to ascribe a semantics to the operations.

3.2.2 Handling of effectful operations

The elimination form for an effectful operation is an effect handler. Effect handlers interpret the effectful segments of a program. The addition of handlers requires us to extend the type algebra of λ_{b} with a kind for handlers and a new syntactic category for

handler types.

$$\begin{array}{ll} \textsf{Kinds} & K \in \mathrm{Kind} ::= \cdots \mid \textsf{Handler} \\ \textsf{Handler types} & F \in \mathrm{HType} ::= C \Rightarrow D \\ \textsf{Types} & T \in \mathrm{Type} ::= \cdots \mid F \end{array}$$

The syntactic category of kinds is augmented with the kind Handler which we will ascribe to handler types F. The arrow, $\Rightarrow$, denotes the handler space. The type structure suggests that a handler is a transformer of computations, since by looking solely at the types a handler takes a computation of type C and returns another computation of type D. As such, we may think of a handler as a sort of generalised function, that work over computations rather than bare values (this observation is exploited in the Frank programming language, where a function is but a special case of a handler [52, 174]). The following kinding rule checks whether a handler type is well-kinded.

$$\textsf{K-Handler}\quad \frac{\Delta \vdash C : \textsf{Comp} \qquad \Delta \vdash D : \textsf{Comp}}{\Delta \vdash C \Rightarrow D : \textsf{Handler}}$$

With the type structure in place, we can move on to the term syntax for handlers. Handlers extend the syntactic category of computations with a new computation form as well as introducing a new syntactic category of handler definitions.

$$\begin{array}{ll} \textsf{Computations} & M, N \in \mathrm{Comp} ::= \cdots \mid \textbf{handle}\ M\ \textbf{with}\ H \\ \textsf{Handlers} & H \in \mathrm{HDef} ::= \{\textbf{return}\ x \mapsto M\} \mid \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \uplus H \\ \textsf{Terms} & t \in \mathrm{Term} ::= \cdots \mid H \end{array}$$

The handle construct ($\textbf{handle}\ M\ \textbf{with}\ H$) is the counterpart to **do**. It runs computation M using handler H. A handler H consists of a return clause $\{\textbf{return}\ x \mapsto M\}$ and a possibly empty set of operation clauses $\{\langle\!\langle \ell\ p_\ell \twoheadrightarrow r_\ell \rangle\!\rangle \mapsto N_\ell\}_{\ell \in \mathcal{L}}$. The return clause $\{\textbf{return}\ x \mapsto M\}$ defines how to interpret the final return value of a handled computation, i.e. a computation that has been fully reduced to $\textbf{return}\ V$ for some value V. The variable x is bound to the final return value in the body M. Each operation clause $\{\langle\!\langle \ell\ p_\ell \twoheadrightarrow r_\ell \rangle\!\rangle \mapsto N_\ell\}_{\ell \in \mathcal{L}}$ defines how to interpret an invocation of some operation ℓ. The variables p_ℓ and r_ℓ are bound in the body N_ℓ. The binding occurrence p_ℓ binds the payload of the operation and r_ℓ binds the resumption of the operation invocation, which is the delimited continuation from the invocation site up of ℓ to and including the enclosing handler.

Given a handler H, we often wish to refer to the clause for a particular operation or the return clause; for these purposes we define two convenient projections on handlers

in the metalanguage.

$$H^{\ell} \stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\}, \quad \text{where } \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \in H$$
$$H^{\text{ret}} \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto N\}, \quad \text{where } \{\mathbf{return}\ x \mapsto N\} \in H$$

The H^{ℓ} projection yields the singleton set consisting of the operation clause in H that handles the operation ℓ, whilst H^{ret} yields the singleton set containing the return clause of H. We define the *domain* of an handler as the set of operation labels it handles, i.e.

$$\mathsf{dom} \ : \ \text{HDef} \to \text{Label}$$
$$\mathsf{dom}(\{\mathbf{return}\ x \mapsto M\}) \stackrel{\text{def}}{=} \emptyset$$
$$\mathsf{dom}(\{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto M\} \uplus H) \stackrel{\text{def}}{=} \{\ell\} \cup \mathsf{dom}(H)$$

3.2.3 Static semantics

There are two typing rules for handlers. The first rule type checks the **handle**-construct and the second rule type checks handler definitions.

$$\textsf{T-Handle}\ \frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

$$\textsf{T-Handler}\ \frac{\begin{array}{c} C = A!\{(\ell_i : A_i \twoheadrightarrow B_i)_i; R\} \\ D = B!\{(\ell_i : P_i)_i; R\} \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\langle\!\langle \ell_i\ p_i \twoheadrightarrow r_i \rangle\!\rangle \mapsto N_i\}_i \\ \Delta; \Gamma, x : A \vdash M : D \\ [\Delta; \Gamma, p_i : A_i, r_i : B_i \to D \vdash N_i : D]_i \end{array}}{\Delta; \Gamma \vdash H : C \Rightarrow D}$$

The T-Handle rule is simply the application rule for handlers. The T-Handler rule is where most of the work happens. The effect rows on the input computation type C and the output computation type D must mention every operation in the domain of the handler. In the output row those operations may be either present ($\mathsf{Pre}(A)$), absent (Abs), or polymorphic in their presence (θ), whilst in the input row they must be mentioned with a present type as those types are used to type operation clauses. In each operation clause the resumption r_i must have the same return type, D, as its handler. In the return clause the binder x has the same type, C, as the result of the input computation.

3.2.4 Dynamic semantics

We augment the operational semantics with two new reduction rules: one for handling return values and another for handling operations.

$$\textsf{S-Ret} \quad \textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H \rightsquigarrow N[V/x], \qquad \text{where } H^{\text{ret}} = \{\textbf{return}\ x \mapsto N\}$$

$$\textsf{S-Op} \quad \textbf{handle}\ \mathcal{E}[\textbf{do}\ \ell\ V]\ \textbf{with}\ H \rightsquigarrow N[V/p, \lambda y.\, \textbf{handle}\ \mathcal{E}[\textbf{return}\ y]\ \textbf{with}\ H/r], \quad \text{where } H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \text{ and } \ell \notin \mathsf{BL}(\mathcal{E})$$

The rule S-Ret invokes the return clause of the current handler H and substitutes V for x in the body N. The rule S-Op handles an operation ℓ subject to two conditions. The first condition ensures that the operation is only captured by a handler if its handler definition H contains a corresponding operation clause for the operation. Otherwise the operation passes seamlessly through the handler such that another suitable handler can handle the operation. This phenomenon is known as *effect forwarding*. It is key to enable modular composition of effectful computations. The second condition ensures the operation ℓ and that the operation does not appear in the *bound labels* (BL) of the inner context $\mathcal{E}$. The bound label condition enforces that an operation is always handled by the nearest enclosing suitable handler. Formally, we define the notion of bound labels, $\mathsf{BL} : \text{Cont} \to \text{Label}$, inductively over the structure of evaluation contexts.

$$\begin{aligned} \mathsf{BL}([\,]) &= \emptyset \\ \mathsf{BL}(\textbf{let}\ x \leftarrow \mathcal{E}\ \textbf{in}\ N) &= \mathsf{BL}(\mathcal{E}) \\ \mathsf{BL}(\textbf{handle}\ \mathcal{E}\ \textbf{with}\ H) &= \mathsf{BL}(\mathcal{E}) \cup \mathsf{dom}(H) \end{aligned}$$

To illustrate the necessity of this condition consider the following example with two nested handlers which both handle the same operation ℓ.

$$\begin{aligned} H_{\mathsf{inner}} &\stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto r\ 42; \textbf{return}\ x \mapsto \textbf{return}\ x\} \\ H_{\mathsf{outer}} &\stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto r\ 0; \textbf{return}\ x \mapsto \textbf{return}\ x\} \end{aligned}$$

$$\textbf{handle}\ (\textbf{handle}\ \textbf{do}\ \ell\ \langle\rangle\ \textbf{with}\ H_{\mathsf{inner}})\ \textbf{with}\ H_{\mathsf{outer}} \rightsquigarrow^{+} \begin{cases} \textbf{return}\ 42 & \text{Innermost} \\ \textbf{return}\ 0 & \text{Outermost} \end{cases}$$

Without the bound label condition there are two possible results as the choice of which handler to pick for ℓ is ambiguous, meaning reduction would be nondeterministic. Conversely, with the bound label condition we obtain that the above term reduces to **return** 42, because ℓ is bound in the computation term of the outermost **handle**.

The decision to always select the nearest enclosing suitable handler for an operation invocation is a conscious choice. In fact, it is the *only* natural and sensible choice as picking any other handler than the nearest enclosing renders programming with effect handlers anti-modular. Consider the other extreme of always selecting the outermost suitable handler, then the meaning of any effectful program fragment depends on the entire ambient context. For example, consider using integer addition as the composition operator to compose the inner handle expression from above with a copy of itself.

$$\mathsf{fortytwo} \stackrel{\text{def}}{=} \textbf{handle do } \ell\ \langle\rangle \textbf{ with } H_{\mathsf{inner}}$$

$$\mathcal{E}[\mathsf{fortytwo} + \mathsf{fortytwo}] \rightsquigarrow^{+} \begin{cases} \textbf{return } 84 & \text{when } \mathcal{E} \text{ is empty} \\ ? & \text{otherwise} \end{cases}$$

Clearly, if the ambient context $\mathcal{E}$ is empty, then we can derive the result by reasoning locally about each constituent separately and subsequently add their results together to obtain the computation term **return** 84. Conversely, if the ambient context is nonempty, then we need to account for the possibility that some handler for ℓ is could be present in the context. For instance if $\mathcal{E} = \textbf{handle } [\,] \textbf{ with } H_{\mathsf{outer}}$ then the result would be **return** 0, which we cannot derive locally from looking at the immediate constituents. Thus we can argue that if we want programming to remain modular and compositional, then we must necessarily always select the nearest enclosing suitable handler for an operation.

The resumption r includes both the captured evaluation context and the handler. Invoking the resumption causes the both the evaluation context and handler to be reinstalled, meaning subsequent invocations of ℓ get handled by the same handler. This is a defining characteristic of deep handlers.

The metatheoretic properties of λ_{b} transfer to λ_{h} with little extra effort, although we must amend the definition of computation normal forms as there are now two ways in which a computation term can terminate: successfully returning a value or getting stuck on an unhandled operation.

Definition 3.7 (Computation normal forms)**.** We say that a computation term N is normal with respect to an effect signature E, if N is either of the form **return** V, or $\mathcal{E}[\textbf{do } \ell\, W]$ where $\ell \in E$ and $\ell \notin \mathsf{BL}(\mathcal{E})$.

Theorem 3.8 (Progress)**.** *Suppose* $\vdash M : C$*, then either there exists* $\vdash N : C$ *such that* $M \rightsquigarrow^{+} N$ *and* N *is normal, or* M *diverges.*

Proof. By induction on the typing derivations. □

Theorem 3.9 (Subject reduction)**.** *Suppose* $\Gamma \vdash M : C$ *and* $M \rightsquigarrow M'$*, then* $\Gamma \vdash M' : C$*.*

Proof. By induction on the typing derivations. □

3.2.5 Effect sugar

The row polymorphism formalism underlying the effect system is rigid with regard to presence information. Every effect row which share the same effect variable must mention the exact same operations to be complete, that is they must mention whether an operation is present, absent, or polymorphic in their its presence. Consequently, in higher-order effectful programming this can cause duplication of information, which in turn can cause effect signatures to become overly verbose. In most cases verbosity is undesirable if the extra information is redundant, and in practice it can be real nuisance in larger codebases. We can retrospectively fix this issue with some syntactic sugar rather than redesigning the entire effect system to rectify this problem.

To this end, I will take inspiration from the effect system of Frank, which allows eliding redundant information in many cases [174]. In the following I will describe an ad-hoc elaboration scheme for effect rows, which is designed to guess the programmer's intent for first-order and second-order functions, but it might not work so well for third-order and above. The reason for focusing on first-order and second-order functions is that many familiar and useful functions are either first-order or second-order, and in the following sections we will mostly be working with first-order and second-order functions (although, it should be noted that there exist useful functions at higher order, e.g. in Chapter 7 we shall use third-order functions; for an example of a sixth order function see Okasaki [209]).

First, let us consider the familiar second-order function map for lists which is completely parametric in its effects.

$$\begin{aligned} &\mathsf{map} : \langle \alpha \to \beta \quad ; \mathsf{List}\ \alpha \rangle \to \mathsf{List}\ \beta \\ \equiv\ &\mathsf{map} : \langle \alpha \to \beta ! \{\varepsilon\} ; \mathsf{List}\ \alpha \rangle \to \mathsf{List}\ \beta ! \{\varepsilon\} \end{aligned}$$

For this type to be correct in λ_{h} (and λ_{b} for that matter) we must annotate the computation type with their effect row. These effect annotations do not convey any additional information, because the function is entirely parametric in all effects, thus the ink spent on the annotations is really wasted in this instance. To fix this we simply need to instantiate each computation type with the same effect variable ε.

A slightly more interesting example is a second-order function which itself performs some operation, but is otherwise parametric in the effects of its argument.

$$\begin{aligned}&(A \to B_1!\{\qquad\qquad \varepsilon\}) \to B_2!\{\ell : A' \twoheadrightarrow B'; \varepsilon\}\\ \equiv\ &(A \to B_1!\{\ell : A' \twoheadrightarrow B'; \varepsilon\}) \to B_2!\{\ell : A' \twoheadrightarrow B'; \varepsilon\}\end{aligned}$$

To be type-correct both rows must mention the ℓ operation. However, this information is redundant on the functional parameter. The idea here is to push the information of the ambient effect row B_2 inwards to B_1 such that the functional argument can be granted the ability to perform ℓ. The following infix function $\lhd$ implements the inward push of information by copying operations from its right parameter to its left parameter.

$$\begin{aligned}\lhd\ &:\ \text{Effect} \times \text{Effect} \to \text{Effect}\\ E \lhd \{\cdot\} &\stackrel{\text{def}}{=} E\\ E \lhd \{\varepsilon\} &\stackrel{\text{def}}{=} E\\ E \lhd \{\ell : P; R\} &\stackrel{\text{def}}{=} \begin{cases}\{\ell : P\} \uplus (E \lhd \{R\}) & \text{if } \ell \notin E\\ \{R\} \lhd E & \text{otherwise}\end{cases}\end{aligned}$$

This function essentially computes the union of the two effect rows.

The most frequent case to occur is a second-order function which handles the effects of its argument.

$$\begin{aligned}&(A \to B_1!\{\ell : A' \twoheadrightarrow B'; \varepsilon\}) \to B_2!\{\qquad \varepsilon\}\\ \equiv\ &(A \to B_1!\{\ell : A' \twoheadrightarrow B'; \varepsilon\}) \to B_2!\{\ell : \theta; \varepsilon\}\end{aligned}$$

To capture the intuition that operations have been handled, we would like to not mention the handled operations in the effect row attached to B_2. The idea is to propagate the information of the effect row attached to B_1 outwards such that this information can be used to complete the effect row on B_2. To complete the row we need to copy the operations unique to the effect row of B_1 into the effect row of B_2 and instantiate them with a fresh presence variable. The following function $\rhd$ propagates information from its left parameter to its right parameter.

$$\begin{aligned}\rhd\ &:\ \text{Effect} \times \text{Effect} \to \text{Effect}\\ \{\cdot\} \rhd E &\stackrel{\text{def}}{=} E\\ \{\varepsilon\} \rhd E &\stackrel{\text{def}}{=} E\\ \{\ell : A \twoheadrightarrow B; R\} \rhd E &\stackrel{\text{def}}{=} \begin{cases}\{\ell : \theta\} \uplus (\{R\} \rhd E) & \text{if } \ell \notin E\\ \{R\} \rhd E & \text{otherwise}\end{cases}\\ \{\ell : P; R\} \rhd E &\stackrel{\text{def}}{=} \begin{cases}\{\ell : P\} \uplus (\{R\} \rhd E) & \text{if } \ell \notin E\\ \{R\} \rhd E & \text{otherwise}\end{cases}\end{aligned}$$

The only subtlety occur in the interesting case which is when an operation is present in the left row, but not in the right row. In this case we instantiate the operation with a fresh presence variable in the output row.

Propagation of information in either direction should only happen if the effect rows share the same effect variable. To avoid erroneous propagation of information we implement and use the following guarded variations of $\triangleleft$ and $\triangleright$.

$$\{R;\varepsilon\} \blacktriangleleft \{R';\varepsilon\} \stackrel{\text{def}}{=} \{R;\varepsilon\} \triangleleft \{R';\varepsilon\} \qquad \{R;\varepsilon\} \blacktriangleright \{R';\varepsilon\} \stackrel{\text{def}}{=} \{R;\varepsilon\} \triangleright \{R';\varepsilon\}$$
$$\{R;\varepsilon\} \blacktriangleleft \{R';\varepsilon'\} \stackrel{\text{def}}{=} \{R;\varepsilon\} \qquad \{R;\varepsilon\} \blacktriangleright \{R';\varepsilon'\} \stackrel{\text{def}}{=} \{R';\varepsilon'\}$$

The following function $I[\![-]\!]$ pushes the ambient effect row E_{amb} inward a given type. I will omit the homomorphic cases as there is only one interesting case.

$$I[\![-]\!] \;:\; \text{CType} \times \text{Effect} \to \text{CType}$$
$$I[\![A!E]\!]_{E_{\mathsf{amb}}} \stackrel{\text{def}}{=} I[\![A]\!]_{E_{\mathsf{amb}}}!E \blacktriangleleft E_{\mathsf{amb}}$$

The following function $O[\![-]\!]$ combines and propagates the effect rows of a type outward. Again, I omit the homomorphic cases.

$$O[\![-]\!] \;:\; \text{VType} \to \text{Effect} \qquad O[\![-]\!] \;:\; \text{CType} \to \text{Effect}$$
$$O[\![\alpha]\!] \stackrel{\text{def}}{=} \{\cdot\} \qquad O[\![A!E]\!] \stackrel{\text{def}}{=} O[\![A]\!] \blacktriangleright E$$
$$O[\![A \to C]\!] \stackrel{\text{def}}{=} O[\![A]\!] \blacktriangleright O[\![C]\!]$$
$$O[\![-]\!] \;:\; \text{Presence} \to \text{Effect} \qquad O[\![-]\!] \;:\; \text{Row} \to \mathsf{Effect}$$
$$O[\![\mathsf{Abs}]\!] \stackrel{\text{def}}{=} O[\![\theta]\!] \stackrel{\text{def}}{=} \{\cdot\} \qquad O[\![\cdot]\!] \stackrel{\text{def}}{=} O[\![\rho]\!] \stackrel{\text{def}}{=} \{\cdot\}$$
$$O[\![\ell : P;R]\!] \stackrel{\text{def}}{=} O[\![P]\!] \blacktriangleright O[\![R]\!]$$

We combine all of the above functions to implement the effect row elaboration for top-level function types.

$$\mathcal{T}[\![-]\!] \;:\; \text{VType} \to \text{VType}$$
$$\mathcal{T}[\![A \to B!E]\!] \stackrel{\text{def}}{=} I[\![A]\!]_{E'} \to I[\![B]\!]_{E'}!E'$$
$$\textbf{where } E' = (O[\![A]\!] \blacktriangleright E) \triangleright (O[\![B]\!] \blacktriangleright E)$$

The function $\mathcal{T}[\![-]\!]$ traverses the abstract syntax of its argument twice. The first traversal propagates effect information outwards to the ambient effect row E. The second traversal pushes the full ambient information E' inwards. The construction of E' makes use of the fact that $E \triangleright E = E$. As a remark, note that the function $\mathcal{T}[\![-]\!]$ do not have to consider handler types, because they cannot appear at the top-level in λ_{h}. With this syntactic sugar in place we can program with second-order effectful functions without having to write down redundant information.

3.3 Shallow effect handling

Shallow handlers are an alternative to deep handlers. Shallow handlers are defined as case-splits over computation trees, whereas deep handlers are defined as folds. Consequently, a shallow handler application unfolds only a single layer of the computation tree. Semantically, the difference between deep and shallow handlers is analogous to the difference between Church [46] and Scott [244] encoding techniques for data types in the sense that the recursion is intrinsic to the former, whilst recursion is extrinsic to the latter. Thus a fixpoint operator is necessary to make programming with shallow handlers practical.

Shallow handlers offer more flexibility than deep handlers as they do not hard wire a particular recursion scheme. Shallow handlers are favourable when catamorphisms are not the natural solution to the problem at hand. A canonical example of when shallow handlers are desirable over deep handlers is UNIX-style pipes, where the natural implementation is in terms of two mutually recursive functions (specifically *mutumorphisms* [97]), which is convoluted to implement with deep handlers [120, 122, 143].

In this section we take the full λ_{b} as our starting point and extend it with shallow handlers, resulting in the calculus $\lambda_{\mathsf{h}^\dagger}$. The calculus borrows some syntax and semantics from λ_{h}, whose presentation will not be duplicated in this section.

3.3.1 Syntax and static semantics

The syntax and semantics for effectful operation invocations are the same as in λ_{h}. Handler definitions and applications also have the same syntax as in λ_{h}, although we shall annotate the application form for shallow handlers with a superscript † to distinguish it from deep handler application.

$$\textsf{Computations} \quad M,N \in \mathrm{Comp} ::= \cdots \mid \mathbf{handle}^\dagger\ M\ \mathbf{with}\ H$$

The static semantics for $\mathbf{handle}^\dagger$ are the same as the static semantics for **handle**.

$$\textsf{T-Handle}^\dagger\ \frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}^\dagger\ M\ \mathbf{with}\ H : D}$$

$$\textsf{T-Handler}^\dagger\ \frac{\begin{array}{c} C = A!\{(\ell_i : A_i \twoheadrightarrow B_i)_i; R\} \\ D = B!\{(\ell_i : P_i)_i; R\} \\ H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\langle\!\langle \ell_i\ p_i \twoheadrightarrow r_i \rangle\!\rangle \mapsto N_i\}_i \\ \Delta;\Gamma,x : A \vdash M : D \\ [\Delta;\Gamma,p_i : A_i, r_i : B_i \to C \vdash N_i : D]_i \end{array}}{\Delta;\Gamma \vdash H : C \Rightarrow D}$$

The T-Handler† rule is remarkably similar to the T-Handler rule. In fact, the only difference is the typing of resumptions r_i. The codomain of r_i is C rather than D, meaning that a resumption returns a value of the same type as the input computation. In general the type C may be different from the output type D, thus it is evident from this typing rule that the handler does not guard invocations of resumptions r_i.

3.3.2 Dynamic semantics

There are two reduction rules.

$$\begin{array}{ll} \textsf{S-Ret}^\dagger & \mathbf{handle}^\dagger\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x], \text{where } H^{\text{ret}} = \{\mathbf{return}\ x \mapsto N\} \\ \textsf{S-Op}^\dagger & \mathbf{handle}^\dagger\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/p, \lambda y.\, \mathcal{E}[\mathbf{return}\ y]/r], \\ & \qquad \text{where } H^\ell = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \\ & \qquad \text{and } \ell \notin \mathsf{BL}(\mathcal{E}) \end{array}$$

The rule S-Ret† is the same as the S-Ret rule for deep handlers — there is no difference in how the return value is handled. The S-Op† rule is almost the same as the S-Op rule the crucial difference being the construction of the resumption r. The resumption consists entirely of the captured context $\mathcal{E}$. Thus an invocation of r does not reinstall its handler as in the setting of deep handlers, meaning is up to the programmer to supply the handler the next invocation of ℓ inside $\mathcal{E}$. This handler may be different from H.

The basic metatheoretic properties of $\lambda_{\mathsf{h}^\dagger}$ are a carbon copy of the basic properties of λ_{h}.

Theorem 3.10 (Progress)**.** *Suppose $\vdash M : C$, then either there exists $\vdash N : C$ such that $M \rightsquigarrow^+ N$ and N is normal, or M diverges.*

Proof. By induction on the typing derivations. □

Theorem 3.11 (Subject reduction)**.** *Suppose $\Gamma \vdash M : C$ and $M \rightsquigarrow M'$, then $\Gamma \vdash M' : C$.*

Proof. By induction on the typing derivations. □

3.4 Parameterised effect handling

Parameterised handlers are a variation of ordinary deep handlers with an embedded functional state cell. This state cell is only accessible locally within the handler. The use of state within the handler is opaque to both the ambient context and the context

of the computation being handled. Semantically, parameterised handlers are defined as folds with state threading over computation trees.

We take the deep handler calculus λ_{h} as our starting point and extend it with parameterised handlers to yield the calculus $\lambda_{\mathsf{h}^\ddagger}$. The parameterised handler extension interacts nicely with shallow handlers, and as such it can be added to $\lambda_{\mathsf{h}^\dagger}$ with low effort.

3.4.1 Syntax and static semantics

In addition to a computation, a parameterised handler also take a value as argument. This argument is the initial value of the state cell embedded inside the handler.

Handler types	$F ::= \cdots \mid \langle C;A\rangle \Rightarrow^\ddagger D$
Computations	$M,N ::= \cdots \mid \mathbf{handle}^\ddagger\ M\ \mathbf{with}\ H^\ddagger(W)$
Parameterised definitions	$H^\ddagger ::= q^A.\ H$

The syntactic category of handler types F is extended with a new kind of handler arrow for parameterised handlers. The left hand side of the arrow is a pair, whose first component denotes the type of the input computation and the second component denotes the type of the handler parameter. The right hand side denotes the return type of the handler. The computations category is extended with a new application form for handlers, which runs a computation M under a parameterised handler H applied to the value W. Finally, a new category is added for parameterised handler definitions. A parameterised handler definition is a new binding form $(q^A.\ H)$, where q is the name of the parameter, whose type is A, and H is an ordinary handler definition H. The parameter q is accessible in the **return** and operation clauses of H.

As with ordinary deep handlers and shallow handlers, there are two typing rules: one for handler application and another for handler definitions.

$$\frac{\Gamma \vdash M : C \qquad \Gamma \vdash W : A \qquad \Gamma \vdash H^\ddagger : \langle C;A\rangle \Rightarrow^\ddagger D}{\Gamma \vdash \mathbf{handle}^\ddagger\ M\ \mathbf{with}\ H^\ddagger(W) : D}\ \mathsf{T\text{-}Handle}^\ddagger$$

The T-Handle‡ rule is similar to the T-Handle and T-Handle† rules, except that it has to account for the parameter W, whose type has to be compatible with the second component of the domain type of the handler definition $H^\ddagger$. The typing rule for parameterised handler definitions adapts the corresponding typing rule T-Handler for ordinary deep

handlers with the addition of a parameter.

$$\frac{\begin{array}{c}\mathsf{T\text{-}Handler}^{\ddagger} \\ C = A!\{(\ell_i : A_i \to B_i)_i; R\} \\ D = B!\{(\ell_i : P)_i; R\} \\ H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\langle\!\langle \ell_i\, p_i \twoheadrightarrow r_i \rangle\!\rangle \mapsto N_i\}_i \\ \Delta; \Gamma, q : A', x : A \vdash M : D \\ [\Delta; \Gamma, q : A', p_i : A_i, r_i : \langle B_i; A' \rangle \to D \vdash N_i : D]_i\end{array}}{\Delta; \Gamma \vdash (q^{A'}.H) : \langle C; A' \rangle \Rightarrow^{\ddagger} D}$$

The key differences between the T-Handler and T-Handler‡ rules are that in the latter the return and operation cases are typed with respect to the parameter q, and that resumptions r_i have type $\langle B_\ell; A' \rangle \to D$, that is a parameterised resumption is a binary function, where the first argument is the interpretation of an operation and the second argument is the (updated) handler state. The return type of r_i is the same as the return type of the handler, meaning that an invocation of r_i is guarded in the same way as an invocation of an ordinary deep resumption.

3.4.2 Dynamic semantics

The two reduction rules for parameterised handlers adapt the reduction rules for ordinary deep handlers with a parameter.

$$\begin{array}{ll}\mathsf{S\text{-}Ret}^{\ddagger} & \mathbf{handle}^{\ddagger}\ (\mathbf{return}\ V)\ \mathbf{with}\ (q.H)(W) \rightsquigarrow N[V/x, W/q], \\ & \qquad \text{where } H^{\mathrm{ret}} = \{\mathbf{return}\, x \mapsto N\} \\ \mathsf{S\text{-}Op}^{\ddagger} & \mathbf{handle}^{\ddagger}\ \mathcal{E}[\mathbf{do}\ \ell\, V]\ \mathbf{with}\ (q.H)(W) \rightsquigarrow N[V/p, W/q, R/r], \\ & \quad \text{where } R = \lambda\langle y; q' \rangle.\mathbf{handle}^{\ddagger}\ \mathcal{E}[\mathbf{return}\, y]\ \mathbf{with}\ (q.H)(q') \\ & \quad \text{and } H^{\ell} = \{\langle\!\langle \ell\, p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \\ & \quad \text{and } \ell \notin \mathsf{BL}(\mathcal{E})\end{array}$$

The rule S-Ret‡ handles the return value of a computation. Just like the rule S-Ret the return value V is substituted for the binder x in the return case body N. Furthermore the value W is substituted for the handler parameter q in N, meaning the handler parameter is accessible in the return case.

The S-Op‡ handles an operation invocation. Both the operation payload V and handler argument W are accessible inside the case body N. As with ordinary deep handlers, the resumption rewraps its handler, but with the slight twist that the parameterised handler definition is applied to the updated parameter value q' rather than the original value

W. This achieves the effect of state passing as the value of q' becomes available upon the next activation of the handler.

The metatheoretic properties of λ_{h} carries over to $\lambda_{\mathsf{h}^\ddagger}$.

Theorem 3.12 (Progress)**.** *Suppose $\vdash M : C$, then either there exists $\vdash N : C$ such that $M \rightsquigarrow^+ N$ and N is normal, or M diverges.*

Proof. By induction on the typing derivations. □

Theorem 3.13 (Subject reduction)**.** *Suppose $\Gamma \vdash M : C$ and $M \rightsquigarrow M'$, then $\Gamma \vdash M' : C$.*

Proof. By induction on the typing derivations. □

3.5 Related work

Row polymorphism Row polymorphism was originally introduced by Wand [272] as a typing discipline for extensible records. The style of row polymorphism used in this chapter is due to Rémy [237]. It was designed to work well with type inference as typically featured in the ML-family of programming languages. Rémy also describes a slight variation of this system, where the presence polymorphism annotations may depend on a concrete type, e.g. $\ell : \theta.\mathsf{Int}; R$ means that the label is polymorphic in its presence, however, if it is present then it has presence type Int.

Either of Rémy's row systems have set semantics, i.e. a row cannot contain duplicated labels. An alternative semantics based on dictionaries is used by Leijen [163]. In Leijen's system labels may be duplicated, which introduces a form of scoping for labels, which for example makes it possible to shadow fields in a record. There is no notion of presence information in [163]'s system, and thus, as a result Leijen-style rows simplify the overall type structure. Leijen [164] has used this system as the basis for the effect system of Koka.

Morris and McKinna [206] have developed a unifying theory of rows, which collects the aforementioned row systems under one umbrella. Their system provides a general account of record extension and projection, and dually, variant injection and branching.

Effect tracking As mentioned in Section 1.2.3 the original effect system was developed by Lucassen and Gifford [183] to provide a lightweight facility for static concurrency analysis. Since then effect systems have been employed to perform a variety of static analyses, e.g. Tofte and Talpin [262, 263] describe a region-based memory management system that makes use of a type and effect system to infer and track lifetimes

of regions; Benton and Kennedy [17] use a monadic effect system to identify opportunities for optimisations in the intermediate language of their ML to Java compiler; and Lindley and Cheney [173] use a variation of the row system presented in this chapter to support abstraction and predicable code generation for database programming in Links. Row types are used to give structural types to SQL rows in queries, whilst their effect system is used to differentiate between *tame* and *wild* functions, where a tame function is one whose body can be translated and run directly on the database, whereas a wild function cannot.

Programming languages with handlers The union of the calculi presented in this chapter make up the essence of the intermediate representation of the Links programming language [119, 120]. A closely related programming language with handlers is Leijen's Koka, which has been retrofitted with ordinary deep and parameterised effect handlers [165]. In Koka effects are nominal, meaning an effect and its constructors must be declared before use, which is unlike the structural approach taken in this chapter. Koka also tracks effects via an effect system based on Leijen-style row polymorphism [163, 164], where rows are interpreted as multisets which means an effect can occur multiple times in an effect row. The ability to repeat effects provide a form for effect scoping in the sense that an effect instance can shadow another. A handler handles only the first instance of a repeated effect, leaving the remaining instances for another handler. Consequently, the order of repeated effect instances matter and it can therefore be situational useful to manipulate the order of repeated instances by way of so-called *effect masking*. The notion of effect masking was formalised by Biernacki et al. [26] and generalised by Convent et al. [52].

Biernacki et al. [26] designed Helium, which is a programming language that features a rich module system, deep handlers, and *lexical* handlers [28]. Lexical handlers *bind* effectful operations to specific handler instances. Operations remain bound for the duration of computation. This makes the nature of lexical handlers more static than ordinary deep handlers, as for example it is not possible to dynamically overload the interpretation of residual effects of a resumption invocation as in Section 2.3. The mathematical foundations for lexical handlers has been developed by Geron [106].

The design of the Effekt language by Brachthäuser et al. [39] resolves around the idea of lexical handlers for efficiency. Effekt takes advantage of the static nature of lexical handlers to eliminate the dynamic handler lookup at runtime by tying the correct handler instance directly to an operation invocation [35, 243]. The effect system

of Effekt is based on intersection types, which provides a limited form of effect polymorphism [39]. A design choice that means it does not feature first-class functions.

The Frank language by Lindley et al. [174] is born and bred on shallow effect handlers. One of the key novelties of Frank is n-ary shallow handlers, which generalise ordinary unary shallow handlers to be able to handle multiple computations simultaneously. Another novelty is the effect system, which is based on a variation of Leijen-style row polymorphism, where the programmer rarely needs to mention effect variables. This is achieved by insisting that the programmer annotates each input argument with the particular effects handled at the particular argument position as well as declaring what effects needs to be handled by the ambient context. Each annotation is essentially an incomplete row. They are made complete by concatenating them and inserting a fresh effect variable.

Bauer and Pretnar's Eff language was the first programming language designed from the ground up with effect handlers in mind. It features only deep handlers [15]. A previous iteration of the language featured an explicit *effect instance* system. An effect instance is a sort of generative interface, where the operations are unique to each instance. As a result it is possible to handle two distinct instances of the same effect differently in a single computation. Their system featured a type-and-effect system with support for effect inference [14, 231], however, the effect instance system was later dropped to in favour of a vanilla nominal approach to effects and handlers.

Multicore OCaml is, at the time of writing, an experimental branch of the OCaml programming language, which aims to extend OCaml with effect handlers for multicore and concurrent programming [70, 71]. The current incarnation features untracked nominal effects and deep handlers with single-use resumptions.

Part II

Implementation

Chapter 4

Continuation-passing style

Continuation-passing style (CPS) is a *canonical* program notation that makes every facet of control flow and data flow explicit. In CPS every function takes an additional function-argument called the *continuation*, which represents the next computation in evaluation position. CPS is canonical in the sense that it is definable in pure λ-calculus without any further primitives. As an informal illustration of CPS let us consider the ever-green factorial function, which may be implemented in λ_{b} as follows.

$$
\begin{array}{l}
\mathsf{fac} : \mathsf{Int} \to \mathsf{Int} \\
\mathsf{fac} \stackrel{\text{def}}{=} \lambda n.\mathbf{let}\ isz \leftarrow n = 0\ \mathbf{in} \\
\qquad\qquad \mathbf{if}\ isz\ \mathbf{then\ return}\ 1 \\
\qquad\qquad \mathbf{else}\ \ \mathbf{let}\ n' \leftarrow n - 1\ \mathbf{in} \\
\qquad\qquad\qquad\ \mathbf{let}\ m \leftarrow \mathsf{fac}\ n'\ \mathbf{in} \\
\qquad\qquad\qquad\ \mathbf{let}\ res \leftarrow n * m\ \mathbf{in} \\
\qquad\qquad\qquad\ \mathbf{return}\ res
\end{array}
$$

The above implementation of the function fac is given in direct-style fine-grain call-by-value. In CPS notation the implementation of this function changes as follows.

$$
\begin{array}{l}
\mathsf{fac}_{\mathsf{cps}} : \mathsf{Int} \to (\mathsf{Int} \to \alpha) \to \alpha \\
\mathsf{fac}_{\mathsf{cps}} \stackrel{\text{def}}{=} \lambda n.\lambda k.(=_{\mathsf{cps}})\ n\ 0\ (\lambda isz.\ \mathbf{if}\ isz\ \mathbf{then}\ k\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (-_{\mathsf{cps}})\ n\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda n'.\mathsf{fac}_{\mathsf{cps}}\ n' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda m.(*_{\mathsf{cps}})\ n\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda res.k\ res))))
\end{array}
$$

There are several worthwhile observations to make about the differences between the two implementations fac and $\mathsf{fac}_{\mathsf{cps}}$. Firstly note that their type signatures differ. The

CPS version has an additional formal parameter of type $\mathsf{Int} \to \alpha$ which is the continuation. By convention the continuation parameter is named k in the implementation. As usual, the continuation represents the remainder of computation. In this specific instance k represents the undelimited current continuation of an application of $\mathsf{fac}_{\mathsf{cps}}$. Given a value of type Int, the continuation produces a result of type α, which is the *answer type* of the entire program. Thus applying $\mathsf{fac}_{\mathsf{cps}}\ 3$ to the identity function $(\lambda x.x)$ yields $6 : \mathsf{Int}$, whilst applying it to the predicate $\lambda x.x > 2$ yields $\mathsf{true} : \mathsf{Bool}$.

Secondly note that every **let**-binding in fac has become a function application in $\mathsf{fac}_{\mathsf{cps}}$. The binding sequence in the **else**-branch has been turned into a series of nested function applications. The functions $=_{\mathsf{cps}}$, $-_{\mathsf{cps}}$, and $*_{\mathsf{cps}}$ denote the CPS versions of equality testing, subtraction, and multiplication respectively. For clarity, I have meticulously written each continuation function on a newline. For instance, the continuation of the $-_{\mathsf{cps}}$-application is another application of $\mathsf{fac}_{\mathsf{cps}}$, whose continuation is an application of $*_{\mathsf{cps}}$, and its continuation is an application of the current continuation, k, of $\mathsf{fac}_{\mathsf{cps}}$. Each **return**-computation has been turned into an application of the current continuation k. In the **then**-branch the continuation applied to 1, whilst in the **else**-branch the continuation is applied to the result obtained by multiplying n and m.

Thirdly note that every function application occurs in tail position (recall Definition 3.1). This is a characteristic property of CPS transforms that make them feasible as a practical implementation strategy, since programs in CPS notation require only a constant amount of stack space to run, namely, a single activation frame [8]. Although, the pervasiveness of closures in CPS means that CPS programs make heavy use of the heap for closure allocation. Some care must be taken when CPS transforming a program as if done naïvely the image may be inflated with extraneous terms [68]. For example in $\mathsf{fac}_{\mathsf{cps}}$ the continuation term $(\lambda res.k\ res)$ is redundant as it is simply an eta expansion of the continuation k. A more optimal transform would simply pass k. Extraneous terms can severely impact the runtime performance of a CPS program. A smart CPS transform recognises and eliminates extraneous terms at translation time [67]. Extraneous terms come in various disguises as we shall see later in this chapter.

The complete exposure of the control flow makes CPS a good fit for implementing control operators such as effect handlers. It is an established intermediate representation used by compilers, providing it with merits as a practical compilation target [8, 144].

The purpose of this chapter is to use the CPS formalism to develop a universal implementation strategy for deep, shallow, and parameterised effect handlers. Section 4.1 defines a suitable target calculus λ_{u} for CPS transformed programs. Section 4.2 demon-

strates how to CPS transform λ_b-programs to λ_u-programs. In Section 4.3 develop a CPS transform for deep handlers through step-wise refinement of the initial CPS transform for λ_b. The resulting CPS transform is adapted in Section 4.4 to support for shallow handlers. As a by-product we develop the notion of *generalised continuation*, which provides a versatile abstraction for implementing effect handlers. We use generalised continuations to implement parameterised handlers in Section 4.5.

Chapter outline

Section 4.1 presents the initial target calculus.

Section 4.2 demonstrates how to translate λ_b to the target calculus via CPS.

Section 4.3 starts from a rather naïve first-order CPS transform for deep handlers. Over the course of several refinements the naïve CPS transform gets progressively more sophisticated, ultimately resulting in a higher-order one-pass CPS transform for deep handlers.

Section 4.4 further extends higher-order CPS to accommodate shallow handlers. During this development the notion of generalised continuations is introduced.

Section 4.5 adapts the translation with generalised continuations to support parameterised handlers as well.

Section 4.6 discusses related work.

Relation to prior work This chapter is based on the following work.

i Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *FSCD*, volume 84 of *LIPIcs*, pages 18:1–18:19, 2017

ii Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018

iii Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020

Section 4.3.4 is based on item i, however, I have adapted it to follow the notation and style of item iii.

4.1 Initial target calculus

The syntax, semantics, and syntactic sugar for the target calculus λ_{u} is given in Figure 4.1. The calculus largely amounts to an untyped variation of λ_{b}, specifically we retain the syntactic distinction between values (V) and computations (M). The values (V) comprise lambda abstractions ($\lambda x.M$), empty tuples ($\langle\rangle$), pairs ($\langle V, W\rangle$), and first-class labels (ℓ). Computations (M) comprise values (V), applications ($M\ V$), pair elimination (**let** $\langle x, y\rangle = V$ **in** N), label elimination (**case** $V\ \{\ell \mapsto M; x \mapsto N\}$), and explicit marking of unreachable code (**absurd**). A key difference from λ_{b} is that the function position of an application is allowed to be a computation (i.e., the application form is $M\ W$ rather than $V\ W$). Later, when we refine the initial CPS translation we will be able to rule out this relaxation.

The reduction semantics follows the trend of the previous reduction semantics in the sense that it is a small-step context-based reduction semantics. Evaluation contexts comprise the empty context and function application.

To make the notation more lightweight, we define syntactic sugar for variant values, record values, list values, let binding, variant eliminators, and record eliminators. We use pattern matching syntax for deconstructing variants, records, and lists. For desugaring records, we assume a failure constant $\ell_\perp$ (e.g. a divergent term) to cope with the case of pattern matching failure.

4.2 Transforming fine-grain call-by-value

We start by giving a CPS translation of λ_{b} in Figure 4.2. Fine-grain call-by-value admits a particularly simple CPS translation due to the separation of values and computations. All constructs from the source language are translated homomorphically into the target language λ_{u}, except for **return** and **let** (and type abstraction because the translation performs type erasure). Lifting a value V to a computation **return** V is interpreted by passing the value to the current continuation k. Sequencing computations with **let** is translated by applying the translation of M to the translation of the continuation N, which is ultimately applied to the current continuation k. In addition, we explicitly η-expand the translation of a type abstraction in order to ensure that value terms in the source calculus translate to value terms in the target.

Syntax

Values	$U, V, W \in \mathrm{UVal} ::= x \mid \lambda x.M \mid\mid \langle\rangle \mid \langle V, W\rangle \mid \ell$
Computations	$M, N \in \mathrm{UComp} ::= V \mid M\,W \mid \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N$
	$\mid\ \mathbf{case}\ V\,\{\ell \mapsto M; y \mapsto N\} \mid \mathbf{absurd}\,V$
Evaluationcontexts	$\mathcal{E} \in \mathrm{UCont} ::= [\,] \mid \mathcal{E}\,W$

Reductions

U-App	$(\lambda x.M)V \rightsquigarrow M[V/x]$
U-Split	$\mathbf{let}\ \langle x, y\rangle = \langle V, W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y]$
U-Case$_1$	$\mathbf{case}\ \ell\ \{\ell \mapsto M; y \mapsto N\} \rightsquigarrow M$
U-Case$_2$	$\mathbf{case}\ \ell\ \{\ell' \mapsto M; y \mapsto N\} \rightsquigarrow N[\ell/y], \quad \text{if } \ell \neq \ell'$
U-Lift	$\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \quad \text{if } M \rightsquigarrow N$

Syntactic sugar

$$
\begin{aligned}
\mathbf{let}\ x = V\ \mathbf{in}\ N &\equiv N[V/x] \\
\ell\ V &\equiv \langle \ell; V\rangle \\
\langle\rangle &\equiv \ell_{\langle\rangle} \\
\langle \ell = V; W\rangle &\equiv \langle \ell, \langle V, W\rangle\rangle \\
[] &\equiv \ell_{[]} \\
V :: W &\equiv \langle \ell_{::}, \langle V, W\rangle\rangle \\
\mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\} &\equiv \mathbf{let}\ y = V\ \mathbf{in}\ \mathbf{let}\ \langle z, x\rangle = y\ \mathbf{in} \\
&\qquad \mathbf{case}\ z\ \{\ell \mapsto M; z' \mapsto N\} \\
\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N &\equiv \mathbf{let}\ \langle z, z'\rangle = V\ \mathbf{in}\ \mathbf{let}\ \langle x, y\rangle = z'\ \mathbf{in} \\
&\qquad \mathbf{case}\ z\ \{\ell \mapsto N; z'' \mapsto \ell_{\perp}\}
\end{aligned}
$$

Figure 4.1: Untyped target calculus for the CPS translations.

Values

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Val} \to \mathrm{UVal} \\
[\![x]\!] &= x \\
[\![\lambda x.M]\!] &= \lambda x.[\![M]\!] \\
[\![\Lambda\alpha.M]\!] &= \lambda k.[\![M]\!]\ k \\
[\![\langle\rangle]\!] &= \langle\rangle \\
[\![\langle \ell = V; W\rangle]\!] &= \langle \ell = [\![V]\!]; [\![W]\!]\rangle \\
[\![\ell\ V]\!] &= \ell\ [\![V]\!]
\end{aligned}
$$

Computations

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
[\![V\,W]\!] &= [\![V]\!]\,[\![W]\!] \\
[\![V\,T]\!] &= [\![V]\!] \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &= \mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!] \\
[\![\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}]\!] &= \mathbf{case}\ [\![V]\!]\ \{\ell\ x \mapsto [\![M]\!]; y \mapsto [\![N]\!]\} \\
[\![\mathbf{absurd}\ V]\!] &= \mathbf{absurd}\ [\![V]\!] \\
[\![\mathbf{return}\ V]\!] &= \lambda k.k\,[\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &= \lambda k.[\![M]\!](\lambda x.[\![N]\!]\,k)
\end{aligned}
$$

Figure 4.2: First-order CPS translation of λ_{b}.

4.3 Transforming deep effect handlers

The translation of a computation term by the basic CPS translation in Section 4.2 takes a single continuation parameter that represents the context. In the presence of effect handlers in the source language, it becomes necessary to keep track of two kinds of contexts in which each computation executes: a *pure context* that tracks the state of pure computation in the scope of the current handler, and an *effect context* that describes how to handle operations in the scope of the current handler. Correspondingly, we have both *pure continuations* (k) and *effect continuations* (h). As handlers can be nested, each computation executes in the context of a *stack* of pairs of pure and effect continuations.

On entry into a handler, the pure continuation is initialised to a representation of the return clause and the effect continuation to a representation of the operation clauses. As pure computation proceeds, the pure continuation may grow, for example when executing a **let**. If an operation is encountered then the effect continuation is invoked. The current continuation pair (k, h) is packaged up as a *resumption* and passed to the current handler along with the operation and its argument. The effect continuation then either handles the operation, invoking the resumption as appropriate, or forwards the operation to an outer handler. In the latter case, the resumption is modified to ensure that the context of the original operation invocation can be reinstated upon invocation of the resumption.

4.3.1 Curried translation

We first consider a curried CPS translation that extends the translation of Figure 4.2. The extension to operations and handlers is localised to the additional features because currying conveniently lets us get away with a shift in interpretation: rather than accepting a single continuation, translated computation terms now accept an arbitrary even number of arguments representing the stack of pure and effect continuations. Thus, we can conservatively extend the translation in Figure 4.2 to cover λ_{h}, where we imagine there being some number of extra continuation arguments that have been η-reduced.

The translation of operations and handlers is as follows.

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \lambda k.\lambda h.h\ \langle \ell, \langle [\![V]\!], \lambda x.k\ x\ h\rangle\rangle \\
[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &\stackrel{\text{def}}{=} [\![M]\!]\ [\![H^{\mathrm{ret}}]\!]\ [\![H^{\mathrm{ops}}]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UComp} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &\stackrel{\text{def}}{=} \lambda x.\lambda h.[\![N]\!] \\
[\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell\in\mathcal{L}}]\!] &\stackrel{\text{def}}{=} \lambda\langle z, \langle p, r\rangle\rangle.\mathbf{case}\ z\ \{(\ell \mapsto [\![N_\ell]\!])_{\ell\in\mathcal{L}}; y \mapsto M_{\mathrm{forward}}(y,p,r)\} \\
M_{\mathrm{forward}}(y,p,r) &\stackrel{\text{def}}{=} \lambda k.\lambda h.h\,\langle y, \langle p, \lambda x.\, r\,x\,k\,h\rangle\rangle
\end{aligned}
$$

The translation of $\mathbf{do}\ \ell\ V$ abstracts over the current pure (k) and effect (h) continuations passing an encoding of the operation into the latter. The operation is encoded as a triple consisting of the name ℓ, parameter $[\![V]\!]$, and resumption $\lambda x.k\ x\ h$, which passes the same effect continuation h to ensure deep handler semantics.

The translation of $\mathbf{handle}\ M\ \mathbf{with}\ H$ invokes the translation of M with new pure and effect continuations for the return and operation clauses of H. The translation of a return clause is a term which garbage collects the current effect continuation h. The translation of a set of operation clauses is a function which dispatches on encoded operations, and in the default case forwards to an outer handler. In the forwarding case, the resumption is extended by the parent continuation pair to ensure that an eventual invocation of the resumption reinstates the handler stack.

The translation of complete programs feeds the translated term the identity pure continuation (which discards its handler argument), and an effect continuation that is never intended to be called.

$$
\begin{aligned}
\top[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
\top[\![M]\!] &\stackrel{\text{def}}{=} [\![M]\!]\ (\lambda x.\lambda h.x)\ (\lambda\langle z, \_\rangle.\mathbf{absurd}\ z)
\end{aligned}
$$

Conceptually, this translation encloses the translated program in a top-level handler with an empty collection of operation clauses and an identity return clause.

A pleasing property of this particular CPS translation is that it is a conservative extension to the CPS translation for λ_{b}. Alas, this translation also suffers two displeasing properties which makes it unviable in practice.

1. The image of the translation is not *properly tail-recursive* [60, 63, 254], meaning not every function application occur in tail position in the image, and thus the image is not stackless. Consequently, the translation cannot readily be used as the basis for an implementation. This deficiency is essentially due to the curried representation of the continuation stack.

2. The image of the translation yields static administrative redexes, i.e. redexes that could be reduced statically. This is a classic problem with CPS translations. This problem can be dealt with by introducing a second pass to clean up the image [220]. By clever means the clean up pass and the translation pass can be fused together to make an one-pass translation [63, 67].

The following minimal example readily illustrates both issues.

$$\begin{aligned}
\top[\![\mathbf{return}\ \langle\rangle]\!] &= (\lambda k.k\,\langle\rangle)\,(\lambda x.\lambda h.x)\,(\lambda\langle z,_\rangle.\mathbf{absurd}\,z) \\
&\rightsquigarrow ((\lambda x.\lambda h.x)\,\langle\rangle)\,(\lambda\langle z,_\rangle.\mathbf{absurd}\,z) && (*4.1)\\
&\rightsquigarrow (\lambda h.\langle\rangle)\,(\lambda\langle z,_\rangle.\mathbf{absurd}\,z) && (*4.2)\\
&\rightsquigarrow \langle\rangle
\end{aligned}$$

The second and third reductions simulate handling **return** $\langle\rangle$ at the top level. The second reduction partially applies the curried function term $\lambda x.\lambda h.x$ to $\langle\rangle$, which must return a value such that the third reduction can be applied. Consequently, evaluation is not tail-recursive. The lack of tail-recursion is also apparent in our relaxation of fine-grain call-by-value in Figure 4.1 as the function position of an application can be a computation. In Section 4.3.2 we will refine this translation to be properly tail-recursive. As for administrative redexes, observe that the first reduction is administrative. It is an artefact introduced by the translation, and thus it has nothing to do with the dynamic semantics of the original term. We can eliminate such redexes statically. We will address this issue in Section 4.3.4.

Nevertheless, we can show that the image of this CPS translation simulates the preimage. Due to the presence of administrative reductions, the simulation is not on the nose, but instead up to congruence. For reduction in the untyped target calculus we write $\rightsquigarrow_{\text{cong}}$ for the smallest relation containing $\rightsquigarrow$ that is closed under the term formation constructs.

Theorem 4.1 (Simulation)**.** *If* $M \rightsquigarrow N$ *then* $\top[\![M]\!] \rightsquigarrow^{+}_{cong} \top[\![N]\!]$.

Proof. The result follows by composing a call-by-value variant of Forster et al.'s translation from effect handlers to delimited continuations [2019] with Materzok and Biernacki's CPS translation for delimited continuations [2012]. □

4.3.2 Uncurried translation

In this section we will refine the CPS translation for deep handlers to make it properly tail-recursive. As remarked in the previous section, the lack of tail-recursion is apparent

Syntax

Computations	$M, N \in \mathrm{UComp} ::= \cdots \mid \cancel{M\,W} \mid V\,W \mid U\,V\,W$
~~Evaluationcontexts~~	~~$\mathcal{E} \in \mathrm{UCont} ::= [\,] \mid \mathcal{E}\,W$~~

Reductions

U-App$_1$	$(\lambda x.M)V \rightsquigarrow M[V/x]$
U-App$_2$	$(\lambda x.\lambda y.M)V\,W \rightsquigarrow M[V/x, W/y]$
~~U-Lift~~	~~$\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]$, if $M \rightsquigarrow N$~~

Figure 4.3: Adjustments to the syntax and semantics of λ_{u}.

---

in the syntax of the target calculus λ_{u} as it permits an arbitrary computation term in the function position of an application term.

As a first step we may restrict the syntax of the target calculus such that the term in function position must be a value. With this restriction the syntax of λ_{u} implements the property that any term constructor features at most one computation term, and this computation term always appears in tail position. This restriction suffices to ensure that every function application will be in tail position. Figure 4.3 contains the adjustments to syntax and semantics of λ_{u}. The target calculus now supports both unary and binary application forms. As we shall see shortly, binary application turns out be convenient when we enrich the notion of continuation. Both application forms are comprised only of value terms. As a result the dynamic semantics of λ_{u} no longer makes use of evaluation contexts. The reduction rule U-App$_1$ applies to unary application and it is the same as the U-App-rule in Figure 4.1. The new U-App$_2$-rule applies to binary application: it performs a simultaneous substitution of the arguments V and W for the parameters x and y, respectively, in the function body M.

These changes to λ_{u} immediately invalidate the curried translation from the previous section as the image of the translation is no longer well-formed. The crux of the problem is that the curried interpretation of continuations causes the CPS translation to produce 'large' application terms, e.g. the translation rule for effect forwarding produces a three-argument application term. To rectify this problem we can adapt the technique of Materzok and Biernacki [190] to uncurry our CPS translation. Uncurrying necessitates a change of representation for continuations: a continuation is now an alternating list of pure continuation functions and effect continuation functions. Thus, we move to an explicit representation of the runtime handler stack. The change of continuation

representation means the CPS translation for effect handlers is no longer a conservative extension. The translation is adjusted as follows to account for the new representation.

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
[\![\mathbf{return}\ V]\!] &\stackrel{\text{def}}{=} \lambda(k :: ks).k\,[\![V]\!]\,ks \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &\stackrel{\text{def}}{=} \lambda(k :: ks).[\![M]\!]((\lambda x.\lambda ks'.[\![N]\!](k :: ks')) :: ks) \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \lambda(k :: h :: ks).h\,\langle \ell, \langle [\![V]\!], \lambda x.\lambda ks'.k\,x\,(h :: ks')\rangle\rangle\,ks \\
[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &\stackrel{\text{def}}{=} \lambda ks.[\![M]\!]([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: ks) \\
\\
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UComp} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &\stackrel{\text{def}}{=} \lambda x.\lambda ks.\mathbf{let}\ (h :: ks') = ks\ \mathbf{in}\ [\![N]\!]\,ks' \\
[\![\{\ell\ p\ r \mapsto N_\ell\}_{\ell \in \mathcal{L}}]\!] &\stackrel{\text{def}}{=} \lambda\langle z, \langle p, r\rangle\rangle.\lambda ks.\mathbf{case}\ z\ \{(\ell \mapsto [\![N_\ell]\!]\,ks)_{\ell \in \mathcal{L}}; \\
&\qquad\qquad y \mapsto M_{\mathrm{forward}}((y,p,r),ks)\} \\
M_{\mathrm{forward}}((y,p,r),ks) &\stackrel{\text{def}}{=} \mathbf{let}\ (k' :: h' :: ks') = ks\ \mathbf{in} \\
&\qquad\qquad h'\,\langle y, \langle p, \lambda x.\lambda ks''.r\,x\,(k' :: h' :: ks'')\rangle\rangle\,ks' \\
\\
\top[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
\top[\![M]\!] &\stackrel{\text{def}}{=} [\![M]\!]\ ((\lambda x.\lambda ks.x) :: (\lambda\langle z, \langle p, r\rangle\rangle.\lambda ks.\mathbf{absurd}\ z) :: [])
\end{aligned}
$$

The other cases are as in the original CPS translation in Figure 4.2. Since we now use a list representation for the stacks of continuations, we have had to modify the translations of all the constructs that manipulate continuations. For **return** and **let**, we extract the top continuation k and manipulate it analogously to the original translation in Figure 4.2. For **do**, we extract the top pure continuation k and effect continuation h and invoke h in the same way as the curried translation, except that we explicitly maintain the stack ks of additional continuations. The translation of **handle**, however, pushes a continuation pair onto the stack instead of supplying them as arguments. Handling of operations is the same as before, except for explicit passing of the ks. Forwarding now pattern matches on the stack to extract the next continuation pair, rather than accepting them as arguments.

Let us revisit the example from Section 4.3.1 to see first hand that our refined translation makes the example properly tail-recursive.

$$
\begin{aligned}
\top[\![\mathbf{return}\ \langle\rangle]\!] &= (\lambda(k :: ks).k\,\langle\rangle\,ks)\,((\lambda x.\lambda ks.x) :: (\lambda\langle z, \_\rangle.\lambda ks.\mathbf{absurd}\,z) :: []) \\
&\rightsquigarrow (\lambda x.\lambda ks.x)\,\langle\rangle\,((\lambda\langle z, \_\rangle.\lambda ks.\mathbf{absurd}\,z) :: []) \\
&\rightsquigarrow \langle\rangle
\end{aligned}
$$

The reduction sequence in the image of this uncurried translation has one fewer steps (disregarding the administrative steps induced by pattern matching) than in the image

of the curried translation. The 'missing' step is precisely the reduction marked ($*$4.2), which was a partial application of the initial pure continuation function that was not in tail position. Note, however, that the first reduction (corresponding to ($*$4.1)) remains administrative, the reduction is entirely static, and as such, it can be dealt with as part of the translation.

Administrative redexes We can determine whether a redex is administrative in the image by determining whether it corresponds to a redex in the preimage. If there is no corresponding redex, then the redex is said to be administrative. We can further classify an administrative redex as to whether it is *static* or *dynamic*.

A static administrative redex is a by-product of the translation that does not contribute to the implementation of the dynamic behaviour of the preimage. The separation between value and computation terms in fine-grain call-by-value makes it evident where static administrative redexes can arise. They arise from computation terms, which can clearly be seen from the translation where each computation term induces a λ-abstraction. Each induced λ-abstraction must necessarily be eliminated by a unary application. These unary applications are administrative; they do not correspond to reductions in the preimage. The applications that do correspond to reductions in the preimage are the binary (continuation) applications.

A dynamic administrative redex is a genuine implementation detail that supports some part of the dynamic behaviour of the preimage. An example of such a detail is the implementation of effect forwarding. In λ_{h} effect forwarding involves no auxiliary reductions, any operation invocation is instantaneously dispatched to a suitable handler (if such one exists). The translation presented above realises effect forwarding by explicitly applying the next effect continuation. This application is an example of a dynamic administrative reduction. Not every dynamic administrative reduction is necessary, though. For instance, the implementation of resumptions as a composition of λ-abstractions gives rise to administrative reductions upon invocation. As we shall see in Section 4.3.3 administrative reductions due to resumption invocation can be dealt with by choosing a more clever implementation of resumptions.

4.3.3 Resumptions as explicit reversed stacks

Thus far resumptions have been represented as functions, and forwarding has been implemented using function composition. The composition of resumption gives rise

to unnecessary dynamic administrative redexes as function composition necessitates the introduction of an additional lambda abstraction. As an illustration of how and where these administrative redexes arise let us consider an example with an operation $\mathsf{Ask} : \langle\rangle \twoheadrightarrow \mathsf{Int}$ and two handlers H_{Reader} and H_{Other} such that $H^{\mathsf{Ask}}_{\mathsf{Reader}} = \{\langle\!\langle \mathsf{Ask}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto r\ 42\}$ whilst $\mathsf{Ask} \notin \mathsf{dom}(H_{\mathsf{Other}})$. We denote the top-level continuation by $ks_\top$.

$$
\begin{aligned}
& \top[\![\mathbf{handle}\ (\mathbf{handle\ do}\ \mathsf{Ask}\,\langle\rangle\ \mathbf{with}\ H_{\mathsf{Other}})\ \mathbf{with}\ H_{\mathsf{Reader}}]\!] \\
= \quad & \text{(definition of } \top[\![-]\!]) \\
& (\lambda ks.(\lambda ks'.(\lambda(k :: h :: ks'').h\,\langle \mathsf{Ask}, \langle\rangle, \lambda x.\lambda ks'''.k\ x\ (h :: ks''')\rangle\rangle\, ks'') \\
& \qquad ([\![H^{\mathrm{ret}}_{\mathsf{Other}}]\!] :: [\![H^{\mathrm{ops}}_{\mathsf{Other}}]\!] :: ks')) \\
& \quad ([\![H^{\mathrm{ret}}_{\mathsf{Reader}}]\!] :: H^{\mathrm{ops}}_{\mathsf{Reader}} :: ks))\, ks_\top \\
\rightsquigarrow^* \quad & \text{(multiple applications of U-App, activation of } H_{\mathsf{Other}}) \\
& [\![H^{\mathrm{ops}}_{\mathsf{Other}}]\!]\,\langle \mathsf{Ask}, \langle\rangle, \lambda x.\lambda ks'''.[\![H^{\mathrm{ret}}_{\mathsf{Other}}]\!]\ x\ ([\![H^{\mathrm{ops}}_{\mathsf{Other}}]\!] :: ks''')\rangle\rangle\ ([\![H^{\mathrm{ret}}_{\mathsf{Reader}}]\!] :: H^{\mathrm{ops}}_{\mathsf{Reader}} :: ks_\top) \\
\rightsquigarrow^* \quad & \text{(effect forwarding to } H_{\mathsf{Reader}}) \\
& H^{\mathrm{ops}}_{\mathsf{Reader}}\,\langle \mathsf{Ask}, \langle\rangle, \lambda x.\lambda ks''.r_{\mathsf{admin}}\ x\ (H^{\mathrm{ret}}_{\mathsf{Reader}} :: H^{\mathrm{ops}}_{\mathsf{Reader}} :: ks'')\rangle\rangle\, ks_\top \\
& \mathbf{where}\ r_{\mathsf{admin}} \stackrel{\mathrm{def}}{=} \lambda x.\lambda ks'''.[\![H^{\mathrm{ret}}_{\mathsf{Other}}]\!]\ x\ ([\![H^{\mathrm{ops}}_{\mathsf{Other}}]\!] :: ks''') \\
\rightsquigarrow^* \quad & \text{(invocation of the administrative resumption)} \\
& r_{\mathsf{admin}}\ 42\ (H^{\mathrm{ret}}_{\mathsf{Reader}} :: H^{\mathrm{ops}}_{\mathsf{Reader}} :: ks_\top) \\
\rightsquigarrow^* \quad & \text{(invocation of the resumption of the operation invocation site)} \\
& [\![H^{\mathrm{ret}}_{\mathsf{Other}}]\!]\ 42\ ([\![H^{\mathrm{ops}}_{\mathsf{Other}}]\!] :: H^{\mathrm{ret}}_{\mathsf{Reader}} :: H^{\mathrm{ops}}_{\mathsf{Reader}} :: ks_\top)
\end{aligned}
$$

Effect forwarding introduces the administrative abstraction r_{admin}, whose sole purpose is to forward the interpretation of the operation to the operation invocation site. In a certain sense r_{admin} is a sort of identity frame. The insertion of identities ought to always trigger the alarm bells as an identity computation is typically extraneous. The amount of identity frames being generated scales linearly with the number of handlers the operation needs to pass through before reaching a suitable handler.

We can avoid generating these administrative resumption redexes by applying a variation of the technique that we used in the previous section to uncurry the curried CPS translation. Rather than representing resumptions as functions, we move to an explicit representation of resumptions as *reversed* stacks of pure and effect continuations. By choosing to reverse the order of pure and effect continuations, we can construct resumptions efficiently using regular cons-lists. We augment the syntax and semantics of λ_{u} with a computation term $\mathbf{let}\ r = \mathbf{res}\, V\ \mathbf{in}\ N$ which allow us to convert these reversed stacks to actual functions on demand.

$$\textsf{U-Res} \qquad \mathbf{let}\ r = \mathbf{res}\,(V_n :: \ldots :: V_1 :: []) \ \mathbf{in}\ N \ \rightsquigarrow\ N[\lambda x k.V_1\, x\, (V_2 :: \ldots :: V_n :: k)/r]$$

This reduction rule reverses the stack, extracts the top continuation V_1, and prepends the remainder onto the current stack W. The stack representing a resumption and the remaining stack W are reminiscent of the zipper data structure for representing cursors in lists [130]. Thus we may think of resumptions as representing pointers into the stack of handlers. The translations of **do**, handling, and forwarding need to be modified to account for the change in representation of resumptions.

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \lambda k :: h :: ks.\, h\,\langle \ell, \langle [\![V]\!], h :: k :: [] \rangle \rangle\, ks \\
\\
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UComp} \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell \in \mathcal{L}}\}]\!] &\stackrel{\text{def}}{=} \lambda \langle z, \langle p, rs \rangle \rangle.\lambda ks.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}\ rs\ \mathbf{in}\ [\![N_\ell]\!]\, ks)_{\ell \in \mathcal{L}}; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \mapsto M_{\text{forward}}((y,p,rs),ks)\} \\
M_{\text{forward}}((y,p,rs),ks) &\stackrel{\text{def}}{=} \mathbf{let}\ (k' :: h' :: ks') = ks\ \mathbf{in}\ h'\,\langle y, \langle p, h' :: k' :: rs \rangle \rangle\, ks'
\end{aligned}
$$

The translation of **do** constructs an initial resumption stack, operation clauses extract and convert the current resumption stack into a function using the **res** construct, and M_{forward} augments the current resumption stack with the current continuation pair.

4.3.4 Higher-order translation for deep effect handlers

In the previous sections, we have seen step-wise refinements of the initial curried CPS translation for deep effect handlers (Section 4.3.1) to be properly tail-recursive (Section 4.3.2) and to avoid yielding unnecessary dynamic administrative redexes for resumptions (Section 4.3.3). There is still one outstanding issue, namely, that the translation yields static administrative redexes. In this section we will further refine the CPS translation to eliminate all static administrative redexes at translation time. Specifically, the translation will be adapted to a higher-order one-pass CPS translation [62] that partially evaluates administrative redexes at translation time. Following Danvy and Nielsen [67], I will use a two-level lambda calculus notation to distinguish between *static* lambda abstraction and application in the meta language and *dynamic* lambda abstraction and application in the target language. To disambiguate syntax constructors in the respective calculi I will mark static constructors with a blue overline, whilst dynamic constructors are marked with a red underline. The principal idea is that redexes marked as static are reduced as part of the translation, whereas those marked as dynamic are reduced at runtime. To facilitate this notation I will write application explicitly using an infix "at" symbol (@) in both calculi.

Values

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Val} \to \mathrm{UVal} \\
[\![x]\!] &\stackrel{\text{def}}{=} x \\
[\![\lambda x.M]\!] &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\mathbf{let}\ (k \mathbin{\underline{::}} h \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h \mathbin{\overline{::}} \uparrow ks') \\
[\![\Lambda\alpha.M]\!] &\stackrel{\text{def}}{=} \underline{\lambda} \langle\rangle\, ks.\mathbf{let}\ (k \mathbin{\underline{::}} h \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h \mathbin{\overline{::}} \uparrow ks') \\
[\![\langle\rangle]\!] &\stackrel{\text{def}}{=} \langle\rangle \\
[\![\langle \ell = V; W\rangle]\!] &\stackrel{\text{def}}{=} \langle \ell = [\![V]\!]; [\![W]\!]\rangle \\
[\![\ell\ V]\!] &\stackrel{\text{def}}{=} \ell\ [\![V]\!]
\end{aligned}
$$

Computations

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{SVal}^* \to \mathrm{UComp} \\
[\![V\,W]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.[\![V]\!]\ \underline{@}\ [\![W]\!]\ \underline{@} \downarrow\kappa \\
[\![V\,T]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.[\![V]\!]\ \underline{@}\ \langle\rangle\ \underline{@} \downarrow\kappa \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.\mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ \kappa \\
[\![\mathbf{case}\ V\ \{\ell\ x \mapsto M; y \mapsto N\}]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.\mathbf{case}\ [\![V]\!]\ \{\ell\ x \mapsto [\![M]\!]\ \overline{@}\ \kappa; y \mapsto [\![N]\!]\ \overline{@}\ \kappa\} \\
[\![\mathbf{absurd}\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.\mathbf{absurd}\ [\![V]\!] \\
[\![\mathbf{return}\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\theta \mathbin{\overline{::}} \kappa.\downarrow\theta\ \underline{@}\ [\![V]\!]\ \underline{@} \downarrow\kappa \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\theta \mathbin{\overline{::}} \kappa.[\![M]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.\mathbf{let}\ (h \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
&\qquad\qquad [\![N]\!]\ \overline{@}\ (\theta \mathbin{\overline{::}} \uparrow h \mathbin{\overline{::}} \uparrow ks'))) \mathbin{\overline{::}} \kappa) \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\theta \mathbin{\overline{::}} \chi \mathbin{\overline{::}} \kappa.\downarrow\chi\ \underline{@}\ \langle \ell, \langle [\![V]\!], \downarrow\chi \mathbin{\underline{::}} \downarrow\theta \mathbin{\underline{::}} [\underline{]}\rangle\rangle\ \underline{@} \downarrow\kappa \\
[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.[\![M]\!]\ \overline{@}\ (\uparrow[\![H^{\mathrm{ret}}]\!] \mathbin{\overline{::}} \uparrow[\![H^{\mathrm{ops}}]\!] \mathbin{\overline{::}} \kappa)
\end{aligned}
$$

Handler definitions

$$
\begin{aligned}
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UVal} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\ \mathbf{let}\ (h \mathbin{\underline{::}} k \mathbin{\underline{::}} h' \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
&\qquad\qquad [\![N]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h' \mathbin{\overline{::}} \uparrow ks') \\
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell \in \mathcal{L}}\}]\!] &\stackrel{\text{def}}{=} \underline{\lambda}\langle z, \langle p, rs\rangle\rangle\, ks.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}\ rs\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ (k \mathbin{\underline{::}} h \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
&\qquad\qquad [\![N_\ell]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h \mathbin{\overline{::}} \uparrow ks'))_{\ell \in \mathcal{L}}; \\
&\qquad\qquad y \mapsto M_{\mathrm{forward}}((y,p,rs),ks)\} \\
M_{\mathrm{forward}}((y,p,rs),ks) &\stackrel{\text{def}}{=} \mathbf{let}\ (k' \mathbin{\underline{::}} h' \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ h'\ \underline{@}\ \langle y, \langle p, h' \mathbin{\underline{::}} k' \mathbin{\underline{::}} rs\rangle\rangle\ \underline{@}\ ks'
\end{aligned}
$$

Top level program

$$
\begin{aligned}
\top[\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
\top[\![M]\!] &= [\![M]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) \mathbin{\overline{::}} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) \mathbin{\overline{::}} \uparrow[\underline{]})
\end{aligned}
$$

Figure 4.4: Higher-order uncurried CPS translation of λ_{h}.

Static terms As in the dynamic target language, continuations are represented as alternating lists of pure continuation functions and effect continuation functions. To ease notation I will make use of pattern matching notation. The static meta language is generated by the following productions.

Static patterns	$\mathcal{P} \in \mathrm{SPat} ::= \kappa \mid \theta \mathbin{\overline{::}} \mathcal{P}$
Static values	$\mathcal{V}, \mathcal{W} \in \mathrm{SVal} ::= \uparrow V \mid \mathcal{V} \mathbin{\overline{::}} \mathcal{W} \mid \overline{\lambda}\mathcal{P}.\mathcal{M}$
Static computations	$\mathcal{M} \in \mathrm{SComp} ::= \mathcal{V} \mid \mathcal{V} \mathbin{\overline{@}} \mathcal{W} \mid \mathcal{V} \mathbin{\underline{@}} V \mathbin{\underline{@}} W$

The patterns comprise only static list deconstructing. We let $\mathcal{P}$ range over static patterns. The static values comprise reflected dynamic values, static lists, and static lambda abstractions. We let $\mathcal{V}, \mathcal{W}$ range over meta language values; by convention we shall use variables θ to denote statically known pure continuations, χ to denote statically known effect continuations, and κ to denote statically known continuations. I shall use $\mathcal{M}$ to range over static computations, which comprise static values, static application and binary dynamic application of a static value to two dynamic values. Static computations are subject to the following equational axioms.

$$(\overline{\lambda}\kappa.\mathcal{M}) \mathbin{\overline{@}} \mathcal{V} \stackrel{\text{def}}{=} \mathcal{M}[\mathcal{V}/\kappa]$$
$$(\overline{\lambda}\theta \mathbin{\overline{::}} \kappa.\mathcal{M}) \mathbin{\overline{@}} (\mathcal{V} \mathbin{\overline{::}} \mathcal{W}) \stackrel{\text{def}}{=} (\overline{\lambda}\kappa.\mathcal{M}[\mathcal{V}/\theta]) \mathbin{\overline{@}} \mathcal{W}$$

The first equation is static β-equivalence, it states that applying a static lambda abstraction with binder κ and body $\mathcal{M}$ to a static value $\mathcal{V}$ is equal to substituting $\mathcal{V}$ for κ in $\mathcal{M}$. The second equation provides a means for applying a static lambda abstraction to a static list component-wise.

Reflected static values are reified as dynamic language values $\downarrow\mathcal{V}$ by induction on their structure.

$$\downarrow\uparrow V \stackrel{\text{def}}{=} V \qquad \downarrow(\mathcal{V} \mathbin{\overline{::}} \mathcal{W}) \stackrel{\text{def}}{=} \downarrow\mathcal{V} \mathbin{\underline{::}} \downarrow\mathcal{W}$$

Higher-order translation As we shall see this translation manipulates the continuation intricate ways; and since we maintain the interpretation of the continuation as an alternating list of pure continuation functions and effect continuation functions it is useful to define the 'parity' of a continuation as follows: a continuation is said to be *odd* if the top element is an effect continuation function, otherwise it is said to *even*.

The complete CPS translation is given in Figure 4.4. In essence, it is the same as the refined first-order uncurried CPS translation, although the notation is slightly more involved due to the separation of static and dynamic parts.

As before, the translation comprises three translation functions, one for each syntactic category: values, computations, and handler definitions. Amongst the three functions, the translation function for computations stands out, because it is the only one that operates on static continuations. Its type signature, $[\![-]\!] : \mathrm{Comp} \to \mathrm{SVal}^* \to \mathrm{UComp}$, signifies that it is a binary function, taking a λ_{h}-computation term as its first argument and a static continuation (a list of static values) as its second argument, and ultimately produces a λ_{u}-computation term. Thus the computation translation function is able to manipulate the continuation. In fact, the translation is said to be higher-order because the continuation parameter is a higher-order: it is a list of functions.

To ensure that static continuation manipulation is well-defined the translation maintains the invariant that the statically known continuation stack (θ) always contains at least two continuation functions, i.e. a complete continuation pair consisting of a pure continuation function and an effect continuation function. This invariant guarantees that all translations are uniform in whether they appear statically within the scope of a handler or not, and this also simplifies the correctness proof (Theorem 4.8). Maintaining this invariant has a cosmetic effect on the presentation of the translation. This effect manifests in any place where a dynamically known continuation stack is passed in (as a continuation parameter *ks*), as it must be deconstructed using a dynamic language **let** to expose the continuation structure and subsequently reconstructed as a static value with reflected variable names.

The translation of λ-abstractions provides an example of this deconstruction and reconstruction in action. The dynamic continuation *ks* is deconstructed to expose to the next pure continuation function *k* and effect continuation *h*, and the remainder of the continuation ks'; these names are immediately reflected and put back together to form a static continuation that is provided to the translation of the body computation *M*.

The only translation rule that consumes a complete reflected continuation pair is the translation of **do**. The effect continuation function, χ, is dynamically applied to an operation package and the reified remainder of the continuation κ. As usual, the operation package contains the payload and the resumption, which is represented as a reversed continuation slice. The only other translation rules that manipulate the continuation are **return** and **let**, which only consume the pure continuation function θ. For example, the translation of **return** is a dynamic application of θ to the translation of the value *V* and the remainder of the continuation κ. The shape of κ is odd, meaning that the top element is an effect continuation function. Thus the pure continuation θ has to account for this odd shape. Fortunately, the possible instantiations of the pure continuation are few.

We can derive the all possible instantiations systematically by using the operational semantics of λ_h. According to the operational semantics the continuation of a **return**-computation is either the continuation of a **let**-expression or a **return**-clause (a bare top-level **return**-computation is handled by the $\top[\![-]\!]$ translation). The translations of **let**-expressions and **return**-clauses each account for odd continuations. For example, the translation of **let** consumes the current pure continuation function and generates a replacement: a pure continuation function which expects an odd dynamic continuation ks, which it deconstructs to expose the effect continuation h along with the current pure continuation function in the translation of N. The modified continuation is passed to the translation of M. To provide a flavour of how this continuation manipulation functions in practice, consider the following example term.

$$
\begin{array}{ll}
 & \top[\![\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N]\!] \\
= & \quad \text{(definition of } \top[\![-]\!]\text{)} \\
 & (\overline{\lambda}\theta :\!\!\overline{:} \kappa.[\![\mathbf{return}\ V]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.\mathbf{let}\ (h :\!\!\underline{:} ks') = ks\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\theta :\!\!\overline{:} \uparrow h :\!\!\overline{:} \uparrow ks'))) :\!\!\overline{:} \kappa) \\
 & \overline{@}(\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\overline{:} \uparrow\underline{[]})) \\
= & \quad \text{(definition of } [\![-]\!]\text{)} \\
 & (\overline{\lambda}\theta :\!\!\overline{:} \kappa.(\overline{\lambda}\theta :\!\!\overline{:} \kappa.\downarrow\theta\ \underline{@}\ [\![V]\!]\ \underline{@}\ \downarrow\kappa)\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.\mathbf{let}\ (h :\!\!\underline{:} ks') = ks\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\theta :\!\!\overline{:} \uparrow h :\!\!\overline{:} \uparrow ks'))) :\!\!\overline{:} \kappa) \\
 & \overline{@}(\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\overline{:} \uparrow\underline{[]})) \\
= & \quad \text{(static } \beta\text{-reduction)} \\
 & (\overline{\lambda}\theta :\!\!\overline{:} \kappa.\downarrow\theta\ \underline{@}\ [\![V]\!]\ \underline{@}\ \downarrow\kappa)\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.\mathbf{let}\ (h :\!\!\underline{:} ks') = ks\ \mathbf{in} \\
 & \qquad\qquad [\![N]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow h :\!\!\overline{:} \uparrow ks')) \\
 & \qquad\qquad :\!\!\overline{:} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\overline{:} \uparrow\underline{[]})) \\
= & \quad \text{(static } \beta\text{-reduction)} \\
 & \quad (\underline{\lambda} x\, ks.\mathbf{let}\ (h :\!\!\underline{:} ks') = ks\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow h :\!\!\overline{:} \uparrow ks')) \\
 & \underline{@}\ [\![V]\!]\ \underline{@}\ ((\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\underline{:} \underline{[]}) \\
\rightsquigarrow & \quad (\mathsf{U\text{-}App}_2) \\
 & \mathbf{let}\ (h :\!\!\underline{:} ks') = (\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\underline{:} \underline{[]}\ \mathbf{in}\ [\![N[V/x]]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow h :\!\!\overline{:} \uparrow ks')) \\
\rightsquigarrow^{+} & \quad \text{(dynamic pattern matching and substitution)} \\
 & [\![N[V/x]]\!]\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) :\!\!\overline{:} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\overline{:} \uparrow\underline{[]})
\end{array}
$$

The translation of **return** provides the generated dynamic pure continuation function with the odd continuation $((\underline{\lambda} z\, ks.\mathbf{absurd}\ z) :\!\!\underline{:} \underline{[]})$. After the $\mathsf{U\text{-}App}_2$ reduction, the pure continuation function deconstructs the odd continuation in order to bind the current effect continuation function to the name h, which would have been used during the

translation of N.

The translation of **handle** applies the translation of M to the current continuation extended with the translation of the **return**-clause, acting as a pure continuation function, and the translation of operation-clauses, acting as an effect continuation function. The translation of a **return**-clause discards the effect continuation h and in addition exposes the next pure continuation k and effect continuation h' which are reflected to form a static continuation for the translation of N. The translation of operation clauses unpacks the provided operation package to perform a case-split on the operation label z. The branch for ℓ deconstructs the continuation ks in order to expose the continuation structure. The forwarding branch also deconstructs the continuation, but for a different purpose; it augments the resumption rs with the next pure and effect continuation functions.

Let us revisit the example from Section 4.3.1 to see that the higher-order translation eliminates the static redex at translation time.

$$\begin{aligned}
\top[\![\mathbf{return}\ \langle\rangle]\!] &= (\overline{\lambda}\theta \mathbin{\overline{::}} \kappa.\theta\ \underline{@}\ \langle\rangle\ \underline{@} \downarrow\kappa)\ \overline{@}\ (\uparrow(\underline{\lambda} x\, ks.x) \mathbin{\overline{::}} \uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) \mathbin{\overline{::}} \uparrow\underline{[]}) \\
&= (\underline{\lambda} x\, ks.x)\ \underline{@}\ \langle\rangle\ \underline{@}\ (\uparrow(\underline{\lambda} z\, ks.\mathbf{absurd}\ z) \mathbin{\underline{::}} \underline{[]}) \\
&\rightsquigarrow \langle\rangle
\end{aligned}$$

In contrast with the previous translations, the reduction sequence in the image of this translation contains only a single dynamic reduction (disregarding the dynamic administrative reductions arising from continuation construction and deconstruction); both ($*$4.1) and ($*$4.2) reductions have been eliminated as part of the translation.

The elimination of static redexes coincides with a refinements of the target calculus. Unary application is no longer a necessary primitive. Every unary application dealt with by the metalanguage, i.e. all unary applications are static.

Implicit lazy continuation deconstruction An alternative to the explicit deconstruction of continuations is to implicitly deconstruct continuations on demand when static pattern matching fails. I took this approach in Hillerström et al. [121]. On one hand this approach leads to a slightly slicker presentation. On the other hand it complicates the proof of correctness as one must account for static pattern matching failure. A practical argument in favour of the explicit eager continuation deconstruction is that it is more accessible from an implementation point of view. No implementation details are hidden away in side conditions. Also, it is not clear that lazy deconstruction has any advantage over eager deconstruction, as the translation must reify the continuation when it trans-

itions from computations to values and reflect the continuation when it transitions from values to computations, in which case static pattern matching would fail.

Correctness

We establish the correctness of the higher-order uncurried CPS translation via a simulation result in the style of Plotkin [220] (Theorem 4.8). Before we can state and prove this result, we first several auxiliary lemmas describing how translated terms behave. First, the higher-order CPS translation commutes with substitution.

Lemma 4.2 (Substitution)**.** *The higher-order uncurried CPS translation commutes with substitution in value terms*

$$[\![W]\!][[\![V]\!]/x] = [\![W[V/x]]\!],$$

and with substitution in computation terms

$$([\![M]\!] \mathbin{\overline{@}} (\theta :: \chi :: \kappa))[[\![V]\!]/x] = [\![M[V/x]]\!] \mathbin{\overline{@}} (\theta :: \chi :: \kappa)[[\![V]\!]/x],$$

and with substitution in handler definitions

$$[\![H^{\mathrm{ret}}]\!][[\![V]\!]/x] = [\![H^{\mathrm{ret}}[V/x]]\!],$$
$$[\![H^{\mathrm{ops}}]\!][[\![V]\!]/x] = [\![H^{\mathrm{ops}}[V/x]]\!].$$

Proof. By mutual induction on the structure of W, M, H^{ret}, and H^{ops}. □

It follows as a corollary that top-level substitution is well-behaved.

Corollary 4.3 (Top-level substitution)**.**

$$\top[\![M]\!][[\![V]\!]/x] = \top[\![M[V/x]]\!].$$

Proof. Follows immediately by the definitions of $\top[\![-]\!]$ and Lemma 4.2. □

In order to reason about the behaviour of the S-Op rule, which is defined in terms of an evaluation context, we need to extend the CPS translation to evaluation contexts.

$$\begin{aligned}
[\![-]\!] &: \mathrm{Cont} \to \mathrm{SVal} \\
[\![[\,]]\!] &\stackrel{\mathrm{def}}{=} \overline{\lambda}\kappa.\kappa \\
[\![\mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N]\!] &\stackrel{\mathrm{def}}{=} \overline{\lambda}\theta :: \kappa.[\![\mathcal{E}]\!] \mathbin{\overline{@}} (\uparrow(\underline{\lambda} x\, ks.\mathbf{let}\ (h :: ks') = ks\ \mathbf{in} \\
&\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\theta :: \uparrow h :: \uparrow ks')) :: \kappa) \\
[\![\mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H]\!] &\stackrel{\mathrm{def}}{=} \overline{\lambda}\kappa.[\![\mathcal{E}]\!] \mathbin{\overline{@}} ([\![H^{\mathrm{ret}}]\!] :: [\![H^{\mathrm{ops}}]\!] :: \kappa)
\end{aligned}$$

The following lemma is the characteristic property of the CPS translation on evaluation contexts. It provides a means for decomposing an evaluation context, such that we can focus on the computation contained within the evaluation context.

Lemma 4.4 (Decomposition)**.**

$$[\![\mathcal{E}[M]]\!] \mathbin{\overline{@}} (\mathcal{V} :\!\!\overline{:}\; \mathcal{W}) = [\![M]\!] \mathbin{\overline{@}} ([\![\mathcal{E}]\!] \mathbin{\overline{@}} (\mathcal{V} :\!\!\overline{:}\; \mathcal{W}))$$

Proof. By structural induction on the evaluation context $\mathcal{E}$. □

Even though we have eliminated the static administrative redexes, we still need to account for the dynamic administrative redexes that arise from pattern matching against a reified continuation. To properly account for these administrative redexes it is convenient to treat list pattern matching as a primitive in λ_u, therefore we introduce a new reduction rule U-SplitList in λ_u.

$$\textsf{U-SplitList} \qquad \textbf{let}\ (k :\!\!\underline{:}\; ks) = V :\!\!\underline{:}\; W\ \textbf{in}\ M \ \rightsquigarrow\ M[V/k, W/ks]$$

Note this rule is isomorphic to the U-Split rule with lists encoded as right nested pairs using unit to denote nil. We write $\rightsquigarrow_a$ for the compatible closure of U-SplitList.

We also need to be able to reason about the computational content of reflection after reification. By definition we have that $\downarrow\uparrow V = V$, the following lemma lets us reason about the inverse composition.

Lemma 4.5 (Reflect after reify)**.** *Reflection after reification may give rise to dynamic administrative reductions, i.e.*

$$[\![M]\!] \mathbin{\overline{@}} (\mathcal{V}_1 :\!\!\overline{:} \ldots \mathcal{V}_n :\!\!\overline{:} \uparrow\downarrow \mathcal{W}) \rightsquigarrow_a^* [\![M]\!] \mathbin{\overline{@}} (\mathcal{V}_1 :\!\!\overline{:} \ldots \mathcal{V}_n :\!\!\overline{:} \mathcal{W})$$

Proof. By induction on the structure of M. □

We next observe that the CPS translation simulates forwarding.

Lemma 4.6 (Forwarding)**.** *If $\ell \notin dom(H_1)$ then*

$$[\![H_1^{\text{ops}}]\!] \mathbin{\underline{@}} \langle \ell, \langle U, V \rangle \rangle \mathbin{\underline{@}} (V_2 :\!\!\underline{:}\; [\![H_2^{\text{ops}}]\!] :\!\!\underline{:}\; W) \rightsquigarrow^+ [\![H_2^{\text{ops}}]\!] \mathbin{\underline{@}} \langle \ell, \langle U, [\![H_2^{\text{ops}}]\!] :\!\!\underline{:}\; V_2 :\!\!\underline{:}\; V \rangle \rangle \mathbin{\underline{@}} W$$

Proof. By direct calculation. □

Now we show that the translation simulates the S-Op rule.

Lemma 4.7 (Handling)**.** *If $\ell \notin BL(\mathcal{E})$ and $H^\ell = \{\ell\, p\, r \mapsto N_\ell\}$ then*

$$\begin{aligned}&[\![\textbf{do}\ \ell\ V]\!] \mathbin{\overline{@}} ([\![\mathcal{E}]\!] \mathbin{\overline{@}} (\uparrow[\![H^{\text{ret}}]\!] :\!\!\overline{:} \uparrow[\![H^{\text{ops}}]\!] :\!\!\overline{:} \mathcal{V})) \rightsquigarrow^+ \rightsquigarrow_a^* \\ &\quad ([\![N_\ell]\!] \mathbin{\overline{@}} \mathcal{V})[[\![V]\!]/p, (\lambda y\, ks.[\![\textbf{return}\ y]\!] \mathbin{\overline{@}} ([\![\mathcal{E}]\!] \mathbin{\overline{@}} (\uparrow[\![H^{\text{ret}}]\!] :\!\!\overline{:} \uparrow[\![H^{\text{ops}}]\!] :\!\!\overline{:} \uparrow ks)))/]\end{aligned}$$

Proof. Follows from Lemmas 4.4, 4.5, and 4.6. □

Finally, we have the ingredients to state and prove the simulation result. The following theorem shows that the only extra behaviour exhibited by a translated term is the bureaucracy of deconstructing the continuation stack.

Theorem 4.8 (Simulation)**.** *If* $M \rightsquigarrow N$ *then* $\top[\![M]\!] \rightsquigarrow^{+} \rightsquigarrow_a^{*} \top[\![N]\!]$.

Proof. By case analysis on the reduction relation using Lemmas 4.4–4.7. The S-Op case follows from Lemma 4.7. □

4.4 Transforming shallow effect handlers

In this section we will continue to build upon the higher-order uncurried CPS translation (Section 4.3.4) in order to add support for shallow handlers. The dynamic nature of shallow handlers pose an interesting challenge, because unlike deep resumption capture, a shallow resumption capture discards the handler leaving behind a dangling pure continuation. The dangling pure continuation must be 'adopted' by whichever handler the resumption invocation occur under. This handler is determined dynamically by the context, meaning the CPS translation must be able to modify continuation pairs.

In Section 4.4.1 I will discuss an attempt at a 'natural' extension of the higher-order uncurried CPS translation for deep handlers, but for various reasons this extension is flawed. However, the insights gained by attempting this extension leads to yet another change of the continuation representation (Section 4.4.2) resulting in the notion of a *generalised continuation*. In Section 4.4.4 we will see how generalised continuations provide a basis for implementing deep and shallow effect handlers simultaneously, solving all of the problems encountered thus far uniformly.

4.4.1 A specious attempt

Initially it is tempting to try to extend the interpretation of the continuation representation in the higher-order uncurried CPS translation for deep handlers to squeeze in shallow handlers. The main obstacle one encounters is how to decouple a pure continuation from its handler such that a it can later be picked up by another handler.

To fully uninstall a handler, we must uninstall both the pure continuation function corresponding to its return clause and the effect continuation function corresponding to its operation clauses. In the current setup it is impossible to reliably uninstall the former as due to the translation of **let**-expressions it may be embedded arbitrarily deep within

the current pure continuation and the extensional representation of pure continuations means that we cannot decompose them.

A quick fix to this problem is to treat pure continuation functions arising from return clauses separately from pure continuation functions arising from **let**-expressions. Thus we can interpret the continuation as a sequence of triples consisting of two pure continuation functions followed by an effect continuation function, where the first pure continuation function corresponds the continuation induced by some **let**-expression and the second corresponds to the return clause of the current handler. To distinguish between the two kinds of pure continuations, we shall write χ^{ret} for statically known pure continuations arising from return clauses, and h^{ret} for dynamically known ones. Similarly, we write χ^{ops} and h^{ops}, respectively, for statically and dynamically, known effect continuations. With this notation in mind, we may translate operation invocation and handler installation using the new interpretation of the continuation representation as follows.

$$\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{SVal}^* \to \mathrm{UComp} \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\theta \mathbin{\overline{::}} \chi^{\mathrm{ret}} \mathbin{\overline{::}} \chi^{\mathrm{ops}} \mathbin{\overline{::}} \kappa.{\downarrow}\chi^{\mathrm{ops}} \mathbin{\underline{@}} \langle \ell, \langle [\![V]\!], {\downarrow}\chi^{\mathrm{ops}} \mathbin{\underline{::}} {\downarrow}\chi^{\mathrm{ret}} \mathbin{\underline{::}} {\downarrow}\theta \mathbin{\underline{::}} [] \rangle \rangle \\
&\qquad\qquad \mathbin{\underline{@}} {\downarrow}\kappa \\
[\![\mathbf{handle}^{\dagger}\ M\ \mathbf{with}\ H]\!] &\stackrel{\text{def}}{=} \overline{\lambda}\kappa.[\![M]\!]\ \overline{@}\ (\uparrow V_{\mathrm{id}} \mathbin{\overline{::}} \uparrow [\![H^{\mathrm{ret}}]\!] \mathbin{\overline{::}} \uparrow [\![H^{\mathrm{ops}}]\!]^{\dagger} \mathbin{\overline{::}} \kappa) \\
V_{\mathrm{id}} &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\mathbf{let}\ (h^{\mathrm{ret}} \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ h^{\mathrm{ret}} \mathbin{\underline{@}} x \mathbin{\underline{@}} ks'
\end{aligned}$$

The only change to the translation of operation invocation is the extra bureaucracy induced by the additional pure continuation. The translation of handler installation is a little more interesting as it must make up an initial pure continuation in order to maintain the sequence of triples interpretation of the continuation structure. As the initial pure continuation we use the administrative function V_{id}, which amounts to a dynamic variation of the translation rule for the trivial computation term **return**: it invokes the next pure continuation with whatever value it was provided.

Although, I will not demonstrate it here, the translation rules for λ-abstractions, Λ-abstractions, and **let**-expressions must also be adjusted accordingly to account for the extra bureaucracy. The same is true for the translation of **return**-clause, thus it is rather straightforward to adapt it to the new continuation interpretation.

$$\begin{aligned}
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UVal} \\
[\![\{\mathbf{return}\ x \mapsto N\}]\!] &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\mathbf{let}\ (_ \mathbin{\underline{::}} k \mathbin{\underline{::}} h^{\mathrm{ret}} \mathbin{\underline{::}} h^{\mathrm{ops}} \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
&\qquad\qquad [\![N]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h^{\mathrm{ret}} \mathbin{\overline{::}} \uparrow h^{\mathrm{ops}} \mathbin{\overline{::}} \uparrow ks')
\end{aligned}$$

As before, the translation ensures that the associated effect continuation is discarded (the first element of the dynamic continuation ks). In addition the next continuation triple is extracted and reified as a static continuation triple. The interesting rule is the translation of operation clauses.

$$
\begin{aligned}
[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!]^\dagger \stackrel{\text{def}}{=} \underline{\lambda}\langle z, \langle p, rs\rangle\rangle\ ks.\ & \\
& \mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ (k \mathbin{\underline{::}} h^{\text{ret}} \mathbin{\underline{::}} h^{\text{ops}} \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
& \qquad \mathbf{let}\ (\_ \mathbin{\underline{::}} \_ \mathbin{\underline{::}} rs') = rs\ \mathbf{in} \\
& \qquad \mathbf{let}\ r = \mathbf{res}\,(V_{\text{ops}} \mathbin{\underline{::}} V_{\text{ret}} \mathbin{\underline{::}} rs')\ \mathbf{in} \\
& \qquad [\![N_\ell]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow h^{\text{ret}} \mathbin{\overline{::}} \uparrow h^{\text{ops}} \mathbin{\overline{::}} \uparrow ks'))_{\ell\in\mathcal{L}} \\
& \quad y \mapsto M_{\text{forward}}((y,p,rs),ks)\} \\
M_{\text{forward}}((y,p,rs),ks) \stackrel{\text{def}}{=} & \ \mathbf{let}\ (k \mathbin{\underline{::}} h^{\text{ret}} \mathbin{\underline{::}} h^{\text{ops}} \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
& \qquad h^{\text{ops}}\ \underline{@}\ \langle y, \langle p, h^{\text{ops}} \mathbin{\underline{::}} h^{\text{ret}} \mathbin{\underline{::}} k \mathbin{\underline{::}} rs\rangle\rangle\ \underline{@}\ ks' \\
V_{\text{ops}} \stackrel{\text{def}}{=} & \ \underline{\lambda}\langle z, \langle p, rs\rangle\rangle\ ks.M_{\text{forward}}((z,p,rs),ks) \\
V_{\text{ret}} \stackrel{\text{def}}{=} & \ \underline{\lambda} x\ ks.\mathbf{let}\ (h^{\text{ops}} \mathbin{\underline{::}} k \mathbin{\underline{::}} ks') = ks\ \mathbf{in}\ k\ \underline{@}\ x\ \underline{@}\ ks'
\end{aligned}
$$

The main difference between this translation rule and the translation rule for deep handler operation clauses is the realisation of resumptions. Recall that a resumption is represented as a reversed slice of a continuation. Thus the deconstruction of the resumption rs effectively ensures that the current handler gets properly uninstalled. However, it presents a new problem as the remainder rs' is not a well-formed continuation slice, because the top element is a pure continuation without a handler.

To rectify this problem, we can insert a dummy identity handler composed from V_{ops} and V_{ret}. The effect continuation V_{ops} forwards every operation, and the pure continuation V_{ret} is an identity clause. Thus every operation and the return value will effectively be handled by the next handler. Unfortunately, insertion of such identity handlers lead to memory leaks [120, 146].

The use of identity handlers is symptomatic for the need of a more general notion of resumptions. During resumption invocation the dangling pure continuation should be composed with the current pure continuation which suggests the need for a shallow variation of the resumption construction primitive **res**.

$$
\begin{aligned}
&\mathbf{let}\ r = \mathbf{res}^\dagger(\_ \mathbin{\underline{::}} \_ \mathbin{\underline{::}} k \mathbin{\underline{::}} h_n^{\text{ops}} \mathbin{\underline{::}} h_n^{\text{ret}} \mathbin{\underline{::}} k_n \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} h_1^{\text{ops}} \mathbin{\underline{::}} h_1^{\text{ret}} \mathbin{\underline{::}} k_1 \mathbin{\underline{::}} \underline{[]})\ \mathbf{in}\ N \rightsquigarrow \\
&\quad N[(\underline{\lambda} x\ ks.\mathbf{let}\ (k' \mathbin{\underline{::}} ks') = ks\ \mathbf{in} \\
&\qquad k_1\ \underline{@}\ x\ \underline{@}\ (h_1^{\text{ret}} \mathbin{\underline{::}} h_1^{\text{ops}} \cdots \mathbin{\underline{::}} k_n \mathbin{\underline{::}} h_n^{\text{ret}} \mathbin{\underline{::}} h_n^{\text{ops}} \mathbin{\underline{::}} (k' \circ k) \mathbin{\underline{::}} ks'))/r]
\end{aligned}
$$

where $\circ$ is defined to be function composition in continuation passing style.

$$g \circ f \stackrel{\text{def}}{=} \lambda x\, ks.\mathbf{let}\ (k :: ks') = ks\ \mathbf{in}\\ f\ @\ x\ @\ ((\lambda x\, ks.g\ @\ x\ @\ (k :: ks)) :: ks')$$

The idea is that $\mathbf{res}^{\dagger}$ uninstalls the appropriate handler and composes the dangling pure continuation k with the next *dynamically determined* pure continuation k', and reverses the remainder of the resumption and composes it with the modified dynamic continuation $((k' \circ k) :: ks')$.

While the underlying idea is correct, this particular realisation of the idea is problematic as the use of function composition reintroduces a variation of the dynamic administrative redexes that we dealt with in Section 4.3.3. In order to avoid generating these administrative redexes we need a more intensional continuation representation. Another telltale sign that we require a more intensional continuation representation is the necessary use of the administrative function V_{id} in the translation of **handle** as a placeholder for the empty pure continuation. In terms of aesthetics, the non-uniform continuation deconstructions also suggest that we could benefit from a more structured interpretation of continuations. Although it is seductive to program with lists, it quickly gets unwieldy.

4.4.2 Generalised continuations

One problem is that the continuation representation used by the higher-order uncurried translation for deep handlers is too extensional to support shallow handlers efficiently. Specifically, the representation of pure continuations needs to be more intensional to enable composition of pure continuations without having to materialise administrative continuation functions.

Another problem is that the continuation representation integrates the return clause into the pure continuations, but the semantics of shallow handlers demands that this return clause is discarded when any of the operations is invoked.

The solution to the first problem is to reuse the key idea of Section 4.3.3 to avoid administrative continuation functions by representing a pure continuation as an explicit list consisting of pure continuation functions. As a result the composition of pure continuation functions can be realised as a simple cons-operation.

The solution to the second problem is to pair the continuation functions corresponding to the **return**-clause and operation clauses in order to distinguish the pure continuation function induced by a **return**-clause from those induced by **let**-expressions.

Plugging these two solutions yields the notion of *generalised continuations*. A generalised continuation is a list of *continuation frames*. A continuation frame is a triple $\langle fs, \langle h^{\text{ret}}, h^{\text{ops}} \rangle \rangle$, where fs is list of stack frames representing the pure continuation for the computation occurring between the current execution and the handler, h^{ret} is the (translation of the) return clause of the enclosing handler, and h^{ops} is the (translation of the) operation clauses.

The change of representation of pure continuations does mean that we can no longer invoke them by simple function application. Instead, we must inspect the structure of the pure continuation fs and act appropriately. To ease notation it is convenient introduce a new computation form for pure continuation application $\mathbf{app}\ V\ W$ that feeds a value W into the continuation represented by V. There are two reduction rules.

$$\begin{array}{ll} \text{U-KAppNil} & \mathbf{app}\ (\langle [], \langle h^{\text{ret}}, h^{\text{ops}} \rangle \rangle :: ks)\ W \rightsquigarrow h^{\text{ret}}\ W\ ks \\ \text{U-KAppCons} & \mathbf{app}\ (\langle f :: fs, h \rangle :: ks)\ W \rightsquigarrow f\ W\ (\langle fs, h \rangle :: ks) \end{array}$$

The first rule describes what happens when the pure continuation is exhausted and the return clause of the enclosing handler is invoked. The second rule describes the case when the pure continuation has at least one element: this pure continuation function is invoked and the remainder of the continuation is passed in as the new continuation.

We must also change how resumptions (i.e. reversed continuations) are converted into functions that can be applied. Resumptions for deep handlers ($\mathbf{res}\ V$) are similar to Section 4.3.3, except that we now use **app** to invoke the continuation. Resumptions for shallow handlers ($\mathbf{res}^{\dagger}\ V$) are more complex. Instead of taking all the frames and reverse appending them to the current stack, we remove the current handler h and move the pure continuation $(f_1 :: \cdots :: f_m :: [])$ into the next frame. This captures the intended behaviour of shallow handlers: they are removed from the stack once they have been invoked. The following two reduction rules describe their behaviour.

$$\begin{array}{ll} \text{U-Res} & \mathbf{let}\ r = \mathbf{res}\ (V_n :: \cdots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow N[\lambda x\, ks.\mathbf{app}\ (V_1 :: \ldots V_n :: ks)\, x/r] \\ \text{U-Res}^{\dagger} & \mathbf{let}\ r = \mathbf{res}^{\dagger}\ (\langle f_1 :: \cdots :: f_m :: [], h \rangle :: V_n :: \cdots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\ & \qquad N[\lambda x\, ks.\mathbf{let}\ \langle fs', h' \rangle :: ks' = ks\ \mathbf{in} \\ & \qquad\qquad \mathbf{app}\ (V_1 :: \cdots :: V_n :: \langle f_1 :: \cdots :: f_m :: fs', h' \rangle :: ks')\, x/r] \end{array}$$

These constructs along with their reduction rules are macro-expressible in terms of the existing constructs. I choose here to treat them as primitives in order to keep the presentation relatively concise.

Essentially, a generalised continuation amounts to a sort of *defunctionalised* continuation, where **app** acts as an interpreter for the continuation structure [66, 240].

4.4.3 Dynamic terms: the target calculus revisited

Let us revisit the target calculus. Figure 4.5 depicts the untyped target calculus with support for generalised continuations. This is essentially the same as the target calculus used for the higher-order uncurried CPS translation for deep effect handlers in Section 4.3.4, except for the addition of recursive functions. The calculus also includes the **app** and **let** $r = \mathbf{res}^{\delta}\ V$ **in** N constructs described in Section 4.4.2. There is a small difference in the reduction rules for the resumption constructs: for deep resumptions we do an additional pattern match on the current continuation (ks). This is required to make the simulation proof for the CPS translation with generalised continuations (Section 4.4.4) go through, because it makes the functions that resumptions get converted to have the same shape as the translation of source level functions – this is required because the operational semantics does not treat resumptions as distinct first-class objects, but rather as a special kinds of functions.

4.4.4 Translation with generalised continuations

The CPS translation is given in Figure 4.6. In essence, it is the same as the CPS translation for deep effect handlers as described in Section 4.3.4, though it is adjusted to account for generalised continuation representation. For notational convenience, we write χ to denote a statically known effect continuation frame $\overline{\langle}\chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}$. The translation of **return** invokes the continuation κ using the continuation application primitive **app**. The translations of deep and shallow handlers differ only in their use of the resumption construction primitive.

A major aesthetic improvement due to the generalised continuation representation is that continuation construction and deconstruction are now uniform: only a single continuation frame is inspected at a time.

Correctness

The correctness of this CPS translation (Theorem 4.14) follows closely the correctness result for the higher-order uncurried CPS translation for deep handlers (Theorem 4.8). Save for the syntactic difference, the most notable difference is the extension of the operation handling lemma (Lemma 4.13) to cover shallow handling in addition to deep handling. Each lemma is stated in terms of static continuations, where the shape of the top element is always known statically, i.e., it is of the form $\overline{\langle}\mathcal{V}_{fs}, \overline{\langle}\mathcal{V}_{\mathrm{ret}}, \mathcal{V}_{\mathrm{ops}}\overline{\rangle}\rangle} :: \mathcal{W}$. Moreover, the static values $\mathcal{V}_{fs}$, $\mathcal{V}_{\mathrm{ret}}$, and $\mathcal{V}_{\mathrm{ops}}$ are all reflected dynamic terms (i.e., of

the form $\uparrow V$). This fact is used implicitly in the proofs. For brevity we write $\mathcal{V}_f$ to denote a statically known continuation frame $\overline{\langle}\mathcal{V}_{fs}, \overline{\langle}\mathcal{V}_{\text{ret}}, \mathcal{V}_{\text{ops}}\overline{\rangle\rangle}$. The full proof details are published in Appendix A of Hillerström et al. [122].

Lemma 4.9 (Substitution)**.** *The CPS translation commutes with substitution in value terms*

$$[\![W]\!][[\![V]\!]/x] = [\![W[V/x]]\!],$$

and with substitution in computation terms

$$\begin{aligned}&([\![M]\!] \mathbin{\overline{@}} (\mathcal{V}_f :\!\!: \mathcal{W}))[[\![V]\!]/x]\\ &= [\![M[V/x]]\!] \mathbin{\overline{@}} (\mathcal{V}_f :\!\!: \mathcal{W})[[\![V]\!]/x],\end{aligned}$$

and with substitution in handler definitions

$$\begin{aligned}[\![H^{\text{ret}}]\!][[\![V]\!]/x] &= [\![H^{\text{ret}}[V/x]]\!],\\ [\![H^{\text{ops}}]\!][[\![V]\!]/x] &= [\![H^{\text{ops}}[V/x]]\!].\end{aligned}$$

In order to reason about the behaviour of the S-Op and S-Op† rules, which are defined in terms of evaluation contexts, we extend the CPS translation to evaluation contexts, using the same translations as for the corresponding constructs in $\lambda_{\mathsf{h}^\dagger}$.

$$\begin{aligned}[\![[\,]]\!] &= \overline{\lambda}\kappa.\kappa\\ [\![\mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N]\!] &= \overline{\lambda}\overline{\langle}\theta, \chi\overline{\rangle} :\!\!: \kappa.\\ &\qquad [\![\mathcal{E}]\!] \mathbin{\overline{@}} (\overline{\langle}\uparrow((\underline{\lambda} x\, ks.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\text{ret}}, h^{\text{ops}}\underline{\rangle\rangle} :\!\!: ks' = ks\ \mathbf{in}\\ &\qquad\qquad [\![N]\!] \mathbin{\overline{@}} (\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\text{ret}}, \uparrow h^{\text{ops}}\overline{\rangle\rangle} :\!\!: \uparrow ks'))\underline{::}\downarrow\theta),\\ &\qquad \chi\overline{\rangle} :\!\!: \kappa)\\ [\![\mathbf{handle}^\delta\ \mathcal{E}\ \mathbf{with}\ H]\!] &= \overline{\lambda}\kappa.[\![\mathcal{E}]\!] \mathbin{\overline{@}} (\overline{\langle}[], [\![H]\!]^\delta\overline{\rangle} :\!\!: \kappa)\end{aligned}$$

The following lemma is the characteristic property of the CPS translation on evaluation contexts. This allows us to focus on the computation within an evaluation context.

Lemma 4.10 (Evaluation context decomposition)**.**

$$[\![\mathcal{E}[M]]\!] \mathbin{\overline{@}} (\mathcal{V}_f :\!\!: \mathcal{W}) = [\![M]\!] \mathbin{\overline{@}} ([\![\mathcal{E}]\!] \mathbin{\overline{@}} (\mathcal{V}_f :\!\!: \mathcal{W}))$$

By definition, reifying a reflected dynamic value is the identity ($\downarrow\uparrow V = V$), but we also need to reason about the inverse composition. As a result of the invariant that the translation always has static access to the top of the handler stack, the translated terms are insensitive to whether the remainder of the stack is statically known or is a reflected version of a reified stack. This is captured by the following lemma. The proof is by

induction on the structure of M (after generalising the statement to stacks of arbitrary depth), and relies on the observation that translated terms either access the top of the handler stack, or reify the stack to use dynamically, whereupon the distinction between reflected and reified becomes void. Again, this lemma holds when the top of the static continuation is known.

Lemma 4.11 (Reflect after reify)**.**

$$[\![M]\!]\,\overline{@}\,(\mathcal{V}_f :: \uparrow\downarrow \mathcal{W}) = [\![M]\!]\,\overline{@}\,(\mathcal{V}_f :: \mathcal{W}).$$

The next lemma states that the CPS translation correctly simulates forwarding. The proof is by inspection of how the translation of operation clauses treats non-handled operations.

Lemma 4.12 (Forwarding)**.** *If $\ell \notin dom(H_1)$ then:*

$$\begin{aligned}[\![H_1^{\mathrm{ops}}]\!]^{\delta}\,@\,\langle \ell, \langle V_p, V_{rs}\rangle\rangle\,@\,(\langle V_{fs}, [\![H_2]\!]^{\delta}\rangle :: W) \rightsquigarrow^{+}\\ [\![H_2^{\mathrm{ops}}]\!]^{\delta}\,@\,\langle \ell, \langle V_p, \langle V_{fs}, [\![H_2]\!]^{\delta}\rangle :: V_{rs}\rangle\rangle\,@\,W.\end{aligned}$$

The following lemma is central to our simulation theorem. It characterises the sense in which the translation respects the handling of operations. Note how the values substituted for the resumption variable r in both cases are in the image of the translation of λ-terms in the CPS translation. This is thanks to the precise way that the reductions rules for resumption construction works in our dynamic language, as described above.

Lemma 4.13 (Handling)**.** *Suppose $\ell \notin BL(\mathcal{E})$ and $H^{\ell} = \{\ell\, p\, r \mapsto N_{\ell}\}$. If H is deep then*

$$\begin{aligned}&[\![\mathbf{do}\ \ell\ V]\!]\,\overline{@}\,([\![\mathcal{E}]\!]\,\overline{@}\,(\overline{\langle}\uparrow[], [\![H]\!]\overline{\rangle} :: \mathcal{V}_f :: \mathcal{W})) \rightsquigarrow^{+}\\ &\quad([\![N_{\ell}]\!]\,\overline{@}\,(\mathcal{V}_f :: \mathcal{W}))[[\![V]\!]/p, \lambda x\, ks.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: ks' = ks\ \mathbf{in}\ [\![\mathbf{return}\ x]\!]\\ &\qquad\qquad\overline{@}([\![\mathcal{E}]\!]\,\overline{@}\,(\overline{\langle}\uparrow[], [\![H]\!]\overline{\rangle} :: \overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow ks'))/r].\end{aligned}$$

Otherwise if H is shallow then

$$\begin{aligned}&[\![\mathbf{do}\ \ell\ V]\!]\,\overline{@}\,([\![\mathcal{E}]\!]\,\overline{@}\,(\overline{\langle}\uparrow[], [\![H]\!]^{\dagger}\overline{\rangle} :: \mathcal{V}_f :: \mathcal{W})) \rightsquigarrow^{+}\\ &\quad([\![N_{\ell}]\!]\,\overline{@}\,(\mathcal{V}_f :: \mathcal{W}))[[\![V]\!]/p, \lambda x\, ks.\mathbf{let}\ \langle fs, \langle h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: ks' = ks\ \mathbf{in}\ [\![\mathbf{return}\ x]\!]\\ &\qquad\qquad\overline{@}([\![\mathcal{E}]\!]\,\overline{@}\,(\overline{\langle}\uparrow fs, \overline{\langle}\uparrow h^{\mathrm{ret}}, \uparrow h^{\mathrm{ops}}\overline{\rangle\rangle} :: \uparrow ks'))/r].\end{aligned}$$

With the aid of the above lemmas we can state and prove the main result for the translation: a simulation result in the style of Plotkin [220].

Theorem 4.14 (Simulation)**.** *If $M \rightsquigarrow N$ then*

$$[\![M]\!]\,\overline{@}\,(\overline{\langle}\mathcal{V}_{fs}, \overline{\langle}\mathcal{V}_{\mathrm{ret}}, \mathcal{V}_{\mathrm{ops}}\overline{\rangle\rangle} :: \mathcal{W}) \rightsquigarrow^{+} [\![N]\!]\,\overline{@}\,(\overline{\langle}\mathcal{V}_{fs}, \overline{\langle}\mathcal{V}_{\mathrm{ret}}, \mathcal{V}_{\mathrm{ops}}\overline{\rangle\rangle} :: \mathcal{W}).$$

Proof. The proof is by case analysis on the reduction relation using Lemmas 4.10–4.13. In particular, the S-Op and S-Op† cases follow from Lemma 4.13. □

In common with most CPS translations, full abstraction does not hold (a function could count the number of handlers it is invoked within by examining the continuation, for example). However, as the semantics is deterministic it is straightforward to show a backward simulation result.

Lemma 4.15 (Backwards simulation)**.** *If* $\top[\![M]\!] \rightsquigarrow^+ V$ *then there exists* W *such that* $M \rightsquigarrow^* W$ *and* $\top[\![W]\!] = V$.

Corollary 4.16. $M \rightsquigarrow^* V$ *if and only if* $\top[\![M]\!] \rightsquigarrow^* \top[\![V]\!]$.

4.5 Transforming parameterised handlers

Generalised continuations provide a versatile implementation strategy for effect handlers as exemplified in the previous section. In this section we add further emphasis on the versatility of generalised continuations by demonstrating how to adapt the continuation structure to accommodate parameterised handlers. In order to support parameterised handlers, each effect continuation must store the current value of the handler parameter. Thus, an effect continuation becomes a triple consisting of the parameter, return clause, and operation clause(s). Furthermore, the return clause gets transformed into a binary function, that takes the current value of the handler parameter as its first argument and the return value of the handled computation as its second argument. Similarly, the operation clauses are transformed into a binary function, that takes the handler parameter first and the operation package second. This strategy effectively amounts to explicit state passing as the parameter value gets threaded through every handler continuation function. Operationally, the pure continuation invocation rule U-KAppNil requires a small adjustment to account for the handler parameter.

$$\mathbf{app}\ (\langle[]\rangle, \langle q, h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle :: ks)\ V \rightsquigarrow h^{\mathrm{ret}}\ @\ \langle q, V\rangle\ @\ ks$$

The pure continuation v is now applied to a pair consisting of the current value of the handler parameter q and the return value V. Similarly, the resumption rule U-Res must also be adapted to update the value of the handler parameter.

$$\begin{aligned}&\mathbf{let}\ r = \mathbf{res}^{\ddagger}\ (\langle q, h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle :: \cdots :: V_1 :: [])\ \mathbf{in}\ N \rightsquigarrow \\ &\quad N[\lambda\langle q', x\rangle\, ks.\mathbf{app}\ (V_1 :: \cdots :: \langle q', h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle :: ks)\ x/r]\end{aligned}$$

The rule is not much different from the original U-Res rule. The difference is that this rule unpacks the current handler parameter q along with the return clause, h^{ret}, and operation clauses, h^{ops}. The reduction constructs a resumption function, whose first parameter q' binds the updated value of the handler parameter. The q' is packaged with the original h^{ret} and h^{ops} such that the next activation of the handler gets the parameter value q' rather than q.

The CPS translation is updated accordingly to account for the triple effect continuation structure. This involves updating the cases that scrutinise the effect continuation structure as it now includes the additional state value. The cases that need to be updated are shown in Figure 4.7. We write ξ to denote static handler parameters. The translation of **do** invokes the effect continuation $\downarrow\chi^{\mathrm{ops}}$ with a triple consisting of the value of the handler parameter, the operation, and the operation payload. The parameter is also pushed onto the reversed resumption stack. This is necessary to account for the case where the effect continuation $\downarrow\chi^{\mathrm{ops}}$ does not handle operation ℓ.

The translation of the return and operation clauses are parameterised by the name of the binder for the handler parameter. Each translation yields functions that take a pair as input in addition to the current continuation. The forwarding case is adjusted in the same way as the translation for **do**. The current continuation k is deconstructed in order to identify the next effect continuation h^{ops} and its parameter q. Then h^{ops} is invoked with the updated resumption stack and the value of its parameter q. The top-level translation adds a 'dummy' unit value, which is ignored by both the pure continuation and effect continuation.We can avoid the use of such values entirely if the target language had proper sums to tag effect continuation frames accordingly. Obviously, this entails performing a case analysis every time an effect continuation frame is deconstructed.

4.6 Related work

CPS transforms for effect handlers The one-pass higher-order CPS translation for deep, shallow, and parameterised handlers draws on insights from the literature on CPS translations for delimited control operators such as shift and reset [62, 63, 67, 190]. Other CPS translations for handlers use a monadic approach. For example, Leijen [165] implements deep and parameterised handlers in Koka [164] by translating them into a free monad primitive in the runtime. Leijen uses a selective CPS translation to lift code into the monad. The selective aspect is important in practice to avoid overhead in code

that does not use effect handlers. Scala Effekt [35, 37] provides an implementation of effect handlers as a library for the Scala programming language. The implementation is based closely on the monadic delimited control framework of Dybvig et al. [75]. A variation of the Scala Effekt library is used to implement effect handlers as an interface for programming with delimited continuations in Java [36]. The implementation of delimited continuations depend on special byte code instructions, inserted via a selective type-driven CPS translation.

The Effekt language (which is distinct from the Effekt library) implements handlers by a translation into capability-passing style, which may more informatively be dubbed *handler-passing style* as handlers are passed downwards to the invocation sites of their respective operations [39, 243]. The translation into capability-passing style is realised by way of a effect-type directed iterated CPS transform, which introduces a continuation argument per handler in scope [243]. The idea of iterated CPS is due to Danvy and Filinski [62], who used it to give develop a CPS transform for shift and reset. Xie et al. [277] have devised an *evidence-passing translation* for deep effect handlers. The basic idea is similar to capability-passing style as evidence for handlers are passed downwards to their operations in shape of a vector containing the handlers in scope through computations. Xie and Leijen [276] have realised handlers by evidence-passing style as a Haskell library.

There are clear connections between the CPS translations presented in this chapter and the continuation monad implementation of Kammar et al. [143]. Whereas Kammar et al. present a practical Haskell implementation depending on sophisticated features such as type classes, which to some degree obscures the essential structure, here we have focused on a foundational formal treatment. Kammar et al. obtain impressive performance results by taking advantage of the second class nature of type classes in Haskell coupled with the aggressive fusion optimisations GHC performs [275].

Plotkin's colon translation The original method for proving the correctness of a CPS translation is by way of a simulation result. Simulation states that every reduction sequence in a given source program is mimicked by its CPS transformation. Static administrative redexes in the image of a CPS translation provide hurdles for proving simulation, since these redexes do not arise in the source program. Plotkin [220] uses the so-called *colon translation* to overcome static administrative reductions. Informally, it is defined such that given some source term M and some continuation k, then the term $M : k$ is the result of performing all static administrative reductions on $[\![M]\!]\, k$, that is to

say $[\![M]\!]\, k \rightsquigarrow_{\mathrm{a}}^{*} M : k$. Thus this translation makes it possible to bypass administrative reductions and instead focus on the reductions inherited from the source program. The colon translation captures precisely the intuition that drives CPS transforms, namely, that if in the source $M \rightsquigarrow^{*} \mathbf{return}\ V$ then in the image $[\![M]\!]\, k \rightsquigarrow^{*} k\, [\![V]\!]$.

Syntax

$$\begin{array}{lll}
\text{Values} & V, W \in \mathrm{UVal} ::= & x \mid \underline{\lambda} x\, ks.M \mid \mathbf{rec}\, g\, x\, ks.M \mid \ell \mid \underline{\langle} V, W \underline{\rangle} \\
\text{Computations} & M, N \in \mathrm{UComp} ::= & V \mid U \mathbin{\underline{@}} V \mathbin{\underline{@}} W \mid \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = V\ \mathbf{in}\ N \\
& \mid & \mathbf{case}\ V\, \{\ell \mapsto M; x \mapsto N\} \mid \mathbf{absurd}\ V \\
& \mid & \mathbf{app}\, V\, W \mid \mathbf{let}\ r = \mathbf{res}^{\delta}\ V\ \mathbf{in}\ M
\end{array}$$

Syntactic sugar

$$\begin{array}{lll}
\mathbf{let}\ x = V\ \mathbf{in}\ N \equiv N[V/x] & \langle\rangle \equiv \ell_{\langle\rangle} & \underline{[]} \equiv \ell_{\underline{[]}} \\
\ell\ V \equiv \underline{\langle} \ell, V \underline{\rangle} & \langle \ell = V; W \rangle \equiv \ell\ \underline{\langle} V, W \underline{\rangle} & V \mathbin{\underline{::}} W \equiv \ell_{\underline{::}}\ \underline{\langle} V, W \underline{\rangle}
\end{array}$$

$$\begin{array}{ll}
\mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\} \equiv & \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \equiv \\
\quad \mathbf{let}\ y = V\ \mathbf{in}\ \mathbf{let}\ \underline{\langle} z, x \underline{\rangle} = y\ \mathbf{in} & \quad \mathbf{let}\ \underline{\langle} z, z' \underline{\rangle} = V\ \mathbf{in}\ \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = z'\ \mathbf{in} \\
\quad \mathbf{case}\ z\ \{\ell \mapsto M; z \mapsto N\} & \quad \mathbf{case}\ z\ \{\ell \mapsto N; z \mapsto \ell_{\perp}\}
\end{array}$$

Standard reductions

$$\begin{array}{lll}
\text{U-App} & (\underline{\lambda} x\, ks.M) \mathbin{\underline{@}} V \mathbin{\underline{@}} W & \rightsquigarrow M[V/x, W/ks] \\
\text{U-Rec} & (\mathbf{rec}\, g\, x\, ks.M) \mathbin{\underline{@}} V \mathbin{\underline{@}} W & \rightsquigarrow M[\mathbf{rec}\, g\, x\, ks.M/g, V/x, W/ks] \\
\text{U-Split} & \mathbf{let}\ \underline{\langle} x, y \underline{\rangle} = \underline{\langle} V, W \underline{\rangle}\ \mathbf{in}\ N & \rightsquigarrow N[V/x, W/y] \\
\text{U-Case}_1 & \mathbf{case}\ \ell\, \{\ell \mapsto M; x \mapsto N\} & \rightsquigarrow M \\
\text{U-Case}_2 & \mathbf{case}\ \ell\, \{\ell' \mapsto M; x \mapsto N\} & \rightsquigarrow N[\ell/x], \qquad \text{if } \ell \neq \ell'
\end{array}$$

Continuation reductions

$$\begin{array}{lll}
\text{U-KAppNil} & \mathbf{app}\ (\underline{\langle} \underline{[]}, \underline{\langle} v, e \underline{\rangle} \underline{\rangle} \mathbin{\underline{::}} ks)\, V & \rightsquigarrow v \mathbin{\underline{@}} V \mathbin{\underline{@}} ks \\
\text{U-KAppCons} & \mathbf{app}\ (\underline{\langle} f \mathbin{\underline{::}} fs, h \underline{\rangle} \mathbin{\underline{::}} ks)\, V & \rightsquigarrow f \mathbin{\underline{@}} V \mathbin{\underline{@}} (\underline{\langle} fs, h \underline{\rangle} \mathbin{\underline{::}} ks)
\end{array}$$

Resumption reductions

$$\begin{array}{ll}
\text{U-Res} & \mathbf{let}\ r = \mathbf{res}(V_n \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} V_1 \mathbin{\underline{::}} \underline{[]})\ \mathbf{in}\ N \rightsquigarrow \\
& \quad N[\underline{\lambda} x\, ks.\mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle} \underline{\rangle} \mathbin{\underline{::}} ks' = ks\ \mathbf{in} \\
& \qquad \mathbf{app}\ (V_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} V_n \mathbin{\underline{::}} \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle} \underline{\rangle} \mathbin{\underline{::}} ks')\ x/r] \\
\text{U-Res}^{\dagger} & \mathbf{let}\ r = \mathbf{res}^{\dagger}(\underline{\langle} f_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} f_m \mathbin{\underline{::}} \underline{[]}, h \underline{\rangle} \mathbin{\underline{::}} V_n \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} V_1 \mathbin{\underline{::}} \underline{[]})\ \mathbf{in}\ N \rightsquigarrow \\
& \quad N[\underline{\lambda} x\, k.\mathbf{let}\ \underline{\langle} fs', h' \underline{\rangle} \mathbin{\underline{::}} ks' = ks\ \mathbf{in} \\
& \qquad \mathbf{app}\ (V_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} V_n \mathbin{\underline{::}} \underline{\langle} f_1 \mathbin{\underline{::}} \cdots \mathbin{\underline{::}} f_m \mathbin{\underline{::}} fs', h' \underline{\rangle} \mathbin{\underline{::}} ks')\ x/r]
\end{array}$$

Figure 4.5: Untyped target calculus supporting generalised continuations.

Values

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Val} \to \mathrm{UVal} \\
[\![x]\!] &\stackrel{\text{def}}{=} x \\
[\![\lambda x.M]\!] &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks') \\
[\![\Lambda \alpha.M]\!] &\stackrel{\text{def}}{=} \underline{\lambda} \langle\rangle\, ks.\mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks') \\
[\![\mathbf{rec}\, g\, x.M]\!] &\stackrel{\text{def}}{=} \mathbf{rec}\, g\, x\, ks.\mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in}\ [\![M]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks') \\
[\![\langle\rangle]\!] &\stackrel{\text{def}}{=} \langle\rangle \qquad [\![\langle \ell = V; W\rangle]\!] \stackrel{\text{def}}{=} \langle \ell = [\![V]\!]; [\![W]\!]\rangle \qquad [\![\ell\, V]\!] \stackrel{\text{def}}{=} \ell\, [\![V]\!]
\end{aligned}
$$

Computations

$$
\begin{aligned}
[\![-]\!] &: \mathrm{Comp} \to \mathrm{SVal}^* \to \mathrm{UComp} \\
[\![V\, W]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.[\![V]\!]\ \underline{@}\ [\![W]\!]\ \underline{@} \downarrow \kappa \\
[\![V\, T]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.[\![V]\!]\ \underline{@}\ \langle\rangle\ \underline{@} \downarrow \kappa \\
[\![\mathbf{let}\ \langle \ell = x; y\rangle = V\ \mathbf{in}\ N]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.\mathbf{let}\ \langle \ell = x; y\rangle = [\![V]\!]\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ \kappa \\
[\![\mathbf{case}\ V\ \{\ell\, x \mapsto M; y \mapsto N\}]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.\mathbf{case}\ [\![V]\!]\ \{\ell\, x \mapsto [\![M]\!]\ \overline{@}\ \kappa; y \mapsto [\![N]\!]\ \overline{@}\ \kappa\} \\
[\![\mathbf{absurd}\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.\mathbf{absurd}\ [\![V]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{return}\, V]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.\mathbf{app}\ (\downarrow \kappa)\ [\![V]\!] \\
[\![\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N]\!] &\stackrel{\text{def}}{=} \overline{\lambda \langle} \theta, \chi \overline{\rangle} \overline{::} \kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow ((\underline{\lambda} x\, ks.\mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in} \\
&\qquad [\![N]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks')) \\
&\qquad \underline{::} \downarrow \theta), \chi \overline{\rangle} \overline{::} \kappa) \\
[\![\mathbf{do}\ \ell\ V]\!] &\stackrel{\text{def}}{=} \overline{\lambda \langle} \theta, \overline{\langle} \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}} \overline{\rangle \rangle} \overline{::} \kappa.\downarrow \chi^{\mathrm{ops}}\ \underline{@} \underline{\langle} \ell, \underline{\langle} [\![V]\!], \underline{\langle} \downarrow \theta, \underline{\langle} \downarrow \chi^{\mathrm{ret}}, \downarrow \chi^{\mathrm{ops}} \underline{\rangle \rangle} \underline{::} [] \underline{\rangle \rangle} \\
&\qquad \underline{@} \downarrow \kappa \\
[\![\mathbf{handle}^{\delta}\, M\ \mathbf{with}\ H]\!] &\stackrel{\text{def}}{=} \overline{\lambda} \kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow [], \overline{\langle} \uparrow [\![H^{\mathrm{ret}}]\!], \uparrow [\![H^{\mathrm{ops}}]\!]^{\delta} \overline{\rangle \rangle} \overline{::} \kappa)
\end{aligned}
$$

Handler definitions

$$
\begin{aligned}
[\![-]\!] &: \mathrm{HDef} \to \mathrm{UVal} \\
[\![\{\mathbf{return}\, x \mapsto N\}]\!] &\stackrel{\text{def}}{=} \underline{\lambda} x\, ks.\mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks') \\
[\![\{(\ell\, p\, r \mapsto N_\ell)_{\ell \in \mathcal{L}}\}]\!]^{\delta} &\stackrel{\text{def}}{=} \underline{\lambda} \underline{\langle} z, \underline{\langle} p, rs \underline{\rangle \rangle}\, ks.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}^{\delta}\, rs\ \mathbf{in} \\
&\qquad \mathbf{let}\ (k \underline{::} ks') = ks\ \mathbf{in} \\
&\qquad [\![N_\ell]\!]\ \overline{@}\ (\uparrow k \overline{::} \uparrow ks'))_{\ell \in \mathcal{L}} \\
&\qquad y \mapsto M_{\mathrm{forward}}((y, p, rs), ks)\} \\
M_{\mathrm{forward}}((y, p, rs), ks) &\stackrel{\text{def}}{=} \mathbf{let}\ \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle \rangle} \underline{::} ks' = ks\ \mathbf{in} \\
&\qquad h^{\mathrm{ops}}\ \underline{@} \underline{\langle} y, \underline{\langle} p, \underline{\langle} fs, \underline{\langle} h^{\mathrm{ret}}, h^{\mathrm{ops}} \underline{\rangle \rangle} \underline{::} rs \underline{\rangle \rangle}\ \underline{@}\ ks'
\end{aligned}
$$

Top-level program

$$
\begin{aligned}
\top [\![-]\!] &: \mathrm{Comp} \to \mathrm{UComp} \\
\top [\![M]\!] &\stackrel{\text{def}}{=} [\![M]\!]\ \overline{@}\ (\overline{\langle} \uparrow [], \overline{\langle} \uparrow \underline{\lambda} x\, ks.x, \uparrow \underline{\lambda} \underline{\langle} z, \underline{\langle} p, rs \underline{\rangle \rangle}\, ks.\mathbf{absurd}\ z \overline{\rangle \rangle} \overline{::} \uparrow [])
\end{aligned}
$$

Figure 4.6: Higher-order uncurried CPS translation for effect handlers.

Computations

$$[\![-]\!] \;:\; \mathrm{Comp} \to \mathrm{SVal}^* \to \mathrm{UComp}$$

$$[\![\mathbf{do}\ \ell\ V]\!] \stackrel{\text{def}}{=} \overline{\lambda}\langle\theta, \overline{\langle}\xi, \chi^{\mathrm{ret}}, \chi^{\mathrm{ops}}\overline{\rangle}\rangle \mathbin{\overline{::}} \kappa.\downarrow\chi^{\mathrm{ops}}\ @\ \langle\downarrow\xi, \ell, \langle[\![V]\!], \langle\downarrow\theta, \langle\downarrow\xi, \downarrow\chi^{\mathrm{ret}}, \downarrow\chi^{\mathrm{ops}}\rangle\rangle :: [] \rangle\rangle\ @\ \downarrow\kappa$$

$$[\![\mathbf{handle}^{\ddagger} M\ \mathbf{with}\ (q.H)(W)]\!] \stackrel{\text{def}}{=} \overline{\lambda}\kappa.[\![M]\!]\ \overline{@}\ (\overline{\langle}\uparrow[], \overline{\langle}\uparrow[\![W]\!], \uparrow[\![H^{\mathrm{ret}}]\!]^{\ddagger}_{q}, \uparrow[\![H^{\mathrm{ops}}]\!]^{\ddagger}_{q}\overline{\rangle}\rangle \mathbin{\overline{::}} \kappa)$$

Handler definitions

$$[\![-]\!] \;:\; \mathrm{HDef} \times \mathrm{UVal} \to \mathrm{UVal}$$

$$[\![\{\mathbf{return}\ x \mapsto N\}]\!]^{\ddagger}_{q} \stackrel{\text{def}}{=} \underline{\lambda}\langle q, x\rangle\ ks.\mathbf{let}\ (k :: ks') = ks\ \mathbf{in}\ [\![N]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow ks')$$

$$[\![\{(\ell\ p\ r \mapsto N_\ell)_{\ell\in\mathcal{L}}\}]\!]^{\ddagger}_{q} \stackrel{\text{def}}{=} \underline{\lambda}\langle q, z, \langle p, rs\rangle\rangle\ ks.\mathbf{case}\ z\ \{(\ell \mapsto \mathbf{let}\ r = \mathbf{res}^{\ddagger}\ rs\ \mathbf{in}\ \mathbf{let}\ (k :: ks') = ks\ \mathbf{in}\ [\![N_\ell]\!]\ \overline{@}\ (\uparrow k \mathbin{\overline{::}} \uparrow ks'))_{\ell\in\mathcal{L}}\ y \mapsto M_{\mathrm{forward}}((y, p, rs), ks)\}$$

$$M_{\mathrm{forward}}((y, p, rs), ks) \stackrel{\text{def}}{=} \mathbf{let}\ \langle fs, \langle q, h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: ks' = ks\ \mathbf{in}\ h^{\mathrm{ops}}\ @\ \langle q, y, \langle p, \langle fs, \langle q, h^{\mathrm{ret}}, h^{\mathrm{ops}}\rangle\rangle :: rs\rangle\rangle\ @\ ks'$$

Top-level program

$$\top[\![M]\!] = [\![M]\!]\ \overline{@}\ (\overline{\langle}[], \overline{\langle}\uparrow\langle\rangle, \uparrow\underline{\lambda}\langle q, x\rangle\ ks.x, \uparrow\underline{\lambda}\langle q, z\rangle\ ks.\mathbf{absurd}\ z\overline{\rangle}\rangle \mathbin{\overline{::}} \uparrow[])$$

Figure 4.7: CPS translation for parameterised handlers.

Chapter 5

Abstract machine semantics

Abstract machine semantics are an operational semantics that makes program control more apparent than context-based reduction semantics. In a some sense abstract machine semantics are a lower level semantics than reduction semantics as they provide a model of computation based on *abstract machines*, which capture some core aspects of how actual computers might go about executing programs. Abstract machines come in different style and flavours, though, a common trait is that they are defined in terms of *configurations*. A configuration includes the essentials to describe the machine state as it were, i.e. some abstract notion of call stack, memory, program counter, etc.

In this chapter I will demonstrate an application of generalised continuations (Section 4.4.2) to abstract machines that emphasises the usefulness of generalised continuations to implement various kinds of effect handlers. The key takeaway from this application is that it is possible to plug the generalised continuation structure into a standard framework to achieve a simultaneous implementation of deep, shallow, and parameterised effect handlers. Specifically I will change the continuation structure of a standard Felleisen and Friedman style *CEK machine* to fit generalised continuations.

The CEK machine (CEK is an acronym for Control, Environment, Kontinuation [83]) is an abstract machine with an explicit environment, which models the idea that processor registers name values as an environment associates names with values. Thus by using the CEK formalism we depart from the substitution-based model of computation used in the preceding chapters and move towards a more 'realistic' model of computation (realistic in the sense of emulating how a computer executes a program). Another significant difference is that in the CEK formalism evaluation contexts are no longer syntactically intertwined with the source program. Instead evaluation contexts are separately managed through the continuation of the CEK machine.

Chapter outline

Section 5.1 augments the standard CEK notion of configurations to accommodate generalised continuations.

Section 5.2 gives the reduction rules for the CEK machine with generalised continuations.

Section 5.3 discusses ways to realise the machine and potential efficiency pitfalls.

Section 5.4 shows that the machine with generalised continuations simulates the substitution-based reduction semantics of λ_{h}.

Section 5.5 discusses related work.

Relation to prior work The work in this chapter is based on work in the following previously published papers.

i Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016

ii Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018

iii Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020

The particular presentation in this chapter is adapted from item iii.

5.1 Configurations with generalised continuations

Syntactically, the CEK machine consists of three components: 1) the control component, which focuses the term currently being evaluated; 2) the environment component, which maps free variables to machine values, and 3) the continuation component, which describes what to evaluate next (some literature uses the term ‘control string’ in lieu of continuation to disambiguate it from programmatic continuations in the source language). Intuitively, the continuation component captures the idea of call stack from actual programming language implementations.

The abstract machine is formally defined in terms of configurations. A configuration $\langle M \mid \gamma \mid \kappa \circ \kappa' \rangle \in \mathrm{Conf}$ is a triple consisting of a computation term $M \in \mathrm{Comp}$, an environment $\gamma \in \mathrm{Env}$, and a pair of generalised continuations $\kappa, \kappa' \in \mathrm{GenCont}$. The complete abstract machine syntax is given in Figure 5.1. The control and environment components are completely standard as they are similar to the components in Felleisen and Friedman's original CEK machine modulo the syntax of the source language. However, the structure of the continuation component is new. This component comprises two generalised continuations, where the latter continuation κ' is an entirely administrative object that materialises only during operation invocations as it is used to construct the reified segment of the continuation up to an appropriate enclosing handler. For the most part κ' is empty, therefore we will write $\langle M \mid \gamma \mid \kappa \rangle$ as syntactic sugar for $\langle M \mid \gamma \mid \kappa \circ [] \rangle$ where $[]$ is the empty continuation (an alternative is to syntactically differentiate between regular and administrative configurations by having both three-place and four-place configurations as for example as Biernacka et al. [23] do).

An environment is either empty, written $\emptyset$, or an extension of some other environment γ, written $\gamma[x \mapsto v]$, where x is the name of a variable and v is a machine value. The machine values consist of function closures, recursive function closures, type function closures, records, variants, and reified continuations. The three abstraction forms are paired with an environment that binds the free variables in the their bodies. The records and variants are transliterated from the value forms of the source calculi. Figure 5.2 defines the value interpretation function, which turns any source language value into a corresponding machine value. A continuation κ is a stack of generalised continuation frames $[\theta_1, \ldots, \theta_n]$. As in Section 4.4.2 each continuation frame $\theta = (\sigma, \chi)$ consists of a pure continuation σ, corresponding to a sequence of let bindings, interpreted under some handler, which in this context is represented by the handler closure χ. A pure continuation is a stack of pure frames. A pure frame (γ, x, N) closes a let-binding **let** $x = [\ ]$ **in** N over environment γ. The pure continuation structure is similar to the continuation structure of Felleisen and Friedman's original CEK machine. There are three kinds of handler closures, one for each kind of handler. A deep handler closure is a pair (γ, H) which closes a deep handler definition H over environment γ. Similarly, a shallow handler closure $(\gamma, H^\dagger)$ closes a shallow handler definition over environment γ. Finally, a parameterised handler closure $(\gamma, (q.H))$ closes a parameterised handler definition over environment γ. As a syntactic shorthand we write H^δ to range over deep, shallow, and parameterised handler definitions. Sometimes H^δ will range over just two kinds of handler definitions; it will be clear from the context which handler definition is

Configurations	$\mathcal{C} \in \mathrm{Conf} ::= \langle M \mid \gamma \mid \kappa \circ \kappa' \rangle$
Value environments	$\gamma \in \mathrm{Env} ::= \emptyset \mid \gamma[x \mapsto v]$
Values	$v, w \in \mathrm{Mval} ::= (\gamma, \lambda x^A.M) \mid (\gamma, \mathbf{rec}^{A\to C} x.M)$
	$\mid \ (\gamma, \Lambda\alpha^K.M)$
	$\mid \ \langle\rangle \mid \langle \ell = v; w \rangle \mid (\ell\, v)^R$
	$\mid \ \kappa^A \mid (\kappa, \sigma)^A$
Continuations	$\kappa \in \mathrm{GenCont} ::= [] \mid \theta :: \kappa$
Continuation frames	$\theta \in \mathrm{GenFrame} ::= (\sigma, \chi)$
Pure continuations	$\sigma \in \mathrm{PureCont} ::= [] \mid \phi :: \sigma$
Pure continuation frames	$\phi \in \mathrm{PureFrame} ::= (\gamma, x, N)$
Handler closures	$\chi \in \mathrm{HClo} ::= (\gamma, H) \mid (\gamma, H^\dagger) \mid (\gamma, (q.H))$

Figure 5.1: Abstract machine syntax.

$$[\![-]\!] : \mathrm{Val} \times \mathrm{Env} \to \mathrm{Mval}$$

$$[\![x]\!]\gamma \stackrel{\text{def}}{=} \gamma(x) \qquad [\![\langle\rangle]\!]\gamma \stackrel{\text{def}}{=} \langle\rangle$$

$$[\![\lambda x^A.M]\!]\gamma \stackrel{\text{def}}{=} (\gamma, \lambda x^A.M) \qquad [\![\langle \ell = V; W \rangle]\!]\gamma \stackrel{\text{def}}{=} \langle \ell = [\![V]\!]\gamma; [\![W]\!]\gamma \rangle$$

$$[\![\mathbf{rec}\, g^{A\to C} x.M]\!]\gamma \stackrel{\text{def}}{=} (\gamma, \mathbf{rec}\, g^{A\to C} x.M) \qquad [\![(\ell\, V)^R]\!]\gamma \stackrel{\text{def}}{=} (\ell\, [\![V]\!]\gamma)^R$$

$$[\![\Lambda\alpha^K.M]\!]\gamma \stackrel{\text{def}}{=} (\gamma, \Lambda\alpha^K.M)$$

Figure 5.2: Value interpretation definition.

omitted. We extend the clause projection notation to handler closures and generalised continuation frames, i.e.

$$\theta^{\mathrm{ret}} \stackrel{\text{def}}{=} (\sigma, \chi^{\mathrm{ret}}) \stackrel{\text{def}}{=} H^{\mathrm{ret}}, \quad \text{where } \chi = (\gamma, H^\delta)$$
$$\theta^{\ell} \stackrel{\text{def}}{=} (\sigma, \chi^{\ell}) \stackrel{\text{def}}{=} H^{\ell}, \quad \text{where } \chi = (\gamma, H^\delta)$$

Values are annotated with types where appropriate to facilitate type reconstruction in order to make the results of Section 5.4 easier to state.

$$\longrightarrow \subseteq \mathrm{Conf} \times \mathrm{Conf}$$

M-App	$\langle V\ W \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \kappa \rangle,$	if $[\![V]\!]\gamma = (\gamma', \lambda x^A.M)$
M-AppRec	$\langle V\ W \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle M \mid \gamma'[g \mapsto (\gamma', \mathbf{rec}\, g^{A\to C} x.M), x \mapsto [\![W]\!]\gamma] \mid \kappa \rangle,$	if $[\![V]\!]\gamma = (\gamma', \mathbf{rec}\, g^{A\to C} x.M)$
M-AppType	$\langle V\ T \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle M[T/\alpha] \mid \gamma' \mid \kappa \rangle,$	if $[\![V]\!]\gamma = (\gamma', \Lambda\alpha^K.M)$
M-Resume	$\langle V\ W \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ \kappa \rangle,$	if $[\![V]\!]\gamma = (\kappa')^A$
M-Resume†	$\langle V\,W \mid \gamma \mid (\sigma,\chi) :: \kappa \rangle$	$\longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ ((\sigma' +\!\!+ \sigma, \chi) :: \kappa) \rangle,$	if $[\![V]\!]\gamma = (\kappa', \sigma')^A$
M-Resume‡	$\langle V \langle W; W' \rangle \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid \kappa' +\!\!+ [(\sigma, (\gamma'[q \mapsto [\![W']\!]\gamma], q.H))] +\!\!+ \kappa \rangle,$ if $[\![V]\!]\gamma = \kappa' +\!\!+ [(\sigma, (\gamma', q.H))])^A$	
M-Split	$\langle \mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \kappa \rangle,$	if $[\![V]\!]\gamma = \langle \ell = v; w \rangle$
M-Case	$\langle \mathbf{case}\ V \{\ell\ x \mapsto M; y \mapsto N\} \mid \gamma \mid \kappa \rangle$	$\longrightarrow \begin{cases} \langle M \mid \gamma[x \mapsto v] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \ell\, v \\ \langle N \mid \gamma[y \mapsto \ell' v] \mid \kappa \rangle, & \text{if } [\![V]\!]\gamma = \ell' v \text{ and } \ell \neq \ell' \end{cases}$	
M-Let	$\langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma,\chi) :: \kappa \rangle$	$\longrightarrow \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle$	
M-Handle$^\delta$	$\langle \mathbf{handle}^\delta\, M\ \mathbf{with}\, H^\delta \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle M \mid \gamma \mid ([], (\gamma, H^\delta)) :: \kappa \rangle$	
M-Handle‡	$\langle \mathbf{handle}^\ddagger M\ \mathbf{with}\ (q.H)(W) \mid \gamma \mid \kappa \rangle$	$\longrightarrow \langle M \mid \gamma \mid ([], (\gamma[q \mapsto [\![W]\!]\gamma], H)) :: \kappa \rangle$	
M-PureCont	$\langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle$	$\longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \chi) :: \kappa \rangle$	
M-GenCont	$\langle \mathbf{return}\ V \mid \gamma \mid ([], (\gamma', H^\delta)) :: \kappa \rangle$	$\longrightarrow \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle,$	if $H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto M\}$
M-Do$^\delta$	$\langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid ((\sigma, (\gamma', H^\delta)) :: \kappa) \circ \kappa' \rangle$	$\longrightarrow \langle M \mid \gamma'[p \mapsto [\![V]\!]\gamma, r \mapsto (\kappa' +\!\!+ [(\sigma, (\gamma', H^\delta))])^B] \mid \kappa \rangle,$ if $\ell : A \to B \in E$ and $H^\ell = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto M\}$	
M-Do†	$\langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid ((\sigma, (\gamma', H)^\dagger) :: \kappa) \circ \kappa' \rangle$	$\longrightarrow \langle M \mid \gamma'[p \mapsto [\![V]\!]\gamma, r \mapsto (\kappa', \sigma)^B] \mid \kappa \rangle,$ if $\ell : A \to B \in E$ and $H^\ell = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto M\}$	
M-Forward	$\langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid (\theta :: \kappa) \circ \kappa' \rangle$	$\longrightarrow \langle (\mathbf{do}\ \ell\ V)^E \mid \gamma \mid \kappa \circ (\kappa' +\!\!+ [\theta]) \rangle,$	if $\theta^\ell = \emptyset$

Figure 5.3: Abstract machine transitions.

Initial continuation

$\kappa_0 \stackrel{\text{def}}{=} [([], (\emptyset, \{\mathbf{return}\, x \mapsto x\}))]$

Initialisation $\longrightarrow \subseteq \text{Comp} \times \text{Conf}$

M-Init $M \longrightarrow \langle M \mid \emptyset \mid \kappa_0 \rangle$

Finalisation $\longrightarrow \subseteq \text{Conf} \times \text{Val}$

M-Halt $\langle \mathbf{return}\, V \mid \gamma \mid [] \rangle \longrightarrow [\![V]\!]\gamma$

Figure 5.4: Machine initialisation and finalisation.

5.2 Generalised continuation-based machine semantics

The semantics of the abstract machine is defined in terms of a transition relation $\longrightarrow \subseteq \text{Conf} \times \text{Conf}$ on machine configurations. The definition of the transition relation is given in Figure 5.3. A fair amount of the transition rules involve manipulating the continuation. We adopt the same stack notation conventions used in the CPS translation with generalised continuations (Section 4.4.4) and write $[]$ for an empty stack, $x :: s$ for the result of pushing x on top of stack s, and $s +\!\!+ s'$ for the concatenation of stack s on top of s'. We use pattern matching to deconstruct stacks.

The first eight rules enact the elimination of values. The first three rules concern closures (M-App, M-AppRec, M-AppType); they all essentially work the same. For example, the M-App uses the value interpretation function $[\![-]\!]$ to interpret the abstractor V in the machine environment γ to obtain the closure. The body M of closure gets put into the control component. Before the closure environment γ' gets installed as the new machine environment, it gets extended with a binding of the formal parameter of the abstraction to the interpretation of argument W in the previous environment γ. The rule M-AppRec behaves the almost the same, the only difference is that it binds the variable g to the recursive closure in the environment. The rule M-AppType does not extend the environment, instead the type is substituted directly into the body. In either rule continuation component remains untouched.

The resumption rules (M-Resume, M-Resume†, M-Resume‡), however, manipulate the continuation component as they implement the context restorative behaviour of deep, shallow, and parameterised resumption application respectively. The M-Resume rule handles deep resumption invocations. A deep resumption is syntactically a generalised continuation, and therefore it can be directly composed with the machine con-

tinuation. Following a deep resumption invocation the argument gets placed in the control component, whilst the reified continuation κ' representing the resumptions gets concatenated with the machine continuation κ in order to restore the captured context. The rule M-Resume† realises shallow resumption invocations. Syntactically, a shallow resumption consists of a pair whose first component is a dangling pure continuation σ', which is leftover after removal of its nearest enclosing handler, and the second component contains a reified generalised continuation κ'. The dangling pure continuation gets adopted by the top-most handler χ as σ' gets appended onto the pure continuation σ running under χ. The resulting continuation gets composed with the reified continuation κ'. The rule M-Resume‡ implements the behaviour of parameterised resumption invocations. Syntactically, a parameterised resumption invocation is generalised continuation just like an ordinary deep resumption. The primary difference between M-Resume and M-Resume‡ is that in the latter rule the top-most frame of κ' contains a parameterised handler definition, whose parameter q needs to be updated following an invocation. The handler closure environment γ' gets extended by a mapping of q to the interpretation of the argument W' such that this value of q is available during the next activation of the handler. Following the environment update the reified continuation gets reconstructed and appended onto the current machine continuation.

The rules M-Split and M-Case concern record destructing and variant scrutinising, respectively. Record destructing binds both the variable x to the value v at label ℓ in the record V and the variable y to the tail of the record in current environment γ. Case splitting dispatches to the first branch with the variable x bound to the variant payload in the environment if the label of the variant V matches ℓ, otherwise it dispatches to the second branch with the variable y bound to the interpretation of V in the environment.

The rules M-Let, M-Handle$^{\delta}$, and M-Handle‡ augment the current continuation with let bindings and handlers. The rule M-Let puts the computation M of a let expression into the control component and extends the current pure continuation with the closure of the (source) continuation of the let expression. The M-Handle$^{\delta}$ rule covers both ordinary deep and shallow handler installation. The computation M is placed in the control component, whilst the continuation is extended by an additional generalised frame with an empty pure continuation and the closure of the handler H. The rule M-Handle‡ covers installation of parameterised handlers. The only difference here is that the parameter q is initialised to the interpretation of W in handler environment γ'.

The current continuation gets shrunk by rules M-PureCont and M-GenCont. If the current pure continuation is nonempty then the rule M-PureCont binds a returned value,

otherwise the rule M-GenCont invokes the return clause of a handler if the pure continuation is empty.

The forwarding continuation is used by rules M-Do$^{\delta}$, M-Do†, and M-Forward. The rule M-Do$^{\delta}$ covers operation invocations under deep and parameterised handlers. If the top-most handler handles the operation ℓ, then corresponding clause computation M gets placed in the control component, and the handler environment γ' is installed with bindings of the operation payload and the resumption. The resumption is the forwarding continuation κ' extended by the current generalised continuation frame. The rule M-Do† is much like M-Do$^{\delta}$, except it constructs a shallow resumption, discarding the current handler but keeping the current pure continuation. The rule M-Forward appends the current continuation frame onto the end of the forwarding continuation.

As a slight abuse of notation, we overload $\longrightarrow$ to inject computation terms into an initial machine configuration as well as projecting values. Figure 5.4 depicts the structure of the initial machine continuation and two additional pseudo transitions. The initial continuation consists of a single generalised continuation frame with an empty pure continuation running under an identity handler. The M-Init rule provides a canonical way to map a computation term onto a configuration, whilst M-Halt provides a way to extract the final value of some computation from a configuration.

5.2.1 Putting the machine into action

To gain a better understanding of how the abstract machine concretely transitions between configurations we will consider a small program consisting of a deep, parameterised, and shallow handler. For the deep handler we will use the nondet handler from Section 2.4 which handles invocations of the operation $\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}$; it is reproduced here in fine-grain call-by-value syntax.

$$
\begin{aligned}
&\mathsf{nondet} : (1 \to \alpha!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}) \to \mathsf{List}\ \alpha \\
&\mathsf{nondet}\ m \stackrel{\text{def}}{=} \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\
&\qquad\qquad \mathbf{return}\ x \mapsto [x] \\
&\qquad\qquad \langle\!\langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow \mathit{resume} \rangle\!\rangle \mapsto \mathbf{let}\ \mathit{xs} \leftarrow \mathit{resume}\ \mathsf{true}\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \mathit{ys} \leftarrow \mathit{resume}\ \mathsf{false}\ \mathbf{in}\ \mathit{xs} +\!\!+ \mathit{ys}
\end{aligned}
$$

As for the parameterised handler we will use a handler, which implements a simple counter that supports one operation $\mathsf{Incr} : 1 \twoheadrightarrow \mathsf{Int}$, which increments the value of the

counter and returns the previous value. It is defined as follows.

$$\begin{array}{l}
\mathsf{incr} : \langle \mathsf{Int}; 1 \to \alpha!\{\mathsf{Incr} : 1 \twoheadrightarrow \mathsf{Int}\}\rangle \to \alpha \\
\mathsf{incr}\,\langle i_0; m\rangle \stackrel{\text{def}}{=} \mathbf{handle}^{\ddagger}\ m\,\langle\rangle\ \mathbf{with} \\
\qquad \left(\begin{array}{ll} i. & \mathbf{return}\ x \mapsto \mathbf{return}\ x \\ & \langle\!\langle \mathsf{Incr}\ \langle\rangle \twoheadrightarrow \mathit{resume}\rangle\!\rangle \mapsto \mathbf{let}\ i' \leftarrow i+1\ \mathbf{in}\ \mathit{resume}\,\langle i'; i\rangle \end{array}\right)\ i_0
\end{array}$$

We will use the pipe and copipe shallow handlers from Section 2.6 to construct a small pipeline.

$$\begin{array}{l}
\mathsf{pipe} : \langle 1 \to \alpha!\{\mathsf{Yield} : \beta \twoheadrightarrow 1\}; 1 \to \alpha!\{\mathsf{Await} : 1 \twoheadrightarrow \beta\}\rangle \to \alpha \\
\mathsf{pipe}\,\langle p; c\rangle \stackrel{\text{def}}{=} \mathbf{handle}^{\dagger}\ c\,\langle\rangle\ \mathbf{with} \\
\qquad \begin{array}{ll} \mathbf{return}\ x & \mapsto \mathbf{return}\ x \\ \langle\!\langle \mathsf{Await}\ \langle\rangle \twoheadrightarrow \mathit{resume}\rangle\!\rangle & \mapsto \mathsf{copipe}\,\langle \mathit{resume}; p\rangle \end{array} \\
\mathsf{copipe} : \langle \beta \to \alpha!\{\mathsf{Await} : 1 \twoheadrightarrow \beta\}; 1 \to \alpha!\{\mathsf{Yield} : \beta \twoheadrightarrow 1\}\rangle \to \alpha \\
\mathsf{copipe}\,\langle c; p\rangle \stackrel{\text{def}}{=} \mathbf{handle}^{\dagger}\ p\,\langle\rangle\ \mathbf{with} \\
\qquad \begin{array}{ll} \mathbf{return}\ x & \mapsto \mathbf{return}\ x \\ \langle\!\langle \mathsf{Yield}\ y \twoheadrightarrow \mathit{resume}\rangle\!\rangle & \mapsto \mathsf{pipe}\,\langle \mathit{resume}; \lambda\langle\rangle.c\,y\rangle \end{array}
\end{array}$$

We use the following the producer and consumer computations for the pipes.

$$\begin{array}{l}
\mathsf{prod} : 1 \to \alpha!\{\mathsf{Incr} : 1 \twoheadrightarrow \mathsf{Int}; \mathsf{Yield} : \mathsf{Int} \twoheadrightarrow 1\} \\
\mathsf{prod}\,\langle\rangle \stackrel{\text{def}}{=} \mathbf{let}\ j \leftarrow \mathbf{do}\ \mathsf{Incr}\,\langle\rangle\ \mathbf{in}\ \mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{Yield}\ j\ \mathbf{in}\ \mathsf{prod}\,\langle\rangle \\
\mathsf{cons} : 1 \to \mathsf{Int}!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}; \mathsf{Await} : 1 \twoheadrightarrow \mathsf{Int}\} \\
\mathsf{cons}\,\langle\rangle \stackrel{\text{def}}{=} \mathbf{let}\ b \leftarrow \mathbf{do}\ \mathsf{Fork}\,\langle\rangle\ \mathbf{in}\ \mathbf{let}\ x \leftarrow \mathbf{do}\ \mathsf{Await}\,\langle\rangle\ \mathbf{in} \\
\qquad \mathbf{if}\ b\ \mathbf{then}\ x * 2\ \mathbf{else}\ x * x
\end{array}$$

The producer computation prod invokes the operation Incr to increment and retrieve the previous value of some counter. This value is supplied as the payload to an invocation of Yield. The consumer computation cons first performs an invocation of Fork to duplicate the stream, and then it performs an invocation Await to retrieve some value. The return value of cons depends on the instance runs in the original stream or forked stream. The original stream multiplies the retrieved value by 2, and the duplicate squares the value. Finally, the top-level computation plugs all of the above together.

$$\mathsf{nondet}\,(\lambda\langle\rangle.\mathsf{incr}\,\langle 1; \lambda\langle\rangle.\mathsf{pipe}\,\langle \mathsf{prod}; \mathsf{cons}\rangle\rangle) \tag{5.1}$$

Function interpretation is somewhat heavy notation-wise as environments need to be built. To make the notation a bit more lightweight I will not define the initial environments for closures explicitly. By convention I will subscript initial environments with

the name of function, e.g. γ_{cons} denotes the initial environment for the closure of cons. Extensions of initial environments will use superscripts to differentiate themselves, e.g. γ'_{cons} is an extension of γ_{cons}. As a final environment simplification, I will take the initial environments to contain the bindings for parameters of their closures, that is, an initial environment is really the environment for the body of its closure. In a similar fashion, I will use superscripts and subscripts to differentiate handler closures, e.g. $\chi^{\dagger}_{\mathsf{pipe}}$ denotes the handler closure for the shallow handler definition in pipe. The environment of a handler closure is to be understood implicitly. Furthermore, the definitions above should be understood to be implicitly **let**-sequenced, whose tail computation is (5.1). Evaluation of this sequence gives rise to a 'toplevel' environment, which binds the closures for the definition. I shall use γ_0 to denote this environment.

The machine executes the top-level computation in an initial configuration with the top-level environment γ_0. The first couple of transitions install the three handlers in order: nondet, incr, and pipe.

$$
\begin{aligned}
& \mathsf{nondet}\,(\lambda\langle\rangle.\mathsf{incr}\,\langle 1;\lambda\langle\rangle.\mathsf{pipe}\,\langle\mathsf{prod};\mathsf{cons}\rangle\rangle) \\
\longrightarrow\quad & (\text{M-Init with } \gamma_0) \\
& \langle \mathsf{nondet}\,(\lambda\langle\rangle.\mathsf{incr}\,\langle 0;\lambda\langle\rangle.\mathsf{pipe}\,\langle\mathsf{prod};\mathsf{cons}\rangle\rangle) \mid \gamma_0 \mid \kappa_0\rangle \\
\longrightarrow^{+}\quad & (3\times(\text{M-App, M-Handle}^{\delta})) \\
& \langle c\,\langle\rangle \mid \gamma_{\mathsf{pipe}} \mid ([],\chi^{\dagger}_{\mathsf{pipe}}) :: ([],\chi^{\ddagger}_{\mathsf{incr}}) :: ([],\chi_{\mathsf{nondet}}) :: \kappa_0\rangle
\end{aligned}
$$

At this stage the continuation consists of four frames. The first three frames each corresponds to an installed handler, whereas the last frame is the identity handler. The control component focuses the application of consumer computation provided as an argument to pipe. The next few transitions get us to the first operation invocation.

$$
\begin{aligned}
\longrightarrow^{+}\quad & (\text{M-App, M-Let}) \\
& \langle \mathbf{do}\ \mathsf{Fork}\,\langle\rangle \mid \gamma_{\mathsf{cons}} \mid ([(\gamma_{\mathsf{cons}}, b, \mathbf{let}\ x \leftarrow \cdots)],\chi^{\dagger}_{\mathsf{pipe}}) :: ([],\chi^{\ddagger}_{\mathsf{incr}}) :: ([],\chi_{\mathsf{nondet}}) :: \kappa_0\rangle \\
\longrightarrow^{+}\quad & (\text{M-Forward, M-Forward}) \\
& \langle \mathbf{do}\ \mathsf{Fork}\,\langle\rangle \mid \gamma_{\mathsf{cons}} \mid [([],\chi_{\mathsf{nondet}}),([],\chi_{\mathsf{id}})] \circ \kappa'\rangle \\
& \text{where } \kappa' = [([(\gamma_{\mathsf{cons}}, b, \mathbf{let}\ x \leftarrow \cdots)],\chi^{\dagger}_{\mathsf{pipe}}),([],\chi^{\ddagger}_{\mathsf{incr}})]
\end{aligned}
$$

The pure continuation under $\chi^{\dagger}_{\mathsf{pipe}}$ has been augmented with the pure frame corresponding to **let**-binding of the invocation of Fork. Operation invocation causes the machine to initiate a search for a suitable handler, as the top-most handler pipe does not handle Fork. The machine performs two M-Forward transitions, which moves the two top-most frames from the program continuation onto the forwarding continuation. As a result the,

now, top-most frame of the program continuation contains a suitable handler for Fork. Thus the following transitions transfer control to Fork-case inside the nondet handler.

$$\begin{aligned}
&\longrightarrow^{+} \quad (\text{M-Do, M-Let}) \\
&\quad \langle \mathit{resume}\ \mathsf{true} \mid \gamma'_{\mathsf{nondet}} \mid \kappa'_0 \rangle \\
&\quad \text{where}\ \ \gamma'_{\mathsf{nondet}} = \gamma_{\mathsf{nondet}}[\mathit{resume} \mapsto \kappa' +\!\!+ [([], \chi_{\mathsf{nondet}})]] \\
&\qquad\qquad \kappa'_0 = [([(\gamma'_{\mathsf{nondet}}, \mathit{xs}, \mathbf{let}\ \mathit{ys} \leftarrow \cdots)], \chi_{\mathrm{id}})]
\end{aligned}$$

The M-Do transition is responsible for activating the handler, and the M-Let transition focuses the first resumption invocation. The resumption *resume* is bound in the environment to the forwarding continuation κ' extended with the frame for the current handler. The pure continuation running under the identity handler gets extended with the **let**-binding containing the first resumption invocation. The next transitions reassemble the program continuation and focuses control on the invocation of Await.

$$\begin{aligned}
&\longrightarrow^{+} \quad (\text{M-Resume, M-PureCont, M-Let}) \\
&\quad \langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle \mid \gamma'_{\mathsf{cons}} \mid ([(\gamma'_{\mathsf{cons}}, x, \mathbf{if}\ b \cdots)], \chi^{\dagger}_{\mathsf{pipe}}) :: ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
&\quad \text{where}\ \ \gamma'_{\mathsf{cons}} = \gamma_{\mathsf{cons}}[b \mapsto \mathsf{true}]
\end{aligned}$$

At this stage the context of cons has been restored with b being bound to the value true. The pure continuation running under pipe has been extended with pure frame corresponding to the continuation of the **let**-binding of the Await invocation. Handling of this invocation requires no use of the forwarding continuation as the top-most frame contains a suitable handler.

$$\begin{aligned}
&\longrightarrow \quad (\text{M-Do}^{\dagger}) \\
&\quad \langle \mathsf{copipe}\ \langle \mathit{resume}; p \rangle \mid \gamma'_{\mathsf{pipe}} \mid ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
&\quad \text{where}\ \ \gamma'_{\mathsf{pipe}} = \gamma_{\mathsf{pipe}}[\mathit{resume} \mapsto ([], [(\gamma'_{\mathsf{cons}}, x, \mathbf{if}\ b \cdots)])]
\end{aligned}$$

Now the Await-case of the pipe handler has been activated. The resumption *resume* is bound to the shallow resumption in the environment. The generalised continuation component of the shallow resumption is empty, because no forwarding was involved in locating the handler. The next transitions install the copipe handler and runs the

producer computation.

$$
\begin{aligned}
\longrightarrow^{+} \quad & (\text{M-App, M-Handle}^{\dagger}) \\
& \langle p\,\langle\rangle \mid \gamma'_{\mathsf{copipe}} \mid ([], \chi^{\dagger}_{\mathsf{copipe}}) :: \kappa' \rangle \\
& \text{where } \gamma'_{\mathsf{copipe}} = \gamma_{\mathsf{copipe}}[c \mapsto ([], [(\gamma'_{\mathsf{cons}}, x, \mathbf{if}\, b \cdots)])] \\
\longrightarrow^{+} \quad & (\text{M-AppRec, M-Let}) \\
& \langle \mathbf{do}\ \mathsf{Incr}\,\langle\rangle \mid \gamma_{\mathsf{prod}} \mid ([(\gamma_{\mathsf{prod}}, j, \mathbf{let}\, x \leftarrow \cdots)], \chi^{\dagger}_{\mathsf{copipe}}) :: \kappa' \rangle \\
\longrightarrow^{+} \quad & (\text{M-Forward, M-Do}^{\ddagger}\text{, M-Let, M-App, M-PureCont}) \\
& \langle \mathit{resume}\,\langle i'; i\rangle \mid \gamma'_{\mathsf{incr}} \mid ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
& \text{where } \gamma'_{\mathsf{incr}} = \gamma_{\mathsf{incr}}[i \mapsto 1, i' \mapsto 2, \\
& \qquad\qquad \mathit{resume} \mapsto [([(\gamma_{\mathsf{prod}}, j, \mathbf{let}\, x \leftarrow \cdots)], \chi^{\dagger}_{\mathsf{copipe}}), ([], \chi^{\ddagger}_{\mathsf{incr}})]]
\end{aligned}
$$

The producer computation performs the Incr operation, which requires one M-Forward transition in order to locate a suitable handler for it. The Incr-case of the incr handler increments the counter i by one. The environment binds the current value of the counter. The following M-Resume‡ transition updates the counter value to be that of i' and continues the producer computation.

$$
\begin{aligned}
\longrightarrow^{+} \quad & (\text{M-Resume}^{\ddagger}\text{, M-PureCont, M-Let}) \\
& \langle \mathbf{do}\ \mathsf{Yield}\, j \mid \gamma''_{\mathsf{prod}} \mid ([(\gamma''_{\mathsf{prod}}, x, \mathsf{prod}\,\langle\rangle)], \chi^{\dagger}_{\mathsf{copipe}}) :: ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
& \text{where } \gamma''_{\mathsf{prod}} = \gamma'_{\mathsf{prod}}[j \mapsto 1] \\
\longrightarrow \quad & (\text{M-Do}^{\dagger}) \\
& \langle \mathsf{pipe}\,\langle \mathit{resume}; \lambda\langle\rangle.c\, y\rangle \mid \gamma'_{\mathsf{pipe}} \mid ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
& \text{where } \gamma'_{\mathsf{pipe}} = \gamma_{\mathsf{pipe}}[y \mapsto 1, \mathit{resume} \mapsto ([], [(\gamma''_{\mathsf{prod}}, x, \mathsf{prod}\,\langle\rangle)])] \\
\longrightarrow^{+} \quad & (\text{M-App, M-Handle}^{\dagger}\text{, M-Resume}^{\dagger}\text{, M-PureCont}) \\
& \langle \mathbf{if}\, b\ \mathbf{then}\, x * 2\ \mathbf{else}\, x * x \mid \gamma''_{\mathsf{cons}} \mid ([], \chi^{\dagger}_{\mathsf{pipe}}) :: ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa'_0 \rangle \\
& \text{where } \gamma''_{\mathsf{cons}} = \gamma'_{\mathsf{cons}}[x \mapsto 1]
\end{aligned}
$$

The Yield operation causes another instance of the pipe to be installed in place of the copipe. The M-Resume† transition occurs because the consumer argument provided to pipe is the resumption of captured by the original instance of pipe, thus invoking it causes the context of the original consumer computation to be restored. Since b is true the **if**-expression will dispatch to the **then**-branch, meaning the computation will ultimately return 2. This return value gets propagated through the handler stack.

$$
\begin{aligned}
\longrightarrow^{+} \quad & (\text{M-Case, M-App, M-GenCont, M-GenCont, M-GenCont}) \\
& \langle \mathbf{return}\ [x] \mid \gamma_{\mathsf{nondet}}[x \mapsto 2] \mid \kappa'_0 \rangle
\end{aligned}
$$

The **return**-clauses of the pipe and incr handlers are identities, and thus, the return value x passes through unmodified. The **return**-case of nondet lifts the value into a singleton

list. Next the pure continuation is invoked, which restores the handling context of the first operation invocation Fork.

$$
\begin{array}{ll}
\longrightarrow & \text{(M-PureCont, M-Let)} \\
& \langle \mathit{resume}\ \mathsf{false} \mid \gamma'_{\mathsf{nondet}} \mid \kappa''_0 \rangle \\
& \text{where}\ \ \gamma''_{\mathsf{nondet}} = \gamma'_{\mathsf{nondet}}[xs \mapsto [3]] \\
& \qquad\qquad \kappa''_0 = [([(\gamma''_{\mathsf{nondet}}, ys, xs +\!\!+ ys)], \chi_{\mathrm{id}})] \\
\longrightarrow^{+} & \text{(M-Resume, M-PureCont, M-Let)} \\
& \langle \mathbf{do}\ \mathsf{Await}\ \langle\rangle \mid \gamma'''_{\mathsf{cons}} \mid ([(\gamma'''_{\mathsf{cons}}, x, \mathbf{if}\ b \cdots)], \chi^{\dagger}_{\mathsf{pipe}}) :: ([], \chi^{\ddagger}_{\mathsf{incr}}) :: ([], \chi_{\mathsf{nondet}}) :: \kappa''_0 \rangle \\
& \text{where}\ \ \gamma'''_{\mathsf{cons}} = \gamma''_{\mathsf{cons}}[b \mapsto \mathsf{false}]
\end{array}
$$

The second invocation of the resumption *resume* interprets Fork as false. The consumer computation is effectively restarted with b bound to false. The previous transitions will be repeated.

$$
\begin{array}{ll}
\longrightarrow^{+} & \text{(same reasoning as above)} \\
& \langle \mathit{resume}\ \langle i'; i \rangle \mid \gamma''_{\mathsf{incr}} \mid ([], \chi_{\mathsf{nondet}}) :: \kappa''_0 \rangle \\
& \text{where}\ \ \gamma'_{\mathsf{incr}} = \gamma_{\mathsf{incr}}[i \mapsto 2, i' \mapsto 3, \\
& \qquad\qquad \mathit{resume} \mapsto [([(\gamma''_{\mathsf{prod}}, i, \mathbf{let}\ x \leftarrow \cdots)], \chi^{\dagger}_{\mathsf{copipe}}), ([], \chi^{\ddagger}_{\mathsf{incr}})]]
\end{array}
$$

After some amount transitions the parameterised handler incr will be activated again. The counter variable i is bound to the value computed during the previous activation of the handler. The machine proceeds as before and eventually reaches concatenation application inside the Fork-case.

$$
\begin{array}{ll}
\longrightarrow^{+} & \text{(same reasoning as above)} \\
& \langle xs +\!\!+ ys \mid \gamma''_{\mathsf{nondet}} \mid \kappa_0 \rangle \\
& \text{where}\ \ \gamma''_{\mathsf{nondet}} = \gamma'_{\mathsf{nondet}}[ys \mapsto [4]] \\
\longrightarrow & \text{(M-App, M-GenCont, M-Halt)} \\
& [3, 4]
\end{array}
$$

5.3 Realisability and efficiency implications

A practical benefit of the abstract machine semantics over the context-based small-step reduction semantics with explicit substitutions is that it provides either a blueprint for a high-level interpreter-based implementation or an outline for how stacks should be manipulated in a low-level implementation along with a more practical and precise cost model. The cost model is more practical in the sense of modelling how actual hardware

might go about executing instructions, and it is more precise as it eliminates the declarative aspect of the contextual semantics induced by the S-Lift rule. For example, the asymptotic cost of handler lookup is unclear in the contextual semantics, whereas the abstract machine clearly tells us that handler lookup involves a linear search through the machine continuation.

The abstract machine is readily realisable using standard persistent functional data structures such as lists and maps [210]. The concrete choice of data structures required to realise the abstract machine is not set in stone, although, its definition is suggestive about the choice of data structures it leaves space for interpretation. For example, generalised continuations can be implemented using lists, arrays, or even heaps. However, the concrete choice of data structure is going to impact the asymptotic time and space complexity of the primitive operations on continuations: continuation augmentation ($::$) and concatenation ($+\!\!+$). For instance, a linked list provides a fast constant time implementations of either operation, whereas a fixed-size array can only provide implementations of either operation that run in linear time due to the need to resize and copy contents in the extreme case. An implementation based on a singly-linked list admits constant time for both continuation augmentation as this operation corresponds directly to list cons. However, it admits only a linear time implementation for continuation concatenation. Alternatively, an implementation based on a Hughes list [131] reverses the cost as a Hughes list uses functions to represent cons cells, thus meaning concatenation is simply function composition, but accessing any element, including the head, always takes linear time in the size of the list. In practice, this difference in efficiency means we can either trade-off fast interpretation of **let**-bindings and **handle**-computations for 'slow' handling and context restoration or vice versa depending on what we expect to occur more frequently.

The pervasiveness of **let**-bindings in fine-grain call-by-value means that the topmost pure continuation is likely to be augmented and shrunk repeatedly, thus it is a sensible choice to simply represent generalisation continuations as singly linked list in order to provide constant time pure continuation augmentation (handler installation would be constant time too). However, the continuation component contains two generalisation continuations. In the rule M-Forward the forwarding continuation is extended using concatenation, thus we may choose to represent the forwarding continuation as a Hughes list for greater efficiency. A consequence of this choice is that upon resumption invocation we must convert the forwarding continuation into singly linked list such that it can be concatenated with the program continuation. Both the conversion

and the concatenation require a full linear traversal of the forwarding continuation. A slightly clever choice is to represent both continuations using Huet's Zipper data structure [130], which essentially boils down to using a pair of singly linked lists, where the first component contains the program continuation, and the second component contains the forwarding continuation. We can make a non-asymptotic improvement by representing the forwarding continuation as a reversed continuation such that we may interpret the concatenation operation ($+\!\!+$) in M-Forward as regular cons (::). In the M-Resume$^{\delta}$ rules we must then interpret concatenation as reverse append, which needs to traverse the forwarding continuation only once.

Continuation copying A convenient consequence of using persistent functional data structure to realise the abstract machine is that multi-shot resumptions become efficiency as continuation copying becomes a constant time operation. However, if we were only interested one-shot or linearly used resumptions, then we may wish to use in-place mutations to achieve greater efficiency. In-place mutations do not exclude support for multi-shot resumptions, however, with mutable data structures the resumptions needs to be copied before use. One possible way to copy resumptions is to expose an explicit copy instruction in the source language. Alternatively, if the source language is equipped with a linear type system, then the linear type information can be leveraged to provide an automatic insertion of copy instructions prior to resumption invocations.

5.4 Simulation of the context-based reduction semantics

We now show that the base abstract machine is correct with respect to the combined context-based small-step semantics of λ_{h}, $\lambda_{\mathsf{h}^{\dagger}}$, and $\lambda_{\mathsf{h}^{\ddagger}}$ via a simulation result.

Initial states provide a canonical way to map a computation term onto the abstract machine. A more interesting question is how to map an arbitrary configuration to a computation term. Figure 5.5 describes such a mapping $(\!|-|\!)$ from configurations to terms via a collection of mutually recursive functions defined on configurations, continuations, handler closures, computation terms, handler definitions, value terms, and machine values. The mapping makes use of a domain operation and a restriction operation on environments.

Definition 5.1. The domain of an environment is defined recursively as follows.

$$\begin{aligned}
&\mathsf{dom} : \mathrm{Env} \to \mathrm{Var} \\
&\mathsf{dom}(\emptyset) &&\stackrel{\text{def}}{=} \emptyset \\
&\mathsf{dom}(\gamma[x \mapsto v]) &&\stackrel{\text{def}}{=} \{x\} \cup \mathsf{dom}(\gamma)
\end{aligned}$$

We write $\gamma \backslash \{x_1, \ldots, x_n\}$ for the restriction of environment γ to $\mathsf{dom}(\gamma) \backslash \{x_1, \ldots, x_n\}$.

The $(\!|-|\!)$ function enables us to classify the abstract machine reduction rules according to how they relate to the context-based small-step semantics. Both the rules M-Let and M-Forward are administrative in the sense that $(\!|-|\!)$ is invariant under either rule. This leaves the β-rules M-App, M-AppRec, M-TyApp, M-Resume$^{\delta}$, M-Split, M-Case, M-PureCont, and M-GenCont. Each of these corresponds directly with performing a reduction in the small-step semantics. We extend the notion of transition to account for administrative steps.

Definition 5.2 (Auxiliary reduction relations)**.** We write $\longrightarrow_{\mathrm{a}}$ for administrative steps and $\simeq_{\mathrm{a}}$ for the symmetric closure of $\longrightarrow_{\mathrm{a}}^{*}$. We write $\longrightarrow_{\beta}$ for β-steps and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_{\mathrm{a}}^{*}\longrightarrow_{\beta}$.

The following lemma describes how we can simulate each reduction in the small-step reduction semantics by a sequence of administrative steps followed by one β-step in the abstract machine.

Lemma 5.3. *Suppose M is a computation and $\mathcal{C}$ is configuration such that $(\!|\mathcal{C}|\!) = M$, then if $M \rightsquigarrow N$ there exists $\mathcal{C}'$ such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $(\!|\mathcal{C}'|\!) = N$, or if $M \not\rightsquigarrow$ then $\mathcal{C} \not\Longrightarrow$.*

Proof. By induction on the derivation of $M \rightsquigarrow N$. □

The correspondence here is rather strong: there is a one-to-one mapping between $\rightsquigarrow$ and the quotient relation of $\Longrightarrow$ and $\simeq_{\mathrm{a}}$. Notice that Lemma 5.3 does not require that M be well-typed. This is mostly a convenience to simplify the lemma. The lemma is used in the following theorem where it is being applied only on well-typed terms.

Theorem 5.4 (Simulation)**.** *If $\vdash M : A!E$ and $M \rightsquigarrow^{+} N$ such that N is normal with respect to E, then $\langle M \mid \emptyset \mid \kappa_0 \rangle \longrightarrow^{+} \mathcal{C}$ such that $(\!|\mathcal{C}|\!) = N$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid \kappa_0 \rangle \not\longrightarrow$.*

Proof. By repeated application of Lemma 5.3. □

5.5 Related work

The literature on abstract machines is vast and rich. I describe here the basic structure of some selected abstract machines from the literature.

Handler machines Chronologically, the machine presented in this chapter was the first abstract machine specifically designed for effect handlers to appear in the literature. Subsequently, this machine has been extended and used to explain the execution model for the Multicore OCaml implementation [252]. Their primary extension captures the finer details of the OCaml runtime as it models the machine continuation as a heterogeneous sequence consisting of interleaved OCaml and C frames.

An alternative machine has been developed by Biernacki et al. [27] for the Helium language. Although, their machine is based on Biernacka et al.'s definitional abstract machine for the control operators shift and reset [24], the continuation structure of the resulting machine is essentially the same as that of a generalised continuation. The primary difference is that in their presentation a generalised frame is either pair consisting of a handler closure and a pure continuation (as in the presentation in this chapter) or a coercion paired with a pure continuation.

SECD machine Landin's SECD machine was the first abstract machine for λ-calculus viewed as a programming language [58, 158]. The machine is named after its structure as it consists of a *stack* component, *environment* component, *control* component, and a *dump* component. The stack component maintains a list of intermediate value. The environment maps free variables to values. The control component holds a list of directives that manipulate the stack component. The dump acts as a caller-saved register as it maintains a list of partial machine state snapshots. Prior to a closure application, the machine snapshots the state of the stack, environment, and control components such that this state can be restored once the stack has been reduced to a single value and the control component is empty. The structure of the SECD machine lends itself to a simple realisation of the semantics of Landin's the J operator as its behaviour can realised by reifying the dump the as value. Plotkin [220] proved the correctness of the machine in style of a simulation result with respect to a reduction semantics [1].

The SECD machine is a precursor to the CEK machine as the latter can be viewed as a streamlined variation of the SECD machine, where the continuation component unifies stack and dump components of the SECD machine. For a deep dive into the

operational details of Landin's SECD machine, the reader may consult Danvy [58], who dissects the SECD machine, and as a follow up on that work Danvy and Millikin [64] perform several rational deconstructions and reconstructions of the SECD machine with the J operator.

Krivine machine The Krivine machine takes its name after its designer Krivine [156]. It is designed for call-by-name λ-calculus computation as it performs reduction to weak head normal form [156, 168]. The structure of the Krivine machine is similar to that of the CEK machine as it features a control component, which focuses the current term under evaluation; an environment component, which binds variables to closures; and a stack component, which contains a list of closures. Evaluation of an application term pushes the argument along with the current environment onto the stack and continues to evaluate the abstractor term. Dually evaluation of a λ-abstraction places the body in the control component, subsequently it pops the top most closure from the stack and extends the current environment with this closure [73].

Krivine [156] has also designed a variation of the machine which supports a call-by-name variation of the callcc control operator. In this machine continuations have the same representation as the stack component, and they can be stored on the stack. Then the continuation capture mechanism of callcc can be realised by popping and installing the top-most closure from the stack, and then saving the tail of the stack as the continuation object, which is to be placed on top of the stack. An application of a continuation can be realised by replacing the current stack with the stack embedded inside the continuation object [156].

ZINC machine The ZINC machine is a strict variation of Krivine's machine, though it was designed independently by Leroy [168]. The machine is used as the basis for the OCaml byte code interpreter [168, 169]. There are some cosmetic difference between Krivine's machine and the ZINC machine. For example, the latter decomposes the stack component into an argument stack, holding arguments to function calls, and a return stack, which holds closures. A peculiar implementation detail of the ZINC machine that affects the semantics of the OCaml language is that for n-ary function application to be efficient, function arguments are evaluated right-to-left rather than left-to-right as customary in call-by-value language [168]. The OCaml manual leaves the evaluation order for function arguments unspecified [169]. However, for a long time the native code compiler for OCaml would emit code utilised left-to-right evaluation order for

function arguments, consequently the compilation method could affect the semantics of a program, as the evaluation order could be observed using effects, e.g. by raising an exception [43]. Anecdotally, Damien Doligez told me in person at ICFP 2017 that unofficially the compiler has been aligned with the byte code interpreter such that code running on either implementation exhibits the same semantics. Even though the evaluation order remains unspecified in the manual any other observable order than right-to-left evaluation order is now considered a bug (subject to some exceptions, notably short-circuiting logical and/or functions).

Mechanical machine derivations There are deep mathematical connections between environment-based abstract machine semantics and standard reduction semantics with explicit substitutions. For example, Ager et al. [1, 2, 3] relate abstract machines and functional evaluators by way of a two-way derivation that consists of closure conversion, transformation into CPS, and defunctionalisation of continuations. Biernacka and Danvy [22] demonstrate how to formally derive an abstract machine from a small-step reduction strategy. Their presentation has been formalised by Swierstra [258] in the dependently-typed programming language Agda. Hutton and Wright [133] demonstrate how to calculate a correct-by-construction abstract machine from a given specification using structural induction. Notably, their example machine supports basic computational effects in the form of exceptions. Ager et al. [4] also extended their technique to derive abstract machines from monadic-style effectful evaluators.

Configurations

$$(\!|\langle M \mid \gamma \mid \kappa \circ \kappa' \rangle|\!) \stackrel{\text{def}}{=} (\!|\kappa' +\!\!+ \kappa|\!)((\!|M|\!)\gamma) \stackrel{\text{def}}{=} (\!|\kappa'|\!)((\!|\kappa|\!)(\!|M|\!)\gamma)$$

Pure continuations

$$(\!|[]|\!)M \stackrel{\text{def}}{=} M \qquad (\!|((\gamma, x, N) :: \sigma)|\!)M \stackrel{\text{def}}{=} (\!|\sigma|\!)(\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\}))$$

Continuations

$$(\!|[]|\!)M \stackrel{\text{def}}{=} M \qquad (\!|(\sigma, \chi) :: \kappa|\!)M \stackrel{\text{def}}{=} (\!|\kappa|\!)((\!|\chi|\!)((\!|\sigma|\!)(M)))$$

Handler closures

$$(\!|(\gamma, H^{\delta})|\!)M \stackrel{\text{def}}{=} \mathbf{handle}^{\delta}\ M\ \mathbf{with}\ (\!|H^{\delta}|\!)\gamma$$

Computation terms

$$\begin{aligned}
(\!|V\,W|\!)\gamma &\stackrel{\text{def}}{=} (\!|V|\!)\gamma(\!|W|\!)\gamma \\
(\!|V\,T|\!)\gamma &\stackrel{\text{def}}{=} (\!|V|\!)\gamma T \\
(\!|\mathbf{let}\ \langle \ell = x; y \rangle = V\ \mathbf{in}\ N|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{let}\ \langle \ell = x; y \rangle = (\!|V|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x, y\}) \\
(\!|\mathbf{case}\ V\,\{\ell\, x \mapsto M; y \mapsto N\}|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{case}\ (\!|V|\!)\gamma\{\ell\, x \mapsto (\!|M|\!)(\gamma \backslash \{x\}); y \mapsto (\!|N|\!)(\gamma \backslash \{y\})\} \\
(\!|\mathbf{return}\ V|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{return}\ (\!|V|\!)\gamma \\
(\!|\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{let}\ x \leftarrow (\!|M|\!)\gamma\ \mathbf{in}\ (\!|N|\!)(\gamma \backslash \{x\}) \\
(\!|\mathbf{do}\ \ell\ V|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{do}\ \ell\ (\!|V|\!)\gamma \\
(\!|\mathbf{handle}^{\delta}\ M\ \mathbf{with}\ H|\!)\gamma &\stackrel{\text{def}}{=} \mathbf{handle}^{\delta}\ (\!|M|\!)\gamma\ \mathbf{with}\ (\!|H|\!)\gamma
\end{aligned}$$

Handler definitions

$$\begin{aligned}
(\!|\{\mathbf{return}\ x \mapsto M\}|\!)\gamma &\stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto (\!|M|\!)(\gamma \backslash \{x\})\} \\
(\!|\{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto M\} \uplus H^{\delta}|\!)\gamma &\stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto (\!|M|\!)(\gamma \backslash \{p, r\}\} \uplus (\!|H^{\delta}|\!)\gamma \\
(\!|(q.H)|\!)\gamma &\stackrel{\text{def}}{=} (\!|H|\!)(\gamma \backslash \{q\})
\end{aligned}$$

Value terms and values

$$\begin{aligned}
(\!|x|\!)\gamma &\stackrel{\text{def}}{=} (\!|v|\!), \quad \text{if } \gamma(x) = v & (\!|\kappa^A|\!) &\stackrel{\text{def}}{=} \lambda x^A.(\!|\kappa|\!)(\mathbf{return}\ x) \\
(\!|x|\!)\gamma &\stackrel{\text{def}}{=} x, \quad \text{if } x \notin \mathsf{dom}(\gamma) & (\!|(\kappa, \sigma)^A|\!) &\stackrel{\text{def}}{=} \lambda x^A.(\!|\sigma|\!)((\!|\kappa|\!)(\mathbf{return}\ x)) \\
(\!|\lambda x^A.M|\!)\gamma &\stackrel{\text{def}}{=} \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) & (\!|(\gamma, \lambda x^A.M)|\!) &\stackrel{\text{def}}{=} \lambda x^A.(\!|M|\!)(\gamma \backslash \{x\}) \\
(\!|\Lambda\alpha^K.M|\!)\gamma &\stackrel{\text{def}}{=} \Lambda\alpha^K.(\!|M|\!)\gamma & (\!|(\gamma, \Lambda\alpha^K.M)|\!) &\stackrel{\text{def}}{=} \Lambda\alpha^K.(\!|M|\!)\gamma \\
(\!|\langle\rangle|\!)\gamma &\stackrel{\text{def}}{=} \langle\rangle & (\!|\langle\rangle|\!) &\stackrel{\text{def}}{=} \langle\rangle \\
(\!|\langle \ell = V; W \rangle|\!)\gamma &\stackrel{\text{def}}{=} \langle \ell = (\!|V|\!)\gamma; (\!|W|\!)\gamma \rangle & (\!|\langle \ell = v; w \rangle|\!) &\stackrel{\text{def}}{=} \langle \ell = (\!|v|\!); (\!|w|\!) \rangle \\
(\!|(\ell\ V)^R|\!)\gamma &\stackrel{\text{def}}{=} (\ell\ (\!|V|\!)\gamma)^R & (\!|(\ell\ v)^R|\!) &\stackrel{\text{def}}{=} (\ell\ (\!|v|\!))^R
\end{aligned}$$

$$(\!|\mathbf{rec}\, g^{A \to C}\, x.M|\!)\gamma \stackrel{\text{def}}{=} \mathbf{rec}\, g^{A \to C}\, x.(\!|M|\!)(\gamma \backslash \{g, x\}) \stackrel{\text{def}}{=} (\!|(\gamma, \mathbf{rec}\, g^{A \to C}\, x.M)|\!)$$

Figure 5.5: Mapping from abstract machine configurations to terms.

Part III

Expressiveness

Chapter 6

Interdefinability of effect handlers

On the surface, shallow handlers seem to offer more flexibility than deep handlers as they do not enforce a particular recursion scheme over effectful computations. An interesting hypothesis worth investigating is whether this flexibility is a mere programming convenience or whether it enables shallow handlers to implement programs that would otherwise be impossible to implement with deep handlers. Put slightly different, the hypothesis to test is whether handlers can implement one another. To test this sort of hypothesis we first need to pin down what it means for 'something to be able to implement something else'.

For example in Section 2.6 I asserted that shallow handlers provide the natural basis for implementing pipes, suggesting that an implementation based on deep handlers would be fiddly. If we were to consider the wider design space of programming language features, then it turns out that deep handlers offer a direct implementation of pipes by shifting recursion from terms to the level of types (the interested reader may consult either Kammar et al. [143] or Hillerström and Lindley [120] for the precise details). Thus in some sense pipes are implementable with deep handlers, however, this particular implementation strategy is not realisable in the λ_{h}-calculus since it has no notion of recursive types, meaning we cannot use this strategy to argue that deep handlers can implement pipes in our setting.

We will restrict our attention to the calculi λ_{h}, $\lambda_{\mathsf{h}^\dagger}$, and $\lambda_{\mathsf{h}^\ddagger}$ and use the notion of *typeability-preserving macro-expressiveness* to determine whether handlers are interdefinable [99]. In our particular setting, typeability-preserving macro-expressiveness asks whether there exists a *local* transformation that can transform one kind of handler into another kind of handler, whilst preserving typeability in the image of the transformation. By mandating that the transform is local we rule out the possibility of rewrit-

ing the entire program in, say, CPS notation to implement deep and shallow handlers as in Chapter 4. In this chapter we use the notion of typeability-preserving macro-expressiveness to show that shallow handlers and general recursion can simulate deep handlers up to congruence, and that deep handlers can simulate shallow handlers up to administrative reductions.

Chapter outline

Section 6.1 develops an encoding of deep handlers in terms of shallow handlers.

Section 6.2 shows an encoding going the other way, i.e. shallow handlers encoded using deep handlers.

Section 6.3 demonstrates that parameterised handlers are encodable using ordinary deep handlers with the state-passing technique.

Section 6.4 discusses related work.

Relation to prior work The results in this chapter has been published previously in the following papers.

i Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018

ii Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020

The results of Sections 6.1 and 6.2 appear in item i, whilst the result of Section 6.3 appears in item ii.

6.1 Deep as shallow

The implementation of deep handlers using shallow handlers (and recursive functions) is by a direct local translation, similar to how one would implement a fold (catamorphism) in terms of general recursion. Each handler is wrapped in a recursive function and each resumption has its body wrapped in a call to this recursive function. Formally, the translation $\mathcal{S}[\![-]\!]$ is defined as the homomorphic extension of the following equations

to all terms and substitutions.

$$\begin{aligned}
&\mathcal{S}[\![-]\!] : \mathrm{Comp} \to \mathrm{Comp}\\
&\mathcal{S}[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!] \stackrel{\text{def}}{=} (\mathbf{rec}\ h\, f.\mathbf{handle}^{\dagger}\ f\,\langle\rangle\ \mathbf{with}\ \mathcal{S}[\![H]\!]_h)\,(\lambda\langle\rangle.\mathcal{S}[\![M]\!])\\
&\mathcal{S}[\![-]\!] : \mathrm{HDef} \times \mathrm{Val} \to \mathrm{HDef}\\
&\mathcal{S}[\![\{\mathbf{return}\ x \mapsto N\}]\!]_h \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto \mathcal{S}[\![N]\!]\}\\
&\mathcal{S}[\![\{\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto N_\ell\}_{\ell\in\mathcal{L}}]\!]_h \stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto \mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h\,(\lambda\langle\rangle.r\,x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathcal{S}[\![N_\ell]\!]\}_{\ell\in\mathcal{L}}
\end{aligned}$$

The translation of **handle** uses a **rec**-abstraction to introduce a fresh name h for the handler H. This name is used by the translation of the handler definitions. The translation of **return**-clauses is the identity, and thus ignores the handler name. However, the translation of operation clauses uses the name to simulate a deep resumption by guarding invocations of the shallow resumption r with h.

In order to exemplify the translation, let us consider a variation of the env handler from Section 2.3, which handles an operation $\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{Int}$.

$$\begin{aligned}
&\mathcal{D}\left[\!\!\left[\begin{array}{l}\mathbf{handle\ do}\ \mathsf{Ask}\,\langle\rangle + \mathbf{do}\ \mathsf{Ask}\,\langle\rangle\ \mathbf{with}\\ \quad \mathbf{return}\ x \mapsto \mathbf{return}\ x\\ \quad \langle\!\langle \mathsf{Ask}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle \mapsto r\ 42\end{array}\right]\!\!\right]\\
&= (\mathbf{rec}\ env\, f.\mathbf{handle}^{\dagger}\ f\,\langle\rangle\ \mathbf{with}\\
&\qquad \mathbf{return}\ x \mapsto \mathbf{return}\ x\\
&\qquad \langle\!\langle \mathsf{Ask}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle \mapsto \mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.env\ (\lambda\langle\rangle.r\ x)\ \mathbf{in}\\
&\qquad\qquad r\ 42)\ (\lambda\langle\rangle.\mathbf{do}\ \mathsf{Ask}\,\langle\rangle + \mathbf{do}\ \mathsf{Ask}\,\langle\rangle)
\end{aligned}$$

The deep semantics are simulated by generating the name *env* for the shallow handlers and recursively apply the handler under the modified resumption.

The translation commutes with substitution and preserves typeability.

Lemma 6.1. *Let* σ *denote a substitution. The translation* $\mathcal{S}[\![-]\!]$ *commutes with substitution, i.e.*

$$\mathcal{S}[\![V]\!]\mathcal{S}[\![\sigma]\!] = \mathcal{S}[\![V\sigma]\!], \quad \mathcal{S}[\![M]\!]\mathcal{S}[\![\sigma]\!] = \mathcal{S}[\![M\sigma]\!], \quad \mathcal{S}[\![H]\!]\mathcal{S}[\![\sigma]\!] = \mathcal{S}[\![H\sigma]\!].$$

Proof. By induction on the structures of V, M, and H. □

Theorem 6.2. *If* $\Delta;\Gamma \vdash M : C$ *then* $\Delta;\Gamma \vdash \mathcal{S}[\![M]\!] : \mathcal{S}[\![C]\!]$.

Proof. By induction on the typing derivations. □

In order to obtain a simulation result, we allow reduction in the simulated term to be performed under lambda abstractions (and indeed anywhere in a term), which is necessary because of the redefinition of the resumption to wrap the handler around its body. Nevertheless, the simulation proof makes minimal use of this power, merely using it to rename a single variable.

Theorem 6.3 (Simulation up to congruence)**.** *If* $M \rightsquigarrow N$ *then* $\mathcal{S}[\![M]\!] \rightsquigarrow^{+}_{\mathrm{cong}} \mathcal{S}[\![N]\!]$.

Proof. By case analysis on $\rightsquigarrow$ using Lemma 6.1. The interesting case is S-Op, which is where we apply a single β-reduction, renaming a variable, under the λ-abstraction representing the resumption. The proof of this case is as follows.

$$
\begin{array}{rl}
 & \mathcal{S}[\![\mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H]\!] \\
= & \quad (\text{definition of } \mathcal{S}[\![-]\!]) \\
 & (\mathbf{rec}\ h\, f.\mathbf{handle}^{\dagger}\ f\ \langle\rangle\ \mathbf{with}\ \mathcal{S}[\![H]\!]_h)\ (\lambda\langle\rangle.\mathcal{S}[\![\mathcal{E}]\!][\mathbf{do}\ \ell\ \mathcal{S}[\![V]\!]]) \\
\rightsquigarrow^{+} & \quad (\textsf{S-Rec}, \textsf{S-App}, \textsf{S-Op}^{\dagger}\ \text{with}\ H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\}) \\
 & (\mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h\,(\lambda\langle\rangle.r\ x)\ \mathbf{in}\ \mathcal{S}[\![N]\!])[\lambda y.\mathcal{S}[\![\mathcal{E}]\!][\mathbf{return}\ y]/r, \mathcal{S}[\![V]\!]/p] \\
= & \quad (\text{definition of [-]}) \\
 & (\mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h\,(\lambda\langle\rangle.(\lambda y.\mathcal{S}[\![\mathcal{E}]\!][\mathbf{return}\ y])\ x)\ \mathbf{in}\ \mathcal{S}[\![N]\!][\mathcal{S}[\![V]\!]/p] \\
\rightsquigarrow_{\mathrm{cong}} & \quad (\textsf{S-App}\ \text{reduction under}\ \lambda x.\cdots) \\
 & (\mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda x.h\,(\lambda\langle\rangle.\mathcal{S}[\![\mathcal{E}]\!][\mathbf{return}\ x])\ \mathbf{in}\ \mathcal{S}[\![N]\!][\mathcal{S}[\![V]\!]/p] \\
\rightsquigarrow & \quad (\textsf{S-Let}\ \text{and Lemma 6.1}) \\
 & \mathcal{S}[\![N[V/p, \lambda x.h\,(\lambda\langle\rangle.\mathcal{E}[\mathbf{return}\ x])/r]]\!]
\end{array}
$$

□

6.2 Shallow as deep

Implementing shallow handlers in terms of deep handlers is slightly more involved than the other way round. It amounts to the encoding of a case split by a fold and involves a translation on handler types as well as handler terms. Formally, the translation $\mathcal{D}[\![-]\!]$ is defined as the homomorphic extension of the following equations to all types, terms,

type environments, and substitutions.

$$\mathcal{D}[\![-]\!] : \mathrm{HType} \to \mathrm{HType}$$
$$\mathcal{D}[\![A!E_1 \Rightarrow B!E_2]\!] \stackrel{\text{def}}{=} \mathcal{D}[\![A!E_1]\!] \Rightarrow \langle 1 \to \mathcal{D}[\![C!E_1]\!]; 1 \to \mathcal{D}[\![B!E_2]\!]\rangle!\mathcal{D}[\![E_2]\!]$$

$$\mathcal{D}[\![-]\!] : \mathrm{Comp} \to \mathrm{Comp}$$
$$\begin{aligned}\mathcal{D}[\![\mathbf{handle}^{\dagger}\ M\ \mathbf{with}\ H]\!] \stackrel{\text{def}}{=}\ & \mathbf{let}\ z \leftarrow \mathbf{handle}\ \mathcal{D}[\![M]\!]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in} \\ & \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ g\,\langle\rangle\end{aligned}$$

$$\mathcal{D}[\![-]\!] : \mathrm{HDef} \to \mathrm{HDef}$$
$$\mathcal{D}[\![\{\mathbf{return}\ x \mapsto N\}]\!] \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{return}\ x; \lambda\langle\rangle.\mathcal{D}[\![N]\!]\rangle\}$$
$$\begin{aligned}\mathcal{D}[\![\{\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto N\}_{\ell\in\mathcal{L}}]\!] \stackrel{\text{def}}{=} \{&\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto \\ & \mathbf{let}\ r \leftarrow \lambda x.\mathbf{let}\ z \leftarrow r\ x\ \mathbf{in}\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle\ \mathbf{in} \\ & \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ p\ \mathbf{in}\ r\ x; \lambda\langle\rangle.\mathcal{D}[\![N]\!]\rangle\}_{\ell\in\mathcal{L}}\end{aligned}$$

As evident from the translation of handler types, each shallow handler is encoded as a deep handler that returns a pair of thunks. It is worth noting that the handler construction is actually pure, yet we need to annotate the pair with the translated effect signature $\mathcal{D}[\![E_2]\!]$, because the calculus has no notion of effect subtyping. Technically we could insert an administrative identity handler to coerce the effect signature. There are practical reasons for avoiding administrative handlers, though, as we shall discuss momentarily the inordinate administrative overhead of this transformation might conceal the additional overhead incurred by the introduction of administrative identity handlers. The first component of the pair forwards all operations, acting as the identity on computations. The second component interprets a single operation before reverting to forwarding. The following example illustrates the translation on an instance of the pipe operator from Section 2.6 using the consumer computation $\mathbf{do}\ \mathsf{Await}\,\langle\rangle + \mathbf{do}\ \mathsf{Await}\,\langle\rangle$

and the suspended producer computation $\mathbf{rec}\ \mathit{ones}\,\langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\,\langle\rangle$.

$$
\begin{aligned}
&\mathcal{D}\left[\!\!\left[\begin{array}{l}\mathbf{handle}^{\dagger}\ \mathbf{do}\ \mathsf{Await}\,\langle\rangle + \mathbf{do}\ \mathsf{Await}\,\langle\rangle\ \mathbf{with}\\ \quad \mathbf{return}\ x \mapsto \mathbf{return}\ x\\ \quad \langle\!\langle \mathsf{Await}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle \mapsto \mathsf{copipe}\,\langle r; \mathbf{rec}\ \mathit{ones}\,\langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\,\langle\rangle\rangle\end{array}\right]\!\!\right]\\
&= \mathbf{let}\ z \leftarrow \mathbf{handle}\ (\lambda\langle\rangle.\mathbf{do}\ \mathsf{Await}\,\langle\rangle + \mathbf{do}\ \mathsf{Await}\,\langle\rangle)\,\langle\rangle\ \mathbf{with}\\
&\qquad \mathbf{return}\ x \mapsto \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{return}\ x; \lambda\langle\rangle.\mathbf{return}\ x\rangle\\
&\qquad \langle\!\langle \mathsf{Await}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle \mapsto\\
&\qquad\quad \mathbf{let}\ r \leftarrow \lambda x.\mathbf{let}\ z \leftarrow r\ x\ \mathbf{in}\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle\ \mathbf{in}\\
&\qquad\quad \mathbf{return}\ \langle\lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ p\ \mathbf{in}\ r\ x;\\
&\qquad\qquad\qquad \lambda\langle\rangle.\mathcal{D}[\![\mathsf{copipe}]\!]\,\langle r; \mathbf{rec}\ \mathit{ones}\,\langle\rangle.\mathbf{do}\ \mathsf{Yield}\ 1; \mathit{ones}\,\langle\rangle\rangle\rangle\\
&\quad \mathbf{in}\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ g\,\langle\rangle
\end{aligned}
$$

Evaluation of both the left hand side and right hand side of the equals sign yields the value $2 : \mathsf{Int}$. The **return**-case in the image contains a redundant pair, because the **return**-case of pipe is the identity. The translation of the Await-case sets up the forwarding component and handling component of the pair of thunks.

The distinction between deep and shallow handlers is that the latter is discharged after handling a single operation, whereas the former is persistent and apt for continual operation interpretations. The persistence of deep handlers means that any handler in the image of the translation remains in place for the duration of the handled computation after handling a single operation, which has noticeable asymptotic performance implications. Each activation of a handler in the image introduces another layer of indirection that any subsequent operation invocation have to follow. Supposing some source program contains n handlers and performs k operation invocations, then the image introduces k additional handlers, meaning the total amount of handlers in the image is $n+k$. Viewed through the practical lens of the CPS translation (Chapter 4) or abstract machine (Chapter 5) it means that in the worst case handler lookup takes $O(n+k)$ time. For example, consider the extreme case where $n = 1$, that is, the handler lookup takes $O(1)$ time in the source, but in the image it takes $O(k)$ time. Thus this translation is more of theoretical significance than practical interest. It also demonstrates that typeability-preserving macro-expressiveness is rather coarse-grained notion of expressiveness, as it blindly considers whether some construct is computable using another construct without considering the computational cost.

The translation commutes with substitution and preserves typeability.

Lemma 6.4. *Let σ denote a substitution. The translation $\mathcal{D}[\![-]\!]$ commutes with substitution, i.e.*

$$\mathcal{D}[\![V]\!]\mathcal{D}[\![\sigma]\!] = \mathcal{D}[\![V\sigma]\!], \quad \mathcal{D}[\![M]\!]\mathcal{D}[\![\sigma]\!] = \mathcal{D}[\![M\sigma]\!], \quad \mathcal{D}[\![H]\!]\mathcal{D}[\![\sigma]\!] = \mathcal{D}[\![H\sigma]\!].$$

Proof. By induction on the structures of V, M, and H. □

Theorem 6.5. *If $\Delta;\Gamma \vdash M : C$ then $\mathcal{D}[\![\Delta]\!];\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{D}[\![M]\!] : \mathcal{D}[\![C]\!]$.*

Proof. By induction on the typing derivations. □

As with the implementation of deep handlers as shallow handlers, the implementation is again given by a typeability-preserving local translation. However, this time the administrative overhead is more significant. Reduction up to congruence is insufficient and we require a more semantic notion of administrative reduction.

Definition 6.6 (Administrative evaluation contexts)**.** An evaluation context $\mathcal{E} \in \mathrm{Cont}$ is administrative, $\mathit{admin}(\mathcal{E})$, when the following two criteria hold.

1. For all values $V \in \mathrm{Val}$, we have: $\mathcal{E}[\mathbf{return}\ V] \rightsquigarrow^* \mathbf{return}\ V$
2. For all evaluation contexts $\mathcal{E}' \in \mathrm{Cont}$, operations $\ell \in \mathsf{BL}(\mathcal{E})\backslash\mathsf{BL}(\mathcal{E}')$, and values $V \in \mathrm{Val}$:

$$\mathcal{E}[\mathcal{E}'[\mathbf{do}\ \ell\ V]] \rightsquigarrow^*_{\mathrm{cong}} \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in}\ \mathcal{E}[\mathcal{E}'[\mathbf{return}\ x]].$$

The intuition is that an administrative evaluation context behaves like the empty evaluation context up to some amount of administrative reduction, which can only proceed once the term in the context becomes sufficiently evaluated. Values annihilate the evaluation context and handled operations are forwarded.

Definition 6.7 (Approximation up to administrative reduction)**.** Define $\gtrsim$ as the compatible closure of the following inference rules.

$$\frac{}{M \gtrsim M} \qquad \frac{M \rightsquigarrow M' \qquad M' \gtrsim N}{M \gtrsim N} \qquad \frac{\mathit{admin}(\mathcal{E}) \qquad M \gtrsim N}{\mathcal{E}[M] \gtrsim N}$$

We say that M approximates N up to administrative reduction if $M \gtrsim N$.

Approximation up to administrative reduction captures the property that administrative reduction may occur anywhere within a term. The following lemma states that the forwarding component of the translation is administrative.

Lemma 6.8. *For all shallow handlers H, the following context is administrative*

$$\mathbf{let}\ z \leftarrow \mathbf{handle}\ [\,]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in\ let}\ \langle f; \_\rangle = z\ \mathbf{in}\ f\,\langle\rangle.$$

Proof. We have to check both conditions of Definition 6.6.

1. Follows by direct calculation.

2. Follows by direct calculation using the assumption that $\ell \notin \mathsf{BL}(\mathcal{E}')$.

$$\begin{array}{ll}
& \mathbf{let}\ z \leftarrow \mathbf{handle}\ \mathcal{E}'[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in\ let}\ \langle f; _\rangle = z\ \mathbf{in}\ f\,\langle\rangle \\
\rightsquigarrow & (\textsf{S-Op using assumption } \ell \notin \mathsf{BL}(\mathcal{E}')) \\
& \mathbf{let}\ z \leftarrow \mathbf{let}\ r \leftarrow \lambda x.\mathbf{let}\ z \leftarrow (\lambda x.\mathbf{handle}\ \mathcal{E}'[\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\ x \\
& \qquad\qquad \mathbf{in\ let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle \\
& \qquad \mathbf{in\ return}\ \langle \lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in}\ r\ x; \lambda\langle\rangle.\mathcal{D}[\![N]\!]\rangle \\
& \mathbf{in}\ \langle f; _\rangle = z\ \mathbf{in}\ f\,\langle\rangle \\
\rightsquigarrow^{+} & (\textsf{S-Let, S-Split, S-App}) \\
& (\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in}\ r\ x)[(\lambda x.\mathbf{let}\ z \leftarrow (\lambda x.\mathbf{handle}\ \mathcal{E}'[\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\,x \\
& \qquad\qquad \mathbf{in\ let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle)/r] \\
\rightsquigarrow_{\mathrm{cong}} & (\textsf{S-App tail position reduction}) \\
& \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in\ let}\ z \leftarrow (\lambda x.\mathbf{handle}\ \mathcal{E}'[\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\,x\ \mathbf{in} \\
& \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle \\
\rightsquigarrow_{\mathrm{cong}} & (\textsf{S-App reduction under binder}) \\
& \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ V\ \mathbf{in\ let}\ z \leftarrow \mathbf{handle}\ \mathcal{E}'[\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in} \\
& \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle
\end{array}$$

□

Theorem 6.9 (Simulation up to administrative reduction)**.** *If $M' \gtrsim \mathcal{D}[\![M]\!]$ and $M \rightsquigarrow N$ then there exists N' such that $N' \gtrsim \mathcal{D}[\![N]\!]$ and $M' \rightsquigarrow^{+} N'$.*

Proof. By induction on $M' \gtrsim \mathcal{D}[\![M]\!]$ and case analysis on $M \rightsquigarrow N$ using Lemma 6.4 and Lemma 6.8. The interesting case is reflexivity of $\gtrsim$ where $M \rightsquigarrow N$ is an application of $\textsf{S-Op}^{\dagger}$, which we will show.

In the reflexivity case we have $M' \gtrsim \mathcal{D}[\![M]\!]$, where $M = \mathbf{handle}^{\dagger}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H$ and $N = N_{\ell}[V/p, \lambda y.\mathcal{E}[\mathbf{return}\ y]/r]$ such that $M \rightsquigarrow N$ where $\ell \notin \mathsf{BL}(\mathcal{E})$ and $H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N_{\ell}\}$. Hence by reflexivity of $\gtrsim$ we have $M' = \mathcal{D}[\![\mathbf{handle}^{\dagger}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H]\!]$.

Now we can compute N' by direct calculation starting from M' yielding

$$
\begin{array}{ll}
& \mathcal{D}[\![\mathbf{handle}^{\dagger}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H]\!] \\
= & \quad \text{(definition of } \mathcal{D}[\![-]\!]) \\
& \mathbf{let}\ z \leftarrow \mathbf{handle}\ \mathcal{D}[\![\mathcal{E}]\!][\mathbf{do}\ \ell\ \mathcal{D}[\![V]\!]]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in} \\
& \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ g\,\langle\rangle \\
\rightsquigarrow^{+} & \quad (\mathsf{S\text{-}Op} \text{ using assumption } \ell \notin \mathsf{BL}(\mathcal{D}[\![\mathcal{E}]\!]), \mathsf{S\text{-}Let}, \mathsf{S\text{-}Let}) \\
& \mathbf{let}\ \langle f; g\rangle = \langle \lambda\langle\rangle.\mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ \mathcal{D}[\![V]\!]\ \mathbf{in}\ r\ x; \\
& \qquad\qquad \lambda\langle\rangle.\mathcal{D}[\![N_\ell]\!]\rangle[\lambda x.\mathbf{let}\ z \leftarrow (\lambda y.\mathbf{handle}\ \mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\ x\ \mathbf{in} \\
& \qquad\qquad\qquad \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle/r, \mathcal{D}[\![V]\!]/p]\ \mathbf{in}\ g\,\langle\rangle \\
\rightsquigarrow^{+} & \quad (\mathsf{S\text{-}Split}, \mathsf{S\text{-}App}) \\
& \mathcal{D}[\![N_\ell]\!][\lambda x.\mathbf{let}\ z \leftarrow (\lambda y.\mathbf{handle}\ \mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\ x\ \mathbf{in} \\
& \qquad \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle/r, \mathcal{D}[\![V]\!]/p] \\
= & \quad \text{(by Lemma 6.4)} \\
& \mathcal{D}[\![N_\ell[\lambda x.\mathbf{let}\ z \leftarrow (\lambda y.\mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y]\ \mathbf{with}\ H)\ x\ \mathbf{in} \\
& \qquad \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle/r, V/p]]\!]
\end{array}
$$

Take the final term to be N'. If the resumption $r \notin \mathrm{FV}(N_\ell)$ then the two terms N' and $\mathcal{D}[\![N_\ell[V/p, \lambda y.\mathcal{E}[\mathbf{return}\ y]/r]]\!]$ are the identical, and thus the result follows immediate by reflexivity of the $\gtrsim$-relation. Otherwise the proof reduces to showing that the larger resumption term simulates the smaller resumption term, i.e (note we lift the $\gtrsim$-relation to value terms).

$$
\begin{array}{l}
(\lambda x.\mathbf{let}\ z \leftarrow (\lambda y.\mathbf{handle}\ \mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y]\ \mathbf{with}\ \mathcal{D}[\![H]\!])\ x\ \mathbf{in} \\
\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle) \gtrsim (\lambda y.\mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y]).
\end{array}
$$

We use the congruence rules to apply a single $\mathsf{S\text{-}App}$ on the left hand side to obtain

$$
\begin{array}{l}
(\lambda x.\mathbf{let}\ z \leftarrow \mathbf{handle}\ \mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in} \\
\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle) \gtrsim (\lambda y.\mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y]).
\end{array}
$$

Now the trick is to define the following context

$$
\mathcal{E}' \stackrel{\text{def}}{=} \mathbf{let}\ z \leftarrow \mathbf{handle}\ [\,]\ \mathbf{with}\ \mathcal{D}[\![H]\!]\ \mathbf{in}\ \mathbf{let}\ \langle f; g\rangle = z\ \mathbf{in}\ f\,\langle\rangle.
$$

The context $\mathcal{E}'$ is an administrative evaluation context by Lemma 6.8. Now it follows by Defintion 6.7 that $(\lambda x.\mathcal{E}'[\mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ x]]) \gtrsim (\lambda y.\mathcal{D}[\![\mathcal{E}]\!][\mathbf{return}\ y])$.

□

6.3 Parameterised handlers as ordinary deep handlers

As mentioned in Section 3.4, parameterised handlers codify the parameter-passing idiom. They may be seen as an optimised form of parameter-passing deep handlers. We now show formally that parameterised handlers are special instances of ordinary deep handlers. We define a local transformation $\mathcal{P}[\![-]\!]$ which translates parameterised handlers into ordinary deep handlers. Formally, the translation is defined on terms, types, environments, and substitutions. We omit the homomorphic cases and show only the interesting cases.

$$\mathcal{P}[\![-]\!] : \mathrm{HType} \to \mathrm{HType}$$
$$\mathcal{P}[\![\langle C;A\rangle \Rightarrow^{\ddagger} B!E]\!] \stackrel{\text{def}}{=} \mathcal{P}[\![C]\!] \Rightarrow (\mathcal{P}[\![A]\!] \to \mathcal{P}[\![B!E]\!])!\mathcal{P}[\![E]\!]$$

$$\mathcal{P}[\![-]\!] : \mathrm{Comp} \to \mathrm{Comp}$$
$$\mathcal{P}[\![\mathbf{handle}^{\ddagger}\ M\ \mathbf{with}\ (q.H)(W)]\!] \stackrel{\text{def}}{=} (\mathbf{handle}\ \mathcal{P}[\![M]\!]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ \mathcal{P}[\![W]\!]$$

$$\mathcal{P}[\![-]\!] : \mathrm{HDef} \times \mathrm{Val} \to \mathrm{HDef}$$
$$\mathcal{P}[\![\{\mathbf{return}\ x \mapsto M\}]\!]_q \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto \lambda q.\mathcal{P}[\![M]\!]\}$$
$$\mathcal{P}[\![\{\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto M\}]\!]_q \stackrel{\text{def}}{=} \{\langle\!\langle \ell\ p \twoheadrightarrow r\rangle\!\rangle \mapsto \lambda q.\mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda\langle x;q'\rangle.r\ x\ q'\ \mathbf{in}\ \mathcal{P}[\![M]\!]\}$$

The parameterised $\mathbf{handle}^{\ddagger}$ construct becomes an application of a **handle** construct to the translation of the parameter. The translation of **return** and operation clauses are parameterised by the name of the handler parameter as each clause body is enclosed in a λ-abstraction whose formal parameter is the handler parameter q. As a result the ordinary deep resumption r is a curried function. However, the uses of r in M expects a binary function. To repair this discrepancy, we construct an uncurried interface of r by embedding it under a binary λ-abstraction.

To illustrate the translation in action consider the following example program that adds the results obtained by performing two invocations of some stateful operation $\mathsf{Incr} : 1 \twoheadrightarrow \mathsf{Int}$, which increments some global counter and returns its prior value.

$$\mathcal{P}\left[\!\!\left[\mathbf{handle}^{\ddagger}\ \mathbf{do}\ \mathsf{Incr}\,\langle\rangle + \mathbf{do}\ \mathsf{Incr}\,\langle\rangle\ \mathbf{with}\ \left(q.\ \begin{array}{ll}\mathbf{return}\ x & \mapsto \mathbf{return}\ \langle x;q\rangle\\ \langle\!\langle \mathsf{Incr}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle & \mapsto r\,\langle q;q+1\rangle\end{array}\right)\ 40\right]\!\!\right]$$
$$= \left(\begin{array}{l}\mathbf{handle}\ \mathbf{do}\ \mathsf{Incr}\,\langle\rangle + \mathbf{do}\ \mathsf{Incr}\,\langle\rangle\ \mathbf{with}\\ \quad \mathbf{return}\ x \mapsto \lambda q.\mathbf{return}\ \langle x;q\rangle\\ \quad \langle\!\langle \mathsf{Incr}\ \langle\rangle \twoheadrightarrow r\rangle\!\rangle \mapsto \lambda q.\mathbf{let}\ r \leftarrow \mathbf{return}\ \lambda\langle x;q\rangle.r\ x\ q\ \mathbf{in}\ r\,\langle q;q+1\rangle\end{array}\right)\ 40$$

Evaluation of the program on either side of the equals sign yields $\langle 81;42\rangle : \mathsf{Int}$. The translation desugars the parameterised handler into an ordinary deep handler that makes use

of the parameter-passing idiom to maintain the state of the handled computation [232].

The translation commutes with substitution and preserves typeability.

Lemma 6.10. *Let σ denote a substitution. The translation $\mathcal{P}[\![-]\!]$ commutes with substitution, i.e.*

$$\mathcal{P}[\![V]\!]\mathcal{P}[\![\sigma]\!] = \mathcal{P}[\![V\sigma]\!], \quad \mathcal{P}[\![M]\!]\mathcal{P}[\![\sigma]\!] = \mathcal{P}[\![M\sigma]\!], \quad \mathcal{P}[\![(q.H)]\!]\mathcal{P}[\![\sigma]\!] = \mathcal{P}[\![(q.H)\sigma]\!].$$

Proof. By induction on the structures of V, M, and $q.H$. □

Theorem 6.11. *If $\Delta;\Gamma \vdash M : C$ then $\mathcal{P}[\![\Delta]\!];\mathcal{D}[\![\Gamma]\!] \vdash \mathcal{P}[\![M]\!] : \mathcal{P}[\![C]\!]$.*

Proof. By induction on the typing derivations. □

This translation of parameterised handlers simulates the native semantics. As with the simulation of deep handlers via shallow handlers in Section 6.1, this simulation is not quite on the nose as the image simulates the source only up to congruence due to the need for an application of a pure function to a variable to be reduced.

Theorem 6.12 (Simulation up to congruence)**.** *If $M \rightsquigarrow N$ then $\mathcal{P}[\![M]\!] \rightsquigarrow^{+}_{\mathrm{cong}} \mathcal{P}[\![N]\!]$.*

Proof. By case analysis on the relation $\rightsquigarrow$ using Lemma 6.10. The interesting case is S-Op‡, which is where we need to reduce under the λ-abstraction representing the

parameterised resumption.

$$
\begin{aligned}
& \mathcal{P}[\![\mathbf{handle}^{\ddagger}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ (q.\ H)(W)]\!] \\
= \quad & \text{(definition of } \mathcal{P}[\![-]\!]) \\
& (\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{do}\ \ell\ \mathcal{P}[\![V]\!]]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ \mathcal{P}[\![W]\!] \\
\rightsquigarrow \quad & (\textsf{S-Op with } H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\}) \\
& ((\lambda q.\mathbf{let}\ r \leftarrow \lambda\langle x; q'\rangle.r\ x\ q\ \mathbf{in} \\
& \qquad \mathcal{P}[\![N]\!])[\mathcal{P}[\![V]\!]/p, \lambda x.\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q/r])\ \mathcal{P}[\![W]\!] \\
= \quad & \text{(definition of } [-]) \\
& (\lambda q.\mathbf{let}\ r \leftarrow \lambda\langle x, q'\rangle.(\lambda x.\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ x\ q'\ \mathbf{in} \\
& \quad \mathcal{P}[\![N]\!][\mathcal{P}[\![V]\!]/p])\ \mathcal{P}[\![W]\!] \\
\rightsquigarrow \quad & (\textsf{S-App}) \\
& \mathbf{let}\ r \leftarrow \lambda\langle x; q'\rangle.(\lambda x.\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ x\ q'\ \mathbf{in} \\
& \mathcal{P}[\![N]\!][\mathcal{P}[\![V]\!]/p, \mathcal{P}[\![W]\!]/q] \\
\rightsquigarrow_{\mathrm{cong}} \quad & (\textsf{S-App under } \lambda\langle x; q'\rangle.\cdots) \\
& \mathbf{let}\ r \leftarrow \lambda\langle x; q'\rangle.(\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ q'\ \mathbf{in} \\
& \mathcal{P}[\![N]\!][\mathcal{P}[\![V]\!]/p, \mathcal{P}[\![W]\!]/q] \\
\rightsquigarrow \quad & (\textsf{S-Let}) \\
& \mathcal{P}[\![N]\!][\mathcal{P}[\![V]\!]/p, \mathcal{P}[\![W]\!]/q, \\
& \qquad \lambda\langle x, q'\rangle.(\mathbf{handle}\ \mathcal{P}[\![\mathcal{E}]\!][\mathbf{return}\ x]\ \mathbf{with}\ \mathcal{P}[\![H]\!]_q)\ q'/r] \\
= \quad & \text{(definition of } \mathcal{P}[\![-]\!] \text{ and Lemma 6.10)} \\
& \mathcal{P}[\![N[V/p, W/q, \lambda\langle x, q'\rangle.\mathbf{handle}^{\ddagger}\ \mathcal{E}[\mathbf{return}\ x]\ \mathbf{with}\ (q.\ H)(q')/r]]\!]
\end{aligned}
$$

□

6.4 Related work

Precisely how effect handlers fit into the landscape of programming language features is largely unexplored in the literature. The most relevant related work in this area is due to my collaborators and myself on the inherited efficiency of effect handlers (c.f. Chapter 7) and Forster et al. [98], who investigate various relationships between effect handlers, delimited control in the form of shift/reset, and monadic reflection using the notions of typeability-preserving macro-expressiveness and untyped macro-expressiveness [98, 99]. They show that in an untyped setting all three are interdefinable, whereas in a simply typed setting effect handlers cannot macro-express either. Piróg et al. [217] build upon the work of Forster et al. as they show that with sufficient polymorphism effect handlers and delimited control can simulate one another.

The work of Shan [247, 248] is related in spirit to the work presented in this chapter. Shan shows that static and dynamic notions of delimited control are interdefinable in an untyped setting. The work in this chapter has a similar flavour to Shan's work as we can view deep handlers as a kind of static control facility and shallow handlers as a kind of dynamic control facility. In order to simulate dynamic control using static control, Shan's translation makes use of recursive delimited continuations to construct the dynamic context surrounding and including the invocation context. A recursive continuation allows the captured context and continuation invocation context to coincide.

Chapter 7

Asymptotic speedup with effect handlers

When extending some programming language $\mathcal{L} \subset \mathcal{L}'$ with some new feature it is desirable to know exactly how the new feature impacts the language. At a bare minimum it is useful to know whether the extended language $\mathcal{L}'$ is unsound as a result of inhabiting the new feature (although, some languages are designed deliberately to be unsound [21]). More fundamentally, it may be useful for theoreticians and practitioners alike to know whether the extended language is more expressive than the base language as it may inform programming practice. Specifically, it may be of interest to know whether the extended language $\mathcal{L}'$ exhibits any *essential* expressivity when compared to the base language $\mathcal{L}$. Questions about essential expressivity fall under three different headings.

Programmability Are there programmable operations that can be done more easily in $\mathcal{L}'$ than in $\mathcal{L}$?

Computability Are there operations of a given type that are programmable in $\mathcal{L}'$ but not expressible at all in $\mathcal{L}$?

Complexity Are there operations programmable in $\mathcal{L}'$ with some asymptotic runtime bound (e.g. '$O(n^2)$') that cannot be achieved in $\mathcal{L}$?

The purpose of this chapter is to give a clear example of an essential complexity difference. Specifically, we will show that if we take a typical PCF-like base language, $\lambda_{\mathsf{b}}^{\rightarrow}$, and extend it with effect handlers, $\lambda_{\mathsf{h}}^{\rightarrow}$, then there exists a class of programs that have asymptotically more efficient realisations in $\lambda_{\mathsf{h}}^{\rightarrow}$ than possible in $\lambda_{\mathsf{b}}^{\rightarrow}$, hence establishing that effect handlers enable an asymptotic speedup for some programs.

To this end, we consider the following *generic count* problem, parametric in n: given a boolean-valued predicate P on the space $\mathbb{B}^n$ of boolean vectors of length n, return the number of such vectors q for which $Pq = \mathsf{true}$. We shall consider boolean vectors of any length to be represented by the type $\mathsf{Nat} \to \mathsf{Bool}$; thus for each n, we are asking for an implementation of a certain third-order function.

$$\mathsf{count}_n : ((\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat}$$

A naïve implementation strategy is simply to apply P to each of the 2^n vectors in turn. However, one can do better with a curious approach due to Berger [19], which achieves the effect of 'pruned search' where the predicate allows it. This should be taken as a warning that counter-intuitive phenomena can arise in this territory. Nonetheless, under the mild condition that P must inspect all n components of the given vector before returning, both these approaches will have a $\Omega(n2^n)$ runtime. Moreover, we shall show that in $\lambda_{\mathsf{b}}^{\to}$, a typical call-by-value language without advanced control features, one cannot improve on this: *any* implementation of count_n must necessarily take time $\Omega(n2^n)$ on *any* predicate P. Conversely, in the extended language $\lambda_{\mathsf{h}}^{\to}$ it becomes possible to bring the runtime down to $O(2^n)$: an asymptotic gain of a factor of n.

The key to enabling the speedup is *backtracking* via multi-shot resumptions. The idea is to memorise the control state at each component inspection to make it possible to quickly backtrack to a prior inspection and make a different decision as soon as one possible result has been computed. Concretely, suppose for example $n = 3$, and suppose that the predicate P always inspects the components of its argument in the order $0, 1, 2$. A naïve implementation of count_3 might start by applying the given predicate P to $q_0 = (\mathsf{true}, \mathsf{true}, \mathsf{true})$, and then to $q_1 = (\mathsf{true}, \mathsf{true}, \mathsf{false})$. Note that there is some duplication here: the computations of Pq_0 and Pq_1 will proceed identically up to the point where the value of the final component is requested. Ideally, we would record the state of the computation of Pq_0 at just this point, so that we can later resume this computation with false supplied as the final component value in order to obtain the value of Pq_1. Of course, a bespoke search function implementation would apply this backtracking behaviour in a standard manner for some *particular* choice of P (e.g. the n-queens problem); but to apply this idea of resuming previous subcomputations in the *generic* setting (i.e. uniformly in P) requires some special control feature such as effect handlers with multi-shot resumptions. Obviously, one can remove the need a special control feature by a change of type for the predicate P, but this such a change shifts the perspective. The intention is precisely to show that the languages differ in an essential

way as regards to their power to manipulate data of type $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$.

The idea of using first-class control achieve backtracking is fairly well-known in the literature [152], and there is a clear programming intuition that this yields a speedup unattainable in languages without such control features.

Chapter outline

Section 7.1 introduces the core calculi $\lambda_{\mathsf{b}}^{\to} \subset \lambda_{\mathsf{h}}^{\to}$, which are essentially simply-typed variations of λ_{b} and λ_{h}, respectively.

Section 7.2 develops an abstract machine for each core calculus. Both machines are simpler variations of the machine developed in Chapter 5.

Section 7.3 presents the gadgetry required to set up and proof the results for the generic count problem. It also shows that there exists an efficient implementation of generic count in $\lambda_{\mathsf{h}}^{\to}$.

Section 7.4 establishes the lower bound runtime result for implementations of generic count in $\lambda_{\mathsf{b}}^{\to}$.

Section 7.5 discusses extensions and variations of the phenomenon.

Section 7.6 investigates an empirical evaluation of the phenomenon.

Section 7.7 discusses related work.

Relation to prior work This chapter is based entirely on the following previously published paper.

> Daniel Hillerström, Sam Lindley, and John Longley. Effects for efficiency: Asymptotic speedup with first-class control. *Proc. ACM Program. Lang.*, 4(ICFP): 100:1–100:29, 2020

The contents of Sections 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6 are almost verbatim copies of Sections 3, 4, 5, 6, 7, and 8 of the above paper. I have made a few stylistic adjustments to make the Sections fit with the rest of this dissertation.

$$
\begin{array}{lrl}
\text{Types} & A,B,C,D \in \text{Type} & ::= \mathsf{Nat} \mid 1 \mid A \to B \mid A \times B \mid A + B \\
\text{Type environments} & \Gamma \in \text{TyEnv} & ::= \cdot \mid \Gamma, x : A \\
\text{Values} & V,W \in \text{Val} & ::= x \mid k \mid c \mid \lambda x^A.M \mid \mathbf{rec}\, f^{A\to B}\, x.M \\
& & \mid \ \langle\rangle \mid \langle V, W\rangle \mid (\mathbf{inl}\, V)^B \mid (\mathbf{inr}\, W)^A \\
\text{Computations} & M,N \in \text{Comp} & ::= V\, W \mid \mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N \\
& & \mid \ \mathbf{case}\, V\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} \\
& & \mid \ \mathbf{return}\, V \mid \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N
\end{array}
$$

Figure 7.1: Syntax of $\lambda_{\mathsf{b}}^{\to}$.

7.1 Simply-typed base and handler calculi

In this section, we present a base language $\lambda_{\mathsf{b}}^{\to}$ and its extension with effect handlers $\lambda_{\mathsf{h}}^{\to}$, both of which amounts to simply-typed variations of λ_{b} and λ_{h}, respectively. Sections 7.1.1–7.2.2 essentially recast the developments of Chapters 3 and 5 to fit the calculi $\lambda_{\mathsf{b}}^{\to}$ and $\lambda_{\mathsf{h}}^{\to}$. I will quickly glance over the details here, only highlighting the key differences as I am making use of a crucial design decision in Section 7.1.2 to make continuation reification a constant time operation.

7.1.1 Base calculus

The base calculus $\lambda_{\mathsf{b}}^{\to}$ is a fine-grain call-by-value [170] variation of PCF [219]. In essence it is a simply-typed variation of λ_{b}. Figure 7.1 depicts the type syntax, type environment syntax, and term syntax of $\lambda_{\mathsf{b}}^{\to}$. The main difference in the type language between λ_{b} and $\lambda_{\mathsf{b}}^{\to}$ is that the latter does not feature polymorphism nor an effect tracking system. At the term level, $\lambda_{\mathsf{b}}^{\to}$ does not feature polymorphic records and variants, but rather plain pairs and sums. For sums the left injection is introduced by $(\mathbf{inl}\ V)^B$, where the type annotation B is the type of the right injection. Similarly, the right injection is introduced by $(\mathbf{inl}\ W)^A$, where A is the type of the left injection. The **case**-construct eliminates sums. The last crucial difference between λ_{b} and $\lambda_{\mathsf{b}}^{\to}$ is that the latter includes natural numbers and primitive operations on natural numbers $(+,-,=)$. We let k range over natural numbers and c range over primitive operations on natural numbers $(+,-,=)$. As usual we let x,y,z range over term variables. For convenience, we also use f, g, and h for variables of function type, i and j for variables of type Nat, and r to denote resumptions.

Values

$$\textsf{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsf{T-Unit}\ \frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \textsf{T-Nat}\ \frac{k \in \mathbb{N}}{\Gamma \vdash k : \mathsf{Nat}} \qquad \textsf{T-Const}\ \frac{c : A \to B}{\Gamma \vdash c : A \to B}$$

$$\textsf{T-Lam}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^{A}.M : A \to B} \qquad \textsf{T-Rec}\ \frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \mathbf{rec}\, f^{A \to B} x.M : A \to B}$$

$$\textsf{T-Prod}\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : B}{\Gamma \vdash \langle V, W\rangle : A \times B} \qquad \textsf{T-Inl}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash (\mathbf{inl}\, V)^{B} : A + B} \qquad \textsf{T-Inr}\ \frac{\Gamma \vdash W : B}{\Gamma \vdash (\mathbf{inr}\, W)^{A} : A + B}$$

Computations

$$\textsf{T-App}\ \frac{\Gamma \vdash V : A \to B \qquad \Gamma \vdash W : A}{\Gamma \vdash V\, W : B} \qquad \textsf{T-Split}\ \frac{\Gamma \vdash V : A \times B \qquad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N : C}$$

$$\textsf{T-Case}\ \frac{\Gamma \vdash V : A + B \qquad \Gamma, x : A \vdash M : C \qquad \Gamma, y : B \vdash N : C}{\Gamma \vdash \mathbf{case}\, V\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} : C}$$

$$\textsf{T-Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\, V : A} \qquad \textsf{T-Let}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : C}{\Gamma \vdash \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : C}$$

Figure 7.2: Typing rules for $\lambda_{\mathsf{b}}^{\to}$.

S-App	$(\lambda x^A.M)V$	$\rightsquigarrow$	$M[V/x]$	
S-AppRec	$(\mathbf{rec}\, f^A\, x.M)V$	$\rightsquigarrow$	$M[(\mathbf{rec}\, f^A\, x.M)/f, V/x]$	
S-Const	$c\ V$	$\rightsquigarrow$	$\mathbf{return}\ (\ulcorner c \urcorner(V))$	
S-Split	$\mathbf{let}\ \langle x, y\rangle = \langle V, W\rangle\ \mathbf{in}\ N$	$\rightsquigarrow$	$N[V/x, W/y]$	
S-Case-inl	$\mathbf{case}\ (\mathbf{inl}\, V)^B\ \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$	$\rightsquigarrow$	$M[V/x]$	
S-Case-inr	$\mathbf{case}\ (\mathbf{inr}\, V)^A\ \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$	$\rightsquigarrow$	$N[V/y]$	
S-Let	$\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N$	$\rightsquigarrow$	$N[V/x]$	
S-Lift	$\mathcal{E}[M]$	$\rightsquigarrow$	$\mathcal{E}[N]$,	if $M \rightsquigarrow N$

Evaluation contexts $\quad \mathcal{E} \in \mathrm{Cont} ::= [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N$

Figure 7.3: Contextual small-step operational semantics.

The typing rules are given in Figure 7.2. These are similar to the ones given in Figure 3.4 modulo the polymorphism. The constants have the following types.

$$\{(+), (-)\} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \qquad (=) : \mathsf{Nat} \times \mathsf{Nat} \to 1 + 1$$

We give a small-step operational semantics for $\lambda_{\mathsf{b}}^{\to}$ with *evaluation contexts* in the style of Felleisen [79]. The reduction rules are given in Figure 7.3. We write $M[V/x]$ for M with V substituted for x and $\ulcorner c \urcorner$ for the usual interpretation of constant c as a meta-level function on closed values. The reduction relation $\rightsquigarrow$ is defined on computation terms. The statement $M \rightsquigarrow N$ reads: term M reduces to term N in one step.

Notation We use the same notation as introduced in Section 3.1.3. Although, we adapt the encoding of booleans to use regular sums.

$$\mathsf{Bool} \stackrel{\text{def}}{=} 1 + 1 \qquad \mathsf{true} \stackrel{\text{def}}{=} \mathbf{inl}\ \langle\rangle \qquad \mathsf{false} \stackrel{\text{def}}{=} \mathbf{inr}\ \langle\rangle$$

$$\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N \stackrel{\text{def}}{=} \mathbf{case}\ V\ \{\mathbf{inl}\ \langle\rangle \mapsto M; \mathbf{inr}\ \langle\rangle \mapsto N\}$$

7.1.2 Handler calculus

We now define $\lambda_{\mathsf{h}}^{\to}$ as an extension of $\lambda_{\mathsf{b}}^{\to}$.

Operation symbols	$\ell \in \mathcal{L}$
Signatures	$\Sigma \in \mathrm{Sig} ::= \cdot \mid \{\ell : A \to B\} \cup \Sigma$
Handler types	$F \in \mathrm{HType} ::= C \Rightarrow D$
Computations	$M, N \in \mathrm{Comp} ::= \cdots \mid \mathbf{do}\ \ell\ V \mid \mathbf{handle}\ M\ \mathbf{with}\ H$
Handlers	$H ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\} \uplus H$

Again, $\lambda_{\mathsf{h}}^{\to}$ is essentially a simply-typed variation of λ_{h}. There are a couple of key differences. The first key difference is that in the absence of an effect system $\lambda_{\mathsf{h}}^{\to}$ uses a nominal notion of effects. Following Pretnar [232], we assume a global effect signature Σ for every program. An effect signature Σ is a map from operation symbols to their types, thus we assume that each operation symbol in a signature is distinct.We write $\mathsf{dom}(\Sigma) \subseteq \mathcal{L}$ for the set of operation symbols in a signature Σ. The second and last key difference is that we adopt Plotkin and Pretnar's convention that a handler with missing operation clauses (with respect to Σ) is syntactic sugar for one in which all missing clauses perform explicit forwarding [228], i.e.

$$\{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell p\ \mathbf{in}\ r x\}.$$

This convention makes effect forwarding explicit, whereas in λ_{h} effect forwarding was implicit. As we shall see soon, an important semantic consequence of making effect forwarding explicit is that the abstract machine model in Section 7.2.2 has no rule for effect forwarding as it instead happens as a sequence of explicit **do** invocations in the term language. As a result, we become able to reason about continuation reification as a constant time operation, because a **do** invocation will just reify the top-most continuation frame.

The typing rules for handlers and operation invocation are similar to those of λ_{h} given in Section 3.2. The main difference is that the type of operations are retrieved from the global effect signature Σ rather than the current effect context. The typing rules are given in Figure 7.4.

The reduction rules for handlers are similar to those of λ_{h}.

S-Ret $\quad \mathbf{handle}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow N[V/x], \qquad \text{where } H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto N\}$

S-Op $\quad \mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \ell\ V]\ \mathbf{with}\ H \rightsquigarrow N[V/p, (\lambda y.\mathbf{handle}\ \mathcal{E}[\mathbf{return}\ y]\ \mathbf{with}\ H)/r],$
$\qquad \text{where } H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N\}$

Computations

$$\textsf{T-Do}\quad \frac{(\ell : A \to B) \in \Sigma \qquad \Gamma \vdash V : A}{\Gamma \vdash \mathbf{do}\ \ell\ V : B} \qquad \textsf{T-Handle}\quad \frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

Handlers

$$\textsf{T-Handler}\quad \frac{H^{\mathrm{ret}} = \{\mathbf{return}\ x \mapsto M\} \qquad [H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto N_{\ell}\}]_{\ell \in dom(\Sigma)} \qquad \Gamma, x : C \vdash M : D \qquad [\Gamma, p : A_{\ell}, r : B_{\ell} \to D \vdash N_{\ell} : D]_{(\ell : A_{\ell} \to B_{\ell}) \in \Sigma}}{\Gamma \vdash H : C \Rightarrow D}$$

Figure 7.4: Additional typing rules for $\lambda_{\mathsf{h}}^{\to}$.

However, the attentive reader may notice that the S-Op is missing the side condition regarding ℓ not appearing in the bound labels of $\mathcal{E}$. The notion of bound labels is of no use to us here due the convention that every handler handles every operation. Instead, we use a different notion of evaluation context. Although, some care must be taken to ensure the language semantics remains deterministic. As if we were naïvely to extend evaluation contexts with the handle construct then our semantics would become nondeterministic, as it may pick an arbitrary handler in scope. In order to ensure that the semantics is deterministic, we add a distinct form of evaluation context for effectful computation, which we call handler contexts.

$$\text{Handler contexts} \quad \mathcal{H} \in \mathrm{HCtx} ::= [\,] \mid \mathbf{handle}\ \mathcal{H}\ \mathbf{with}\ H \mid \mathbf{let}\ x \leftarrow \mathcal{H}\ \mathbf{in}\ N$$

We replace the S-Lift rule with a corresponding rule for handler contexts.

$$\mathcal{H}[M] \rightsquigarrow \mathcal{H}[N], \qquad \text{if } M \rightsquigarrow N$$

The separation between pure evaluation contexts $\mathcal{E}$ and handler contexts $\mathcal{H}$ ensures that the S-Op rule always selects the innermost handler.

The computation normal forms and type soundness property of λ_{h} carry over with modest changes. A computation term is now normal with respect to the global signature Σ rather than the current effect context.

Definition 7.1 (Computation normal forms)**.** A computation term N is normal with respect to Σ, if $N = \mathbf{return}\ V$ for some V or $N = \mathcal{E}[\mathbf{do}\ \ell\, W]$ for some $\ell \in dom(\Sigma)$, $\mathcal{E}$, and W.

Theorem 7.2 (Type soundness)**.** *If* $\vdash M : C$*, then either there exists* $\vdash N : C$ *such that* $M \rightsquigarrow^* N$ *and* N *is normal with respect to* Σ*, or* M *diverges.*

7.1.3 The role of types

Readers familiar with backtracking search algorithms may wonder where types come into the expressiveness picture. Types will not play a direct role in our proofs but rather in the characterisation of which programs can be meaningfully compared. In particular, types are used to rule out global approaches such as continuation passing style (CPS): without types one could obtain an efficient pure generic count program by CPS transforming the entire program.

Readers familiar with effect handlers may wonder why our handler calculus does not include an effect type system. As types frame the comparison of programs between languages, we require that types be fixed across languages; hence $\lambda_{\mathsf{h}}^{\rightarrow}$ does not include effect types. Future work includes reconciling effect typing with our approach to expressiveness.

7.2 A practical model of computation

The calculi $\lambda_{\mathsf{b}}^{\rightarrow}$ and $\lambda_{\mathsf{h}}^{\rightarrow}$ use a substitution model for evaluation. Whilst this model is semantically pleasing, it falls short of providing a realistic account of practical computation as substitution is an expensive operation. Instead we shall use a slightly simpler variation of the abstract machine from Chapter 5 as it provides a more practical model of computation (it is simpler, because the source language is simpler).

7.2.1 Base machine

The base machine is similar to the abstract machine from Chapter 5. The main differences are that the base machine does not feature support for effect handlers, and therefore it has a considerably simpler continuation structure, and it interprets the base language $\lambda_{\mathsf{b}}^{\rightarrow}$ rather than λ_{b}.

Formally, the base machine operates on configurations of the form $\langle M \mid \gamma \mid \sigma \rangle$. The first component contains the computation currently being evaluated. The second component contains the environment γ which binds free variables. The third component contains the continuation which instructs the machine how to proceed once evaluation

of the current computation is complete. The syntax of abstract machine states is as follows.

$$
\begin{array}{lll}
\textsf{Configurations} & \mathcal{C} \in \textsf{Conf} ::= & \langle M \mid \gamma \mid \sigma \rangle \\
\textsf{Environments} & \gamma \in \textsf{Env} ::= & \emptyset \mid \gamma[x \mapsto v] \\
\textsf{Machine values} & v, w \in \textsf{Mval} ::= & x \mid n \mid c \mid \langle\rangle \mid \langle v, w \rangle \\
& & \mid \ (\gamma, \lambda x^A.M) \mid (\gamma, \mathbf{rec} f^{A \to B} x.M) \\
& & \mid \ (\mathbf{inl}\, v)^B \mid (\mathbf{inr}\, w)^A \\
\textsf{Pure continuations} & \sigma \in \textsf{PureCont} ::= & [] \mid (\gamma, x, N) :: \sigma
\end{array}
$$

Values consist of function closures, constants, pairs, and left or right tagged values. We refer to continuations of the base machine as *pure*. A pure continuation is a stack of pure continuation frames. A pure continuation frame (γ, x, N) closes a let-binding $\mathbf{let}\, x \leftarrow [\,]\, \mathbf{in}\, N$ over environment γ. We write $[]$ for an empty pure continuation and $\phi :: \sigma$ for the result of pushing the frame ϕ onto σ. We use pattern matching to deconstruct pure continuations.

The abstract machine semantics is given in Figure 7.5. The transition relation ($\longrightarrow$) makes use of the value interpretation ($[\![-]\!]$) from value terms to machine values. The machine is initialised by placing a term in a configuration alongside the empty environment ($\emptyset$) and identity pure continuation ($[]$). The rules (M-App), (M-AppRec), (M-Const), (M-Split), (M-CaseL), and (M-CaseR) eliminate values. The (M-Let) rule extends the current pure continuation with let bindings. The (M-PureCont) rule pops the top frame of the pure continuation and extends the environment with the returned value. Given an input of a well-typed closed computation term $\vdash M : A$, the machine will either diverge or return a value of type A. A final state is given by a configuration of the form $\langle \mathbf{return}\, V \mid \gamma \mid [] \rangle$ in which case the final return value is given by the denotation $[\![V]\!]\gamma$ of V under environment γ.

Correctness The base machine faithfully simulates the operational semantics for $\lambda_{\mathsf{b}}^{\to}$; most transitions correspond directly to β-reductions, but M-Let performs an administrative step to bring the computation M into evaluation position. The proof of correctness is similar to the proof of Theorem 5.4 and the required proof gadgetry is the same. The full details are published in Appendix A of Hillerström et al. [124].

Transition relation $\longrightarrow \subseteq \mathrm{Conf} \times \mathrm{Conf}$

$$
\begin{array}{lll}
\textsf{M-App} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x^A.M) \\
\textsf{M-AppRec} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma'[f \mapsto (\gamma', \mathbf{rec} f^{A\to B} x.M), x \mapsto [\![W]\!]\gamma] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec} f^{A\to B} x.M) \\
\textsf{M-Const} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle \mathbf{return}\ (\ulcorner c \urcorner([\![W]\!]\gamma)) \mid \gamma \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = c \\
\textsf{M-Split} & \langle \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N \mid \gamma \mid \sigma\rangle \longrightarrow \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \langle v; w\rangle \\
\textsf{M-CaseL} & \langle \mathbf{case}\ V\ \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma[x \mapsto v] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inl}\, v \\
\textsf{M-CaseR} & \langle \mathbf{case}\ V\ \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} \mid \gamma \mid \sigma\rangle \longrightarrow \langle N \mid \gamma[y \mapsto v] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inr}\, v \\
\textsf{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma \mid (\gamma, x, N) :: \sigma\rangle & \\
\textsf{M-PureCont} & \langle \mathbf{return}\ V \mid \gamma \mid (\gamma', x, N) :: \sigma\rangle \longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \sigma\rangle &
\end{array}
$$

Value interpretation $[\![-]\!] : \mathrm{Val} \times \mathrm{Env} \to \mathrm{Mval}$

$$
\begin{array}{lll}
[\![x]\!]\gamma \stackrel{\text{def}}{=} \gamma(x) & [\![n]\!]\gamma \stackrel{\text{def}}{=} n & [\![\lambda x^A.M]\!]\gamma \stackrel{\text{def}}{=} (\gamma, \lambda x^A.M) \\
[\![\langle\rangle]\!]\gamma \stackrel{\text{def}}{=} \langle\rangle & [\![c]\!]\gamma \stackrel{\text{def}}{=} c & [\![\mathbf{rec} f^{A\to B} x.M]\!]\gamma \stackrel{\text{def}}{=} (\gamma, \mathbf{rec} f^{A\to B} x.M)
\end{array}
$$

$$
[\![\langle V, W\rangle]\!]\gamma \stackrel{\text{def}}{=} \langle [\![V]\!]\gamma, [\![W]\!]\gamma\rangle \qquad
\begin{array}{l}
[\![(\mathbf{inl}\, V)^B]\!]\gamma \stackrel{\text{def}}{=} (\mathbf{inl}\ [\![V]\!]\gamma)^B \\
[\![(\mathbf{inr}\, V)^A]\!]\gamma \stackrel{\text{def}}{=} (\mathbf{inr}\ [\![V]\!]\gamma)^A
\end{array}
$$

Figure 7.5: Abstract machine semantics for $\lambda_{\mathsf{b}}^{\to}$.

7.2.2 Handler machine

The abstract machine for $\lambda_{\mathsf{h}}^{\rightarrow}$ closely follows the machine definition in Chapter 5, though, this machine supports only deep handlers. The syntax is extended as follows.

Configurations	$\mathcal{C} \in \mathsf{Conf} ::= \langle M \mid \gamma \mid \kappa \rangle$
Resumptions	$\rho \in \mathsf{Res} ::= (\sigma, \chi)$
Continuations	$\kappa \in \mathrm{GenCont} ::= [] \mid \rho :: \kappa$
Handler closures	$\chi \in \mathrm{GenFrame} ::= (\gamma, H)$
Machine values	$v, w \in \mathrm{Mval} ::= \cdots \mid \rho$

The notion of configurations changes slightly as the continuation component is replaced by a generalised continuation $\kappa \in \mathrm{GenCont}$, in the same way as described in Chapter 5.The identity continuation is a singleton list containing the identity resumption, which is an empty pure continuation paired with the identity handler closure:

$$\kappa_0 \stackrel{\text{def}}{=} [([], (\emptyset, \{\mathbf{return}\ x \mapsto x\}))]$$

Besides the structure of the continuation component, the primary difference between the base machine and this machine is that the machine values are augmented to include resumptions as an operation invocation causes the topmost frame of the machine continuation to be reified (and bound to the resumption parameter in the operation clause). In addition, the handler machine adds transition rules for handlers, and modifies (M-Let) and (M-PureCont) from the base machine to account for the richer continuation structure. Figure 7.6 depicts the new and modified rules. The (M-Handle) rule pushes a handler closure along with an empty pure continuation onto the continuation stack. The (M-GenCont) rule transfers control to the success clause of the current handler once the pure continuation is empty. The (M-Op) rule transfers control to the matching operation clause on the topmost handler, and during the process it reifies the handler closure. Finally, the (M-Resume) rule applies a reified handler closure, by pushing it onto the continuation stack. The handler machine has two possible final states: either it yields a value or it gets stuck on an unhandled operation.

Correctness The handler machine faithfully simulates the operational semantics of $\lambda_{\mathsf{h}}^{\rightarrow}$. The proof of correctness is almost a carbon copy of the proof of Theorem 5.4. The full details are published in Appendix B of Hillerström et al. [124].

Transition relation

$$
\begin{array}{lrcl}
\textsf{M-Resume} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle \textbf{return}\ W \mid \gamma \mid (\sigma,\chi) :: \kappa \rangle, \\
& & & \quad \text{if } [\![V]\!]\gamma = (\sigma,\chi) \\
\textsf{M-Let} & \langle \textbf{let}\ x \leftarrow M\ \textbf{in}\ N \mid \gamma \mid (\sigma,\chi) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ((\gamma,x,N) :: \sigma,\chi) :: \kappa \rangle \\
\textsf{M-PureCont} & \langle \textbf{return}\ V \mid \gamma \mid ((\gamma',x,N) :: \sigma,\chi) :: \kappa \rangle & \longrightarrow & \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma,\chi) :: \kappa \rangle \\
\textsf{M-Handle} & \langle \textbf{handle}\ M\ \textbf{with}\ H \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ([],(\gamma,H)) :: \kappa \rangle \\
\textsf{M-GenCont} & \langle \textbf{return}\ V \mid \gamma \mid ([],(\gamma',H)) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle, \\
& & & \quad \text{if } H^{\text{ret}} = \{\textbf{return}\ x \mapsto M\} \\
\textsf{M-Op} & \langle \textbf{do}\ \ell\ V \mid \gamma \mid (\sigma,(\gamma',H)) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[p \mapsto [\![V]\!]\gamma, \\
& & & \qquad r \mapsto (\sigma,(\gamma',H))] \mid \kappa \rangle, \\
& & & \quad \text{if } \ell : A \to B \in \Sigma \\
& & & \quad \text{and } H^{\ell} = \{\langle\!\langle \ell\ p \twoheadrightarrow r \rangle\!\rangle \mapsto M\}
\end{array}
$$

Figure 7.6: Abstract machine semantics for $\lambda_{\mathsf{h}}^{\rightarrow}$.

7.2.3 Realisability and asymptotic complexity

As discussed in Section 5.3 the machine is readily realisable using standard persistent functional data structures. Pure continuations on the base machine and generalised continuations on the handler machine can be implemented using linked lists with a time complexity of $O(1)$ for the extension operation $(\_ :: \_)$. The topmost pure continuation on the handler machine may also be extended in time $O(1)$, as extending it only requires reaching under the topmost handler closure. Environments, γ, can be realised using a map, with a time complexity of $O(\log|\gamma|)$ for extension and lookup [210].

The worst-case time complexity of a single machine transition is exhibited by rules which involve operations on the environment, since any other operation is constant time, hence the worst-time complexity of a transition is $O(\log|\gamma|)$. The value interpretation function $[\![-]\!]\gamma$ is defined structurally on values. Its worst-time complexity is exhibited by a nesting of pairs of variables $[\![\langle x_1,\ldots,x_n\rangle]\!]\gamma$ which has complexity $O(n\log|\gamma|)$.

Continuation copying On the handler machine the topmost continuation frame can be copied in constant time due to the persistent runtime and the layout of machine continuations. An alternative design would be to make the runtime non-persistent in which case copying a continuation frame $((\sigma,\_) :: \_)$ would be a $O(|\sigma|)$ time operation.

Primitive operations on naturals Our model assumes that arithmetic operations on arbitrary natural numbers take $O(1)$ time. This is common practice in the study of algorithms when the main interest lies elsewhere [54, Section 2.2]. If desired, one could adopt a more refined cost model that accounted for the bit-level complexity of arithmetic operations; however, doing so would have the same impact on both of the situations we are wishing to compare, and thus would add nothing but noise to the overall analysis.

7.3 Predicates, decision trees, and generic count

We now come to the crux of the chapter. In this section and the next, we prove that $\lambda_{\mathsf{h}}^{\to}$ supports implementations of certain operations with an asymptotic runtime bound that cannot be achieved in $\lambda_{\mathsf{b}}^{\to}$ (Section 7.4). While the positive half of this claim essentially consolidates a known piece of folklore, the negative half appears to be new. To establish our result, it will suffice to exhibit a single 'efficient' program in $\lambda_{\mathsf{h}}^{\to}$, then show that no equivalent program in $\lambda_{\mathsf{b}}^{\to}$ can achieve the same asymptotic efficiency. We take *generic search* as our example.

Generic search is a modular search procedure that takes as input a predicate P on some multi-dimensional search space, and finds all points of the space satisfying P. Generic search is agnostic to the specific instantiation of P, and as a result is applicable across a wide spectrum of domains. Classic examples such as Sudoku solving [30], the n-queens problem [16] and graph colouring can be cast as instances of generic search, and similar ideas have been explored in connection with Nash equilibria and exact real integration [55, 249].

For simplicity, we will restrict attention to search spaces of the form $\mathbb{B}^n$, the set of bit vectors of length n. To exhibit our phenomenon in the simplest possible setting, we shall actually focus on the *generic count* problem: given a predicate P on some $\mathbb{B}^n$, return the *number of* points of $\mathbb{B}^n$ satisfying P. However, we shall explain why our results are also applicable to generic search proper.

We shall view $\mathbb{B}^n$ as the set of functions $\mathbb{N}_n \to \mathbb{B}$, where $\mathbb{N}_n \stackrel{\text{def}}{=} \{0,\ldots,n-1\}$. In both $\lambda_{\mathsf{b}}^{\to}$ and $\lambda_{\mathsf{h}}^{\to}$ we may represent such functions by terms of type $\mathsf{Nat} \to \mathsf{Bool}$. We will often informally write Nat_n in place of Nat to indicate that only the values $0,\ldots,n-1$ are relevant, but this convention has no formal status since our setup does not support dependent types.

To summarise, in both $\lambda_{\mathsf{b}}^{\rightarrow}$ and $\lambda_{\mathsf{h}}^{\rightarrow}$ we will be working with the types

$$\mathsf{Point} \stackrel{\text{def}}{=} \mathsf{Nat} \to \mathsf{Bool} \qquad \mathsf{Point}_n \stackrel{\text{def}}{=} \mathsf{Nat}_n \to \mathsf{Bool}$$
$$\mathsf{Predicate} \stackrel{\text{def}}{=} \mathsf{Point} \to \mathsf{Bool} \qquad \mathsf{Predicate}_n \stackrel{\text{def}}{=} \mathsf{Point}_n \to \mathsf{Bool}$$

and will be looking for programs

$$\mathsf{count}_n : \mathsf{Predicate}_n \to \mathsf{Nat}$$

such that for suitable terms P representing semantic predicates $\Pi : \mathbb{B}^n \to \mathbb{B}$, $\mathsf{count}_n\ P$ finds the number of points of $\mathbb{B}^n$ satisfying Π.

Before formalising these ideas more closely, let us look at some examples, which will also illustrate the machinery of *decision trees* that we will be using.

7.3.1 Examples of points, predicates and trees

Consider first the following terms of type Point:

$$\mathsf{q}_0 \stackrel{\text{def}}{=} \lambda\_.\mathsf{true} \qquad \mathsf{q}_1 \stackrel{\text{def}}{=} \lambda i.i = 0$$

$$\mathsf{q}_2 \stackrel{\text{def}}{=} \lambda i.\,\textbf{if}\ i = 0\ \textbf{then}\ \mathsf{true}\ \textbf{else if}\ i = 1\ \textbf{then}\ \mathsf{false}\ \textbf{else}\ \bot$$

(Here $\bot$ is the diverging term $(\textbf{rec}\, f\, i.f\, i)\, \langle\rangle$.) Then q_0 represents $\langle \mathsf{true}, \ldots, \mathsf{true} \rangle \in \mathbb{B}^n$ for any n; q_1 represents $\langle \mathsf{true}, \mathsf{false}, \ldots, \mathsf{false} \rangle \in \mathbb{B}^n$ for any $n \geq 1$; and q_2 represents $\langle \mathsf{true}, \mathsf{false} \rangle \in \mathbb{B}^2$.

Next some predicates. First, the following terms all represent the constant true predicate $\mathbb{B}^2 \to \mathbb{B}$:

$$\mathsf{T}_0 \stackrel{\text{def}}{=} \lambda q.\mathsf{true} \qquad \mathsf{T}_1 \stackrel{\text{def}}{=} \lambda q.(q\,1; q\,0; \mathsf{true}) \qquad \mathsf{T}_2 \stackrel{\text{def}}{=} \lambda q.(q\,0; q\,0; \mathsf{true})$$

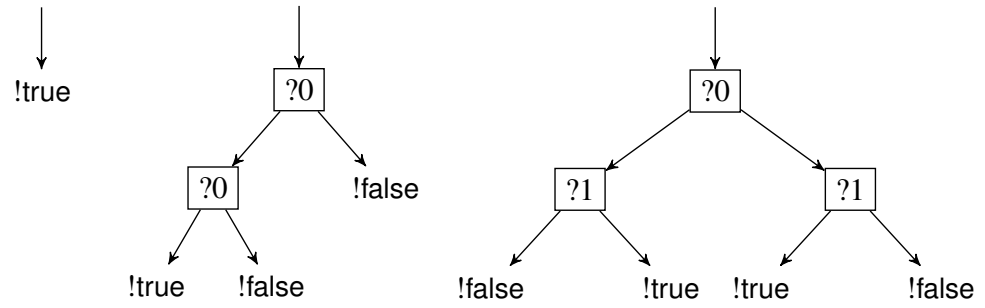
These illustrate that in the course of evaluating a predicate term P at a point q, for each $i < n$ the value of q at i may be inspected zero, one or many times.

Likewise, the following all represent the 'identity' predicate $\mathbb{B}^1 \to \mathbb{B}$ (here && is shortcut 'and'):

$$\mathsf{I}_0 \stackrel{\text{def}}{=} \lambda q.q\,0 \qquad \mathsf{I}_1 \stackrel{\text{def}}{=} \lambda q.\,\textbf{if}\ q\,0\ \textbf{then}\ \mathsf{true}\ \textbf{else}\ \mathsf{false} \qquad \mathsf{I}_2 \stackrel{\text{def}}{=} \lambda q.(q\,0)\,\&\&\,(q\,0)$$

Slightly more interestingly, for each n we have the following program which determines whether a point contains an odd number of true components:

$$\mathsf{odd}_n \stackrel{\text{def}}{=} \lambda q.\,\mathsf{fold} \otimes \mathsf{false}\ (\mathsf{map}\ q\ [0, \ldots, n-1])$$



(a) T_0 (b) I_2 (c) odd_2

Figure 7.7: Examples of decision trees.

Here fold and map are the standard combinators on lists, and $\otimes$ is exclusive-or. Applying odd_2 to q_0 yields false; applying it to q_1 or q_2 yields true.

We can think of a predicate term P as participating in a 'dialogue' with a given point $Q : \mathsf{Point}_n$. The predicate may *query* Q at some coordinate k; Q may *respond* with true or false and this returned value may influence the future course of the dialogue. After zero or more such query/response pairs, the predicate may return a final *answer* (true or false).

The set of possible dialogues with a given term P may be organised in an obvious way into an unrooted binary *decision tree*, in which each internal node is labelled with a query $?k$ (with $k < n$), and with left and right branches corresponding to the responses true, false respectively. Any point will thus determine a path through the tree, and each leaf is labelled with an answer !true or !false according to whether the corresponding point or points satisfy the predicate.

Decision trees for a sample of the above predicate terms are depicted in Figure 7.7; the relevant formal definitions are given in the next subsection. In the case of I_2, one of the !false leaves will be 'unreachable' if we are working in $\lambda_{\mathsf{b}}^{\rightarrow}$ (but reachable in a language supporting mutable state).

We think of the edges in the tree as corresponding to portions of computation undertaken by P between queries, or before delivering the final answer. The tree is unrooted (i.e. starts with an edge rather than a node) because in the evaluation of PQ there is potentially some 'thinking' done by P even before the first query or answer is reached. For the purpose of our runtime analysis, we will also consider *timed* variants of these decision trees, in which each edge is labelled with the number of computation

steps involved.

It is possible that for a given P the construction of a decision tree may hit trouble, because at some stage P either goes undefined or gets stuck at an unhandled operation. It is also possible that the decision tree is infinite because P can keep asking queries forever. However, we shall be restricting our attention to terms representing *total* predicates: those with finite decision trees in which every path leads to a leaf.

In order to present our complexity results in a simple and clear form, we will give special prominence to certain well-behaved decision trees. For $n \in \mathbb{N}$, we shall say a tree is *n-standard* if it is total (i.e. every maximal path leads to a leaf labelled with an answer) and along any path to a leaf, each coordinate $k < n$ is queried once and only once. Thus, an n-standard decision tree is a complete binary tree of depth $n+1$, with $2^n - 1$ internal nodes and 2^n leaves. However, there is no constraint on the order of the queries, which indeed may vary from one path to another. One pleasing property of this notion is that for a predicate term with an n-standard decision tree, the number of points in $\mathbb{B}^n$ satisfying the predicate is precisely the number of !true leaves in the tree.

Of the examples we have given, the tree for T_0 is 0-standard; those for I_0 and I_1 are 1-standard; that for odd_n is n-standard; and the rest are not n-standard for any n.

7.3.2 Formal definitions

We now formalise the above notions. We will present our definitions in the setting of $\lambda_{\mathsf{h}}^{\to}$, but everything can clearly be relativised to $\lambda_{\mathsf{b}}^{\to}$ with no change to the meaning in the case of $\lambda_{\mathsf{b}}^{\to}$ terms. For the purpose of this subsection we fix $n \in \mathbb{N}$, set $\mathbb{N}_n \stackrel{\text{def}}{=} \{0, \ldots, n-1\}$, and use k to range over $\mathbb{N}_n$. We write $\mathbb{B}$ for the set of booleans, which we shall identify with the (encoded) boolean values of $\lambda_{\mathsf{h}}^{\to}$, and use b to range over $\mathbb{B}$.

As suggested by the foregoing discussion, we will need to work with both syntax and semantics. For points, the relevant definitions are as follows.

Definition 7.3 (n-points)**.** A closed value $Q : \mathsf{Point}$ is said to be a *syntactic n-point* if:

$$\forall k \in \mathbb{N}_n. \exists b \in \mathbb{B}.\ Q\ k \rightsquigarrow^* \mathbf{return}\ b$$

A *semantic n-point* π is simply a mathematical function $\pi : \mathbb{N}_n \to \mathbb{B}$. (We shall also write $\pi \in \mathbb{B}^n$.) Any syntactic n-point Q is said to *denote* the semantic n-point $[\![Q]\!]$ given by:

$$\forall k \in \mathbb{N}_n, b \in \mathbb{B}.\ [\![Q]\!](k) = b \Leftrightarrow Q\ k \rightsquigarrow^* \mathbf{return}\ b$$

Any two syntactic n-points Q and Q' are said to be *distinct* if $[\![Q]\!] \neq [\![Q']\!]$.

By default, the unqualified term *n-point* will from now on refer to syntactic n-points.

Likewise, we wish to work with predicates both syntactically and semantically. By a *semantic n-predicate* we shall mean simply a mathematical function $\Pi : \mathbb{B}^n \to \mathbb{B}$. One slick way to define syntactic n-predicates would be as closed terms $P : \mathsf{Predicate}$ such that for every n-point Q, PQ evaluates to either **return** true or **return** false. For our purposes, however, we shall favour an approach to n-predicates via *decision trees*, which will yield more information on their behaviour.

We will model decision trees as certain partial functions from *addresses* to *labels*. An address will specify the position of a node in the tree via the path that leads to it, while a label will represent the information present at a node. Formally:

Definition 7.4 (untimed decision tree)**.**

(i) The address set Addr is simply the set $\mathbb{B}^*$ of finite lists of booleans. If $bs, bs' \in \mathsf{Addr}$, we write $bs \sqsubseteq bs'$ (resp. $bs \sqsubset bs'$) to mean that bs is a prefix (resp. proper prefix) of bs'.

(ii) The label set Lab consists of *queries* parameterised by a natural number and *answers* parameterised by a boolean:

$$\mathsf{Lab} \stackrel{\text{def}}{=} \{?k \mid k \in \mathbb{N}\} \cup \{!b \mid b \in \mathbb{B}\}$$

(iii) An (untimed) decision tree is a partial function $\tau : \mathsf{Addr} \rightharpoonup \mathsf{Lab}$ such that:

- The domain of τ (written $dom(\tau)$) is prefix closed.
- Answer nodes are always leaves: if $\tau(bs) = !b$ then $\tau(bs')$ is undefined whenever $bs \sqsubset bs'$.

As our goal is to reason about the time complexity of generic count programs and their predicates, it is also helpful to decorate decision trees with timing data that records the number of machine steps taken for each piece of computation performed by a predicate:

Definition 7.5 (timed decision tree)**.** A timed decision tree is a partial function $\tau : \mathsf{Addr} \rightharpoonup \mathsf{Lab} \times \mathbb{N}$ such that its first projection $bs \mapsto \tau(bs).1$ is a decision tree. We write $\mathsf{labs}(\tau)$ for the first projection $(bs \mapsto \tau(bs).1)$ and $\mathsf{steps}(\tau)$ for the second projection $(bs \mapsto \tau(bs).2)$ of a timed decision tree.

Here we think of $\mathsf{steps}(\tau)(bs)$ as the computation time associated with the edge whose *target* is the node addressed by bs.

We now come to the method for associating a specific tree with a given term P. One may think of this as a kind of denotational semantics, but here we shall extract a tree from a term by purely operational means using our abstract machine model. The key idea is to try applying P to a distinguished free variable $q : \mathsf{Point}$, which we think of as an 'abstract point'. Whenever P wants to interrogate its argument at some index i, the computation will get stuck at some term $q\,i$: this both flags up the presence of a query node in the decision tree, and allows us to explore the subsequent behaviour under both possible responses to this query.

The core of our definition is couched in terms of abstract machine configurations. We write Conf_q for the set of λ_h configurations possibly involving q (but no other free variables). We write $a \simeq b$ for Kleene equality: either both a and b are undefined or both are defined and $a = b$.

It is convenient to define the timed tree and then extract the untimed one from it:

Definition 7.6.

(i) Define $\mathcal{T} : \mathsf{Conf}_q \to \mathsf{Addr} \rightharpoonup (\mathsf{Lab} \times \mathbb{N})$ to be the minimal family of partial functions satisfying the following equations:

$$\begin{aligned}
\mathcal{T}(\langle \mathbf{return}\ W \mid \gamma \mid [] \rangle)\,[] &= (!b, 0), && \text{if } [\![W]\!]\gamma = b \\
\mathcal{T}(\langle z\,V \mid \gamma \mid \kappa \rangle)\,[] &= (?[\![V]\!]\gamma, 0), && \text{if } \gamma(z) = q \\
\mathcal{T}(\langle z\,V \mid \gamma \mid \kappa \rangle)\,(b :: bs) &\simeq \mathcal{T}(\langle \mathbf{return}\ b \mid \gamma \mid \kappa \rangle)\,bs, && \text{if } \gamma(z) = q \\
\mathcal{T}(\langle M \mid \gamma \mid \kappa \rangle)\,bs &\simeq \mathsf{inc}\,(\mathcal{T}(\langle M' \mid \gamma' \mid \kappa' \rangle)\,bs), \\
& && \text{if } \langle M \mid \gamma \mid \kappa \rangle \longrightarrow \langle M' \mid \gamma' \mid \kappa' \rangle
\end{aligned}$$

Here $\mathsf{inc}(\ell, s) = (\ell, s+1)$, and in all of the above equations $\gamma(q) = \gamma'(q) = q$. Clearly $\mathcal{T}(\mathcal{C})$ is a timed decision tree for any $\mathcal{C} \in \mathsf{Conf}_q$.

(ii) The timed decision tree of a computation term is obtained by placing it in the initial configuration: $\mathcal{T}(M) \stackrel{\text{def}}{=} \mathcal{T}(\langle M, \emptyset[q \mapsto q], \kappa_0 \rangle)$.

(iii) The timed decision tree of a closed value $P : \mathsf{Predicate}$ is $\mathcal{T}(P\,q)$. Since q plays the role of a dummy argument, we will usually omit it and write $\mathcal{T}(P)$ for $\mathcal{T}(P\,q)$.

(iv) The untimed decision tree $\mathcal{U}(P)$ is obtained from $\mathcal{T}(P)$ via first projection: $\mathcal{U}(P) = \mathsf{labs}(\mathcal{T}(P))$.

If the execution of a configuration $\mathcal{C}$ runs forever or gets stuck at an unhandled operation, then $\mathcal{T}(\mathcal{C})(bs)$ will be undefined for all bs. Although this is admitted by our definition of decision tree, we wish to exclude such behaviours for the terms we accept as valid predicates. Specifically, we frame the following definition:

Definition 7.7. A decision tree τ is an *n-predicate tree* if it satisfies the following:

- For every query $?k$ appearing in τ, we have $k \in \mathbb{N}_n$.
- Every query node has both children present:

$$\forall bs \in \mathsf{Addr}, k \in \mathbb{N}_n, b \in \mathbb{B}.\ \tau(bs) = ?k \Rightarrow bs +\!\!+ [b] \in dom(\tau)$$

- All paths in τ are finite (so every maximal path terminates in an answer node).

A closed term $P : \mathsf{Predicate}$ is a *(syntactic) n-predicate* if $\mathcal{U}(P)$ is an n-predicate tree.

If τ is an n-predicate tree, clearly any semantic n-point π gives rise to a path $b_0 b_1 \ldots$ through τ, given inductively by:

$$\forall j.\ \text{if } \tau(b_0 \ldots b_{j-1}) = ?k_j \text{ then } b_j = \pi(k_j)$$

This path will terminate at some answer node $b_0 b_1 \ldots b_{r-1}$ of τ, and we may write $\tau \bullet \pi \in \mathbb{B}$ for the answer at this leaf.

Proposition 7.8. *If P is an n-predicate and Q is an n-point, then $PQ \rightsquigarrow^*$* **return** *b where $b = \mathcal{U}(P) \bullet [\![Q]\!]$.*

Proof. By interleaving the computation for the relevant path through $\mathcal{U}(P)$ with computations for queries to Q, and appealing to the correspondence between the small-step reduction and abstract machine semantics. We omit the routine details. □

It is thus natural to define the *denotation* of an n-predicate P to be the semantic n-predicate $[\![P]\!]$ given by $[\![P]\!](\pi) = \mathcal{U}(P) \bullet \pi$.

As mentioned earlier, we shall also be interested in a more constrained class of trees and predicates:

Definition 7.9 (n-standard trees and predicates)**.** An n-predicate tree τ is said to be n-standard if the following hold:

- The domain of τ is precisely Addr_n, the set of bit vectors of length $\leq n$.

- There are no repeated queries along any path in τ:

$$\forall bs, bs' \in dom(\tau), k \in \mathbb{N}_n.\ bs \sqsubseteq bs' \wedge \tau(bs) = \tau(bs') = ?k \Rightarrow bs = bs'$$

A timed decision tree τ is n-standard if its underlying untimed decision tree $(bs \mapsto \tau(bs).1)$ is so. An n-predicate P is n-standard if $\mathcal{T}(P)$ is n-standard.

Clearly, in an n-standard tree, each of the n queries $?0, \ldots, ?(n-1)$ appears exactly once on the path to any leaf, and there are 2^n leaves, all of them answer nodes.

7.3.3 Specification of counting programs

We can now specify what it means for a program $K : \mathsf{Predicate} \to \mathsf{Nat}$ to implement counting.

Definition 7.10. (i) The *count* of a semantic n-predicate Π, written $\sharp\Pi$, is simply the number of semantic n-points $\pi \in \mathbb{B}^n$ for which $\Pi(\pi) = \mathsf{true}$.

(ii) If P is any n-predicate, we say that K *correctly counts* P if $K\,P \rightsquigarrow^*$ **return** m, where $m = \sharp[\![P]\!]$.

This definition gives us the flexibility to talk about counting programs that operate on various classes of predicates, allowing us to state our results in their strongest natural form. On the positive side, we shall shortly see that there is a single 'efficient' program in $\lambda_{\mathsf{h}}^{\to}$ that correctly counts all n-standard $\lambda_{\mathsf{h}}^{\to}$ predicates for every n; in Section 7.5.1 we improve this to one that correctly counts *all* n-predicates of $\lambda_{\mathsf{h}}^{\to}$. On the negative side, we shall show that an n-indexed family of counting programs written in $\lambda_{\mathsf{b}}^{\to}$, even if only required to work correctly on n-standard λ_b predicates, can never compete with our $\lambda_{\mathsf{h}}^{\to}$ program for asymptotic efficiency even in the most favourable cases.

7.3.4 Efficient generic count with effects

Now we are ready to implement a generic count function using effect handlers. In fact, the implementation is so generic that it works on all n-standard predicates.

The program uses a variation of the handler for nondeterministic computation from Section 2.4. The main idea is to implement points as nondeterministic computations using the Branch operation such that the handler may respond to every query twice, by invoking the provided resumption with true and subsequently false. The key insight is that the resumption restarts computation at the invocation site of Branch, which means

that prior computation need not be repeated. In other words, the resumption ensures that common portions of computations prior to any query are shared between both branches.

We assert that $\mathsf{Branch} : 1 \to \mathsf{Bool} \in \Sigma$ is a distinguished operation that may not be handled in the definition of any input predicate (it has to be forwarded according to the default convention). The algorithm is then as follows.

$$
\begin{array}{l}
\mathsf{effcount} : ((\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\
\mathsf{effcount}\, pred \stackrel{\text{def}}{=} \mathbf{handle}\ pred\,(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with} \\
\qquad\qquad \begin{array}{lcl}
\mathbf{return}\, x & \mapsto & \mathbf{if}\ x\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 0 \\
\langle\!\langle \mathsf{Branch}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \mathbf{in} \\
 & & \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}
\end{array}
\end{array}
$$

The handler applies predicate *pred* to a single 'generic point' defined using Branch. The boolean return value is interpreted as a single solution, whilst Branch is interpreted by alternately supplying true and false to the resumption and summing the results. The sharing enabled by the use of the resumption is exactly the 'magic' we need to make it possible to implement generic count more efficiently in $\lambda_{\mathsf{h}}^{\to}$ than in $\lambda_{\mathsf{b}}^{\to}$. A curious feature of effcount is that it works for all n-standard predicates without having to know the value of n. This is because the generic point $(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)$ informally serves as a 'superposition' of all possible points.

We may now articulate the crucial correctness and efficiency properties of effcount.

Theorem 7.11. *The following hold for any $n \in \mathbb{N}$ and any n-standard predicate P of $\lambda_{\mathsf{h}}^{\to}$:*

1. effcount *correctly counts P.*
2. *The number of machine steps required to evaluate* effcount *P is*

$$\left(\sum_{bs \in \mathsf{Addr}_n} \mathsf{steps}(\mathcal{T}(P))(bs)\right) \;+\; O(2^n)$$

Proof outline. Suppose $bs \in \mathsf{Addr}_n$, with $|bs| = j$. From the construction of $\mathcal{T}(P)$, one may easily read off a configuration $\mathcal{C}_{bs}$ whose execution is expected to compute the count for the subtree below node bs, and we can explicitly describe the form $\mathcal{C}_{bs}$ will have. We write $\mathsf{Hyp}(bs)$ for the claim that $\mathcal{C}_{bs}$ correctly counts this subtree, and does so within the following number of steps:

$$\left(\sum_{bs' \in \mathsf{Addr}_n,\, bs' \sqsupset bs} \mathsf{steps}(\mathcal{T}(P))(bs')\right) \;+\; 9 * (2^{n-j} - 1) + 2 * 2^{n-j}$$

The $9 * (2^{n-j} - 1)$ expression is the number of machine steps contributed by the Branch-case inside the handler, whilst the $2 * 2^{n-j}$ expression is the number of machine steps contributed by the **return**-case. We prove $\mathsf{Hyp}(bs)$ by a laborious but routine downwards induction on the length of bs. The proof combines counting of explicit machine steps with 'oracular' appeals to the assumed behaviour of P as modelled by $\mathcal{T}(P)$. Once $\mathsf{Hyp}([])$ is established, both halves of the theorem follow easily. The proof details and development of the proof gadgets are in Appendix C. □

The above formula can clearly be simplified for certain reasonable classes of predicates. For instance, suppose we fix some constant $c \in \mathbb{N}$, and let $\mathcal{P}_{n,c}$ be the class of all n-standard predicates P for which all the edge times $\mathsf{steps}(\mathcal{T}(P))(bs)$ are bounded by c. (Clearly, many reasonable predicates will belong to $\mathcal{P}_{n,c}$ for some modest value of c.) Since the number of sequences bs in question is less than 2^{n+1}, we may read off from the above formula that for predicates in $\mathcal{P}_{n,c}$, the runtime of effcount is $O(c2^n)$.

Alternatively, should we wish to use the finer-grained cost model that assigns an $O(\log|\gamma|)$ runtime to each abstract machine step (see Section 7.2.3), we may note that any environment γ arising in the computation contains at most n entries introduced by the let-bindings in effcount, and (if $P \in \mathcal{P}_{n,c}$) at most $O(cn)$ entries introduced by P. Thus, the time for each step in the computation remains $O(\log c + \log n)$, and the total runtime for effcount is $O(c2^n(\log c + \log n))$.

One might also ask about the execution time for an implementation of $\lambda_{\mathsf{h}}^{\rightarrow}$ that performs genuine copying of continuations, as in systems such as MLton [96]. As MLton copies the entire continuation (stack), whose size is $O(n)$, at each of the 2^n branches, continuation copying alone takes time $O(n2^n)$ and the effectful implementation offers no performance benefit (Tables 7.3 and 7.4). More refined implementations [78, 92] that are able to take advantage of delimited control operators or sharing in copies of the stack can bring the complexity of continuation copying back down to $O(2^n)$.

Finally, one might consider another dimension of cost, namely the space used by effcount. Consider a class $Q_{n,c,d}$ of n-standard predicates P for which the edge times in $\mathcal{T}(P)$ never exceed c and the sizes of pure continuations never exceed d. If we consider any $P \in Q_{n,c,d}$ then the total number of environment entries is bounded by cn, taking up space $O(cn(\log cn))$. We must also account for the pure continuations. There are n of these, each taking at most d space. Thus the total space is $O(n(d + c(\log c + \log n)))$.

7.4 Pure generic count: a lower bound

We have shown that there is an implementation of generic count in $\lambda_{\mathsf{h}}^{\rightarrow}$ with a runtime bound of $O(2^n)$ for certain well-behaved predicates. We now prove that no implementation in $\lambda_{\mathsf{b}}^{\rightarrow}$ can match this: in fact, we establish a lower bound of $\Omega(n2^n)$ for the runtime of any counting program on *any* n-standard predicate. This mathematically rigorous characterisation of the efficiency gap between languages with and without effect handlers is the objective of this chapter.

To get a feel for the issues that the proof must address, let us consider how one might construct a counting program in $\lambda_{\mathsf{b}}^{\rightarrow}$. The naïve approach, of course, would be simply to apply the given predicate P to all 2^n possible n-points in turn, keeping a count of those on which P yields true. It is a routine exercise to implement this approach in $\lambda_{\mathsf{b}}^{\rightarrow}$, yielding (parametrically in n) a program

$$\mathsf{naivecount}_n \ : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat}$$

Since the evaluation of an n-standard predicate on an individual n-point must clearly take time $\Omega(n)$, we have that the evaluation of $\mathsf{naivecount}_n$ on any n-standard predicate P must take time $\Omega(n2^n)$. If P is not n-standard, the $\Omega(n)$ lower bound need not apply, but we may still say that the evaluation of $\mathsf{naivecount}_n$ on *any* predicate P (at level n) must take time $\Omega(2^n)$.

One might at first suppose that these properties are inevitable for any implementation of generic count within $\lambda_{\mathsf{b}}^{\rightarrow}$, or indeed any purely functional language: surely, the only way to learn something about the behaviour of P on every possible n-point is to apply P to each of these points in turn? It turns out, however, that the $\Omega(2^n)$ lower bound can sometimes be circumvented by implementations that cleverly exploit *nesting* of calls to P. The germ of the idea may be illustrated within $\lambda_{\mathsf{b}}^{\rightarrow}$ itself. Suppose that we first construct some program

$$\mathsf{bestshot}_n \ : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to (\mathsf{Nat}_n \to \mathsf{Bool})$$

which, given a predicate P, returns some n-point Q such that $P\ Q$ evaluates to true, if such a point exists, and any point at all if no such point exists. (In other words, $\mathsf{bestshot}_n$ embodies Hilbert's choice operator ε on predicates.) It is once again routine to construct such a program by naïve means; and we may moreover assume that for any P, the evaluation of $\mathsf{bestshot}_n\ P$ takes only constant time, all the real work being deferred until the argument of type Nat_n is supplied.

Now consider the following program:

$$\mathsf{lazycount}_n \stackrel{\text{def}}{=} \lambda \mathit{pred}.\ \textbf{if}\ \mathit{pred}\ (\mathsf{bestshot}_n\ \mathit{pred})\ \textbf{then}\ \mathsf{naivecount}_n\ \mathit{pred}\ \textbf{else return}\ 0$$

Here the term $\mathit{pred}\ (\mathsf{bestshot}_n\ \mathit{pred})$ serves to test whether there exists an n-point satisfying *pred*: if there is not, our count program may return 0 straightaway. It is thus clear that $\mathsf{lazycount}_n$ is a correct implementation of generic count, and also that if *pred* is the predicate $\lambda q.\mathsf{false}$ then $\mathsf{lazycount}_n\ \mathit{pred}$ returns 0 within $O(1)$ time, thus violating the $\Omega(2^n)$ lower bound suggested above.

This might seem like a footling point, as $\mathsf{lazycount}_n$ offers this efficiency gain *only* on (certain implementations of) the constantly false predicate. However, it turns out that by a recursive application of this nesting trick, we may arrive at a generic count program that spectacularly defies the $\Omega(2^n)$ lower bound for an interesting class of (non-n-standard) predicates, and indeed proves quite viable for counting solutions to 'n-queens' and similar problems. We shall refer to this program as BergerCount, as it is modelled largely on Berger's PCF implementation of the so-called *fan functional* [19, 180]. This program is of interest in its own right and is briefly presented in Appendix D. It actually requires a mild extension of $\lambda_{\mathsf{b}}^{\rightarrow}$ with a 'memoisation' primitive to achieve the effect of call-by-need evaluation; but such a language can still be seen as purely 'functional' in the same sense as Haskell.

In the meantime, however, the moral is that the use of *nesting* can lead to surprising phenomena which sometimes defy intuition (Escardó [76] gives some striking further examples). What we now wish to show is that for *n-standard* predicates, the naïve lower bound of $\Omega(n2^n)$ cannot in fact be circumvented. The example of BergerCount both highlights the need for a rigorous proof of this and tells us that such a proof will need to pay particular attention to the possibility of nesting.

We now proceed to the proof itself. We here present the argument in the basic setting of $\lambda_{\mathsf{b}}^{\rightarrow}$; later we will see how a more delicate argument applies to languages with mutable state (Section 7.5.3).

As a first step, we note that where lower bounds are concerned, it will suffice to work with the small-step operational semantics of $\lambda_{\mathsf{b}}^{\rightarrow}$ rather than the more elaborate abstract machine model employed in Section 7.2.1. This is because, as observed in Section 7.2.1, there is a tight correspondence between these two execution models such that for the evaluation of any closed term, the number of abstract machine steps is always at least the number of small-step reductions. Thus, if we are able to show that the number of small-step reductions for any generic program program in $\lambda_{\mathsf{b}}^{\rightarrow}$ on

any n-standard predicate is $\Omega(n2^n)$, this will establish the desired lower bound on the runtime.

Let us suppose, then, that K is a program of $\lambda_{\mathfrak{b}}^{\to}$ that correctly counts all n-standard predicates of $\lambda_{\mathfrak{b}}^{\to}$ for some specific n. We now establish a key lemma, which vindicates the naïve intuition that if P is n-standard, the only way for K to discover the correct value for $\sharp[\![P]\!]$ is to perform 2^n separate applications $P\ Q$ (allowing for the possibility that these applications need not be performed 'in turn' but might be nested in some complex way).

Lemma 7.12 (No shortcuts)**.** *Suppose K correctly counts all n-standard predicates of $\lambda_{\mathfrak{b}}^{\to}$. If P is an n-standard predicate, then K applies P to at least 2^n distinct n-points. More formally, for any of the 2^n possible semantic n-points $\pi : \mathbb{N}_n \to \mathbb{B}$, there is a term $\mathcal{E}[P\ Q]$ appearing in the small-step reduction of $K\ P$ such that Q is an n-point and $[\![Q]\!] = \pi$.*

Proof. Suppose for a contradiction that π is some semantic n-point such that no application $P\ Q$ with $[\![Q]\!] = \pi$ ever arises in the course of computing $K\ P$. Let τ be the untimed decision tree for P. Let l be the maximal path through τ associated with π: that is, the one we construct by responding to each query $?k$ with $\pi(k)$. Then l is a leaf node such that $\tau(l) = !(\tau \bullet \pi)$. We now let τ' be the tree obtained from τ by simply negating this answer value at l.

It is a simple matter to construct a $\lambda_{\mathfrak{b}}^{\to}$ n-standard predicate P' whose decision tree is τ'. This may be done just by mirroring the structure of τ' by nested **if** statements; we omit the easy details.

Since the numbers of true-leaves in τ and τ' differ by 1, it is clear that if K indeed correctly counts all n-standard predicates, then the values returned by $K\ P$ and $K\ P'$ will have an absolute difference of 1. On the other hand, we shall argue that if the computation of $K\ P$ never actually 'visits' the leaf l in question, then K will be unable to detect any difference between P and P'.

The situation is reminiscent of Milner's *context lemma* [198], which (loosely) says that essentially the only way to observe a difference between two programs is to apply them to some argument on which they differ. Traditional proofs of the context lemma reason by induction on length of reduction sequences, and our present proof is closely modelled on these.

We shall make frequent use of term contexts $M[-]$ with a hole of type Predicate (which may appear zero, one or more times in M) in order to highlight particular occur-

rences of P within a term. The following definition enables us to talk about computations that avoid the critical point π:

Definition 7.13 (Safe terms)**.** If $M[-]$ is such a context of ground type, let us say $M[-]$ is *safe* if

- $M[P]$ is closed, and $M[P] \rightsquigarrow^* \mathbf{return}\ W$ for some closed ground type value W;
- For any term $\mathcal{E}[P\ Q]$ appearing in the reduction of $M[P]$, where the applicand P in $P\ Q$ is a residual of one of the abstracted occurrences in $M[P]$, we have that $[\![Q]\!] \neq \pi$.

We may express this as '$M[P]$ is safe' when it is clear which occurrences of P we intend to abstract.

For example, our current hypotheses imply that $K\ P$ is safe (formally, $K'[-] \stackrel{\text{def}}{=} K\ -$ is safe). We may now prove the following:

Lemma 7.14.

(i) Suppose $Q[-] : \mathsf{Point}$ *and* $k : \mathsf{Nat}$ *are values such that* $Q[P]\ k$ *is safe, and suppose* $Q[P]\ k \rightsquigarrow^m \mathbf{return}\ b$ *where* $m \in \mathbb{N}$*. Then also* $Q[P']\ k \rightsquigarrow^* \mathbf{return}\ b$*.*

(ii) Suppose $P\ Q[P]$ *is safe and* $P\ Q[P] \rightsquigarrow^m \mathbf{return}\ b$*. Then also* $P'\ Q[P'] \rightsquigarrow^* \mathbf{return}\ b$*.*

We prove these claims by simultaneous induction on the computation length m. Both claims are vacuous when $m = 0$ as neither $Q[P]\ k$ nor $P\ Q[P]$ is a **return** term. We therefore assume $m > 0$ where both claims hold for all $m' < m$.

(i) Let $p : \mathsf{Predicate}$ be a distinguished free variable, and consider the behaviour of $Q[p]\ k$. If this reduces to a value $\mathbf{return}\ W$, then also $Q[P]\ k \rightsquigarrow^* \mathbf{return}\ W$, whence $W = b$ and also $Q[P']\ k \rightsquigarrow \mathbf{return}\ b$ as required. Otherwise, the reduction of $Q[p]\ k$ will get stuck at some term $M_0 = \mathcal{E}_0[p\ Q_0[p], p]$. Here the first hole in $\mathcal{E}_0[-,-]$ is in the evaluation position, and the second hole abstracts all remaining occurrences of p within M_0. We may also assume that $Q_0[-]$ abstracts all occurrences of p in $Q_0[p]$.

Correspondingly, the reduction of $Q[P]\ k$ will reach $\mathcal{E}_0[P\ Q_0[P], P]$ and then proceed with the embedded reduction of $P\ Q_0[P]$. Note that $P\ Q_0[P]$ will be safe because $Q[P]\ k$ is. So let us suppose that $P\ Q_0[P] \rightsquigarrow^* \mathbf{return}\ b_0$, whence $Q[P]\ k \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P]$.

We may now investigate the subsequent reduction behaviour of $Q[P]\ k$ by considering the reduction of $\mathcal{E}_0[\mathbf{return}\ b_0, p]$. Once again, this may reduce to a value $\mathbf{return}\ W$, in which case $W = b$ and our computation is complete. Otherwise, the reduction of

$\mathcal{E}_0[\mathbf{return}\ b_0, p]$ will get stuck at some $M_1 = \mathcal{E}_1[p\ Q_1[p], p]$, and we may again proceed as above.

By continuing in this way, we may analyse the reduction of $Q[P]\ k$ as follows.

$$\begin{aligned} Q[P]\ k &\rightsquigarrow^* \mathcal{E}_0[P\ Q_0[P], P] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P] \rightsquigarrow^* \mathcal{E}_1[P\ Q_1[P], P] \\ &\rightsquigarrow^* \mathcal{E}_1[\mathbf{return}\ b_1, P] \rightsquigarrow^* \ldots \rightsquigarrow^* \mathcal{E}_{r-1}[P\ Q_{r-1}[P], P] \\ &\rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P] \rightsquigarrow \mathbf{return}\ b \end{aligned}$$

Here the terms $P\ Q_j[P]$ will be safe, and the reductions $P\ Q_j[P] \rightsquigarrow^* \mathbf{return}\ b_j$ each have length $< m$. We may therefore apply part (ii) of the induction hypothesis and conclude that also $P'\ Q_j[P'] \rightsquigarrow^* \mathbf{return}\ b_j$. Furthermore, the remaining segments of the above computation are all obtained as instantiations of 'generic' reduction sequences involving p, so these segments will remain valid if p is instantiated to P'. Reassembling everything, we have a valid reduction sequence:

$$\begin{aligned} Q[P']\ k &\rightsquigarrow^* \mathcal{E}_0[P'\ Q_0[P'], P'] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P'] \rightsquigarrow^* \mathcal{E}_1[P'\ Q_1[P'], P'] \\ &\rightsquigarrow^* \mathcal{E}_1[\mathbf{return}\ b_1, P'] \rightsquigarrow^* \ldots \rightsquigarrow^* \mathcal{E}_{r-1}[P'\ Q_{r-1}[P'], P'] \\ &\rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P'] \rightsquigarrow \mathbf{return}\ b \end{aligned}$$

This establishes the induction step for part (i).

(ii) We may apply a similar analysis to the computation of $P\ Q[P]$ to detect the places where $Q[P]$ is applied to an argument. We do this by considering the reduction behaviour of $P\ q$, where $q : \mathsf{Point}$ is the distinguished variable that featured in Definition 7.6. In this way we may analyse the computation of $P\ Q[P]$ as:

$$\begin{aligned} P\ Q[P] &\rightsquigarrow^* \mathcal{E}_0[Q[P]\ k_0, Q[P]] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, Q[P]] \rightsquigarrow^* \mathcal{E}_1[Q[P]\ k_1, Q[P]] \rightsquigarrow^* \ldots \\ &\rightsquigarrow^* \mathcal{E}_{r-1}[Q[P]\ k_{r-1}, Q[P]] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, Q[P]] \rightsquigarrow \mathbf{return}\ b \end{aligned}$$

where for each j, the first hole in $\mathcal{E}_j[-,-]$ is in evaluation position, the term $Q[P]\ k_j$ is safe, the reduction $Q[P]\ k_j \rightsquigarrow^* \mathbf{return}\ b_j$ has length $< m$, and the remaining portions of computation are instantiations of generic reductions involving q. By part (i) of the induction hypothesis we may conclude that also $Q[P']\ k_j \rightsquigarrow^* \mathbf{return}\ b_j$ for each j, and for the remaining segments of computation we may instantiate q to $Q[P']$. We thus obtain a computation exhibiting that $P\ Q[P'] \rightsquigarrow^* \mathbf{return}\ b$.

It remains to show that the applicand P may be replaced by P' here without affecting the result. The idea here is that the booleans $b_0, \ldots, b_{r-1}$ trace out a path through the decision tree for P; but since $P\ Q[P]$ is safe, we have that $[\![Q[P]]\!] \neq \pi$, and so this path does *not* lead to the critical leaf l. We now have everything we need to establish that $P'\ Q[P'] \rightsquigarrow^* \mathbf{return}\ b$ as required.

More formally, in view of the correspondence between small-step reduction and abstract machine semantics, we may readily correlate the above computation of $P\ Q[P]$ with an exploration of the path $bs = b_0 \dots b_{r-1}$ in $\tau = \mathcal{U}(P)$, leading to a leaf with label $!b$. Since P is n-standard, this correlation shows that $r = n$, that for each j we have $\tau(b_0 \dots b_{j-1}) = ?k_j$, and that $\{k_0, \dots, k_{r-1}\} = \{0, \dots, n-1\}$. Furthermore, we have already ascertained that the values of $Q[P]$ and $Q[P']$ at k_j are both b_j, whence $[\![Q[P]]\!] = [\![Q[P']]\!] = \pi'$ where $\pi'(k_j) = b_j$ for all j. But $P\ Q[P]$ is safe, so in particular $\pi' = [\![Q[P]]\!] \neq \pi$. We therefore also have $\tau'(b_0 \dots b_{j-1}) = ?k_j$ for each $j \leq r$ and $\tau'(b_0 \dots b_{r-1}) = b$. Since $\tau' = \mathcal{U}(P')$ and $[\![Q[P']]\!] = \pi'$, we may conclude by Proposition 7.8 that $P'\ Q[P'] \rightsquigarrow^*$ **return** b. This completes the proof of Lemma 7.14.

To finish off the proof of Lemma 7.12, we apply the same analysis one last time to the reduction of $K\ P$ itself. This will have the form

$$\begin{aligned} K\ P &\rightsquigarrow^* \mathcal{E}_0[P\ Q_0[P], P] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P] \rightsquigarrow^* \dots \\ &\rightsquigarrow^* \mathcal{E}_{r-1}[P\ Q_{r-1}[P], P] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P] \rightsquigarrow^* \mathbf{return}\ c \end{aligned}$$

where, by hypothesis, each $P\ Q_j[P]$ is safe. Using Lemma 7.14 we may replace each subcomputation $P\ Q_j[P] \rightsquigarrow^*$ **return** b_j with $P'\ Q_j[P'] \rightsquigarrow^*$ **return** b_j, and so construct a computation exhibiting that $K\ P' \rightsquigarrow^*$ **return** c.

This gives our contradiction, as the values of $K\ P$ and $K\ P'$ are supposed to differ by 1. □

Corollary 7.15. *Suppose K and P are as in Lemma 7.12. For any semantic n-point π and any natural number $k < n$, the reduction sequence for $K\ P$ contains a term $\mathcal{F}[Q\ k]$, where $\mathcal{F}$ is an evaluation context and $[\![Q]\!] = \pi$.*

Proof. Suppose $\pi \in \mathbb{B}^n$. By Lemma 7.12, the computation of $K\ P$ contains some $\mathcal{E}[P\ Q]$ where $[\![Q]\!] = \pi$, and the above analysis of the computation of $P\ Q$ shows that it contains a term $\mathcal{E}'[Q\ k]$ for each $k < n$. The corollary follows, taking $\mathcal{F}[-] \stackrel{\text{def}}{=} \mathcal{E}[\mathcal{E}'[-]]$. □

This gives our desired lower bound. Since our n-points Q are values, it is clearly impossible that $\mathcal{F}[Q\ k] = \mathcal{F}'[Q'\ k']$ (where $\mathcal{F}, \mathcal{F}'$ are evaluation contexts) unless $Q = Q'$ and $k = k'$. We may therefore read off π from $\mathcal{F}[Q\ k]$ as $[\![Q]\!]$. There are thus at least $n2^n$ distinct terms in the reduction sequence for $K\ P$, so the reduction has length $\geq n2^n$. We have thus proved:

Theorem 7.16. *If K is a $\lambda_{\mathsf{b}}^{\rightarrow}$ program that correctly counts all n-standard $\lambda_{\mathsf{b}}^{\rightarrow}$ predicates, and P is any n-standard $\lambda_{\mathsf{b}}^{\rightarrow}$ predicate, then the evaluation of $K\ P$ must take time $\Omega(n2^n)$.* □

Although we shall not go into details, it is not too hard to apply our proof strategy with minor adjustments to certain richer languages: for instance, an extension of $\lambda_{\mathsf{b}}^{\to}$ with exceptions, or one containing the memoisation primitive required for BergerCount (Appendix D). A deeper adaptation is required for languages with state: we will return to this in Section 7.5.

It is worth noting where the above argument breaks down if applied to $\lambda_{\mathsf{h}}^{\to}$. In $\lambda_{\mathsf{b}}^{\to}$, in the course of computing $K\ P$, every Q to which P is applied will be a self-contained closed term denoting some specific point π. This is intuitively why we may only learn about one point at a time. In $\lambda_{\mathsf{h}}^{\to}$, this is not the case, because of the presence of operation symbols. For instance, our effcount program from Section 7.3.4 will apply P to the 'generic point' $\lambda\_.$**do** Branch $\langle\rangle$. Thus, for example, in our treatment of Lemma 7.14(i), it need no longer be the case that the reduction of $Q[p]\ k$ either yields a value or gets stuck at some $\mathcal{E}_0[p\ Q_0[p], p]$: a third possibility is that it gets stuck at some invocation of ℓ, so that control will then pass to the effect handler.

7.5 Extensions and variations

Our complexity result is robust in that it continues to hold in more general settings. We outline here how it generalises: beyond n-standard predicates, from generic count to generic search, and from pure $\lambda_{\mathsf{b}}^{\to}$ to stateful $\lambda_{\mathsf{s}}^{\to}$.

7.5.1 Beyond n-standard predicates

The n-standard restriction on predicates serves to make the efficiency phenomenon stand out as clearly as possible. However, we can relax the restriction by tweaking effcount to handle repeated queries and missing queries. The trade off is that the analysis of effcount becomes more involved. The key to relaxing the n-standard restriction is the use of state to keep track of which queries have been computed. We can give stateful implementations of effcount without changing its type signature by using *parameter-passing* [143, 232] to internalise state within a handler. Parameter-passing abstracts every handler clause such that the current state is supplied before the evaluation of a clause continues and the state is threaded through resumptions: a resumption becomes a two-argument curried function $r : B \to S \to D$, where the first argument of type B is the return type of the operation and the second argument is the updated state of type S.

Repeated queries We can generalise effcount to handle repeated queries by memoising previous answers. First, we generalise the type of Branch such that it carries an index of a query.

$$\mathsf{Branch} : \mathsf{Nat} \to \mathsf{Bool}$$

We assume a family of natural number to boolean maps, Map_n with the following interface.

$$\begin{aligned} \mathsf{empty}_n &: \mathsf{Map}_n \\ \mathsf{add}_n &: (\mathsf{Nat}_n \times \mathsf{Bool}) \to \mathsf{Map}_n \to \mathsf{Map}_n \\ \mathsf{lookup}_n &: \mathsf{Nat}_n \to \mathsf{Map}_n \to (1 + \mathsf{Bool}) \end{aligned}$$

Invoking lookup i *map* returns **inl** $\langle\rangle$ if i is not present in *map*, and **inr** *ans* if i is associated by *map* with the value *ans* : Bool. Allowing ourselves a few extra constant-time arithmetic operations, we can realise suitable maps in $\lambda_{\mathsf{b}}^{\to}$ such that the time complexity of add_n and lookup_n is $O(\log n)$ [210]. We can then use parameter-passing to support repeated queries as follows.

$$\begin{aligned}
&\mathsf{effcount}'_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathsf{effcount}'_n\ \mathit{pred} \stackrel{\text{def}}{=} \mathbf{let}\ h \leftarrow \mathbf{handle}\ \mathit{pred}\ (\lambda i.\mathbf{do}\ \mathsf{Branch}\ i)\ \mathbf{with} \\
&\qquad \mathbf{return}\ x \qquad\qquad \mapsto \lambda s.\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
&\qquad \langle\!\langle \mathsf{Branch}\ i \twoheadrightarrow r \rangle\!\rangle \mapsto \lambda s.\mathbf{case}\ \mathsf{lookup}_n\ i\ s\ \{ \\
&\qquad\qquad\qquad \mathbf{inl}\ \langle\rangle \mapsto \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (\mathsf{add}_n\ \langle i, \mathsf{true}\rangle\ s)\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (\mathsf{add}_n\ \langle i, \mathsf{false}\rangle\ s)\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad x_{\mathsf{true}} + x_{\mathsf{false}}; \\
&\qquad\qquad\qquad \mathbf{inr}\ x \mapsto r\ x\ s\ \} \\
&\qquad \mathbf{in}\ h\ \mathsf{empty}_n
\end{aligned}$$

The state parameter s memoises query results, thus avoiding double-counting and enabling $\mathsf{effcount}'_n$ to work correctly for predicates performing the same query multiple times.

Missing queries Similarly, we can use parameter-passing to support missing queries.

$$
\begin{array}{l}
\mathsf{effcount}''_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\
\mathsf{effcount}''_n\ \mathit{pred} \stackrel{\text{def}}{=} \mathbf{let}\ h \leftarrow \mathbf{handle}\ \mathit{pred}\,(\lambda i.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with} \\
\qquad\qquad \mathbf{return}\ x \mapsto \lambda d.\mathbf{let}\ \mathit{result} \leftarrow \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathit{result} \times 2^{n-d} \\
\qquad\qquad \langle\!\langle \mathsf{Branch}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \lambda d.\mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (d+1)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (d+1)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad (x_{\mathsf{true}} + x_{\mathsf{false}}) \\
\qquad \mathbf{in}\ h\ 0
\end{array}
$$

The parameter d tracks the depth and the returned result is scaled by 2^{n-d} accounting for the unexplored part of the current subtree. This enables $\mathsf{effcount}''_n$ to operate correctly on predicates that inspect n points at most once. We leave it as an exercise for the reader to combine $\mathsf{effcount}'_n$ and $\mathsf{effcount}''_n$ in order to handle both repeated queries and missing queries.

7.5.2 From generic count to generic search

We can generalise the problem of generic counting to generic searching. The main operational difference is that a generic search procedure must materialise a list of solutions, thus its type is

$$\mathsf{search}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{List}_{\mathsf{Nat}_n \to \mathsf{Bool}}$$

where List_A is the type of cons-lists whose elements have type A. We modify effcount to return a list of solutions rather than the number of solutions by lifting each result into a singleton list and using list concatenation instead of addition to combine partial results xs_{true} and xs_{false} as follows.

$$
\begin{array}{l}
\mathsf{effsearch}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{List}_{\mathsf{Nat}_n \to \mathsf{Bool}} \\
\mathsf{effsearch}_n\ \mathit{pred} \stackrel{\text{def}}{=} \mathbf{let}\ f \leftarrow \mathbf{handle}\ \mathit{pred}\,(\lambda i.\mathbf{do}\ \mathsf{Branch}\ i)\ \mathbf{with} \\
\qquad\qquad \mathbf{return}\ x \mapsto \lambda q.\mathbf{if}\ x\ \mathbf{then}\ \mathsf{singleton}\ q\ \mathbf{else}\ \mathsf{nil} \\
\qquad\qquad \langle\!\langle \mathsf{Branch}\ i \twoheadrightarrow r \rangle\!\rangle \mapsto \\
\qquad\qquad\quad \lambda q.\mathbf{let}\ xs_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ q\ j)\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{let}\ xs_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ q\ j)\ \mathbf{in} \\
\qquad\qquad\qquad \mathsf{concat}\ \langle xs_{\mathsf{true}}, xs_{\mathsf{false}} \rangle \\
\qquad \mathbf{in}\ \mathsf{toConsList}\ (f\ (\lambda j.\bot))
\end{array}
$$

The Branch operation is now parameterised by an index i. The handler is now parameterised by the current path as a point q, which is output at a leaf iff it is in the predicate. A little care is required to ensure that $\mathsf{effsearch}_n$ has runtime $O(2^n)$; naïve use of cons-list concatenation would result in $O(n2^n)$ runtime, as cons-list concatenation is linear in its first operand. In place of cons-lists we use Hughes lists [131], which admit constant time concatenation: $\mathsf{HList}_A \stackrel{\text{def}}{=} \mathsf{List}_A \to \mathsf{List}_A$. The empty Hughes list $\mathsf{nil} : \mathsf{HList}_A$ is defined as the identity function: $\mathsf{nil} \stackrel{\text{def}}{=} \lambda xs.xs$.

$$\mathsf{singleton}_A : A \to \mathsf{HList}_A \qquad \mathsf{concat}_A : \mathsf{HList}_A \times \mathsf{HList}_A \to \mathsf{HList}_A$$
$$\mathsf{singleton}_A\ x \stackrel{\text{def}}{=} \lambda xs.x :: xs \qquad \mathsf{concat}_A\ f\ g \stackrel{\text{def}}{=} \lambda xs.g\ (f\ xs)$$
$$\mathsf{toConsList}_A : \mathsf{HList} \to \mathsf{List}_A$$
$$\mathsf{toConsList}_A\ f \stackrel{\text{def}}{=} f\ []$$

We use the function toConsList to convert the final Hughes list to a standard cons-list at the end; this conversion has linear time complexity (it just conses all of the elements of the list together).

7.5.3 From pure $\lambda_{\mathsf{b}}^{\to}$ to stateful $\lambda_{\mathsf{s}}^{\to}$

Mutable state is a staple ingredient of many practical programming languages. We now outline how our main lower bound result can be extended to a language with state. We will not give full details, but merely point out the respects in which our earlier treatment needs to be modified.

We have in mind an extension $\lambda_{\mathsf{s}}^{\to}$ of $\lambda_{\mathsf{b}}^{\to}$ with ML-style reference cells: we extend our grammar for types with a reference type ($\mathsf{Ref}\,A$), and that for computation terms with forms for creating references (**letref** $x = V$ **in** N), dereferencing ($!x$), and destructive update ($x := V$), with the familiar typing rules. We also add a new kind of value, namely *locations* l^A, of type $\mathsf{Ref}\ A$. We adopt a basic Scott-Strachey [1971] model of store: a location is a natural number decorated with a type, and the execution of a stateful program allocates locations in the order $0, 1, 2, \ldots$, assigning types to them as it does so. A *store* s is a type-respecting mapping from some set of locations $\{0, \ldots, l-1\}$ to values. For the purposes of small-step operational semantics, a *configuration* will be a triple (M, l, s), where M is a computation, l is a 'location counter', and s is a store with domain $\{0, \ldots, l-1\}$. A reduction relation $\rightsquigarrow$ on configurations is defined in a familiar way (again we omit the details).

Certain aspects of our setup require care in the presence of state. For instance, there is in general no unique way to assign an (untimed) decision tree to a closed value P :

$\mathsf{Predicate}_n$, since the behaviour of P on a value $q : \mathsf{Point}_n$ may depend both on the initial state when P is invoked, and on the ways in which the associated computations $q\ V \rightsquigarrow^*$ **return** W modify the state. In this situation, there is not even a clear specification for what an n-count program ought to do.

The simplest way to circumvent this difficulty is to restrict attention to predicates P *within the sublanguage* $\lambda_{\mathsf{b}}^{\rightarrow}$. For such predicates, the notions of decision tree, counting and n-standardness are unproblematic. Our result will establish a runtime lower bound of $\Omega(n2^n)$ for programs $K \in \lambda_{\mathsf{s}}^{\rightarrow}$ that correctly count predicates P of this kind. On the other hand, since K itself may be stateful, we cannot exclude the possibility that $K\ P$ will apply P to a term Q that is itself stateful. Such a Q will no longer unambiguously denote a semantic point π, hence the proof of Section 7.4 must be adapted.

To adapt our proof to the setting of $\lambda_{\mathsf{s}}^{\rightarrow}$, some more machinery is needed. If K is an n-count program and P an n-standard predicate, we expect that the evaluation of $K\ P$ will feature terms $\mathcal{E}[P\ Q]$ which are then reduced to some $\mathcal{E}[\mathbf{return}\ b]$, via a reduction sequence which, modulo $\mathcal{E}[-]$, has the following form:

$$\begin{aligned} P\,Q &\rightsquigarrow^* \mathcal{E}_0[Q\ k_0] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\, b_0] \rightsquigarrow^* \cdots \rightsquigarrow^* \mathcal{E}_{n-1}[Q\ k_{n-1}] \\ &\rightsquigarrow^* \mathcal{E}_{n-1}[\mathbf{return}\, b_{n-1}] \rightsquigarrow^* \mathbf{return}\ b \end{aligned}$$

(For notational clarity, we suppress mention of the location and store components here.) Informally we think of this as a dialogue in which control passes back and forth between P and Q. We shall refer to the portions $\mathcal{E}_j[Q\ k_j] \rightsquigarrow^* \mathcal{E}_j[\mathbf{return}\ b_j]$ of the above reduction as *Q-sections*, and to the remaining portions (including the first and the last) as *P-sections*. We refer to the totality of these P-sections and Q-sections as the *thread* arising from the given occurrence of the application $P\,Q$. An important point to note is that since Q may contain other occurrences of P, it is quite possible for the Q-sections above to contain further threads corresponding to other applications $P\ Q'$.

Since P is n-standard, we know that each thread will consist of $n+1$ P-sections separated by n Q-sections. Indeed, it is clear that this computation traces the path $b_0 \ldots b_{n-1}$ through the decision tree for P, with $k_0, \ldots, k_{n-1}$ the corresponding internal node labels. We may now, 'with hindsight', construe this as a semantic point $\pi : \mathbb{N}_n \to \mathbb{B}$ (where $\pi(k_j) = b_j$ for each j), and call it the semantic point *associated with* (the thread arising from) the application occurrence $P\ p$.

The following lemma now serves as a surrogate for Lemma 7.12:

Lemma 7.17. *Let P be an n-standard predicate. For any semantic point $\pi \in \mathbb{B}^n$, the evaluation of $K\ P$ involves an application occurrence $P\ Q$ with which π is associated.*

	Queens					
	First solution			**All solutions**		
Parameter	20	24	28	8	10	12
Naïve	–	–	–	217.74	–	–
Berger	11.24	15.70	–	2.06	2.86	3.64
Pruned	2.13	2.54	2.91	1.04	1.24	1.39
Bespoke	0.12	0.12	0.12	0.13	0.13	0.12

Table 7.1: SML/NJ: n-Queens benchmark runtime relative to effectful implementation.

	Integration								
	Id	**Squaring**			**Logistic**				
Parameter	20	14	17	20	1	2	3	4	5
Naïve	12.89	45.04	57.80	69.86	–	–	–	–	–
Berger	5.18	20.62	22.37	23.46	22.51	28.97	30.14	29.30	27.94
Pruned	2.07	3.78	4.05	4.24	4.10	5.44	6.42	7.26	7.94

Table 7.2: SML/NJ: integration benchmark runtime relative to effectful implementation.

The proof of this lemma is not too different from that of Lemma 7.12: if π were a point with no associated thread, there would be an unvisited leaf in the decision tree, and we could manufacture an n-standard predicate P' whose tree differed from that of P only at this leaf. We can then show, by induction on length of reductions, that any portion of the evaluation of $K\ P$ can be suitably mimicked with P replaced by P'. Naturally, this idea now needs to be formulated at the level of *configurations* rather than plain terms: in the course of reducing $(K\ P, 0, [])$, we may encounter configurations (M, l, s) in which residual occurrences of P have found their way into s as well as M, so in order to replace P by P' we must abstract on all these occurrences via an evident notion of *configuration context*. With this adjustment, however, the argument of Lemma 7.12 goes through.

A further argument is then needed to show that any two threads are indeed 'disjoint' as regards their P-sections, so that there must be at least $n2^n$ steps in the overall reduction sequence.

	Queens					
	First solution			**All solutions**		
Parameter	20	24	28	8	10	12
Naïve	–	–	–	17.31	–	–
Berger	0.52	0.66	–	0.19	0.22	0.20
Pruned	0.11	0.11	0.13	0.10	0.10	0.08
Bespoke	0.005	0.004	0.004	0.01	0.009	0.006

Table 7.3: MLton: *n*-Queens benchmark runtime relative to effectful implementation.

	Integration								
	Id	**Squaring**			**Logistic**				
Parameter	20	14	17	20	1	2	3	4	5
Naïve	1.45	4.51	5.13	5.82	–	–	–	–	–
Berger	0.43	2.02	1.95	1.92	2.17	3.59	4.24	4.34	4.28
Pruned	0.14	0.39	0.35	0.35	0.39	0.63	0.86	1.03	1.21

Table 7.4: MLton: integration benchmark runtime relative to effectful implementation.

	Queens					
	First solution			**All solutions**		
Parameter	20	24	28	8	10	12
Naïve	–	–	–	0.49	–	–
Berger	0.62	0.64	–	0.73	0.65	0.68
Pruned	0.70	0.68	0.71	0.74	0.70	0.71
Effectful	12.87	13.99	14.90	8.00	8.60	12.19
Bespoke	0.56	0.56	0.56	0.69	0.63	0.59

Table 7.5: MLton: *n*-Queens benchmark runtime relative to SML/NJ.

	Integration								
	Id	**Squaring**			**Logistic**				
Parameter	20	14	17	20	1	2	3	4	5
Naïve	0.55	0.35	0.35	0.35	–	–	–	–	–
Berger	0.41	0.35	0.34	0.34	0.37	0.37	0.37	0.37	0.37
Pruned	0.34	0.36	0.35	0.35	0.36	0.35	0.35	0.35	0.36
Effectful	4.93	3.53	3.95	4.20	3.80	3.00	2.62	2.46	2.37

Table 7.6: MLton: integration benchmarks runtime relative to SML/NJ.

7.6 Experiments

The theoretical efficiency gap between realisations of $\lambda_{\mathsf{b}}^{\rightarrow}$ and $\lambda_{\mathsf{h}}^{\rightarrow}$ manifests in practice. We observe it empirically on instantiations of n-queens and exact real number integration, which can be cast as generic search. Tables 7.1 and 7.2 show the speedup of using an effectful implementation of generic search over various pure implementations of the n-Queens and integration benchmarks, respectively. We discuss the benchmarks and results in further detail below.

Methodology We evaluated an effectful implementation of generic search against three "pure" implementations which are realisable in $\lambda_{\mathsf{b}}^{\rightarrow}$ extended with mutable state:

- Naïve: a simple, and rather naïve, functional implementation;
- Pruned: a generic search procedure with space pruning based on Longley's technique [175] (uses local state);
- Berger: a lazy pure functional generic search procedure based on Berger's algorithm.

Each benchmark was run 11 times. The reported figure is the median runtime ratio between the particular implementation and the baseline effectful implementation. Benchmarks that failed to terminate within a threshold (1 minute for single solution, 8 minutes for enumerations), are reported as $-$. The experiments were conducted in SML/NJ [9] v110.97 64-bit with factory settings on an Intel Xeon CPU E5-1620 v2 @ 3.70GHz powered workstation running Ubuntu 16.04. The effectful implementation uses an encoding of delimited control akin to effect handlers based on top of SML/NJ's

call/cc. The complete source code for the benchmarks and instructions on how to run them are available at:

```
https://dl.acm.org/do/10.1145/3410231/abs/
```

Queens We phrase the n-queens problem as a generic search problem. As a control we include a bespoke implementation hand-optimised for the problem. We perform two experiments: finding the first solution for $n \in \{20, 24, 28\}$ and enumerating all solutions for $n \in \{8, 10, 12\}$. The speedup over the naïve implementation is dramatic, but less so over the Berger procedure. The pruned procedure is more competitive, but still slower than the baseline. Unsurprisingly, the baseline is slower than the bespoke implementation.

Exact real integration The integration benchmarks are adapted from Simpson [249]. We integrate three different functions with varying precision in the interval $[0, 1]$. For the identity function (Id) at precision 20 the speedup relative to Berger is $5.18\times$. For the squaring function the speedups are larger at higher precisions: at precision 14 the speedup is $3.78\times$ over the pruned integrator, whilst it is $4.24\times$ at precision 20. The speedups are more extreme against the naïve and Berger integrators. We also integrate the logistic map $x \mapsto 1 - 2x^2$ at a fixed precision of 15. We make the function harder to compute by iterating it up to 5 times. Between the pruned and effectful integrator the speedup ratio increases as the function becomes harder to compute.

MLton SML/NJ is compiled into CPS, thus providing a particularly efficient implementation of call/cc. MLton [96], a whole program compiler for SML, implements call/cc by copying the stack. We repeated our experiments using MLton 20180207. Tables 7.3 and 7.4 show the results. The effectful implementation performs much worse under MLton than SML/NJ, being surpassed in nearly every case by the pruned search procedure and in some cases by the Berger search procedure. Tables 7.5 and 7.6 summarise the runtime of MLton relative to SML/NJ for both benchmarks. Berger, Pruned, and Bespoke run between 1 and 3 times as fast with MLton compared to SML/NJ. However, the effectful implementation runs between 2 and 14 times as fast with SML/NJ compared with MLton.

7.7 Related work

There are relatively little work in the present literature on expressivity that has focused on complexity difference. Pippenger [216] gives an example of an online operation on infinite sequences of atomic symbols (essentially a function from streams to streams) such that the first n output symbols can be produced within time $O(n)$ if one is working in an effectful version of Lisp (one which allows mutation of cons pairs) but with a worst-case runtime no better than $\Omega(n \log n)$ for any implementation in pure Lisp (without such mutation). This example was reconsidered by Bird et al. [29] who showed that the same speedup can be achieved in a pure language by using lazy evaluation. Jones [136] explores the approach of manifesting expressivity and efficiency differences between certain languages by artificially restricting attention to 'cons-free' programs; in this setting, the classes of representable first-order functions for the various languages are found to coincide with some well-known complexity classes.

The vast majority of work in this area has focused on computability differences. One of the best known examples is the *parallel if* operation which is computable in a language with parallel evaluation but not in a typical sequential programming language [219]. It is also well known that the presence of control features or local state enables observational distinctions that cannot be made in a purely functional setting: for instance, there are programs involving call/cc that detect the order in which a (call-by-name) '$+$' operation evaluates its arguments [43]. Such operations are 'non-functional' in the sense that their output is not determined solely by the extension of their input (seen as a mathematical function $\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$); however, there are also programs with 'functional' behaviour that can be implemented with control or local state but not without them [175]. More recent results have exhibited differences lower down in the language expressivity spectrum: for instance, in a purely functional setting *à la* Haskell, the expressive power of *recursion* increases strictly with its type level [178], and there are natural operations computable by low-order recursion but not by high-order iteration [179]. Much of this territory, including the mathematical theory of some of the natural notions of higher-order computability that arise in this way, is mapped out by Longley and Normann [180].

Part IV

Conclusions

Chapter 8

Conclusions and future work

I will begin this chapter with a brief summary of this dissertation. The following sections each elaborates and spells out directions for future work.

In Part I I have presented the design of a ML-like programming language equipped an effect-and-type system and a structural notion of effectful operations and effect handlers. In this language I have demonstrated how to implement the essence of a UNIX-like operating system by making, almost, zealous use of deep, shallow, and parameterised effect handlers.

In Part II I have devised two canonical implementation strategies for the language, one based an transformation into continuation passing style, and another based on abstract machine semantics. Both strategies make key use of the notion of generalised continuations, which provide a high-level model of segmented runtime stacks.

In Part III I have explored how effect handlers fit into the wider landscape of programming abstractions. I have shown that deep, shallow, and parameterised effect handlers are macro-expressible. Furthermore, I shown that effect handlers endow its host language with additional computational power that provides an asymptotic improvement in runtime performance for some class of programs.

8.1 Programming with effect handlers

Chapter 2 presents a case study of effect handler oriented programming, which reproduces the essence of the UNIX operating system by making crucial use of effect handlers. The case study demonstrates how effect handlers provide a high-degree of modularity and flexibility that enable substantial behavioural changes to be retrofitted onto programs without altering the existing the code. Thus effect handlers provide a

mechanism for building small task-oriented programs that later can be scaled to interact with other programs in a larger context. The case study also demonstrates how one might ascribe a handler semantics to a UNIX-like operating system. The resulting operating system Tiny UNIX captures the essential features of a true operating system including support for managing multiple concurrent user environments simultaneously, process parallelism, file I/O. The case study also shows how each feature can be implemented in terms of some standard effect.

Chapter 3 presents the design of a core calculus that forms the basis for Links, which is a practical programming language with deep, shallow, and parameterised effect handlers. A distinguishing feature of the core calculus is that it is based on a structural notion of data and effects, whereas other literature predominantly consider nominal data and effects. In the setting of structural effects the effect system play a pivotal role in ensuring that the standard safety and soundness properties of statically typed programming languages hold as the effect system is used to track type and presence information about effectful operations. In a nominal setting an effect system is not necessary to ensure soundness (e.g. Section 7.1.2 presents a sound core calculus with nominal effects, but without an effect system). Irrespective of nominal or structural notions of effects, an effect system is a valuable asset when programming with effect handlers as an effect system enables modular reasoning about the composition of functions. The effect system provides crucial information about the introduction and elimination of effects. In the absence of an effect system programmers are essentially required to reason globally their programs as for instance the composition of any two functions may introduce arbitrary effects that need to be handled accordingly. Alternatively, a composition of any two functions may inadvertently eliminate arbitrary effects, and as such, programming with effect handlers without an effect system is prone to error. The UNIX case study in Chapter 2 demonstrates how the effect system assists to ensure that effectful function compositions are meaningful.

The particular effect system that I have used throughout this dissertation is based on Rémy-style row polymorphism formalism [236]. Whilst Rémy-style row polymorphism provides a suitable basis for structural records and variants, its suitability as a basis for practical effect systems is questionable. From a practical point of view the problem with this form of row polymorphism is that it leads to verbose type-and-effect signature due to the presence and absence annotations. In many cases annotations are redundant, e.g. in second-order functions like map for lists, where the effect signature of the function is the same as the signature of its functional argument. From a theoretical point of

view this verbosity is not a concern. However, in practice verbosity may lead to 'an overload of unequivocal information' by which I mean the programmer is presented with too many trivial facts about the program. Too much information can hinder both readability and writability of programs. For instance, in most mainstream programming languages with System F-style type polymorphism programmers normally do not have to annotate type variables with kinds, unless they happen to be doing something special. Similarly, programmers do not have to write type variable quantifiers, unless they do not appear in prenex position. In practice some defaults are implicitly understood and it is only when programmers deviate from those defaults that programmers ought to supply the compiler with explicit information. In Section 3.2.5 introduces some ad-hoc syntactic sugar for effect signature that tames the verbosity of an effect system based on Rémy-style row polymorphism to the degree that second-order functions like map do not duplicate information. Rather than back-patching the effect system in hindsight, a possibly better approach is to design the effect system for practical programming from the ground up as Lindley et al. [174] did for the Frank programming language. Nevertheless, the UNIX case study is indicative of the syntactic sugar being adequate in practice to build larger effect-oriented applications.

8.1.1 Future work

Operating systems via effect handlers In the UNIX case study we explored the paradigmatic reading of *effect handlers as composable operating systems* in practice by composing a UNIX-like operating system out of several effects and handlers. Obviously, the resulting system Tiny UNIX has been implemented in the combined core calculus consisting of λ_{h}, $\lambda_{\mathsf{h}^\dagger}$, and $\lambda_{\mathsf{h}^\ddagger}$ calculi. There also exists an actual runnable implementation of it in Links. It would be interesting to implement the system in other programming languages with support for effect handlers as at the time of writing most languages with effect handlers have some unique trait, e.g. lexical handlers, special effect system, etc. Ultimately, re-implementing the case study can help collect more data points about programming with effect handlers, which can potentially serve to inform the design of future effect handler-oriented languages.

I have made no attempts at formally proving the correctness of Tiny UNIX with respect to some specification. Although, I have consciously opted to implement Tiny UNIX using standard effects with well-known equations. Furthermore, the effect handlers have been implemented such that they ought to respect the equations of their effects.

Thus, perhaps it is possible to devise an equational specification for the operating system and prove the implementation correct with respect to that specification.

One important feature that is arguably missing from Tiny UNIX is external signal handling. Effect handlers as signal handlers is not a new idea. In a previous paper we have outlined an idea for using effect handlers to handle POSIX signals [72]. Signal handling is a delicate matter as signals introduce a form of preemption, thus some care needs to be taken to ensure that the interpretation of a signal does not interrupted by another signal instance. The essence of the idea is to have a *mask* primitive, which is a form of critical section for signals that permits some block of code to suppress signal interruptions. A potential starting point would be to combine Ahman and Pretnar's calculus of asynchronous effects with λ_{h} to explore this idea more formally [6].

Another interesting thought is to implement an actual operating system using effect handlers. Although, it might be a daunting task, the idea is maybe not so far fetched. With the advent of effect handlers in OCaml [252] it may be possible for MirageOS project [188], which is a unikernel based operating system written in OCaml, to take advantage of effect handlers to implement features such as concurrency.

Effect-based optimisations In this dissertation I have not considered any effect-based optimisations. However if effect handler oriented programming is to succeed in practice, then runtime performance will matter. Optimisation of program structure is one way to improve runtime performance. At our disposal we have the effect system and the algebraic structure of effects and handlers. Taking advantage of the information provided by the effect system to optimise programs is an old idea that has been explored previously in the literature [141, 142, 242]. Other work has attempted to exploit the algebraic structure of (deep) effect handlers to fuse nested handlers [275]. An obvious idea is to apply these lines of work to the handler calculi of Chapter 3. Moreover, I hypothesise there is untapped potential in the combination of effect-dependent analysis with respect to equational theories to optimise effectful programs. A potential starting point for testing out this hypothesis is to take Lukšič and Pretnar's a core calculus where effects are equipped with equations [185] and combine it with techniques for effect-dependent optimisations [142].

Multi handlers In this dissertation I have solely focused on so-called *unary* handlers, which handle a *single* effectful computation. A natural generalisation is *n-ary* handlers, which allow n effectful computations to be handled simultaneously. In the literature

n-ary handlers are called *multi handlers*, and unary handlers are simply called handlers. The ability to handle two or more computations simultaneously make for a straightforward way to implement synchronisation between two or more computations. For example, the pipes example of Section 2.6 can be expressed using a single handler rather than two dual handlers [174]. Shallow multi handlers are an ample feature of the Frank programming language [174]. The design space of deep and parameterised notions of multi handlers have yet to be explored as well as their applications domains. Thus an interesting future direction of research would be to extend λ_{h} with multi handlers and explore their practical programming applicability. Retrofitting the effect system of λ_{h} to provide a good programmer experience for programming with multi handlers pose an interesting design challenge as any quirks that occur with unary handlers only get amplified in the setting of multi handlers.

Handling linear resources The implementation of effect handlers in Links makes the language unsound, because the naïve combination of effect handlers and session typing is unsound. The combined power of being able to discard some resumptions and resume others multiple times can make for bad interactions with sessions. For instance, suppose some channel supplies only one value, then it is possible to break session fidelity by twice resuming some resumption that closes over a receive operation. Similarly, it is possible to break type safety by using a combination of exceptions and multi-shot resumptions, e.g. suppose some channel first expects an integer followed by a boolean, then the running the program $\mathbf{do}\ \mathsf{Fork}\ \langle\rangle; \mathbf{send}\ 42; \mathbf{absurd}\ \mathbf{do}\ \mathsf{Fail}\ \langle\rangle$ under the composition of the nondeterminism handler and default failure handler from Chapter 2 will cause the primitive **send** operation to supply two integers in succession, thus breaking the session protocol. Figuring out how to safely combine linear resources, such as channels, and handlers with multi-shot resumptions is an interesting unsolved problem.

8.2 Canonical implementation strategies for handlers

Chapter 4 carries out a comprehensive study of CPS translations for deep, shallow, and parameterised notions of effect handlers. We arrive at a higher-order CPS translation through step-wise refinement of an initial standard first-order fine-grain call-by-value CPS translation, which we extended to support deep effect handlers. Firstly, we refined the first-order translation by uncurrying it in order to yield a properly tail-recursive translation. Secondly, we adapted it to a higher-order one-pass translation that statically

eliminates administrative redexes. Thirdly, we solidified the structure of continuations to arrive at the notion of *generalised continuation*, which provides the basis for implementing shallow and parameterised handlers. The CPS translations have been proven correct with respect to the contextual small-step semantics of λ_{h}, $\lambda_{\mathsf{h}^\dagger}$, and $\lambda_{\mathsf{h}^\ddagger}$.

Generalised continuations are a succinct syntactic framework for modelling low-level stack manipulations. The structure of generalised continuations closely mimics the structure of Hieb et al. and Bruggeman et al.'s segmented stacks [41, 117], which is a state-of-art technique for implementing first-class control [95]. Each generalised continuation frame consists of a pure continuation and a handler definition. The pure continuation represents an execution stack delimited by some handler. Thus chaining together generalised continuation frames yields a sequence of segmented stacks.

The versatility of generalised continuations is illustrated in Chapter 5, where we plugged the notion of generalised continuation into Felleisen and Friedman's CEK machine to obtain an adequate execution runtime with simultaneous support for deep, shallow, and parameterised effect handlers [83]. The resulting abstract machine is proven correct with respect to the reduction semantics of the combined calculus of λ_{h}, $\lambda_{\mathsf{h}^\dagger}$, and $\lambda_{\mathsf{h}^\ddagger}$. The abstract machine provides a blueprint for both high-level interpreter-based implementations of effect handlers as well as low-level implementations based on stack manipulations. The server-side implementation of effect handlers in the Links programming language is a testimony to the former [119], whilst the Multicore OCaml implementation of effect handlers is a testimony to the latter [252].

8.2.1 Future work

Functional correspondence The CPS translations and abstract machine have been developed separately. Even though, the abstract machine is presented as an application of generalised continuations in Chapter 5 it did appear before the CPS translations. The idea of generalised continuation first solidified during the design of higher-order CPS translation for shallow handlers [120], where we adapted the continuation structure of our initial abstract machine design [119]. Thus it seems that there ought to be a formal functional correspondence between higher-order CPS translation and the abstract machine, however, the existence of such a correspondence has yet to be established.

Abstracting continuations It is evident from the step-wise refinement of the CPS translations in Chapter 4 that each translation has a certain structure to it. In fact, this is

how the CPS translation for effect handlers in Links has been implemented. Concretely, the translation is implemented as a functor, which is parameterised by a continuation interface. The continuation interface has monoidal operation for continuation extension and an application operation for applying the continuation to a value argument. Theoretically, it would be interesting to pin down and understand the precise algebraic nature of this nature would be interesting with respect to abstracting the notion of continuations. Practically, it would keep the code base modular and pave the way for rapid compilation of new control structures. Ideally one would simply have to implement a standard CPS translation, which keeps the notion of continuation abstract such that any conforming continuation can be plugged in.

Generalising generalised continuations The incarnation of generalised continuations in this dissertation has been engineered for unary handlers. An obvious extension to investigate is support for multi handlers. With multi handlers, handler definitions enter a one-to-many relationship with pure continuations rather than an one-to-one relationship with unary handlers. Thus at minimum the structure of generalised continuation frames needs to be altered such that each handler definition is paired with a list of pure continuations, where each pure continuation represents a distinct computation running under the handler.

Ad-hoc generalised continuations The literature contains plenty of ad-hoc techniques for realising continuations. For instance, Pettyjohn et al.'s technique for implementing undelimited continuations via exception handlers and state [214], and James and Sabry's technique for implementing delimited control via generators and iterators [135]. Such techniques may be used to implement effect handlers in control hostile environments by simulating the structure of generalised continuations. By using these techniques to implement effect handlers we may be able to bring effect handler oriented programming to programming languages that do not offer programmers much control.

Typed CPS for effect handlers The image of each translation developed in Chapter 4 is untyped. Typing the translations may provide additional insight into the semantic content of the translations. Effect forwarding poses a challenge in typing the image. In order to encode forwarding we need to be able to parametrically specify what a default case does. The Appendix B of the paper by Hillerström et al. [121] outlines a possible typing for the CPS translation for deep handlers. The extension we propose

to our row type system is to allow a row type to be given a *shape* (something akin to Berthomieu and le Moniès de Sagazan's tagged types [20]), which constrains the form of the ordinary types it contains. A full formalisation of this idea remains to be done.

8.3 On the expressive power of effect handlers

In Chapter 6 we investigated the interdefinability of deep, shallow, and parameterised handlers through the lens of typed macro expressiveness. We establish that every kind of handler is interdefinable. Although, the handlers are interdefinable it may matter in practice which kind of handler is being employed. For example, the encoding of shallow handlers using deep handlers is rather inefficient. The encoding suffers from space leaks as demonstrated empirically in Appendix B.3 of Hillerström and Lindley [120]. Similarly, the runtime and memory performance of between native parameterised handlers and encoding parameterised handlers as ordinary deep handlers may be observable in practice as the latter introduce a new closure per operation invocation.

Chapter 7 explores the relative efficiency of a base language, $\lambda_{\mathsf{b}}^{\to}$, and its extension with effect handlers, $\lambda_{\mathsf{h}}^{\to}$, through the lens of type-respecting expressivity. Concretely, we used the example program of *generic count* to show that $\lambda_{\mathsf{h}}^{\to}$ admits realisations of this program whose asymptotic efficiency is better than any possible realisation in $\lambda_{\mathsf{b}}^{\to}$. Concretely, we established that the lower bound of generic count on n-standard predicates in $\lambda_{\mathsf{b}}^{\to}$ is $\Omega(n2^n)$, whilst the worst case upper bound in $\lambda_{\mathsf{h}}^{\to}$ is $O(2^n)$. Hence there is a strict efficiency gap between the two languages. We observed this efficiency gap in practice on several benchmarks. The lower runtime bound also applies to a language $\lambda_{\mathsf{s}}^{\to}$ which extends $\lambda_{\mathsf{b}}^{\to}$ with state. Although, I have not spelled out the details here, in Hillerström et al. [123] we have verified that the lower bound also applies to a language $\lambda_{\mathsf{e}}^{\to}$ with Benton and Kennedy-style *exceptions* and handlers [18]. The lower bound also applies to the combined language $\lambda_{\mathsf{se}}^{\to}$ with both state and exceptions — this seems to bring us close to the expressive power of real languages such as Standard ML, Java, and Python, strongly suggesting that the speedup we have discussed is unattainable in these languages.

The positive result for $\lambda_{\mathsf{h}}^{\to}$ extends to other control operators by appeal to existing results on interdefinability of handlers and other control operators [99, 217].

From a practical point of view one might be tempted to label the efficiency result as merely of theoretical interest, since an $\Omega(2^n)$ runtime is already infeasible. However, what has been presented is an example of a much more pervasive phenomenon, and the

generic count example serves merely as a convenient way to bring this phenomenon into sharp formal focus. For example, suppose that our programming task was not to count all solutions to P, but to find just one of them. It is informally clear that for many kinds of predicates this would in practice be a feasible task, and also that we could still gain our factor n speedup here by working in a language with first-class control. However, such an observation appears less amenable to a clean mathematical formulation, as the runtimes in question are highly sensitive to both the particular choice of predicate and the search order employed.

8.3.1 Future work

Efficiency of handler encodings Although, I do not give a formal proof for the efficiency of the shallow as deep encoding in Chapter 6 it seems intuitively clear that the encoding is rather inefficient. In fact in Appendix B.2 and B.3 of Hillerström and Lindley [120] we show empirically that the encoding is inefficient. An interesting question is whether there exists an efficient encoding of shallow handlers using deep handlers. Formally proving the absence of an efficient encoding would give a strong indication of the relative computational expressive power between shallow and deep handlers. Likewise discovering that an efficient encoding does exist would tell us that it may not matter computationally whether a language incorporates shallow or deep handlers.

Effect tracking breaks asymptotic improvement The result of Chapter 7 does not immediately carry over to a language with an effect system as the implementation of generic search in $\lambda_{\mathsf{h}}^{\rightarrow}$ would introduce an effectful operation, which requires a change of types. In order to state and prove the result in the presence of an effect system some other refined, possibly new, notion of expressivity seems necessary.

Asymptotic improvement with affine handlers The result of Chapter 7 does not immediately remain true in the presence of affine effect handlers (handlers which their resumptions at most once) as they make it possible to encode coroutines. The present proof method does not readily adapt to a situation with coroutines, because the proof depend at various points on an orderly nesting of subcomputations which corouting would break.

Efficiency hierarchy of control The definability hierarchy of various control constructs such as iteration, recursion, recursion with state, and first class control is fairly well-understood [178–180]. However, the relative asymptotic efficiency between them is less well-understood. It would be interesting to formally establish a hierarchy of relative asymptotic efficiency between various control constructs in the style of Chapter 7.

Part V

Appendices

Appendix A

Continuations

A continuation represents the control state of computation at a given point during evaluation. The control state contains the necessary operational information for evaluation to continue. As such, continuations drive computation. Continuations are one of those canonical ideas, that have been discovered multiple times and whose definition predates their use [239]. The term 'continuation' first appeared in the literature in 1974, when Strachey and Wadsworth [255] used continuations to give a denotational semantics to programming languages with unrestricted jumps [256].

The inaugural use of continuations came well before Strachey and Wadsworth's definition. About a decade earlier continuation passing style had already been conceived, if not in name then in spirit, as a compiler transformation for eliminating labels and goto statements [239]. In the mid 1960s Landin [161] introduced the J operator as a programmatic mechanism for manipulating continuations.

Landin's J operator is an instance of a first-class control operator, which is a mechanism that lets programmers reify continuations as first-class objects, that can be invoked, discarded, or stored for later use. There exists a wide variety of control operators, which expose continuations of varying extent and behaviour.

The purpose of this chapter is to examine control operators and their continuations in programming. Section A.1 examines different notions of continuations by characterising their extent and behaviour operationally. Section A.2 contains a detailed overview of various control operators that appear in programming languages and in the literature. Section A.3 summarises some applications of continuations, whilst Section A.4 contains a brief summary of ideas for constraining the power of continuations. Lastly, Section A.5 outlines some implementation strategies for continuations.

A.1 Classifying continuations

The term ‘continuation’ is really an umbrella term that covers several distinct notions of continuations. It is common in the literature to find the word ‘continuation’ accompanied by a qualifier such as full, partial, abortive, escape, undelimited, delimited, composable, or functional (in Chapter 4 I will extend this list by three new ones). Some of these notions of continuations are synonymous, whereas others have distinct meanings. Common to all notions of continuations is that they represent the control state. However, the extent and behaviour of continuations differ widely from notion to notion. The essential notions of continuations are undelimited/delimited and abortive/composable. To tell them apart, we will classify them according to their operational behaviour.

The extent and behaviour of a continuation in programming are determined by its introduction and elimination forms, respectively. Programmatically, a continuation is introduced via a control operator, which reifies the control state as a first-class object, e.g. a function, that can be eliminated via some form of application.

A.1.1 Introduction of continuations

The extent of a continuation determines how much of the control state is contained with the continuation. The extent can be either undelimited or delimited, and it is determined at the point of capture by the control operator.

We need some notation for control operators in order to examine the introduction of continuations operationally. We will use the syntax $\mathbf{ctrl}\ k.M$ to denote a control operator, or control reifier, which that reifies the control state and binds it to k in the computation M. Here the control state will simply be an evaluation context. We will denote continuations by a special value $\mathbf{cont}_{\mathcal{E}}$, which is indexed by the reified evaluation context $\mathcal{E}$ to make it notationally convenient to reflect the context again. To characterise delimited continuations we also need a control delimiter. We will write $\mathbf{del}.M$ to denote a syntactic marker that delimits some computation M.

Undelimited continuation The extent of an undelimited continuation is indefinite as it ranges over the entire remainder of computation. In functional programming languages undelimted control operators most commonly expose the *current* continuation, which is the precisely continuation following the control operator. The following is the characteristic reduction for the introduction of the current continuation.

$$\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow \mathcal{E}[M[\mathbf{cont}_{\mathcal{E}}/k]]$$

The evaluation context $\mathcal{E}$ is the continuation of **ctrl**. The evaluation context on the left hand side gets reified as a continuation object, which is accessible inside of M via k. On the right hand side the entire context remains in place after reification. Thus, the current continuation is evaluated regardless of whether the continuation object is invoked. This is an instance of non-abortive undelimited control. Alternatively, the control operator can abort the current continuation before proceeding as M, i.e.

$$\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow M[\mathbf{cont}_{\mathcal{E}}/k]$$

This is the characteristic reduction rule for abortive undelimited control. The rule is nearly the same as the previous, except that on the right hand side the evaluation context $\mathcal{E}$ is discarded after reification. Now, the programmer has control over the whole continuation, since it is entirely up to the programmer whether $\mathcal{E}$ gets evaluated.

Imperative statement-oriented programming languages commonly expose the *caller* continuation, typically via a return statement. The caller continuation is the continuation of the invocation context of the control operator. Characterising undelimited caller continuations is slightly more involved as we have to remember the continuation of the invocation context. We will use a bold lambda $\boldsymbol{\lambda}$ as a syntactic runtime marker to remember the continuation of an application. In addition we need three reduction rules, where the first is purely administrative, the second is an extension of regular application, and the third is the characteristic reduction rule for undelimited control with caller continuations.

$$\begin{aligned}
\boldsymbol{\lambda}.V &\rightsquigarrow V \\
(\lambda x.N)\,V &\rightsquigarrow \boldsymbol{\lambda}.N[V/x] \\
\mathcal{E}[\boldsymbol{\lambda}.\mathcal{E}'[\mathbf{ctrl}\ k.M]] &\rightsquigarrow \mathcal{E}[\boldsymbol{\lambda}.\mathcal{E}'[M[\mathbf{cont}_{\mathcal{E}}/k]]], \quad \text{where } \mathcal{E}' \text{ contains no } \boldsymbol{\lambda}
\end{aligned}$$

The first rule accounts for the case where $\boldsymbol{\lambda}$ marks a value, in which case the marker is eliminated. The second rule marks inserts a marker after an application such that this position can be recalled later. The third rule is the interesting rule. Here an occurrence of **ctrl** reifies $\mathcal{E}$, the continuation of some application, rather than its current continuation $\mathcal{E}'$. The side condition ensures that **ctrl** reifies the continuation of the inner most application. This rule characterises a non-abortive control operator as both contexts, $\mathcal{E}$ and $\mathcal{E}'$, are left in place after reification. It is straightforward to adapt this rule to an abortive operator. Although, there is no abortive undelimited control operator that captures the caller continuation in the literature.

It is worth noting that the two first rules can be understood locally, that is without mentioning the enclosing context, whereas the third rule must be understood globally.

In the literature an undelimited continuation is also known as a 'full' continuation.

Delimited continuation A delimited continuation is in some sense a refinement of a undelimited continuation as its extent is definite. A delimited continuation ranges over some designated part of computation. A delimited continuation is introduced by a pair operators: a control delimiter and a control reifier. The control delimiter acts as a barrier, which prevents the reifier from reaching beyond it, e.g.

$$\mathbf{del}.V \rightsquigarrow V$$
$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow \mathbf{del}.\mathcal{E}[M[\mathbf{cont}_{\mathcal{E}}/k]]$$

The first rule applies whenever the control delimiter delimits a value, in which case the delimiter is eliminated. The second rule is the characteristic reduction rule for a non-abortive delimited control reifier. It reifies the context $\mathcal{E}$ up to the control delimiter, and then continues as M under the control delimiter. Note that the continuation of **del** is invisible to **ctrl**, and thus, the behaviour of **ctrl** can be understood locally. Most commonly, the control reifier is abortive, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow \mathbf{del}.M[\mathbf{cont}_{\mathcal{E}}/k].$$

The design space of delimited control is somewhat richer than that of undelimited control, as the control delimiter may remain in place after reification, as above, be discarded, be included in the continuation, or a combination. Similarly, the control reifier may reify the continuation up to and including the delimiter or, as above, without the delimiter. Dybvig et al. [75] use a taxonomy for delimited abortive control reifiers, which classifies them according to how they interact with their respective control delimiter. They identify four variations.

$+$**ctrl**$+$ The control reifier includes a copy of the control delimiter in the reified context, and leaves the original in place, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow \mathbf{del}.M[\mathbf{cont}_{\mathbf{del}.\mathcal{E}}/k]$$

$+$**ctrl**$-$ The control delimiter remains in place after reification as the control reifier reifies the context up to, but not including, the delimiter, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow \mathbf{del}.M[\mathbf{cont}_{\mathcal{E}}/k]$$

$-$**ctrl**$+$ The control reifier includes a copy of the control delimiter in the reified context, but discards the original instance, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow M[\mathbf{cont}_{\mathbf{del}.\mathcal{E}}/k]$$

−**ctrl**− The control reifier reifies the context up to, but not including, the delimiter and subsequently discards the delimiter, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{ctrl}\ k.M] \rightsquigarrow M[\mathbf{cont}_{\mathcal{E}}/k]$$

In the literature a delimited continuation is also known as a 'partial' continuation.

A.1.2 Elimination of continuations

The purpose of continuation application is to reinstall the captured context. However, a continuation application may affect the control state in various ways. The literature features two distinct behaviours of continuation application: abortive and composable. We need some notation for application of continuations in order to characterise abortive and composable behaviours. We will write $\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ V$ to denote the application of some continuation object **cont** to some value V.

Abortive continuation Upon invocation an abortive continuation discards the entire evaluation context before reinstalling the captured context. In other words, an abortive continuation replaces the current context with its captured context, i.e.

$$\mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}'}\ V] \rightsquigarrow \mathcal{E}'[V]$$

The current context $\mathcal{E}$ is discarded in favour of the captured context $\mathcal{E}'$ (whether the two contexts coincide depends on the control operator). Abortive continuations are a global phenomenon due to their effect on the current context. However, in conjunction with a control delimiter the behaviour of an abortive continuation can be localised, i.e.

$$\mathbf{del}.\mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}'}\ V] \rightsquigarrow \mathcal{E}'[V]$$

Here, the behaviour of continuation does not interfere with the context of **del**, and thus, the behaviour can be understood and reasoned about locally with respect to **del**.

A key characteristic of an abortive continuation is that composition is meaningless. For example, composing an abortive continuation with itself have no effect.

$$\mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}'}\ (\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}'}\ V)] \rightsquigarrow \mathcal{E}'[V]$$

The innermost application erases the outermost application term, consequently only the first application of **cont** occurs during runtime. It is as if the first application occurred in tail position.

The continuations introduced by the early control operators were all abortive, since they were motivated by modelling unrestricted jumps akin to **goto** in statement-oriented programming languages.

An abortive continuation is also known as an 'escape' continuation in the literature.

Composable continuation A composable continuation splices its captured context with the its invocation context, i.e.

$$\textbf{resume cont}_{\mathcal{E}}\ V \rightsquigarrow \mathcal{E}[V]$$

The application of a composable continuation can be understood locally, because it has no effect on its invocation context. A composable continuation behaves like a function in the sense that it returns to its caller, and thus composition is well-defined, e.g.

$$\textbf{resume cont}_{\mathcal{E}}\ (\textbf{resume cont}_{\mathcal{E}}\ V) \rightsquigarrow \textbf{resume cont}_{\mathcal{E}}\ \mathcal{E}[V]$$

The innermost application composes the captured context with the outermost application. Thus, the outermost application occurs when $\mathcal{E}[V]$ has been reduced to a value.

In the literature, virtually every delimited control operator provides composable continuations. However, the notion of composable continuation is not intimately connected to delimited control. It is perfect possible to conceive of a undelimited composable continuation, just as a delimited abortive continuation is conceivable.

A composable continuation is also known as a 'functional' continuation in the literature.

A.2 Controlling continuations

As suggested in the previous section, the design space for continuation is rich. This richness is to an extent reflected by the large amount of control operators that appear in the literature and in practice. The purpose of this section is to survey a considerable subset of the first-class *sequential* control operators that occur in the literature and in practice. Control operators for parallel programming will not be considered here. Tables A.1 and A.2 provide classifications of some of the undelimited control operators and delimited control operators, respectively, that appear in the literature.

Note that a *first-class* control operator is typically not itself a first-class citizen, rather, the label 'first-class' means that the reified continuation is a first-class object. Control operators that reify the current continuation can be made first-class by enclosing

Name	Continuation behaviour	Canonical reference
J	Abortive	Landin [161]
escape	Abortive	Reynolds [240]
catch	Abortive	Sussman and Steele [257]
callcc	Abortive	Clinger et al. [50]
F	Composable	Felleisen et al. [85]
C	Abortive	Felleisen and Friedman [83]
callcomp	Composable	Flatt and PLT [93]

Table A.1: Classification of first-class undelimited control operators (listed in chronological order).

Name	Taxonomy	Continuation behaviour	Canonical reference
control/prompt	+**ctrl**−	Composable	Felleisen [81]
shift/reset	+**ctrl**+	Composable	Danvy and Filinski [62]
spawn	−**ctrl**+	Composable	Hieb and Dybvig [116]
splitter	−**ctrl**−	Abortive, composable	Queinnec and Serpette [234]
fcontrol	−**ctrl**−	Composable	Sitaram [250]
cupto	−**ctrl**−	Composable	Gunter et al. [111]
catchcont	−**ctrl**−	Composable	Longley [177]
effect handlers	−**ctrl**+	Composable	Plotkin and Pretnar [228]

Table A.2: Classification of first-class delimited control operators (listed in chronological order).

them in a λ-abstraction. Obviously, this trick does not work for operators that reify the caller continuation.

To study the control operators we will make use of a small base language.

A small calculus for control To look at control we will use a simply typed fine-grain call-by-value calculus. Although, we will sometimes have to discard the types, as many of the control operators were invented and studied in a untyped setting. The calculus is essentially the same as the one used in Chapter 7, except that here we will have an explicit invocation form for continuations. Although, in practice most systems disguise continuations as first-class functions, but for a theoretical examination it is convenient to treat them specially such that continuation invocation is a separate

Types	$A,B \in \text{Type} ::= 1 \mid A \to B \mid A \times B \mid \mathsf{Cont}\,\langle A;B\rangle \mid A+B$
Values	$V,W \in \text{Val} ::= \langle\rangle \mid \lambda x^A.M \mid \langle V;W\rangle \mid \mathbf{cont}_{\mathcal{E}} \mid \mathbf{inl}\ V \mid \mathbf{inr}\ W \mid x$
Computations	$M,N \in \text{Comp} ::= \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \mathbf{let}\ \langle x;y\rangle = V\ \mathbf{in}\ N$ $\mid\ V\,W \mid \mathbf{resume}\ V\ W$
Evaluation contexts	$\mathcal{E} \in \text{Ctx} ::= [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N$

Figure A.1: Types and term syntax

reduction rule from ordinary function application. Figure A.1 depicts the syntax of types and terms in the calculus. The types are the standard simple types with the addition of the continuation object type $\mathsf{Cont}\,\langle A;B\rangle$, which is parameterised by an argument type and a result type, respectively. The static semantics is standard as well, except for the continuation invocation primitive **resume**.

$$\frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : \mathsf{Cont}\,\langle A;B\rangle}{\Gamma \vdash \mathbf{resume}\ W\ V : B}$$

Although, it is convenient to treat continuation application specially for operational inspection, it is rather cumbersome to do so when studying encodings of control operators. Therefore, to obtain the best of both worlds, the control operators will reify their continuations as first-class functions, whose body is **resume**-expression. To save some ink, we will use the following notation.

$$\ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner \stackrel{\text{def}}{=} \lambda x.\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ x$$

We will permit ourselves various syntactic sugar to keep the examples relative concise, e.g. we write the examples in ordinary call-by-value.

A.2.1 Undelimited control operators

The early inventions of undelimited control operators were driven by the desire to provide a 'functional' equivalent of jumps as provided by the infamous goto in imperative programming.

In 1965 Peter Landin unveiled the *first* first-class control operator: the J operator [159–161]. Later in 1972 influenced by Landin's J operator John Reynolds designed the escape operator [240]. Influenced by escape, Sussman and Steele designed,

implemented, and standardised the catch operator in Scheme in 1975. A while thereafter the perhaps most famous undelimited control operator appeared: callcc. It was designed in 1982 and standardised in 1985 as a core feature of Scheme. Following on from callcc a wide range of different control operators was designed. A common characteristic of the early control operators is that their capture mechanisms are abortive and their captured continuations are abortive, save for one, namely, Felleisen's F operator. Though, it is worth remarking that Flatt et al. [94] devised a non-abortive and composable variant of callcc. Another common characteristic is that every operator, except for Landin's J operator, capture the current continuation.

Reynolds' escape The escape operator was introduced by Reynolds in 1972 [240] to make statement-oriented control mechanisms such as jumps and labels programmable in an expression-oriented language. The operator introduces a new computation form.

$$M, N \in \mathrm{Comp} ::= \cdots \mid \mathbf{escape}\ k\ \mathbf{in}\ M$$

The variable k is called the *escape variable* and it is bound in M. The escape variable exposes the current continuation of the **escape**-expression to the programmer. The captured continuation is abortive, thus an invocation of the escape variable in the body M has the effect of performing a non-local exit.

In terms of jumps and labels the **escape**-expression can be understood as corresponding to a kind of label and an application of the escape variable k can be understood as corresponding to a jump to the label.

Reynolds' original treatise of escape was untyped, and as such, the escape variable could escape its captor, e.g.

$$\mathbf{let}\ k \leftarrow (\mathbf{escape}\ k\ \mathbf{in}\ k)\ \mathbf{in}\ N$$

Here the current continuation, N, gets bound to k in the **escape**-expression, which returns k as-is, and thus becomes available for use within N. Reynolds recognised the power of this idiom and noted that it could be used to implement coroutines and backtracking [240]. Reynolds did not develop the static semantics for **escape**, however, it is worth noting that this idiom require recursive types to type check. Even in a language without recursive types, the continuation may propagate outside its binding **escape**-expression if the language provides an escape hatch such as mutable references.

An invocation of the continuation discards the invocation context and plugs the

argument into the captured context.

$$\begin{array}{ll} \textsf{Capture} & \mathcal{E}[\textbf{escape}\ k\ \textbf{in}\ M] \rightsquigarrow \mathcal{E}[M[\ulcorner \textbf{cont}_{\mathcal{E}} \urcorner / k]] \\ \textsf{Resume} & \mathcal{E}[\textbf{resume}\ \textbf{cont}_{\mathcal{E}'}\ V] \rightsquigarrow \mathcal{E}'[V] \end{array}$$

The Capture rule leaves the context intact such that if the body M does not invoke k then whatever value M reduces is plugged into the context. The Resume discards the current context $\mathcal{E}$ and installs the captured context $\mathcal{E}'$ with the argument V plugged in.

Sussman and Steele's catch In 1975 Sussman and Steele [257] designed and implemented the catch operator in Scheme. It is a more powerful variant of the catch operator in MacLisp [204]. The MacLisp catch operator had a companion throw operation, which would unwind the evaluation stack until it was caught by an instance of catch. Sussman and Steele's catch operator dispenses with the throw operation and instead provides the programmer with access to the current continuation. Their operator is identical to Reynolds' escape operator, save for the syntax.

$$M, N \in \text{Comp} ::= \cdots \mid \textbf{catch}\ k.M$$

Although, their syntax differ, their dynamic semantics are the same.

$$\begin{array}{ll} \textsf{Capture} & \mathcal{E}[\textbf{catch}\ k.M] \rightsquigarrow \mathcal{E}[M[\ulcorner \textbf{cont}_{\mathcal{E}} \urcorner / k]] \\ \textsf{Resume} & \mathcal{E}[\textbf{resume}\ \textbf{cont}_{\mathcal{E}'}\ V] \rightsquigarrow \mathcal{E}'[V] \end{array}$$

As an aside it is worth to mention that Cartwright and Felleisen [43] used a variation of **catch** to show that control operators enable programs to observe the order of evaluation.

Call-with-current-continuation In 1982 the Scheme implementors observed that they could dispense of the special syntax for **catch** in favour of a higher-order function that would apply its argument to the current continuation, and thus callcc was born (callcc is short for call-with-current-continuation) [50].

Unlike the previous operators, callcc augments the syntactic categories of values.

$$V, W \in \text{Val} ::= \cdots \mid \textbf{callcc}$$

The value **callcc** is essentially a hard-wired function name. Being a value means that the operator itself is a first-class entity which entails it can be passed to functions, returned from functions, and stored in data structures. Operationally, **callcc** captures the current continuation and aborts it before applying it on its argument.

$$\begin{array}{ll} \textsf{Capture} & \mathcal{E}[\textbf{callcc}\ V] \rightsquigarrow \mathcal{E}[V\ulcorner\textbf{cont}_{\mathcal{E}}\urcorner] \\ \textsf{Resume} & \mathcal{E}[\textbf{resume cont}_{\mathcal{E}'}\ V] \rightsquigarrow \mathcal{E}'[V] \end{array}$$

From the dynamic semantics it is evident that **callcc** is a syntax-free alternative to **catch** (although, it is treated as a special value form here; in actual implementation it suffices to recognise the object name of **callcc**). They are trivially macro-expressible.

$$[\![\textbf{catch}\ k.M]\!] \stackrel{\text{def}}{=} \textbf{callcc}\,(\lambda k.[\![M]\!])$$
$$[\![\textbf{callcc}]\!] \stackrel{\text{def}}{=} \lambda f.\textbf{catch}\ k.f\,k$$

Call-with-composable-continuation A variation of callcc is call-with-composable-continuation, abbreviated callcomp. As the name suggests the captured continuation is composable rather than abortive. It was introduced by Flatt et al. [94] in 2007, and implemented in November 2006 according to the history log of Racket (Racket was then known as MzScheme, version 360) [93]. The history log classifies it as a delimited control operator. Truth to be told nowadays in Racket virtually all control operators are delimited, even callcc, because they are parameterised by an optional prompt tag. If the programmer does not supply a prompt tag at invocation time then the optional parameter assume the actual value of the top-level prompt, effectively making the extent of the captured continuation undelimited. In other words its default mode of operation is undelimited, hence the justification for categorising it as such.

Like **callcc** this operator is a value.

$$V, W \in \text{Val} ::= \cdots \mid \textbf{callcomp}$$

Unlike **callcc**, captured continuations behave as functions.

$$\begin{array}{ll} \textsf{Capture} & \mathcal{E}[\textbf{callcomp}\ V] \rightsquigarrow \mathcal{E}[V\ulcorner\textbf{cont}_{\mathcal{E}}\urcorner] \\ \textsf{Resume} & \textbf{resume cont}_{\mathcal{E}}\ V \rightsquigarrow \mathcal{E}[V] \end{array}$$

The capture rule for **callcomp** is identical to the rule for **callcc**, but the resume rule is different. The effect of continuation invocation can be understood locally as it does not erase the global evaluation context, but rather composes with it. To make this more

tangible consider the following example reduction sequence.

$$
\begin{aligned}
& 1 + \mathbf{callcomp}\,(\lambda k.\mathbf{resume}\ k\ (\mathbf{resume}\ k\ 0)) \\
\rightsquigarrow^{+}\quad & (\mathsf{Capture}\ \mathcal{E} = 1 + [\,]) \\
& 1 + (\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ (\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ 0)) \\
\rightsquigarrow^{+}\quad & (\mathsf{Resume\ with}\ \mathcal{E}[0]) \\
& 1 + (\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ 1) \\
\rightsquigarrow^{+}\quad & (\mathsf{Resume\ with}\ \mathcal{E}[1]) \\
& 1 + 2 \rightsquigarrow 3
\end{aligned}
$$

The operator reifies the current evaluation context as a continuation object and passes it to the function argument. The evaluation context is left in place. As a result an invocation of the continuation object has the effect of duplicating the context. In this particular example the context has been duplicated twice to produce the result 3. Contrast this result with the result obtained by using **callcc**.

$$
\begin{aligned}
& 1 + \mathbf{callcc}\,(\lambda k.\mathbf{absurd}\ \mathbf{resume}\ k\ (\mathbf{absurd}\ \mathbf{resume}\ k\ 0)) \\
\rightsquigarrow^{+}\quad & (\mathsf{Capture}\ \mathcal{E} = 1 + [\,]) \\
& 1 + (\mathbf{absurd}\ \mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ (\mathbf{absurd}\ \mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ 0)) \\
\rightsquigarrow\quad & (\mathsf{Resume\ with}\ \mathcal{E}[0]) \\
& 1
\end{aligned}
$$

The second invocation of $\mathbf{cont}_{\mathcal{E}}$ never enters evaluation position, because the first invocation discards the entire evaluation context. Our particular choice of syntax and static semantics already makes it immediately obvious that **callcc** cannot be directly substituted for **callcomp**, and vice versa, in a way that preserves operational behaviour.

An interesting question is whether **callcc** and **callcomp** are interdefinable. Presently, the literature does not seem to answer to this question. I conjecture that the operators exhibit essential differences, meaning they cannot encode each other. The intuition behind this conjecture is that for any encoding of **callcomp** in terms of **callcc** must be able to preserve the current evaluation context, e.g. using a state cell akin to how Filinski [87] encodes composable continuations using abortive continuations and state. The other way around also appears to be impossible, because neither the base calculus nor **callcomp** has the ability to discard an evaluation context.

C and F The C operator is a variation of callcc that provides control over the whole continuation as it aborts the current continuation after capture, whereas callcc implicitly invokes the current continuation on the value of its argument. The C operator was

introduced by Felleisen et al. in two papers during 1986 [83, 84]. The following year, Felleisen et al. [85] introduced the F operator which is a variation of C, whose captured continuation is composable.

In our framework both operators are value forms.

$$V, W \in \mathrm{Val} ::= \cdots \mid \mathbf{C} \mid \mathbf{F}$$

The dynamic semantics of **C** and **F** are as follows.

$$\begin{array}{lrcl}
\textsf{C-Capture} & \mathcal{E}[\mathbf{C}\,V] & \rightsquigarrow & V\,\ulcorner\mathbf{cont}_{\mathcal{E}}\urcorner \\
\textsf{C-Resume} & \mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}'}\ V] & \rightsquigarrow & \mathcal{E}'[V] \\
\textsf{F-Capture} & \mathcal{E}[\mathbf{F}\,V] & \rightsquigarrow & V\,\ulcorner\mathbf{cont}_{\mathcal{E}}\urcorner \\
\textsf{F-Resume} & \mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ V & \rightsquigarrow & \mathcal{E}[V]
\end{array}$$

Their capture rules are identical. Both operators abort the current continuation upon capture. This is what sets **F** apart from the other composable control operator **callcomp**. The resume rules of **C** and **F** show the difference between the two operators. The **C** operator aborts the current continuation and reinstall the then-current continuation just like **callcc**, whereas the resumption of a continuation captured by **F** composes the current continuation with the then-current continuation.

Felleisen et al. [85] show that **F** can simulate **C**.

$$[\![\mathbf{C}]\!] \stackrel{\text{def}}{=} \lambda m.\mathbf{F}\,(\lambda k.m\,(\lambda v.\mathbf{F}\,(\lambda\_.k\ v)))$$

The first application of **F** has the effect of aborting the current continuation, whilst the second application of **F** aborts the invocation context.

Felleisen et al. [85] also postulate that **C** cannot express **F**.

Landin's J operator The J operator was introduced by Peter Landin in 1965 (making it the world's *first* first-class control operator) as a means for translating jumps and labels in the statement-oriented language Algol 60 into an expression-oriented language [159–161]. Landin used the J operator to account for the meaning of Algol 60 labels. The following example due to Danvy and Millikin [64] provides a flavour of the correspondence between labels and J.

$$\begin{array}{l}
\quad \mathcal{S}[\![\mathbf{begin}\ s_1;\ \mathbf{goto}\ L;\ L:\ s_2\ \mathbf{end}]\!] \\
= \lambda\langle\rangle.\mathbf{let}\ L \leftarrow \mathbf{J}\,\mathcal{S}[\![s_2]\!]\ \mathbf{in\ let}\ \langle\rangle \leftarrow \mathcal{S}[\![s_1]\!]\,\langle\rangle\ \mathbf{in\ resume}\ L\,\langle\rangle
\end{array}$$

Here $\mathcal{S}[\![-]\!]$ denotes the translation of statements. In the image, the label L manifests as an application of **J** and the **goto** manifests as an application of continuation captured by **J**. The operator extends the syntactic category of values with a new form.

$$V, W \in \mathrm{Val} ::= \cdots \mid \mathbf{J}$$

The previous example hints at the fact that the J operator is quite different to the previously considered undelimited control operators in that the captured continuation is *not* the current continuation, but rather, the continuation of statically enclosing λ-abstraction. In other words, **J** provides access to the continuation of its the caller. To this effect, the continuation object produced by an application of **J** may be thought of as a first-class variation of the return statement commonly found in statement-oriented languages. Since it is a first-class object it can be passed to another function, meaning that any function can endow other functions with the ability to return from it, e.g.

$$\mathsf{f} \stackrel{\text{def}}{=} \lambda g.\mathbf{let}\ \mathit{return} \leftarrow \mathbf{J}\,(\lambda x.x)\ \mathbf{in}\ g\ \mathit{return};\ \mathsf{true}$$

If the function g does not invoke its argument, then f returns true, e.g.

$$\mathsf{f}\ (\lambda \mathit{return}.\mathsf{false}) \rightsquigarrow^{+} \mathsf{true}$$

However, if g does apply its argument, then the value provided to the application becomes the return value of f, e.g.

$$\mathsf{f}\ (\lambda \mathit{return}.\mathbf{resume}\ \mathit{return}\ \mathsf{false}) \rightsquigarrow^{+} \mathsf{false}$$

The function argument gets post-composed with the continuation of the calling context. The particular application $\mathbf{J}\,(\lambda x.x)$ is so idiomatic that it has its own name: $\mathbf{J}\,\mathbf{I}$, where $\mathbf{I}$ is the identity function.

Any meaningful applications of **J** must appear under a λ-abstraction, because the application captures its caller's continuation. In order to capture the caller's continuation we annotate the evaluation contexts for ordinary applications.

Annotate	$\mathcal{E}[(\lambda x.M)\,V] \rightsquigarrow \mathcal{E}_\lambda[M[V/x]]$
Capture	$\mathcal{E}_\lambda[\mathcal{D}[\mathbf{J}\,W]] \rightsquigarrow \mathcal{E}_\lambda[\mathcal{D}[\ulcorner \mathbf{cont}_{\langle \mathcal{E}_\lambda; W\rangle} \urcorner]]$
Resume	$\mathcal{E}[\mathbf{resume}\ \mathbf{cont}_{\langle \mathcal{E}'; W\rangle}\ V] \rightsquigarrow \mathcal{E}'[W\,V]$

The Capture rule only applies if the application of **J** takes place inside an annotated evaluation context. The continuation object produced by a **J** application encompasses the caller's continuation $\mathcal{E}_\lambda$ and the value argument W. This continuation object may be

invoked in *any* context. An invocation discards the current continuation $\mathcal{E}$ and installs $\mathcal{E}'$ instead with the **J**-argument W applied to the value V.

Landin and Thielecke noticed that **J** can be recovered from the special form **J I** [261]. Taking **J I** to be a primitive, we can translate **J** to a language with **J I** as follows.

$$[\![\mathbf{J}]\!] \stackrel{\text{def}}{=} (\lambda k.\lambda f.\lambda x.\mathbf{resume}\, k\,(f\,x))\,(\mathbf{J}\,\mathbf{I})$$

The term **J I** captures the caller continuation, which gets bound to k. The shape of the residual term is as expected: when $[\![\mathbf{J}]\!]$ is applied to a function, it returns another function, which when applied ultimately invokes the captured continuation.

Let us end by remarking that the J operator is expressive enough to encode a familiar control operator like **callcc** [260].

$$[\![\mathbf{callcc}]\!] \stackrel{\text{def}}{=} \lambda f.f\,\mathbf{J}\,\mathbf{I}$$

Felleisen [80] has shown that the J operator can be syntactically embedded using callcc.

$$[\![\lambda x.M]\!] \stackrel{\text{def}}{=} \lambda x.\mathbf{callcc}\,(\lambda k.[\![M]\!][\mathbf{J} \mapsto \lambda f.\lambda y.k\ (f\,y)])$$

The key point here is that λ-abstractions are not translated homomorphically. The occurrence of **callcc** immediately under the binder reifies the current continuation of the function, which is the precisely the caller continuation in the body M. In M the symbol **J** is substituted with a function that simulates **J** by post-composing the captured continuation with the function argument provided to **J**.

A.2.2 Delimited control operators

The main problem with undelimited control is that it is the programmatic embodiment of the proverb *all or nothing* in the sense that an undelimited continuation always represent the entire residual program from its point of capture. In its basic form undelimited control does not offer the flexibility to reify only some segments of the evaluation context. Delimited control rectifies this problem by associating each control operator with a control delimiter such that designated segments of the evaluation context can be captured individually without interfering with the context beyond the delimiter. This provides a powerful and modular programmatic tool that enables programmers to isolate the control flow of specific parts of their programs, and thus enables local reasoning about the behaviour of control infused program segments. One may argue that delimited control to an extent is more first-class than undelimited control, because, in contrast to undelimited control, it provides more fine-grain control over the evaluation context.

In 1988 Felleisen introduced the first control delimiter known as ‘prompt’, as a companion to the composable control operator F (alias control) [81]. Felleisen’s line of work was driven by a dynamic interpretation of composable continuations in terms of algebraic manipulation of control component of abstract machines. In the context of abstract machines, a continuation is defined as a sequence of frames, whose end is denoted by a prompt, and continuation composition is concatenation of their sequences [79, 83, 86]. The natural outcome of this interpretation is the control phenomenon known as *dynamic delimited control*, where the control operator is dynamically bound by its delimiter. An application of a control operator causes the machine to scour through control component to locate the corresponding delimiter.

The following year, Danvy and Filinski [61] introduced an alternative pair of operators known as ‘shift’ and ‘reset’, where ‘shift’ is the control operator and ‘reset’ is the control delimiter. Their line of work were driven by a static interpretation of composable continuations in terms of continuation passing style (CPS). In ordinary CPS a continuation is represented as a function, however, there is no notion of composition, because every function call must appear in tail position. The ‘shift’ operator enables composition of continuation functions as it provides a means for abstracting over control contexts. Technically, this works by iterating the CPS transform twice on the source program, where ‘shift’ provides access to continuations that arise from the second transformation. The ‘reset’ operator acts as the identity for continuation functions, which effectively delimits the extent of ‘shift’ as in terms of CPS the identity function denotes the top-level continuation. This interpretation of composable continuations as functions naturally leads to the control phenomenon known as *static delimited control*, where the control operator is statically bound by its delimiter.

The machine interpretation and continuation passing style interpretation of composable continuations were eventually connected through defunctionalisation and refunctionalisation in a line of work by Danvy and collaborators [1, 3, 4, 58, 59, 65, 66].

Since control/prompt and shift/reset a whole variety of alternative delimited control operators has appeared.

Felleisen’s control and prompt Control and prompt were introduced by Felleisen in 1988 [81]. The control operator ‘control’ is a rebranding of the F operator. Although, the name ‘control’ was first introduced a little later by Sitaram and Felleisen [251]. A prompt acts as a control-flow barrier that delimits different parts of a program, enabling programmers to manipulate and reason about control locally in different parts of a

program. The name 'prompt' is intended to draw connections to shell prompts, and how they act as barriers between the user and operating system.

In this presentation both control and prompt appear as computation forms.

$$M, W \in \text{Comp} ::= \cdots \mid \textbf{control}\ k.M \mid \#\ M$$

The **control** $k.M$ expression reifies the context up to the nearest, dynamically determined, enclosing prompt and binds it to k inside of M. A prompt is written using the sharp (#) symbol. The prompt remains in place after the reification, and thus any subsequent application of **control** will be delimited by the same prompt. Presenting **control** as a binding form may conceal the fact that it is same as **F**. However, the presentation here is close to Sitaram and Felleisen's presentation, which in turn is close to actual implementations of **control**.

The static semantics of control and prompt were absent in Felleisen's original treatment. Later, Kameyama and Yonezawa [139] have given a polymorphic type system with answer type modifications for control and prompt (we will discuss answer type modification when discussing shift/reset). It is also worth mentioning that Dybvig et al. [75] present a typed embedding of control and prompts in Haskell (actually, they present an entire general monadic framework for implementing control operators based on the idea of *multi-prompts*, which are a slight generalisation of prompts — we will revisit multi-prompts when we discuss splitter and cupto).

The dynamic semantics for control and prompt consist of three rules: 1) handle return through a prompt, 2) continuation capture, and 3) continuation invocation.

Value	$\#\ V \rightsquigarrow V$
Capture	$\#\ \mathcal{E}[\textbf{control}\ k.M] \rightsquigarrow \#\ M[\ulcorner \textbf{cont}_{\mathcal{E}} \urcorner / k]$, where $\mathcal{E}$ contains no #
Resume	$\textbf{resume}\ \textbf{cont}_{\mathcal{E}}\ V \rightsquigarrow \mathcal{E}[V]$

The Value rule accounts for the case when the computation constituent of # has been reduced to a value, in which case the prompt is removed and the value is returned. The Capture rule states that an application of **control** captures the current continuation up to the nearest enclosing prompt. The current continuation (up to the nearest prompt) is also aborted. If we erase # from the rule, then it is clear that **control** has the same dynamic behaviour as **F**. It is evident from the Resume rule that control and prompt are an instance of a dynamic control operator, because resuming the continuation object produced by **control** does not install a new prompt.

To illustrate # and **control** in action, let us consider a few simple examples.

$$
\begin{array}{rl}
 & 1 + \#\, 2 + (\textbf{control}\ k.3 + k\ 0) + (\textbf{control}\ k'.k'\ 4) \\
\rightsquigarrow^{+} & (\text{Capture } \mathcal{E} = 2 + [\,] + (\textbf{control}\ k'.k'\ 4)) \\
 & 1 + \#\, 3 + \textbf{resume cont}_{\mathcal{E}}\ 0 \\
\rightsquigarrow & (\text{Resume with } 0) \\
 & 1 + \#\, 3 + (2 + 0) + (\textbf{control}\ k'.k'\ 4) \\
\rightsquigarrow^{+} & (\text{Capture } \mathcal{E}' = 5 + [\,]) \\
 & 1 + \#\, \textbf{resume cont}_{\mathcal{E}'}\ 4 \\
\rightsquigarrow^{+} & (\text{Resume with } 4) \\
 & 1 + \#\, 5 + 4 \\
\rightsquigarrow^{+} & (\textsf{Value rule}) \\
 & 1 + 9 \rightsquigarrow 10
\end{array}
$$

The continuation captured by the either application of **control** is oblivious to the continuation $1 + [\,]$ of #. Since the captured continuation is composable it returns to its call site. The invocation of the captured continuation k returns the value 0, but splices the captured context into the context $3 + [\,]$. The second application of **control** captures the new context up to the delimiter. The continuation is immediately applied to the value 4, which causes the captured context to be reinstated with the value 4 plugged in. Ultimately the delimited context reduces to the value 9, after which the prompt # gets eliminated, and the continuation of the # is applied to the value 9, resulting in the final result 10.

Let us consider a slight variation of the previous example.

$$
\begin{array}{rl}
 & 1 + \#\, 2 + (\textbf{control}\ k.3 + k\ 0) + (\textbf{control}\ k'.4) \\
\rightsquigarrow^{+} & (\text{Capture } \mathcal{E} = 2 + [\,] + (\textbf{control}\ k'.4)) \\
 & 1 + \#\, 3 + \textbf{resume cont}_{\mathcal{E}}\ 0 \\
\rightsquigarrow & (\text{Resume with } 0) \\
 & 1 + \#\, 3 + (2 + 0) + (\textbf{control}\ k'.4) \\
\rightsquigarrow^{+} & (\text{Capture } \mathcal{E}' = 5 + [\,]) \\
 & 1 + \#\, 4 \\
\rightsquigarrow^{+} & (\textsf{Value rule}) \\
 & 1 + 4 \rightsquigarrow 5
\end{array}
$$

Here the computation constituent of the second application of **control** drops the captured continuation, which has the effect of erasing the previous computation, ultimately resulting in the value 5 rather than 10. The continuation captured by the first application

of **control** contains another application of **control**. The application of the continuation immediate reinstates the captured context filling the hole left by the first instance of **control** with the value 0. The second application of **control** captures the remainder of the computation of to #. However, the captured context gets discarded, because the continuation k' is never invoked.

A slight variation on control and prompt is $\mathbf{control_0}$ and $\#_0$ [247]. The main difference is that $\mathbf{control_0}$ removes its corresponding prompt, i.e.

$$\mathsf{Capture}_0 \qquad \#_0\ \mathcal{E}[\mathbf{control_0}\ k.M] \ \rightsquigarrow\ M[\ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner / k], \text{ where } \mathcal{E} \text{ contains no } \#_0$$

Higher-order programming with control and prompt (and delimited control in general) is fragile, because the body of a higher-order function may inadvertently trap instances of control in its functional arguments. This observation led Sitaram and Felleisen [251] to define an indexed family of control and prompt pairs such that instances of control and prompt can be layered on top of one another. The idea is that the index on each pair denotes their level i such that $\mathbf{control}^i$ matches $\#^i$ and may capture any other instances of $\#^j$ where $j < i$.

Danvy and Filinski's shift and reset Shift and reset first appeared in a technical report by Danvy and Filinski in 1989. Although, perhaps the most widely known account of shift and reset appeared in Danvy and Filinski's seminal work on abstracting control the following year [62]. Shift and reset differ from control and prompt in that the contexts abstracted by shift are statically scoped by reset.

In our setting both shift and reset appear as computation forms.

$$M, N \in \text{Comp} ::= \cdots \mid \mathbf{shift}\ k.M \mid \langle M \rangle$$

The **shift** construct captures the continuation delimited by an enclosing $\langle - \rangle$ and binds it to k in the computation M.

Danvy and Filinski's original development of shift and reset stands out from the previous developments of control operators, as they presented a type system for shift and reset, whereas previous control operators were originally studied in untyped settings. The standard inference-based approach to type checking [222, 223] is inadequate for type checking shift and reset, because shift may alter the *answer type* of the expression (the terminology 'answer type' is adopted from typed continuation passing style transforms, where the codomain of every function is transformed to yield the type of whatever answer the entire program yields [196]). To capture the potent power of shift

in the type system they introduced the notion of *answer type modification* [61]. The addition of answer type modification changes type judgement to be a five place relation.

$$\Gamma; B \vdash M : A; B'$$

This would be read as: in a context Γ where the original result type was B, the type of M is A, and modifies the result type to B'. In this system the typing rule for **shift** is as follows.

$$\frac{\Gamma, k : A/C \to B/C; D \vdash M : D; B'}{\Gamma; B \vdash \mathbf{shift}\ k.M : A; B'}$$

Here the function type constructor $-/- \to -/-$ has been endowed with the domain and codomain of the continuation. The left hand side of $\to$ contains the domain type of the function and the codomain of the continuation, respectively. The right hand side contains the domain of the continuation and the codomain of the function, respectively.

Answer type modification is a powerful feature that can be used to type embedded languages, an illustrious application of this is Danvy's typed printf [57]. A polymorphic extension of answer type modification has been investigated by Asai and Kameyama [11], Kiselyov and Shan [148] developed a substructural type system with answer type modification, whilst Kobori et al. [155] demonstrated how to translate from a source language with answer type modification into a system without using typed multi-prompts.

Differences between shift/reset and control/prompt manifest in the dynamic semantics as well.

Value	$\langle V \rangle$	$\rightsquigarrow$	V
Capture	$\langle \mathcal{E}[\mathbf{shift}\ k.M] \rangle$	$\rightsquigarrow$	$\langle M[\ulcorner \mathbf{cont}_{\langle \mathcal{E} \rangle} \urcorner / k] \rangle$, where $\mathcal{E}$ contains no $\langle - \rangle$
Resume	$\mathbf{resume}\ \mathbf{cont}_{\langle \mathcal{E} \rangle}\ V$	$\rightsquigarrow$	$\langle \mathcal{E}[V] \rangle$

The key difference between Felleisen's control/prompt and shift/reset is that the Capture rule for the latter includes a copy of the delimiter in the reified continuation. This delimiter gets installed along with the captured context $\mathcal{E}$ when the continuation object is resumed. The extra reset has ramifications for the operational behaviour of subsequent occurrences of **shift** in $\mathcal{E}$. To put this into perspective, let us revisit the second con-

trol/prompt example with shift/reset instead.

$$
\begin{array}{rl}
 & 1 + \langle 2 + (\mathbf{shift}\ k.3 + k\,0) + (\mathbf{shift}\ k'.4)\rangle \\
\rightsquigarrow^{+} & \quad (\text{Capture } \mathcal{E} = 2 + [\,] + (\mathbf{shift}\ k.4)) \\
 & 1 + \langle \mathbf{resume\ cont}_{\mathcal{E}}\ 0\rangle \\
\rightsquigarrow & \quad (\text{Resume with } 0) \\
 & 1 + \langle 3 + \langle 2 + 0 + (\mathbf{shift}\ k'.4)\rangle\rangle \\
\rightsquigarrow^{+} & \quad (\text{Capture } \mathcal{E}' = 2 + [\,]) \\
 & 1 + \langle 3 + \langle 4\rangle\rangle \\
\rightsquigarrow^{+} & \quad (\text{Value rule}) \\
 & 1 + \langle 7\rangle \rightsquigarrow^{+} 8
\end{array}
$$

Contrast this result with the result 5 obtained when using control/prompt. In essence the insertion of a new reset after resumption has the effect of remembering the local context of the previous continuation invocation.

This difference naturally raises the question whether shift/reset and control/prompt are interdefinable or exhibit essential expressivity differences. Shan [247] answered this question demonstrating that shift/reset and control/prompt are macro-expressible. The translations are too intricate to be reproduced here, however, it is worth noting that Shan were working in the untyped setting of Scheme and the translation of control/prompt made use of recursive continuations. Biernacki et al. [25] typed and reimplemented this translation in Standard ML New Jersey [9], using Filinski's encoding of shift/reset in terms of callcc and state [87].

As with control and prompt there exist various variation of shift and reset. Danvy and Filinski [61] also considered $\mathbf{shift_0}$ and $\langle - \rangle_0$. The operational difference between $\mathbf{shift_0}/\langle - \rangle_0$ and $\mathbf{shift}/\langle - \rangle$ manifests in the capture rule.

$$\mathsf{Capture}_0 \qquad \langle \mathcal{E}[\mathbf{shift_0}\, k.M]\rangle_0 \rightsquigarrow M[\ulcorner \mathbf{cont}_{\langle \mathcal{E}\rangle_0} \urcorner / k], \text{ where } \mathcal{E} \text{ contains no } \langle - \rangle_0$$

The control reifier captures the continuation up to and including its delimiter, however, unlike **shift**, it removes the control delimiter from the current evaluation context. Thus $\mathbf{shift_0}/\langle - \rangle_0$ are 'dynamic' variations on $\mathbf{shift}/\langle - \rangle$. Materzok and Biernacki [190] introduced $\langle - \mid - \rangle$ (pronounced "dollar0") as an alternative control delimiter for $\mathbf{shift_0}$.

$$
\begin{array}{lrcl}
\mathsf{Value}_{\$_0} & \langle V \mid x.N\rangle & \rightsquigarrow & N[V/x] \\
\mathsf{Capture}_{\$_0} & \langle \mathcal{E}[\mathbf{shift_0}\, k.M] \mid x.N\rangle & \rightsquigarrow & M[\ulcorner \mathbf{cont}_{(x.N\,\$_0\,\mathcal{E})} \urcorner / k], \\
 & & & \qquad \text{where } \mathcal{E} \text{ contains no } \langle - \mid - \rangle \\
\mathsf{Resume}_{\$_0} & \mathbf{resume\ cont}_{(\langle \mathcal{E} \mid x.N\rangle)}\ V & \rightsquigarrow & \langle \mathcal{E}[V] \mid x.N\rangle
\end{array}
$$

The intuition here is that $\langle M \mid x.N \rangle$ evaluates M to some value V in a fresh context, and then continues as N with x bound to V. Thus it builds in a form of "success continuation" that makes it possible to post-process the result of a reset0 term. In fact, reset0 is macro-expressible in terms of dollar0 [190].

$$[\![\langle M \rangle_0]\!] \stackrel{\text{def}}{=} \langle [\![M]\!] \mid x.x \rangle$$

By taking the success continuation to be the identity function dollar0 becomes operationally equivalent to reset0. As it turns out reset0 and shift0 (together) can macro-express dollar0 [190].

$$[\![\langle M \mid x.N \rangle]\!] \stackrel{\text{def}}{=} (\lambda k.\langle (\lambda x.\mathbf{shift_0}\ z.k\ x)\ [\![M]\!] \rangle_0)\ (\lambda x.[\![N]\!])$$

This translation is a little more involved. The basic idea is to first explicit pass in the success continuation, then evaluate M under a reset to yield value which gets bound to x, and then subsequently uninstall the reset by invoking $\mathbf{shift_0}$ and throwing away the captured continuation, afterwards we invoke the success continuation with the value x.

Queinnec and Serpette's splitter The 'splitter' control operator reconciles abortive continuations and composable continuations. It was introduced by Queinnec and Serpette [234] in 1991. The name 'splitter' is derived from it operational behaviour, as an application of 'splitter' marks evaluation context in order for it to be split into two parts, where the context outside the mark represents the rest of computation, and the context inside the mark may be reified into a delimited continuation. The operator supports two operations 'abort' and 'calldc' to control the splitting of evaluation contexts. The former has the effect of escaping to the outer context, whilst the latter reifies the inner context as a delimited continuation (the operation name is short for "call with delimited continuation").

Splitter and the two operations abort and calldc are value forms.

$$V, W \in \mathrm{Val} ::= \cdots \mid \mathbf{splitter} \mid \mathbf{abort} \mid \mathbf{calldc}$$

In their treatment of splitter, Queinnec and Serpette gave three different presentations of splitter. The presentation that I have opted for here is close to their second presentation, which is in terms of multi-prompt continuations. This variation of splitter admits a pleasant static semantics too. Thus, we further extend the syntactic categories with the

machinery for first-class prompts.

$$A, B \in \text{Type} ::= \cdots \mid \mathsf{Prompt}\ A$$
$$V, W \in \text{Val} ::= \cdots \mid p$$
$$M, N \in \text{Comp} ::= \cdots \mid \#_V\ M$$

The type $\mathsf{Prompt}\ A$ classifies prompts whose answer type is A. Prompt names are first-class values and denoted by p. The computation $\#_V\ M$ denotes a computation M delimited by a parameterised prompt, whose value parameter V is supposed to be a prompt name. The static semantics of **splitter**, **abort**, and **calldc** are as follows.

$$\frac{}{\Gamma \vdash \mathbf{splitter} : (\mathsf{Prompt}\ A \to A) \to A} \qquad \frac{}{\Gamma \vdash \mathbf{abort} : \mathsf{Prompt}\ A \times (1 \to A) \to B}$$

$$\frac{}{\Gamma \vdash \mathbf{calldc} : \mathsf{Prompt}\ A \times ((B \to A) \to B) \to B}$$

In this presentation, the operator and the two operations all amount to special higher-order function symbols. The argument to **splitter** is parameterised by a prompt name. This name is injected by **splitter** upon application. The operations **abort** and **calldc** both accept as their first argument the name of the delimiting prompt. The second argument of **abort** is a thunk, whilst the second argument of **calldc** is a higher-order function, which accepts a continuation as its argument.

For the sake of completeness the prompt primitives are typed as follows.

$$\frac{}{\Gamma, p : \mathsf{Prompt}\ A \vdash p : \mathsf{Prompt}\ A} \qquad \frac{\Gamma \vdash V : \mathsf{Prompt}\ A \qquad \Gamma \vdash M : A}{\Gamma \vdash \#_V\ M : A}$$

The dynamic semantics of this presentation require a bit of generativity in order to generate fresh prompt names. Therefore the reduction relation is extended with an additional component to keep track of which prompt names have already been allocated.

$$\begin{array}{lrcl}
\mathsf{AppSplitter} & \mathbf{splitter}\ V, \rho & \rightsquigarrow & \#_p\ V\,p, \rho \uplus \{p\} \\
\mathsf{Value} & \#_p\ V, \rho & \rightsquigarrow & V, \rho \\
\mathsf{Abort} & \#_p\ \mathcal{E}[\mathbf{abort}\ \langle p; V\rangle], \rho & \rightsquigarrow & V\,\langle\rangle, \rho \\
\mathsf{Capture} & \#_p\ \mathcal{E}[\mathbf{calldc}\ \langle p; V\rangle] & \rightsquigarrow & V\ulcorner\mathbf{cont}_{\mathcal{E}}\urcorner, \rho \\
\mathsf{Resume} & \mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ V, \rho & \rightsquigarrow & \mathcal{E}[V], \rho
\end{array}$$

We see by the AppSplitter rule that an application of **splitter** generates a fresh named prompt, whose name is applied on the function argument. The Value rule is completely

standard. The Abort rule show that an invocation of **abort** causes the current evaluation context $\mathcal{E}$ up to and including the nearest enclosing prompt. The next rule Capture show that **calldc** captures and aborts the context up to the nearest enclosing prompt. The captured context is applied on the function argument of **calldc**. As part of the operation the prompt is removed.

It is clear by the prompt semantics that an invocation of either **abort** and **calldc** is only well-defined within the dynamic extent of **splitter**. Since the prompt is eliminated after use of either operation subsequent operation invocations must be guarded by a new instance of **splitter**.

Let us consider an example using both **calldc** and **abort**.

$$
\begin{array}{ll}
 & 2+\mathbf{splitter}\,(\lambda p.2+\mathbf{splitter}\,(\lambda p'.3+\mathbf{calldc}\,\langle p;\lambda k.k\ 0+\mathbf{abort}\,\langle p';\lambda\langle\rangle.k\ 1\rangle\rangle)),\emptyset \\
\rightsquigarrow & \quad(\mathsf{AppSplitter}) \\
 & 2+\#_p\ 2+\mathbf{splitter}\,(\lambda p'.3+\mathbf{calldc}\,\langle p;\lambda k.k\ 0+\mathbf{abort}\,\langle p';\lambda\langle\rangle.k\ 1\rangle\rangle),\{p\} \\
\rightsquigarrow & \quad(\mathsf{AppSplitter}) \\
 & 2+\#_p\ 2+\#_{p'}\ 3+\mathbf{calldc}\,\langle p;\lambda k.k\ 0+\mathbf{abort}\,\langle p';\lambda\langle\rangle.k\ 1\rangle\rangle,\{p,p'\} \\
\rightsquigarrow & \quad(\mathsf{Capture}\ \mathcal{E}=2+\#_{p'}\ 3+[\,]) \\
 & 2+k\ 0+\mathbf{abort}\,\langle p';\lambda\langle\rangle.k\ 1\rangle,\{p,p'\} \\
\rightsquigarrow & \quad(\mathsf{Resume}\ \mathcal{E}\ \mathrm{with}\ 0) \\
 & 2+2+\#_{p'}\ 3+\mathbf{abort}\,\langle p';\lambda\langle\rangle.\ulcorner\mathbf{cont}_{\mathcal{E}}\urcorner 1\rangle,\{p,p'\} \\
\rightsquigarrow^{+} & \quad(\mathsf{Abort}) \\
 & 4+\ulcorner\mathbf{cont}_{\mathcal{E}}\urcorner 1,\{p,p'\} \\
\rightsquigarrow & \quad(\mathsf{Resume}\ \mathcal{E}\ \mathrm{with}\ 1) \\
 & 4+2+\#_{p'}\ 3+1,\{p,p'\} \\
\rightsquigarrow^{+} & \quad(\mathsf{Value}) \\
 & 6+4,\{p,p'\}\rightsquigarrow 10,\{p,p'\}
\end{array}
$$

The important thing to observe here is that the application of **calldc** skips over the inner prompt and reifies it as part of the continuation. This behaviour stands differ from the original formulations of control/prompt, shift/reset. The first application of k restores the context with the prompt. The **abort** application erases the evaluation context up to this prompt, however, the body of the functional argument to **abort** reinvokes the continuation k which restores the prompt context once again.

Moreau and Queinnec [205] proposed a variation of splitter called *marker*, which is also built on top of multi-prompt semantics. The key difference is that the control reifier strips the reified context of all prompts.

Spawn The spawn control operator appeared in a paper published by Hieb and Dybvig [116] in 1990. It is designed for using continuations to program tree-based concurrency. Syntactically, spawn is just a function symbol (like callcc), whose operational behaviour establishes the root of a process tree, and passes the *controller* for the tree to its argument. As we will see shortly a controller is a higher-order function, which grants its argument access to the continuation of a process.

We add **spawn** as a value form.

$$V, W \in \mathrm{Val} ::= \mathbf{spawn}$$

Hieb and Dybvig [116] do not give a static semantics for **spawn**. Their dynamic semantics depend on an extension reminiscent of multi-prompts.

$$\ell \in \mathcal{L}$$
$$M, N \in \mathrm{Comp} ::= \ell : M \mid V \uparrow \ell$$

The set $\mathcal{L}$ is some countably set of labels. The expression $(\ell : M)$ is called a *labelled* expression. It essentially plays the role of prompt. The expression $(V \uparrow \ell)$ is called a control expression. The operator $\uparrow$ is a control reifier which captures the continuation up to the label ℓ and supplies this continuation to V.

AppSpawn	$\mathbf{spawn}\ V, \rho$	$\rightsquigarrow$	$\ell : V\,(\lambda f.f \uparrow \ell), \{\ell\} \uplus \rho$
Value	$\ell : V, \rho$	$\rightsquigarrow$	V, ρ
Capture	$\ell : \mathcal{E}[V \uparrow \ell], \rho$	$\rightsquigarrow$	$V\ulcorner \mathbf{cont}_{\ell:\mathcal{E}} \urcorner, \rho$
Resume	$\mathbf{resume}\ \mathbf{cont}_{\ell:\mathcal{E}}\ V, \rho$	$\rightsquigarrow$	$\ell : \mathcal{E}[V], \rho$

The AppSpawn rule generates a fresh ℓ and applies the functional value V the controller for process tree. By the Capture rule, an invocation of the controller causes the evaluation context up to the matching label ℓ to be reified as a continuation. This continuation gets passed to the functional value of the control expression. The captured continuation contains the label ℓ, and as specified by the Resume rule an invocation of the continuation causes this label to be reinstalled.

The following example usage of **spawn** is a slight variation on an example due to Hieb et al. [118].

$$\begin{array}{ll} & 1 :: (\mathbf{spawn}\,(\lambda c.2 :: (c\,(\lambda k.3 :: k\,(k\,[])))))), \emptyset \\ \rightsquigarrow & (\mathsf{AppSpawn}) \\ & 1 :: (\ell : (\lambda c.2 :: (c\,(\lambda k.3 :: k\,(k\,[])))))\,(\lambda f.f \uparrow \ell)), \{\ell\} \\ \rightsquigarrow & (\beta\text{-reduction}) \\ & 1 :: (\ell : 2 :: ((\lambda f.f \uparrow \ell)\,(\lambda k.3 :: k\,(k\,[])))), \{\ell\} \end{array}$$

$$
\begin{array}{ll}
\rightsquigarrow & (\beta\text{-reduction}) \\
 & 1 :: (\ell : 2 :: ((\lambda k.3 :: k\,(k\,[]))\uparrow\ell)), \{\ell\} \\
\rightsquigarrow & (\mathsf{Capture}\ \mathcal{E} = 2 :: [\,]) \\
 & 1 :: 3 :: \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner (\ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner [\,]), \{\ell\} \\
\rightsquigarrow & (\mathsf{Resume}\ \mathcal{E}\ \mathsf{with}\ [\,]) \\
 & 1 :: 3 :: \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner (\ell : 2 :: [\,]), \{\ell\} \\
\rightsquigarrow^{+} & (\mathsf{Value}) \\
 & 1 :: 3 :: \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner [2], \{\ell\} \\
\rightsquigarrow^{+} & (\mathsf{Resume}\ \mathcal{E}\ \mathsf{with}\ [2]) \\
 & 1 :: 3 :: (\ell : 2 :: [2]), \{\ell\} \\
\rightsquigarrow^{+} & (\mathsf{Value}) \\
 & 1 :: 3 :: [2,2], \{\ell\} \rightsquigarrow^{+} [1,3,2,2], \{\ell\}
\end{array}
$$

When the controller c is invoked the current continuation is $1 :: (\ell : 2 :: [\,])$. The control expression reifies the $\ell : 2 :: [\,]$ portion of the continuation and binds it to k. The first invocation of k reinstates the reified portion and computes the singleton list $[2]$ which is used as argument to the second invocation of k.

Both Hieb and Dybvig [116] and Hieb et al. [118] give several concurrent programming examples with spawn. They show how parallel-or [219] can be codified as a macro using spawn (and a parallel invocation primitive *pcall*).

Sitaram's fcontrol The control operator 'fcontrol' was introduced by Sitaram [250] in 1993. It is a refinement of control0/prompt0, and thus, it is a dynamic delimited control operator. The main novelty of fcontrol is that it shifts the handling of continuations from control capture operator to the control delimiter. The prompt interface for fcontrol lets the programmer attach a handler to it. This handler is activated whenever a continuation captured. Sitaram's observation was that with previous control operators the handling of control happens at continuation capture point, meaning that the control handling logic gets intertwined with application logic. The inspiration for the interface of fcontrol and its associated prompt came from exception handlers, where the handling of exceptions is separate from the invocation site of exceptions [250].

The operator fcontrol is a value and prompt with handler is a computation.

$$
\begin{array}{rl}
V, W \in \mathrm{Val} & ::= \cdots \mid \mathbf{fcontrol} \\
M, N \in \mathrm{Comp} & ::= \cdots \mid \%\ V.M
\end{array}
$$

As with **callcc**, the value **fcontrol** may be regarded as a special unary function symbol. The syntax % denotes a prompt (in Sitaram's terminology it is called run). The value

constituent of $\%$ is the control handler. It is a binary function, that gets applied to the argument of **fcontrol** and the continuation up to the prompt. The dynamic semantics elucidate this behaviour formally.

$$
\begin{array}{lrcl}
\textsf{Value} & \% \; V.W & \rightsquigarrow & W \\
\textsf{Capture} & \% \; V.\mathcal{E}[\mathbf{fcontrol}\; W] & \rightsquigarrow & V \; W \; \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner, \text{ where } \mathcal{E} \text{ contains no } \% \\
\textsf{Resume} & \mathbf{resume}\; \mathbf{cont}_{\mathcal{E}} \; V & \rightsquigarrow & \mathcal{E}[V]
\end{array}
$$

The Value is similar to the previous Value rules. The interesting rule is the Capture. When **fcontrol** is applied to some value W the enclosing context $\mathcal{E}$ gets reified and aborted up to the nearest enclosing prompt, which invokes the handler V with the argument W and the continuation.

Consider the following, slightly involved, example.

$$
\begin{array}{rl}
 & 2 + \%\,(\lambda y\; k'.1 + k'\; y).\%\,(\lambda x\; k.x + k\,(\mathbf{fcontrol}\; k)).3 + \mathbf{fcontrol}\; 1 \\
\rightsquigarrow & \quad (\textsf{Capture } \mathcal{E} = 3 + [\,]) \\
 & 2 + \%\,(\lambda y\; k'.1 + k'\; y).(\lambda x\; k.x + k\,(\mathbf{fcontrol}\; k))\,1 \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner \\
\rightsquigarrow^{+} & \quad (\textsf{Capture } \mathcal{E}' = 1 + \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner [\,]) \\
 & 2 + (\lambda k\; k'.k'\,(k\; 1)) \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner \\
\rightsquigarrow^{+} & \quad (\textsf{Resume } \mathcal{E} \text{ with } 1) \\
 & 2 + \ulcorner \mathbf{cont} \mathcal{E}' \urcorner (3 + 1) \\
\rightsquigarrow^{+} & \quad (\textsf{Resume } \mathcal{E}' \text{ with } 4) \\
 & 2 + 1 + \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner 4 \\
\rightsquigarrow^{+} & \quad (\textsf{Resume } \mathcal{E} \text{ with } 4) \\
 & 3 + 3 + 4 \rightsquigarrow^{+} 10
\end{array}
$$

This example makes use of nontrivial control manipulation as it passes a captured continuation around. However, the point is that the separation of the handling of continuations from their capture makes it considerably easier to implement complicated control idioms, because the handling code is compartmentalised.

Cupto The control operator cupto is a variation of control0/prompt0 designed to fit into the typed ML-family of languages. It was introduced by Gunter et al. [111] in 1995. The name cupto is an abbreviation for "control up to" [111]. The control operator comes with a set of companion constructs, and thus, augments the syntactic categories

of types, values, and computations.

$$A, B \in \text{Type} ::= \cdots \mid \mathsf{Prompt}\, A$$
$$V, W \in \text{Val} ::= \cdots \mid p \mid \mathbf{newPrompt}$$
$$M, N \in \text{Comp} ::= \cdots \mid \mathbf{set}\ V\ \mathbf{in}\ N \mid \mathbf{cupto}\ V\ k.M$$

The type $\mathsf{Prompt}\, A$ is the type of prompts. It is parameterised by an answer type A for the prompt context. Prompts are first-class values, which we denote by p. The construct **newPrompt** is a special function symbol, which returns a fresh prompt. The computation form **set** V **in** N activates the prompt V to delimit the dynamic extent of continuations captured inside N. The **cupto** $V\ k.M$ computation binds k to the continuation up to (the first instance of) the active prompt V in the computation M.

Gunter et al. [111] gave a Hindley-Milner type system [128, 199] for **cupto**, since they were working in the context of ML languages. I do not reproduce the full system here, only the essential rules for the **cupto** constructs.

$$\frac{}{\Gamma, p : \mathsf{Prompt}\, A \vdash p : \mathsf{Prompt}\, A} \qquad \frac{}{\Gamma \vdash \mathbf{newPrompt} : 1 \to \mathsf{Prompt}\, A}$$

$$\frac{\Gamma \vdash V : \mathsf{Prompt}\, A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathbf{set}\ V\ \mathbf{in}\ N : A} \qquad \frac{\Gamma \vdash V : \mathsf{Prompt}\, B \qquad \Gamma, k : A \to B \vdash M : B}{\Gamma \vdash \mathbf{cupto}\ V\ k.M : A}$$

The typing rule for **set** uses the type embedded in the prompt to fix the type of the whole computation N. Similarly, the typing rule for **cupto** uses the prompt type of its value argument to fix the answer type for the continuation k.

The dynamic semantics is generative to accommodate generation of fresh prompts. Formally, the reduction relation is augmented with a store ρ that tracks which prompt names have already been allocated.

Value	$\mathbf{set}\ p\ \mathbf{in}\ V, \rho$	$\rightsquigarrow\ V, \rho$
NewPrompt	$\mathbf{newPrompt}\ \langle\rangle, \rho$	$\rightsquigarrow\ p, \rho \uplus \{p\}$
Capture	$\mathbf{set}\ p\ \mathbf{in}\ \mathcal{E}[\mathbf{cupto}\ p\ k.M], \rho$	$\rightsquigarrow\ M[\ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner / k], \rho,$ where p is not active in $\mathcal{E}$
Resume	$\mathbf{resume}\ \mathbf{cont}_{\mathcal{E}}\ V, \rho$	$\rightsquigarrow\ \mathcal{E}[V], \rho$

The Value rule is akin to value rules of shift/reset and control/prompt. The rule NewPrompt allocates a fresh prompt name p and adds it to the store ρ. The Capture rule reifies and aborts the evaluation context up to the nearest enclosing active prompt p. After reification the prompt is removed and evaluation continues as M. The Resume rule reinstalls the captured context $\mathcal{E}$ with the argument V plugged in.

Gunter et al.'s cupto provides similar behaviour to Queinnec and Serpette's splitter in regards to being able to 'jump over prompt'. However, the separation of prompt creation from the control reifier coupled with the ability to set prompts manually provide a considerable amount of flexibility. For instance, consider the following example which illustrates how control reifier **cupto** may escape a matching control delimiter. Let us assume that two distinct prompts p and p' have already been created.

$$
\begin{array}{rl}
 & 2 + \textbf{set}\ p\ \textbf{in}\ 3 + \textbf{set}\ p'\ \textbf{in}\ (\textbf{set}\ p\ \textbf{in}\ \lambda\langle\rangle.\textbf{cupto}\ p\ k.k\,(k\ 1))\ \langle\rangle, \{p, p'\} \\
\rightsquigarrow & \quad (\text{Value}) \\
 & 2 + \textbf{set}\ p\ \textbf{in}\ 3 + \textbf{set}\ p'\ \textbf{in}\ (\lambda\langle\rangle.\textbf{cupto}\ p\ k.k\,(k\ 1))\ \langle\rangle, \{p, p'\} \\
\rightsquigarrow^{+} & \quad (\text{Capture}\ \mathcal{E} = 3 + \textbf{set}\ p'\ \textbf{in}\ [\,]) \\
 & 2 + \ulcorner \textbf{cont}_{\mathcal{E}} \urcorner (\ulcorner \textbf{cont}_{\mathcal{E}} \urcorner 1), \{p, p'\} \\
\rightsquigarrow & \quad (\text{Resume}\ \mathcal{E}\ \text{with}\ 1) \\
 & 2 + \ulcorner \textbf{cont}_{\mathcal{E}} \urcorner (3 + \textbf{set}\ p'\ \textbf{in}\ 1), \{p, p'\} \\
\rightsquigarrow^{+} & \quad (\text{Value}) \\
 & 2 + \ulcorner \textbf{cont}_{\mathcal{E}} \urcorner 4, \{p, p'\} \\
\rightsquigarrow & \quad (\text{Resume}\ \mathcal{E}\ \text{with}\ 4) \\
 & 2 + \textbf{set}\ p'\ \mathit{In}\ 4, \{p, p'\} \\
\rightsquigarrow & \quad (\text{Value}) \\
 & 2 + 4, \{p, p'\} \rightsquigarrow 6, \{p, p'\}
\end{array}
$$

The prompt p is used twice, and the dynamic scoping of **cupto** means when it is evaluated it reifies the continuation up to the nearest enclosing usage of the prompt p. Contrast this with the morally equivalent example using splitter, which would get stuck on the application of the control reifier, because it has escaped the dynamic extent of its matching delimiter.

Plotkin and Pretnar's effect handlers In 2009, Plotkin and Pretnar [227] introduced handlers for Plotkin and Power's algebraic effects [224, 226, 228]. In contrast to the previous control operators, the mathematical foundations of handlers were not an afterthought, rather, their origin is deeply rooted in mathematics. Nevertheless, they turn out to provide a pragmatic interface for programming with control. Operationally, effect handlers can be viewed as a small extension to exception handlers, where exceptions are resumable. Effect handlers are similar to fcontrol in that handling of control happens at the delimiter and not at the point of control capture. Unlike fcontrol, the interface of effect handlers provide a mechanism for handling the return value of a computation similar to Benton and Kennedy's exception handlers with success continuations [18].

Effect handler definitions occupy their own syntactic category.

$$A, B \in \text{VType} ::= \cdots \mid A \Rightarrow B$$
$$H \in \text{HDef} ::= \{\mathbf{return}\, x \mapsto M\} \mid \{\langle\!\langle \ell\, p \twoheadrightarrow k \rangle\!\rangle \mapsto N\} \uplus H$$

An effect handler consists of a **return**-clause and zero or more operation clauses. Each operation clause binds the payload of the matching operation ℓ to p and the continuation of the operation invocation to k in N.

Effect handlers introduces a new syntactic category of signatures, and extends the value types with operation types. Operation and handler application both appear as computation forms.

$$\Sigma \in \mathsf{Sig} ::= \emptyset \mid \{\ell : A \twoheadrightarrow B\} \uplus \Sigma$$
$$A, B, C, D \in \text{VType} ::= \cdots \mid A \twoheadrightarrow B$$
$$M, N \in \text{Comp} ::= \cdots \mid \mathbf{do}\, \ell\, V \mid \mathbf{handle}\, M\, \mathbf{with}\, H$$

A signature is a collection of labels with operation types. An operation type $A \twoheadrightarrow B$ is similar to the function type in that A denotes the domain (type of the argument) of the operation, and B denotes the codomain (return type). For simplicity, we will just assume a global fixed signature. The form $\mathbf{do}\, \ell\, V$ is the application form for operations. It applies an operation ℓ with payload V. The construct $\mathbf{handle}\, M\, \mathbf{with}\, H$ handles a computation M with handler H.

$$\frac{\begin{array}{c}\{\ell_i : A_i \twoheadrightarrow B_i\}_i \in \Sigma \\ H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\langle\!\langle \ell_i\, p_i \twoheadrightarrow k_i \rangle\!\rangle \mapsto N_i\}_i \\ \Gamma, x : C; \Sigma \vdash M : D \\ [\Gamma, p_i : A_i, k_i : B_i \to D; \Sigma \vdash N_i : D]_i\end{array}}{\Gamma; \Sigma \vdash H : C \Rightarrow D}$$

$$\frac{\{\ell : A \twoheadrightarrow B\} \in \Sigma \qquad \Gamma; \Sigma \vdash V : A}{\Gamma; \Sigma \vdash \mathbf{do}\, \ell\, V : B} \qquad \frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma; \Sigma \vdash \mathbf{handle}\, M\, \mathbf{with}\, H : D}$$

The first typing rule checks that the operation label of each operation clause is declared in the signature Σ. The signature provides the necessary information to construct the type of the payload parameters p_i and the continuations k_i. Note that the domain of each continuation k_i is compatible with the codomain of ℓ_i, and the codomain of k_i is compatible with the codomain of the handler. The second and third typing rules are application of operations and handlers, respectively. The rule for operation application

simply inspects the signature to check that the operation is declared, and that the type of the payload is compatible with the declared type.

This particular presentation is nominal, because operations are declared up front. Nominal typing is the only sound option in the absence of an effect system (unless we restrict operations to work over a fixed type, say, an integer). Chapter 3 provides a different presentation based on structural typing.

The dynamic semantics of effect handlers are similar to that of **fcontrol**, though, the Value rule is more interesting.

$$\begin{array}{lrcl}
\textsf{Value} & \textbf{handle}\ V\ \textbf{with}\ H & \rightsquigarrow & M[V/x], \text{ where } \{\textbf{return}\ x \mapsto M\} \in H \\
\textsf{Capture} & \textbf{handle}\ \mathcal{E}[\textbf{do}\ \ell\ V]\ \textbf{with}\ H & \rightsquigarrow & M[V/p, \ulcorner \textbf{cont}_{\langle \mathcal{E};H \rangle} \urcorner / k], \\
 & & & \quad \text{where } \ell \text{ is not handled in } \mathcal{E} \\
 & & & \quad \text{and } \{\langle\!\langle \ell\ p \twoheadrightarrow k \rangle\!\rangle \mapsto M\} \in H \\
\textsf{Resume} & \textbf{resume}\ \textbf{cont}_{\langle \mathcal{E};H \rangle}\ V & \rightsquigarrow & \textbf{handle}\ \mathcal{E}[V]\ \textbf{with}\ H
\end{array}$$

The Value rule differs from previous operators as it is not just the identity. Instead the **return**-clause of the handler definition is applied to the return value of the computation. The Capture rule handles operation invocation by checking whether the handler H handles the operation ℓ, otherwise the operation implicitly passes through the term to the context outside the handler. This behaviour is similar to how exceptions pass through the context until a suitable handler has been found. If H handles ℓ, then the context $\mathcal{E}$ from the operation invocation up to and including the handler H are reified as a continuation object, which gets bound in the corresponding clause for ℓ in H along with the payload of ℓ. This form of effect handlers is known as *deep* handlers. They are deep in the sense that they embody a structural recursion scheme akin to fold over computation trees induced by effectful operations. The recursion is evident from Resume rule, as continuation invocation causes the same handler to be reinstalled along with the captured context.

A classic example of handlers in action is handling of nondeterminism. Let us fix a signature with two operations.

$$\Sigma \stackrel{\text{def}}{=} \{\mathsf{Fail} : 1 \twoheadrightarrow 0; \mathsf{Choose} : 1 \twoheadrightarrow \mathsf{Bool}\}$$

The Fail operation is essentially an exception as its codomain is the empty type, meaning that its continuation can never be invoked. The Choose operation returns a boolean.

We will define a handler for each operation.

$$
\begin{aligned}
&H_f^A : A \Rightarrow \mathsf{Option}\ A \\
&H_f \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto \mathsf{Some}\ x;\ \langle\!\langle \mathsf{Fail}\ \langle\rangle \twoheadrightarrow k \rangle\!\rangle \mapsto \mathsf{None}\} \\
&H_c^B : B \Rightarrow \mathsf{List}\ B \\
&H_c \stackrel{\text{def}}{=} \{\mathbf{return}\ x \mapsto [x];\qquad \langle\!\langle \mathsf{Choose}\ \langle\rangle \twoheadrightarrow k \rangle\!\rangle \mapsto k\ \mathsf{true} +\!\!+ k\ \mathsf{false}\}
\end{aligned}
$$

The handler H_f handles an invocation of Fail by dropping the continuation and simply returning None (due to the lack polymorphism, the definitions are parameterised by types A and B respectively. We may consider them as universal type variables). The **return**-case of H_f tags its argument with Some. The H_c definition handles an invocation of Choose by first invoking the continuation k with true and subsequently with false. The two results are ultimately concatenated. The **return**-case lifts its argument into a singleton list. Now, let us define a simple nondeterministic coin tossing computation with failure (by convention let us interpret true as heads and false as tails).

$$
\begin{aligned}
&\mathsf{toss} : 1 \to \mathsf{Bool} \\
&\mathsf{toss}\ \langle\rangle \stackrel{\text{def}}{=} \mathbf{if\ do}\ \mathsf{Choose}\ \langle\rangle \\
&\qquad\qquad \mathbf{then\ do}\ \mathsf{Choose}\ \langle\rangle \\
&\qquad\qquad \mathbf{else\ absurd\ do}\ \mathsf{Fail}\ \langle\rangle
\end{aligned}
$$

The computation toss first performs Choose in order to branch. If it returns true then a second instance of Choose is performed. Otherwise, it raises the Fail exception. If we apply toss outside of H_c and H_f then the computation gets stuck as either Choose or Fail, or both, would be unhandled. Thus, we have to run the computation in the context of both handlers. However, we have a choice to make as we can compose the handlers in either order. Let us first explore the composition, where H_c is the outermost handler. Thus we instantiate H_c at type Option Bool and H_f at type Bool.

$$
\begin{aligned}
&\quad \mathbf{handle}\ (\mathbf{handle}\ \mathsf{toss}\ \langle\rangle\ \mathbf{with}\ H_f)\ \mathbf{with}\ H_c \\
&\rightsquigarrow \quad (\beta\text{-reduction},\ \mathcal{E} = \mathbf{if}\ [\,]\ \mathbf{then}\cdots) \\
&\quad \mathbf{handle}\ (\mathbf{handle}\ \mathcal{E}[\mathbf{do}\ \mathsf{Choose}\ \langle\rangle]\ \mathbf{with}\ H_f)\ \mathbf{with}\ H_c \\
&\rightsquigarrow \quad (\mathsf{Capture},\ \{\langle\!\langle \mathsf{Choose}\ \langle\rangle \twoheadrightarrow k \rangle\!\rangle \mapsto \cdots\} \in H_c,\ \mathcal{E}' = (\mathbf{handle}\ \mathcal{E}\ \cdots)) \\
&\quad k\ \mathsf{true} +\!\!+ k\ \mathsf{false}, \qquad \text{where } k = \ulcorner \mathbf{cont}_{\langle \mathcal{E}'; H_c \rangle} \urcorner \\
&\rightsquigarrow^{+} \quad (\mathsf{Resume\ with\ true}) \\
&\quad (\mathbf{handle}\ (\mathbf{handle}\ \mathcal{E}[\mathsf{true}]\ \mathbf{with}\ H_f)\ \mathbf{with}\ H_c) +\!\!+ k\ \mathsf{false} \\
&\rightsquigarrow \quad (\beta\text{-reduction}) \\
&\quad (\mathbf{handle}\ (\mathbf{handle\ do}\ \mathsf{Choose}\ \langle\rangle\ \mathbf{with}\ H_f)\ \mathbf{with}\ H_c) +\!\!+ k\ \mathsf{false}
\end{aligned}
$$

$$
\begin{aligned}
&\rightsquigarrow \quad (\mathsf{Capture}, \{\langle\!\langle \mathsf{Choose}\ \langle\rangle \twoheadrightarrow k' \rangle\!\rangle \mapsto \cdots\} \in H_c,\ \mathcal{E}'' = (\textbf{handle}\ [\,]\ \cdots))\\
&\quad (k'\ \mathsf{true} +\!\!+ k'\ \mathsf{false}) +\!\!+ k\ \mathsf{false}, \qquad \text{where } k' = \ulcorner \textbf{cont}_{\langle \mathcal{E}''; H_c \rangle} \urcorner\\
&\rightsquigarrow \quad (\mathsf{Resume\ with\ true})\\
&\quad ((\textbf{handle}\ (\textbf{handle}\ \mathsf{true}\ \textbf{with}\ H_f)\ \textbf{with}\ H_c) +\!\!+ k'\ \mathsf{false}) +\!\!+ k\ \mathsf{false}\\
&\rightsquigarrow \quad (\mathsf{Value}, \{\textbf{return}\ x \mapsto \cdots\} \in H_f)\\
&\quad ((\textbf{handle}\ \mathsf{Some\ true}\ \textbf{with}\ H_c) +\!\!+ k'\ \mathsf{false}) +\!\!+ k\ \mathsf{false}\\
&\rightsquigarrow \quad (\mathsf{Value}, \{\textbf{return}\ x \mapsto \cdots\} \in H_c)\\
&\quad ([\mathsf{Some\ true}] +\!\!+ k'\ \mathsf{false}) +\!\!+ k\ \mathsf{false}\\
&\rightsquigarrow^{+} \quad (\mathsf{Resume\ with\ false, Value, Value})\\
&\quad [\mathsf{Some\ true}] +\!\!+ [\mathsf{Some\ false}] +\!\!+ k\ \mathsf{false}\\
&\rightsquigarrow^{+} \quad (\mathsf{Resume\ with\ false})\\
&\quad [\mathsf{Some\ true}, \mathsf{Some\ false}] +\!\!+ (\textbf{handle}\ (\textbf{handle absurd do}\ \mathsf{Fail}\ \langle\rangle\ \textbf{with}\ H_f)\ \textbf{with}\ H_c)\\
&\rightsquigarrow \quad (\mathsf{Capture}, \{\langle\!\langle \mathsf{Fail}\ \langle\rangle \twoheadrightarrow k \rangle\!\rangle \mapsto \cdots\} \in H_f)\\
&\quad [\mathsf{Some\ true}, \mathsf{Some\ false}] +\!\!+ (\textbf{handle}\ \mathsf{None}\ \textbf{with}\ H_c)\\
&\rightsquigarrow \quad (\mathsf{Value}, \{\textbf{return}\ x \mapsto \cdots\} \in H_c)\\
&\quad [\mathsf{Some\ true}, \mathsf{Some\ false}] +\!\!+ [\mathsf{None}] \rightsquigarrow [\mathsf{Some\ true}, \mathsf{Some\ false}, \mathsf{None}]
\end{aligned}
$$

Note how the invocation of Choose passes through H_f, because H_f does not handle the operation. This is a key characteristic of handlers, and it is called *effect forwarding*. Any handler will implicitly forward every operation that it does not handle.

Suppose we were to swap the order of H_c and H_f, then the computation would yield None, because the invocation of Fail would transfer control to H_f, which is the now the outermost handler, and it would drop the continuation and simply return None.

The alternative to deep handlers is known as *shallow* handlers. They do not embody a particular recursion scheme, rather, they correspond to case splits to over computation trees. To distinguish between applications of deep and shallow handlers, we will mark the latter with a dagger superscript, i.e. $\textbf{handle}^{\dagger}\ -\ \textbf{with}\ -$. Syntactically deep and shallow handler definitions are identical, however, their typing differ.

$$
\frac{\begin{array}{c}
\{\ell_i : A_i \twoheadrightarrow B_i\}_i \in \Sigma\\
H = \{\textbf{return}\ x \mapsto M\} \uplus \{\langle\!\langle \ell_i\ p_i \twoheadrightarrow k_i \rangle\!\rangle \mapsto N_i\}_i\\
\Gamma, x : C; \Sigma \vdash M : D\\
[\Gamma, p_i : A_i, k_i : B_i \to C; \Sigma \vdash N_i : D]_i
\end{array}}{\Gamma; \Sigma \vdash H : C \Rightarrow D}
$$

The difference is in the typing of the continuation k_i. The codomains of continuations must now be compatible with the return type C of the handled computation. The typing

suggests that an invocation of k_i does not reinstall the handler. The dynamic semantics reveal that a shallow handler does not reify its own definition.

$$\begin{array}{lll} \textsf{Capture} & \textbf{handle}^{\dagger}\ \mathcal{E}[\textbf{do}\ \ell\ V]\ \textbf{with}\ H \rightsquigarrow M[V/p, \ulcorner \textbf{cont}_{\mathcal{E}} \urcorner/k], & \\ & \qquad \text{where } \ell \text{ is not handled in } \mathcal{E} & \\ & \qquad \text{and } \{\ell\ p\ k \mapsto M\} \in H & \\ \textsf{Resume} & \textbf{resume}\ \textbf{cont}_{\mathcal{E}}\ V \rightsquigarrow \mathcal{E}[V] & \end{array}$$

The Capture reifies the continuation up to the handler, and thus the Resume rule can only reinstate the captured continuation without the handler.

Chapter 2 contains further examples of deep and shallow handlers in action.

Longley's catch-with-continue The control operator *catch-with-continue* (abbreviated catchcont) is a delimited extension of the **catch** operator. It was designed by Longley [177] in 2008 [181]. Its origin is in game semantics, in which program evaluation is viewed as an interactive dialogue with the ambient environment [134] — this view aligns neatly with the view of effect handler oriented programming. Curiously, we can view catchcont and effect handlers as "siblings" in the sense that Longley and Plotkin and Pretnar them respectively, during the same time, whilst working in the same department. However, the relationship is presently just 'spiritual' as no formal connections have been drawn between the two operators.

The catchcont operator appears as a computation form in our calculus.

$$M, N \in \text{Comp} ::= \cdots \mid \textbf{catchcont}\, f.M$$

Unlike other delimited control operators, **catchcont** does not introduce separate explicit syntactic constructs for the control delimiter and control reifier. Instead it leverages the higher-order facilities of λ-calculus: the syntactic construct **catchcont** play the role of control delimiter and the name f of function type is the name of the control reifier. Longley and Wolverson [181] describe f as a 'dummy variable'.

The typing rule for **catchcont** is as follows.

$$\frac{\Gamma, f : A \to B \vdash M : C \times D \qquad \textsf{ground}\ C}{\Gamma \vdash \textbf{catchcont}\, f.M : C \times ((A \to B) \to D) + (A \times (B \to (A \to B) \to C \times D))}$$

The computation handled by **catchcont** must return a pair, where the first component must be a ground value. This restriction ensures that the value is not a λ-abstraction, which means that the value cannot contain any further occurrence of the control reifier f.

The second component is unrestricted, and thus, it may contain further occurrences of f. If M fully reduces then **catchcont** returns a pair consisting of a ground value (i.e. an answer from M) and a continuation function which allow M to yield further 'answers'. Alternatively, if M invokes the control reifier f, then **catchcont** returns a pair consisting of the argument supplied to f and the current continuation of the invocation of f.

The operational rules for **catchcont** are as follows.

$$\begin{array}{lrcl}
\mathsf{Value} & \mathbf{catchcont}\, f.\langle V; W\rangle & \rightsquigarrow & \mathbf{inl}\, \langle V; \lambda f.W\rangle \\
\mathsf{Capture} & \mathbf{catchcont}\, f.\mathcal{E}[f\, V] & \rightsquigarrow & \mathbf{inr}\, \langle V; \lambda x.\lambda f.\mathbf{resume}\, \mathbf{cont}_{\mathcal{E}}\, x\rangle \\
\mathsf{Resume} & \mathbf{resume}\, \mathbf{cont}_{\mathcal{E}}\, V & \rightsquigarrow & \mathcal{E}[V]
\end{array}$$

The Value makes sure to bind any lingering instances of f in W before escaping the delimiter. The Capture rule reifies and aborts the current evaluation up to, but no including, the delimiter, which gets uninstalled. The reified evaluation context gets stored in the second component of the returned pair. Importantly, the second λ-abstraction makes sure to bind any instances of f in the captured evaluation context once it has been reinstated by the Resume rule.

Let us consider an example use of catchcont to compute a tree representing the interaction between a second-order function and its first-order parameter.

$$\begin{array}{l}
\mathsf{odd} : (\mathsf{Int} \to \mathsf{Bool}) \to \mathsf{Bool} \times 1 \\
\mathsf{odd}\, f \stackrel{\text{def}}{=} \langle \mathsf{xor}\, (f\ 0)\, (f\ 1); \langle\rangle\rangle
\end{array}$$

The function odd expects its environment to provide it with an implementation of a single operation of type $\mathsf{Int} \to \mathsf{Bool}$. The body of odd invokes, or queries, this operation twice with arguments 0 and 1, respectively. The results are tested using exclusive-or.

Now, let us implement the environment for odd.

$$\begin{array}{l}
\mathsf{Dialogue} \stackrel{\text{def}}{=} [! : \mathsf{Int}; ? : \langle \mathsf{Bool}, \mathsf{Dialogue}, \mathsf{Dialogue}\rangle] \\
\mathsf{env} : ((\mathsf{Int} \to \mathsf{Bool}) \to \mathsf{Bool} \times 1) \to \mathsf{Dialogue} \\
\mathsf{env}\ m \stackrel{\text{def}}{=} \mathbf{case}\, \mathbf{catchcont}\, f.m\, f\ \{\mathbf{inl}\, \langle ans; \langle\rangle\rangle \mapsto\ !ans; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{inr}\, \langle q; k\rangle \quad \mapsto\ ?q\, (\mathsf{env}\ k\ \mathsf{true})\, (\mathsf{env}\ k\ \mathsf{false})\}
\end{array}$$

Type Dialogue represents the dialogue between odd and its parameter. The data structure is a standard binary tree with two constructors: ! constructs a leaf holding a value of type Int and ? constructs an interior node holding a value of type Bool and two subtrees. The function env implements the environment that odd will be run in. This function evaluates its parameter m under **catchcont** which injects the operation f. If

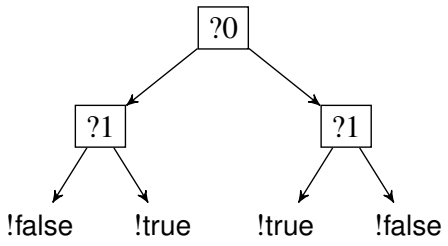


Figure A.2: Visualisation of the result obtained by env odd.

---

m returns, then the left component gets tagged with !, otherwise the argument to the operation *q* gets tagged with a ? along with the subtrees constructed by the two recursive applications of env.

The following derivation gives the high-level details of how evaluation proceeds.

$$
\begin{aligned}
& \mathsf{env}\ \mathsf{odd} \\
\rightsquigarrow^{+}\quad & (\beta\text{-reduction}) \\
& \mathbf{case}\ \mathbf{catchcont}\, f.\, \langle \mathsf{xor}\,(f\ 0)\,(f\ 1); \langle\rangle\rangle \{\cdots\} \\
\rightsquigarrow\quad & (\mathsf{Capture}\ \mathcal{E} = \langle \mathsf{xor}\,[\,]\,(f\ 1), \langle\rangle\rangle) \\
& \mathbf{case}\ \mathbf{inr}\, \langle 0; \lambda x.\lambda f.\ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner x\rangle\ \{\cdots\} \\
\rightsquigarrow^{+}\quad & (\mathsf{Resume}\ \mathcal{E}\ \mathsf{with}\ \mathsf{true}) \\
& ?0\,(\mathbf{case}\ \mathbf{catchcont}\, f.\, \langle \mathsf{xor}\ \mathsf{true}\,(f\ 1); \langle\rangle\rangle \{\cdots\})\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner\ \mathsf{false}) \\
\rightsquigarrow^{+}\quad & (\mathsf{Capture}\ \mathcal{E}' = \langle \mathsf{xor}\ \mathsf{true}\,[\,], \langle\rangle\rangle) \\
& ?0\,(?1\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner\ \mathsf{true})\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner\ \mathsf{false}))\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner\ \mathsf{false}) \\
\rightsquigarrow^{+}\quad & (\mathsf{Resume}\ \mathcal{E}'\ \mathsf{with}\ \mathsf{true}) \\
& ?0\,(?1\,(\mathbf{case}\ \mathbf{catchcont}\, f.\langle \mathsf{xor}\ \mathsf{true}\ \mathsf{true}; \langle\rangle\rangle\ \{\cdots\})\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner\ \mathsf{false}))\\
& \quad (\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner\ \mathsf{false}) \\
\rightsquigarrow^{+}\quad & (\mathsf{Value}) \\
& ?0\,(?1\,(\mathbf{case}\ \mathbf{inl}\ \langle \mathsf{false}; \langle\rangle\rangle\ \{\cdots\})\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner\ \mathsf{false}))\\
& \quad (\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner\ \mathsf{false}) \\
\rightsquigarrow\quad & (\beta\text{-reduction}) \\
& ?0\,(?1\,!\mathsf{false}\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}'} \urcorner\ \mathsf{false}))\,(\mathsf{env}\ \ulcorner \mathbf{cont}_{\mathcal{E}} \urcorner\ \mathsf{false}) \\
\rightsquigarrow^{+}\quad & (\mathsf{Same\ reasoning}) \\
& ?0\,(?1\,!\mathsf{false}\,!\mathsf{true})\,(?1\,!\mathsf{true}\,!\mathsf{false})
\end{aligned}
$$

Figure A.2 visualises this result as a binary tree. The example here does not make use of the 'continuation component', the interested reader may consult Longley and Wolverson [181] for an example usage.

A.3 Programming continuations

Amongst the first uses of continuations were modelling of unrestricted jumps, such as Landin's modelling of Algol 60 labels and gotos using the J operator [159–161, 239].

Backtracking is another early and prominent use of continuations. For example, Burstall [42] used the J operator to implement a heuristic-driven search procedure with continuation-backed backtracking for tree-based search. Somewhat related to backtracking, Friedman et al. [104] posed the *devils and angels problem* as an example that has no direct solution in a programming language without first-class control. Any solution to the devils and angels problem involves extensive manipulation of control to jump both backwards and forwards to resume computation.

If the reader ever find themselves in a quiz show asked to single out a canonical example of continuation use, then implementation of concurrency would be a qualified guess. Cooperative concurrency in terms of various forms of coroutines as continuations occur so frequently in the literature and in the wild that they have become routine. Haynes et al. [115] published one of the first implementations of coroutines using first-class control. Preemptive concurrency in the form of engines were implemented by Dybvig and Hieb [74]. An engine is a control abstraction that runs computations with an allotted time budget [113]. They used continuations to represent strands of computation and timer interrupts to suspend continuations. Kiselyov and Shan [149] used delimited continuations to explain various phenomena of operating systems, including multi-tasking and file systems. On the web, Queinnec [233] used continuations to model the client-server interactions. This model was adapted by Cooper et al. [53] in Links with support for an Erlang-style concurrency model [10]. Leijen [166] and Dolan et al. [72] gave two different ways of implementing the asynchronous programming operator async/await as a user-definable library. In the setting of distributed programming, Bracevac et al. [33] describe a modular event correlation system that makes crucial use of effect handlers. Bracevac's PhD dissertation explicates the theory, design, and implementation of event correlation by way of effect handlers [32].

Continuations have also been used in meta-programming to speed up partial evaluation and multi-staging [140, 162, 208, 274, 279]. Let insertion is a canonical example of use of continuations in multi-staging [279].

Probabilistic programming is yet another application domain of continuations. Kiselyov and Shan [150] used delimited continuations to speed up probabilistic programs. Gorinova et al. [110] used continuations to achieve modularise probabilistic programs and to

provide a simple and efficient mechanism for reparameterisation of inference algorithms. In the subject of differentiable programming Wang et al. [273] explained reverse-mode automatic differentiation operators in terms of delimited continuations.

The aforementioned applications of continuations are by no means exhaustive, though, the diverse application spectrum underlines the versatility of continuations.

A.4 Constraining continuations

Friedman and Haynes [102] advocated for constraining the power of (undelimited) continuations [114]. Even though, they were concerned with callcc and undelimited continuations some of their arguments are applicable to other control operators and delimited continuations. For example, they argued in favour of restricting continuations to be one-shot, which means continuations may only be invoked once. Firstly, because one-shot continuations admit particularly efficient implementations. Secondly, many applications involve only single use of continuations. Thirdly, one-shot continuations interact more robustly with resources, such as file handles, than general multi-shot continuations, because multiple use of a continuation may accidentally interact with a resource after it has been released.

One-shot continuations by themselves are no saving grace for avoiding resource leakage as they may be dropped or used to perform premature exits from a block with resources. For example, Racket provides the programmer with a facility known as *dynamic-wind* to protect a context with resources such that non-local exits properly release whatever resources the context has acquired [93]. An alternative approach is taken by Multicore OCaml, whose implementation of effect handlers with one-shot continuations provides both a *continue* primitive for continuing a given continuation and a *discontinue* primitive for aborting a given continuation [71, 72]. The latter throws an exception at the operation invocation site to which can be caught by local exception handlers to release resources properly. This approach is also used by Fowler [100], who uses a substructural type system to statically enforce the use of continuations, either by means of a continue or a discontinue.

A.5 Implementing continuations

There are numerous strategies for implementing continuations. There is no best implementation strategy. Each strategy has different trade-offs, and as such, there is no

Language	Control operators	Implementation strategies
Eff	Effect handlers	Virtual machine, interpreter
Effekt	Lexical effect handlers	CPS
Frank	N-ary effect handlers	CEK machine
Helium	Effect handlers	CEK machine
Koka	Effect handlers	Continuation monad
Links	Effect handlers, escape	CEK machine, CPS
MLton	callcc	Stack copying
Multicore OCaml	Affine effect handlers	Segmented stacks
OchaCaml	shift/reset	Virtual machine
Racket	callcc, callcomp, cupto, fcontrol, control/prompt, shift/reset, splitter, spawn	Segmented stacks
Scala	shift/reset	CPS
SML/NJ	callcc	CPS
Wasm/k	control/prompt	Virtual machine

Table A.3: Some languages and their implementation strategies for continuations.

"best" strategy. In this section, I will briefly outline the gist of some implementation strategies and their trade-offs. For an in depth analysis the interested reader may consult the respective work of Clinger et al. [49] and Farvardin and Reppy [78], which contain thorough studies of implementation strategies for first-class continuations. Table A.3 lists some programming languages with support for first-class control operators and their implementation strategies.

The control stack provides a adequate runtime representation of continuations as the contiguous sequence of activation records quite literally represent what to do next. Thus continuation capture can be implemented by making a copy of the current stack (possibly up to some delimiter), and continuation invocation as reinstatement of the stack. This implementation strategy works well if continuations are captured infrequently. The MLton implementation of Standard ML utilises this strategy [96]. A slight variation is to defer the first copy action until the continuation is invoked, which requires marking the stack to remember which sequence of activation records to copy.

Obviously, frequent continuation use on top of a stack copying implementation can be expensive time wise as well as space wise, because with undelimited continuations

multiple copies of the stack may be alive simultaneously. Typically the prefix of copies will be identical, which suggests they ought to be shared. One way to achieve optimal sharing is to move from a contiguous stack to a non-contiguous stack representation, e.g. representing the stack as a heap allocated linked list of activation records [56]. With such a representation copying is a constant time and space operation, because there is no need to actually copy anything as the continuation is just a pointer into the stack. The disadvantage of this strategy is that it turns every operation into an indirection.

Segmented stacks provide a middle ground between contiguous stack and non-contiguous stack representations. With this representation the control stack is represented as a linked list of contiguous stacks which makes it possible to only copy a segment of the stack. The stacks grown and shrink dynamically as needed. This representation is due to Hieb et al. [117]. It is used by Chez Scheme, which is the runtime that powers Racket [92]. For undelimited continuations the basic idea is to create a pointer to the current stack upon continuation capture, and then allocate a new stack where subsequent computation happens. For delimited continuations the control delimiter identify when a new stack should be allocated. A potential problem with this representation is *stack thrashing*, which is a phenomenon that occurs when a stack is being continuously resized. This problem was addressed by Bruggeman et al. [41], who designed a slight variation of segmented stacks optimised for one-shot continuations, which has been adapted by Multicore OCaml [72].

Full stack copying and segmented stacks both depend on being able to manipulate the stack directly. This is seldom possible if the language implementer do not have control over the target runtime, e.g. compilation to JavaScript. However, it is possible to emulate stack copying and segmented stacks in lieu of direct stack access. For example, Pettyjohn et al. [214] describe a technique that emulates stack copying by piggybacking on the facile stack inception facility provided by exception handlers in order to lazily reify the control stack. Kumar et al. [157] emulated segmented stacks via threads. Each thread has its own local stack, and as such, a collection of threads effectively models segmented stacks. To actually implement continuations as threads Kumar et al. also made use of standard synchronisation primitives. The advantage of these techniques is that they are generally applicable and portable. The disadvantage is the performance overhead induced by emulation.

Abstract and virtual machines are a form of full machine emulation. An abstract machine is an idealised machine. Abstract machines, such as the CEK machine [83], are attractive because they provide a suitably high-level framework for defining lan-

guage semantics in terms of control string manipulations, whilst admitting a direct implementation. We will discuss abstract machines in more detail in Chapter 5. The term virtual machine typically connotes an abstract machine that works on a byte code representation of programs, whereas the default connotation of abstract machine is a machine that works on a rich abstract syntax tree representation of programs. The disadvantage of abstract machines is their interpretative overhead, although, techniques such as just-in-time compilation can be utilised to reduce this overhead.

Continuation passing style (CPS) is a canonical implementation strategy for continuations — the word 'continuation' even features in its name. CPS is a particular idiomatic notation for programs, where every function takes an additional argument, the current continuation, as input and every function call appears in tail position. Consequently, every aspect of control flow is made explicit, which makes CPS a good fit for implementing control abstraction. In classic CPS the continuation argument is typically represented as a heap allocated closure [8], however, as we shall see in Chapter 4 richer representations of continuations are possible. At first thought it may seem that CPS will not work well in environments that lack proper tail calls such as JavaScript. However, the contrary is true, because the stackless nature of CPS means it can readily be implemented with a trampoline [105]. Alas, at the cost of the indirection induced by the trampoline.

Appendix B

Get get is redundant

The global state effect is often presented with following four equations.

$$\begin{array}{lrcl}
\textsf{Get-get} & x \leftarrow \textsf{get}; y \leftarrow \textsf{get}; M & = & x \leftarrow \textsf{get}; M[x/y] \\
\textsf{Get-put} & x \leftarrow \textsf{get}; \textsf{put}\ x; M & = & M \\
\textsf{Put-get} & \textsf{put}\ V; x \leftarrow \textsf{get}; M & = & \textsf{put}\ V; M[V/x] \\
\textsf{Put-put} & \textsf{put}\ V; \textsf{put}\ W; M & = & \textsf{put}\ W; M
\end{array}$$

However, the first equation is derivable from the second and third equations. I first learned this from Paul-André Melliès during Shonan Seminar No.103 *Semantics of Effects, Resources, and Applications*. I have been unable to find a proof of this fact in the literature, though, Melliès does have a published paper, which states only the three necessary equations [195]. Therefore I include a proof of this fact here (thanks to Sam Lindley for helping me relearning this fact from first principles).

Theorem B.1. Get-put *and* Put-get *implies* Get-get

Proof.

$$\begin{array}{ll}
& x \leftarrow \textsf{get}; y \leftarrow \textsf{get}; M \\
= & \quad (\textsf{Get-put right-to-left};\ z \notin \mathrm{FV}(M)) \\
& z \leftarrow \textsf{get}; \textsf{put}\ z; x \leftarrow \textsf{get}; y \leftarrow \textsf{get}; M \\
= & \quad (\textsf{Put-get}) \\
& z \leftarrow \textsf{get}; \textsf{put}\ z; y \leftarrow \textsf{get}; M[z/x] \\
= & \quad (\textsf{Put-get}) \\
& z \leftarrow \textsf{get}; \textsf{put}\ z; M[z/x, z/y] \\
= & \quad (\text{composition of substitution}) \\
& z \leftarrow \textsf{get}; \textsf{put}\ z; (M[x/y])[z/x]
\end{array}$$

$$
\begin{array}{ll}
= & \quad (\textsf{Put-get right-to-left}) \\
& \quad z \leftarrow \textsf{get}; \textsf{put}\ z; x \leftarrow \textsf{get}; M[x/y] \\
= & \quad (\textsf{Get-put}) \\
& \quad x \leftarrow \textsf{get}; M[x/y]
\end{array}
$$

□

Appendix C

Proof details for the complexity of effectful generic count

In this appendix I give the proof details and artefacts for Theorem 7.11.

Relation to prior work This appendix is imported from Appendix C of Hillerström et al. [124].

Throughout this section we let H_{count} denote the handler definition of count, that is

$$H_{\mathsf{count}} \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} \textbf{return}\ x & \mapsto & \textbf{if}\ x\ \textbf{then return}\ 1\ \textbf{else return}\ 0 \\ \mathsf{Branch}\ \langle\rangle\ r & \mapsto & \textbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \textbf{in} \\ & & \textbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \textbf{in} \\ & & x_{\mathsf{true}} + x_{\mathsf{false}} \end{array} \right\}$$

The timed decision tree model embeds timing information. For the proof we must also know the abstract machine environment and the pure continuation. Thus we decorate timed decision trees with this information.

Definition C.1 (decorated timed decision trees)**.** A decorated timed decision tree is a partial function $\tau : \mathsf{Addr} \rightharpoonup (\mathsf{Lab} \times \mathsf{Nat}) \times \mathsf{Conf}_q$ such that its first projection $bs \mapsto \tau(bs).1$ is a timed decision tree.

We extend the projections labs and steps in the obvious way to work over decorated timed decision trees. We define three further projections. The first $\mathsf{comp}(\tau) \stackrel{\text{def}}{=} bs \mapsto \tau(bs).2.1$ projects the computation component of the configuration, the second $\mathsf{env}(\tau) \stackrel{\text{def}}{=} bs \mapsto \tau(bs).2.2$ projects the environment, and finally the third $\mathsf{pure}(\tau) \stackrel{\text{def}}{=} bs \mapsto \mathsf{head}(t(bs).2.3).1$ projects the pure continuation.

The following definition gives a procedure for constructing a decorated timed decision tree. The construction is analogous to that of Definition 7.6.

Definition C.2. (i) Define $\mathcal{D} : \mathsf{Conf}_q \rightharpoonup \mathsf{Addr} \rightharpoonup (\mathsf{Lab} \times \mathsf{Nat}) \times \mathsf{Conf}_q$ to be the minimal family of partial functions satisfying the following equations:

$$\begin{aligned}
\mathcal{D}(\langle \mathbf{return}\ W \mid \gamma \mid [] \rangle)\,[] &= ((!b, 0), \langle \mathbf{return}\ W \mid \gamma \mid [] \rangle), && \text{if } [\![W]\!]\gamma = b \\
\mathcal{D}(\langle z\,V \mid \gamma \mid \kappa \rangle)\,[] &= ((?[\![V]\!]\gamma, 0), \langle z\,V \mid \gamma \mid \kappa \rangle), && \text{if } \gamma(z) = q \\
\mathcal{D}(\langle z\,V \mid \gamma \mid \kappa \rangle)\,(b :: bs) &\simeq \mathcal{D}(\langle \mathbf{return}\ b \mid \gamma \mid \kappa \rangle)\,bs, && \text{if } \gamma(z) = q \\
\mathcal{D}(\langle M \mid \gamma \mid \kappa \rangle)\,bs &\simeq \mathsf{inc}\,(\mathcal{D}(\langle M' \mid \gamma' \mid \kappa' \rangle)\,bs), && \text{if } \langle M \mid \gamma \mid \kappa \rangle \longrightarrow \langle M' \mid \gamma' \mid \kappa' \rangle
\end{aligned}$$

Here $\mathsf{inc}((\ell, s), \mathcal{C}) = ((\ell, s+1), \mathcal{C})$, and in all of the above equations $\gamma(q) = \gamma'(q) = q$. Clearly $\mathcal{D}(\mathcal{C})$ is a decorated timed decision tree for any $\mathcal{C} \in \mathsf{Conf}_q$.

(ii) The decorated timed decision tree of a computation term is obtained by placing it in the initial configuration: $\mathcal{D}(M) \stackrel{\text{def}}{=} \mathcal{D}(\langle M, \emptyset[q \mapsto q], \kappa_0 \rangle)$.

(iii) The decorated timed decision tree of a closed value $P : \mathsf{Predicate}$ is $\mathcal{D}(P\,q)$. Since q plays the role of a dummy argument, we will usually omit it and write $\mathcal{D}(P)$ for $\mathcal{D}(P\,q)$.

We define some functions, that given a list of booleans and a n-standard predicate, compute configurations of the effectful abstract machine at particular points of interest during evaluation of the given predicate. Let $\chi_{\mathsf{count}}(V) \stackrel{\text{def}}{=} (\emptyset[pred \mapsto [\![V]\!]\emptyset], H_{\mathsf{count}})$ denote the handler closure of H_{count}.

Notation. For an n-standard predicate P we write $|P| = n$ for the size of the predicate. Furthermore, we define χ_{id} for the identity handler closure $(\emptyset, \{\mathbf{return}\ x \mapsto x\})$.

Definition C.3 (computing machine configurations)**.** For any n-standard predicate P and a list of booleans bs, such that $|bs| \leq n$, we can compute machine configurations at points of interest during evaluation of $\mathsf{count}\ P$.

To make the notation slightly simpler we use the following conventions whenever n, τ, and c appear free: $n = |P|$, $\tau = \mathcal{D}(P)$, and $c(bs) = \sharp(bs' \mapsto [\![P]\!]\ (bs +\!\!+ bs'))$. The definitions are presented in a top-down manner.

- The function arrive either computes the configuration at a query node, if $|bs| < n$,

or the configuration at an answer node.

$$\begin{array}{l}
\qquad \mathsf{arrive} \;:\; \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Conf} \\
\mathsf{arrive}(bs,P) \stackrel{\text{def}}{=} \langle z\ V \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs,P) \rangle, \quad \text{if } |bs| < n \\
\quad \text{where } \; z\ V = \mathsf{comp}(\tau)(bs), \gamma = \mathsf{env}(\tau)(bs), \gamma(z) = (\mathsf{env}^{\perp}(P), \lambda_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \\
\qquad ?k = \mathsf{labs}(\tau)(bs), [\![V]\!]\gamma = k, \text{ and } \sigma = \mathsf{pure}(\tau)(bs) \\
\mathsf{arrive}(bs,P) \stackrel{\text{def}}{=} \langle \mathbf{return}\ W \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs,P) \rangle, \quad \text{if } |bs| = n \\
\text{where } \mathbf{return}\ W = \mathsf{comp}(\tau)(bs), \gamma = \mathsf{env}(\tau)(bs), !b = \mathsf{labs}(\tau)(bs), \text{ and } [\![W]\!]\gamma = b
\end{array}$$

- Correspondingly, the depart function computes the configuration either after the completion of a query or handling of an answer.

$$\begin{array}{l}
\qquad \mathsf{depart} \;:\; \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Conf} \\
\mathsf{depart}(bs,P) \stackrel{\text{def}}{=} \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(bs,P) \rangle, \quad \text{if } |bs| < n \\
\qquad\qquad\qquad\qquad \text{where } \gamma = \mathsf{env}^{\uparrow}_{\mathsf{false}}(bs,P) \text{ and } m = c(bs) \\
\mathsf{depart}(bs,P) \stackrel{\text{def}}{=} \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(bs,P) \rangle, \quad \text{if } |bs| = n \\
\text{where } m = c(bs), b = \begin{cases} \mathsf{true} & \text{if } m = 1 \\ \mathsf{false} & \text{if } m = 0 \end{cases}, \text{ and } \gamma = \mathsf{env}^{\perp}(P)[x \mapsto b]
\end{array}$$

The two clauses of depart yield slightly different configurations. The first clause computes a configuration inside the operation clause of H_{count}. The configuration is exactly tail-configuration after summing up the two respective values returned by the two invocations of resumption. Whilst the second clause computes the tail-configuration inside of the success clause of H_{count} after handling a return value of the predicate.

- The residual function computes the residual continuation structure which contains the bits of computations to perform after handling a complete path in a decision tree.

$$\begin{array}{l}
\qquad \mathsf{residual} \;:\; \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Cont} \\
\mathsf{residual}(bs,P) \stackrel{\text{def}}{=} [(\mathsf{purecont}(bs,P), \chi_{id})]
\end{array}$$

- The function purecont computes the pure continuation.

$$\begin{aligned}
\mathsf{purecont} &: \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{PureCont} \\
\mathsf{purecont}([],P) &\stackrel{\text{def}}{=} [] \\
\mathsf{purecont}(bs +\!\!+ [\mathsf{true}],P) &\stackrel{\text{def}}{=} (\gamma, x_{\mathsf{true}}, \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}) \\
&\quad :: \mathsf{purecont}(bs,P), \\
&\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{true}}(bs +\!\!+ [\mathsf{true}],P) \\
\mathsf{purecont}(bs +\!\!+ [\mathsf{false}],P) &\stackrel{\text{def}}{=} (\gamma, x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) \\
&\quad :: \mathsf{purecont}(bs,P), \\
&\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs +\!\!+ [\mathsf{false}],P)
\end{aligned}$$

- The function $\mathsf{env}^{\perp}$ computes the initial environment of the handler. The family of functions $\mathsf{env}^{\downarrow}_{b\in\mathbb{B}}$ contains two functions, one for each instantiation of b, which describe how to compute the environment prior *descending* down a branch as the result of invoking a resumption with b. Analogously, the functions in the family $\mathsf{env}^{\uparrow}_{b\in\mathbb{B}}$ describe how to compute the environment after *ascending* from the resumptive exploration of a branch.

$$\begin{aligned}
\mathsf{env}^{\perp} &: \mathrm{Val} \to \mathsf{Env} \\
\mathsf{env}^{\perp}(P) &\stackrel{\text{def}}{=} \emptyset[\mathit{pred} \mapsto [\![P]\!]\emptyset]
\end{aligned}$$

$$\begin{aligned}
\mathsf{env}^{\downarrow}_{\mathsf{true}} &: \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Env} \\
\mathsf{env}^{\downarrow}_{\mathsf{true}}(bs,P) &\stackrel{\text{def}}{=} \mathsf{env}^{\perp}(P)[r \mapsto (\sigma, \chi_{\mathsf{count}}(P))], \\
&\quad \text{where } \sigma = \mathsf{pure}(\tau)(bs)
\end{aligned}$$

$$\begin{aligned}
\mathsf{env}^{\downarrow}_{\mathsf{false}} &: \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Env} \\
\mathsf{env}^{\downarrow}_{\mathsf{false}}(bs,P) &\stackrel{\text{def}}{=} \mathsf{env}^{\uparrow}_{\mathsf{true}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{env}^{\uparrow}_{\mathsf{true}} &: \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Env} \\
\mathsf{env}^{\uparrow}_{\mathsf{true}}(bs,P) &\stackrel{\text{def}}{=} \gamma[x_{\mathsf{true}} \mapsto i], \\
&\quad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{true}}(bs,P) \\
&\quad \text{and } i = c(bs +\!\!+ [\mathsf{true}])
\end{aligned}$$

$$\begin{aligned}
\mathsf{env}^{\uparrow}_{\mathsf{false}} &: \mathsf{Addr} \times \mathrm{Val} \rightharpoonup \mathsf{Env} \\
\mathsf{env}^{\uparrow}_{\mathsf{false}}(bs,P) &\stackrel{\text{def}}{=} \gamma[x_{\mathsf{false}} \mapsto j], \\
&\quad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs,P) \\
&\quad \text{and } j = c(bs +\!\!+ [\mathsf{false}])
\end{aligned}$$

The proof of Theorem 7.11 works by alternating between two different modes of reasoning: intensional and extensional. The former is used to reason directly about the steps taken by effcount program and the latter is used to reason about steps taken by the provided predicate. The number of steps taken by an n-standard predicate is readily available by constructing its corresponding decorated timed decision tree model. The model is constructed using a distinguished free variable q to denote a point. The following lemma lets us reason about the number of steps taken by a predicate between

its initial application and its first query, between subsequent queries, and between final query and answer when q is instantiated to $\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle$.

Lemma C.4. *Suppose P is an n-standard predicate, $bs \in \mathsf{Addr}$ is a list of booleans, and for all $\chi \in \mathsf{HClo}$ and $\kappa \in \mathsf{Cont}$. Let q denote the distinguished free variable used to construct the decorated timed decision tree τ of P.*

1. *If $|bs| = 0$ then*

$$\begin{aligned}&\langle \mathit{pred}\ q \mid \mathsf{env}^{\perp}(P)[q \mapsto q] \mid ([],\chi) :: \kappa\rangle \\ \longrightarrow^{\mathsf{steps}(\tau)([])}\ &\langle z\ V \mid \gamma[q \mapsto q] \mid (\sigma,\chi) :: \kappa\rangle\end{aligned}$$

   where $z\ V = \mathsf{comp}(\tau)([])$, $\gamma = \mathsf{env}(\tau)([])$, $?k = \mathsf{labs}(\tau)([])$, $[\![V]\!]\gamma = k$, $\gamma(z) = q$, and $\sigma = \mathsf{pure}(\tau)([])$; implies

$$\begin{aligned}&\langle \mathit{pred}\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \mid \mathsf{env}^{\perp}(P) \mid ([],\chi) :: \kappa\rangle \\ \longrightarrow^{\mathsf{steps}(\tau)([])}\ &\langle z\ V \mid \gamma[z \mapsto (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)] \mid (\sigma,\chi) :: \kappa\rangle\end{aligned}$$

2. *If $|bs| < n-1$ then for all $b \in \mathbb{B}$ and $W \in \mathit{Val}$*

$$\begin{aligned}&\langle \mathbf{return}\ W \mid \mathsf{env}^{\downarrow}_b(bs,P) \mid (\sigma,\chi) :: \kappa\rangle \\ \longrightarrow^{\mathsf{steps}(\tau)(bs{+}{+}[b])}\ &\langle z\ V \mid \gamma[q \mapsto q] \mid (\sigma',\chi) :: \kappa\rangle\end{aligned}$$

   where $[\![W]\!](\mathsf{env}^{\downarrow}_b(bs,P)) = b$, $\sigma = \mathsf{pure}(\tau)(bs)$, $z\ V = \mathsf{comp}(\tau)(bs +\!\!+ [b])$, $\gamma = \mathsf{env}(\tau)(bs +\!\!+ [b])$, $\gamma(z) = q$, $?k = \mathsf{labs}(\tau)(bs +\!\!+ [b])$, $[\![V]\!]\gamma = k$, and $\sigma' = \mathsf{pure}(\tau)(bs +\!\!+ [b])$; implies

$$\begin{aligned}&\langle \mathbf{return}\ W \mid \mathsf{env}^{\downarrow}_b(bs,P) \mid (\sigma,\chi) :: \kappa\rangle \\ \longrightarrow^{\mathsf{steps}(\tau)(bs{+}{+}[b])}\ &\langle z\ V \mid \gamma[z \mapsto (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)] \mid (\sigma',\chi) :: \kappa\rangle\end{aligned}$$

3. *If $|bs| = n-1$ then for all $b \in \mathbb{B}$ and $W \in \mathit{Val}$*

$$\begin{aligned}&\langle \mathbf{return}\ W \mid \mathsf{env}^{\downarrow}_b(bs,P) \mid (\sigma,\chi) :: \kappa\rangle \\ \longrightarrow^{\mathsf{steps}(\tau)(bs{+}{+}[b])}\ &\langle \mathbf{return}\ W' \mid \gamma[q \mapsto q] \mid ([],\chi) :: \kappa\rangle\end{aligned}$$

where $[\![W]\!](\mathsf{env}^{\downarrow}_b(bs,P)) = b$, $\sigma = \mathsf{pure}(\tau)(bs)$, **return** $W' = \mathsf{comp}(\tau)(bs +\!\!+ [b])$, $\gamma = \mathsf{env}(\tau)(bs +\!\!+ [b])$, $!b' = \mathsf{labs}(\tau)(bs +\!\!+ [b])$, *and* $[\![W']\!]\gamma = b'$*; implies*

$$\begin{array}{l} \langle \mathbf{return}\ W \mid \mathsf{env}^{\downarrow}_b(bs,P) \mid (\sigma,\chi) :: \kappa \rangle \\ \longrightarrow^{\mathsf{steps}(\tau)(bs +\!\!+ [b])} \\ \langle \mathbf{return}\ W' \mid \gamma \mid ([],\chi) :: \kappa \rangle \end{array}$$

Proof. By unfolding Definition C.2. □

Let $\mathsf{control} : \mathsf{Conf} \rightharpoonup \mathrm{Val}$ denote a partial function that hoists a value out of a given machine configuration, that is

$$\mathsf{control}(\langle M \mid \gamma \mid \kappa \rangle) \stackrel{\text{def}}{=} \begin{cases} [\![V]\!]\gamma & \text{if } M = \mathbf{return}\ V \\ \bot & \text{otherwise} \end{cases}$$

Notation For a given predicate P we write $\chi_{\mathsf{count}}(P)^{\mathbf{return}}$ to mean $\chi_{\mathsf{count}}(P)^{\mathbf{return}} = (\emptyset[pred \mapsto [\![P]\!]\emptyset], H_{\mathsf{count}})^{\mathbf{return}} = H^{\mathbf{return}}_{\mathsf{count}}$, that is the projection of the success clause of H_{count}.

The following lemma performs most of the heavy lifting for the proof of Theorem 7.11.

Lemma C.5. *Suppose P is an n-standard predicate, then for any list of booleans $bs \in \mathsf{Addr}$ such that $|bs| \leq n$*

$$\mathsf{arrive}(bs,P) \rightsquigarrow^{T(bs,n)} \mathsf{depart}(bs,P),$$

and $\mathsf{control}(\mathsf{depart}(bs,P)) \leq 2^{n-|bs|}$ *with the function T defined as*

$$T(bs,n) = \begin{cases} 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \leq |bs'| \leq n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') & \text{if } |bs| < n \\ 2 & \text{if } |bs| = n \end{cases}$$

Proof. By downward induction on bs.

Base step We have that $|bs| = n$. Since the predicate is n-standard we further have that $n \geq 1$. We proceed by direct calculation.

$$\begin{array}{ll} & \mathsf{arrive}(bs,P) \\ = & \quad \text{(definition of arrive when } n = |bs|) \\ & \langle \mathbf{return}\ W \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs,P) \rangle \\ & \text{where } \mathbf{return}\ W = \mathsf{comp}(\tau)(bs), \gamma = \mathsf{env}(\tau)(bs), !b = \mathsf{labs}(\tau)(bs), \text{ and } [\![W]\!]\gamma = b \\ \longrightarrow & \quad (\text{M-RetHandler}, \chi_{\mathsf{count}}(P)^{\mathbf{return}} = \{\mathbf{return}\ x \mapsto \cdots\}) \\ & \langle \mathbf{if}\ x\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 0 \mid \gamma'[x \mapsto [\![b]\!]\gamma'] \mid \mathsf{residual}(bs,P) \rangle \\ & \text{where } \gamma' = \chi_{\mathsf{count}}(P).1 \end{array}$$

The value b can assume either of two values. We consider first the case $b = \mathsf{true}$.

$$
\begin{array}{rl}
= & \text{(assumption } b = \mathsf{true}\text{, definition of } [\![-]\!] \text{ (2 value steps))} \\
& \langle \textbf{if } x \textbf{ then return } 1 \textbf{ else return } 0 \mid \gamma'[x \mapsto \mathsf{true}] \mid \mathsf{residual}(bs, P) \rangle \\
\longrightarrow & \text{(M-Case} - \mathsf{Inl} \text{ (and } \log|\gamma'[x \mapsto \mathsf{true}]| = 1 \text{ environment operations))} \\
& \langle \textbf{return } 1 \mid \gamma'[x \mapsto \mathsf{true}] \mid \mathsf{residual}(bs, P) \rangle \\
= & \text{(definition of depart when } n = |bs|) \\
& \mathsf{depart}(bs, P)
\end{array}
$$

We have that $\mathsf{control}(\mathsf{depart}(bs, P)) = 1 \leq 2^0 = 2^{n-|bs|}$. Next, we consider the case when $b = \mathsf{false}$.

$$
\begin{array}{rl}
= & \text{(assumption } b = \mathsf{false}\text{, definition of } [\![-]\!] \text{ (2 value steps))} \\
& \langle \textbf{if } x \textbf{ then return } 1 \textbf{ else return } 0 \mid \gamma'[x \mapsto \mathsf{false}] \mid \mathsf{residual}(bs, P) \rangle \\
\longrightarrow & \text{(M-Case} - \mathsf{Inl} \text{ (and } \log|\gamma'[x \mapsto \mathsf{false}]| = 1 \text{ environment operations))} \\
& \langle \textbf{return } 0 \mid \gamma'[x \mapsto \mathsf{false}] \mid \mathsf{residual}(bs, P) \rangle \\
= & \text{(definition of depart when } n = |bs|) \\
& \mathsf{depart}(bs, P)
\end{array}
$$

Again, we have that $\mathsf{control}(\mathsf{depart}(bs, P)) = 0 \leq 2^0 = 2^{n-|bs|}$.

Step analysis In either case, the machine uses exactly 2 transitions. Thus we get that

$$2 = T(bs, n), \quad \text{when } |bs| = n$$

Inductive step The induction hypothesis states that for all $b \in \mathbb{B}$ and $|bs| < n$

$$\mathsf{arrive}(bs +\!\!+ [b], P) \rightsquigarrow^{T(bs +\!\!+ [b], n)} \mathsf{depart}(bs +\!\!+ [b], P),$$

such that $\mathsf{control}(\mathsf{depart}(bs +\!\!+ [b], P)) \leq 2^{n-|bs +\!\!+ [b]|}$. We proceed by direct calculation.

$$
\begin{array}{rl}
& \mathsf{arrive}(bs, P) \\
= & \text{(definition of arrive when } n < |bs|) \\
& \langle z\ V \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P) \rangle \\
& \text{where } z\ V = \mathsf{comp}(\tau)(bs), \gamma = \mathsf{env}(\tau)(bs)[z \mapsto (\mathsf{env}^{\perp}(P), \lambda\_.\textbf{do } \mathsf{Branch}\ \langle\rangle)], \\
& \qquad ?k = \mathsf{labs}(\tau)(bs), [\![V]\!]\gamma = k, \text{ and } \sigma = \mathsf{pure}(\tau)(bs) \\
\longrightarrow & \text{(M-App)} \\
& \langle \textbf{do } \mathsf{Branch}\ \langle\rangle \mid \gamma'[\_ \mapsto k] \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P) \rangle \\
& \text{where } \gamma' = \mathsf{env}^{\perp}(P)
\end{array}
$$

$$
\begin{array}{ll}
\longrightarrow & (\text{M-Handle}-\text{Op},\ \chi_{\text{count}}(P)^{\text{Branch}} = \{\text{Branch}\ \langle\rangle\ r \mapsto \cdots\}) \\
& \left\langle \begin{array}{l} \textbf{let}\ x_{\text{true}} \leftarrow r\ \text{true}\ \textbf{in} \\ \textbf{let}\ x_{\text{false}} \leftarrow r\ \text{false}\ \textbf{in} \\ x_{\text{true}} + x_{\text{false}} \end{array} \mid \gamma[r \mapsto [\![(\sigma, \chi_{\text{count}}(P))]\!]\gamma] \mid \text{residual}(bs, P) \right\rangle \\
\text{where } \gamma = \text{env}^{\perp}(P) & \\
= & (\text{definition of } [\![-]\!] \text{ (1 value step)}) \\
& \left\langle \begin{array}{l} \textbf{let}\ x_{\text{true}} \leftarrow r\ \text{true}\ \textbf{in} \\ \textbf{let}\ x_{\text{false}} \leftarrow r\ \text{false}\ \textbf{in} \\ x_{\text{true}} + x_{\text{false}} \end{array} \mid \gamma' \mid \text{residual}(bs, P) \right\rangle \\
\text{where } \gamma' = \gamma[r \mapsto (\sigma, \chi_{\text{count}}(P))] & \\
\longrightarrow & (\text{M-Let, definition of residual}) \\
& \langle r\ \text{true} \mid \gamma' \mid \text{residual}(bs +\!\!+ [\text{true}]bs, P)\rangle \\
\longrightarrow & (\text{M-Resume},\ [\![r]\!]\gamma' = (\sigma, \chi_{\text{count}}(P))) \\
& \langle \textbf{return}\ \text{true} \mid \gamma' \mid (\sigma, \chi_{\text{count}}(P)) :: \text{residual}(bs +\!\!+ [\text{true}], P)\rangle
\end{array}
$$

We now use Lemma C.4 to reason about the progress of the predicate computation σ. There are two cases consider, either $1 + |bs| < n$ or $1 + |bs| = n$.

Case $1 + |bs| < n$. We obtain the following internal node configuration.

$$
\begin{array}{ll}
\longrightarrow^{\text{steps}(\tau)(bs +\!\!+ [\text{true}])} & (\text{by Lemma C.4}) \\
& \langle z\ V \mid \gamma'' \mid (\sigma', \chi_{\text{count}}(P)) :: \text{residual}(bs +\!\!+ [\text{true}], P)\rangle \\
\text{where } & z\ V = \text{comp}(\tau)(bs), \\
& \gamma'' = \text{env}(\tau)(bs +\!\!+ [\text{true}])[z \mapsto (\text{env}^{\perp}(P), \lambda\_.\textbf{do}\ \text{Branch}\ \langle\rangle)], \\
& ?k = \text{labs}(\tau)(bs +\!\!+ [\text{true}]), [\![V]\!]\gamma'' = k, \text{ and } \sigma' = \text{pure}(\tau)(bs +\!\!+ [\text{true}]) \\
= & (\text{definition of arrive when } 1 + |bs| < n) \\
& \text{arrive}(bs +\!\!+ [\text{true}], P) \\
\longrightarrow^{T(bs +\!\!+ [\text{true}], n)} & (\text{induction hypothesis}) \\
& \text{depart}(bs +\!\!+ [\text{true}], P) \\
= & (\text{definition of depart when } 1 + |bs| < n) \\
& \langle \textbf{return}\ i \mid \gamma \mid \text{residual}(bs +\!\!+ [\text{true}], P)\rangle \\
\text{where } & i = c(bs +\!\!+ [\text{true}] +\!\!+ [\text{true}]) + c(bs +\!\!+ [\text{true}] +\!\!+ [\text{false}]) \\
& \text{and } \gamma = \text{env}^{\uparrow}_{\text{false}}(bs +\!\!+ [\text{true}], P) \\
= & (\text{definition of residual and purecont}) \\
& \langle \textbf{return}\ i \mid \gamma \mid [((\gamma', x_{\text{true}}, \textbf{let}\ x_{\text{false}} \leftarrow r\ \text{false}\ \textbf{in}\ x_{\text{true}} + x_{\text{false}}) \\
& \qquad :: \text{purecont}(bs, P), \chi_{id})]\rangle \\
\text{where } & \gamma' = \text{env}^{\downarrow}_{\text{true}}(bs, P)
\end{array}
$$

$\longrightarrow$ (M-RetCont)

$\langle \textbf{let } x_{\mathsf{false}} \leftarrow r \text{ false } \textbf{in } x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs,P),\chi_{id})]\rangle$

where $\gamma'' = \gamma'[x_{\mathsf{true}} \mapsto [\![i]\!]\gamma']$

$\longrightarrow$ (M-Let)

$\langle r \text{ false} \mid \gamma'' \mid [((\gamma'', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs,P),\chi_{id})]\rangle$

$=$ (definition of purecont and residual)

$\langle r \text{ false} \mid \gamma'' \mid \mathsf{residual}(bs +\!\!+ [\mathsf{false}],P)\rangle$

$\longrightarrow$ (M-Resume)

$\langle \textbf{return } \mathsf{false} \mid \gamma'' \mid (\sigma,\chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs +\!\!+ [\mathsf{false}],P)\rangle$

where $\sigma = \mathsf{pure}(\tau)(bs)$

$\longrightarrow^{\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}])}$ (by Lemma C.4)

$\langle z\ V \mid \gamma \mid (\sigma,\chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs +\!\!+ [\mathsf{false}],P)\rangle$

where $z\ V = \mathsf{comp}(\tau)(bs)$,

$\gamma = \mathsf{env}(\tau)(bs +\!\!+ [\mathsf{false}])[q \mapsto (\mathsf{env}^{\perp}(P), \lambda\_.\textbf{do Branch } \langle\rangle)]$,

$?k = \mathsf{labs}(\tau)(bs +\!\!+ [\mathsf{false}]), [\![V]\!]\gamma = k,$ and $\sigma = \mathsf{pure}(\tau)(bs +\!\!+ [\mathsf{false}])$

$=$ (definition of arrive when $1 + |bs| < n$)

$\mathsf{arrive}(bs +\!\!+ [\mathsf{false}],P)$

$\longrightarrow^{T(bs +\!\!+ [\mathsf{false}],n)}$ (induction hypothesis)

$\mathsf{depart}(bs +\!\!+ [\mathsf{false}],P)$

$=$ (definition of depart when $1 + |bs| < n$)

$\langle \textbf{return } j \mid \gamma \mid \mathsf{residual}(bs +\!\!+ [\mathsf{false}],P)\rangle$

where $j = c(bs +\!\!+ [\mathsf{false}] +\!\!+ [\mathsf{true}]) + c(bs +\!\!+ [\mathsf{false}] +\!\!+ [\mathsf{false}])$

and $\gamma = \mathsf{env}^{\uparrow}_{\mathsf{false}}(bs +\!\!+ [\mathsf{false}],P)$

$=$ (definition of residual and purecont)

$\langle \textbf{return } j \mid \gamma \mid [((\gamma'', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs,P),\chi_{id})]\rangle$

$\longrightarrow$ (M-RetCont)

$\langle x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathsf{residual}(bs,P)\rangle$

$\longrightarrow$ (M-Plus)

$\langle \textbf{return } m \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathsf{residual}(bs,P)\rangle$

where $m = c(bs +\!\!+ [\mathsf{true}] +\!\!+ [\mathsf{true}]) + c(bs +\!\!+ [\mathsf{true}] +\!\!+ [\mathsf{false}])$

$+ c(bs +\!\!+ [\mathsf{false}] +\!\!+ [\mathsf{true}]) + c(bs +\!\!+ [\mathsf{false}] +\!\!+ [\mathsf{false}])$

$= c(bs +\!\!+ [\mathsf{true}]) + c(bs +\!\!+ [\mathsf{false}]) = c(bs) \leq 2^{n-|bs|}$

$=$ (definition of depart when $|bs| < n$)

$\mathsf{depart}(bs,P)$

Step analysis The total number of machine steps is given by

$$
\begin{aligned}
& 9+\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + T(bs +\!\!+ [\mathsf{true}], n) \\
& \quad +\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) + T(bs +\!\!+ [\mathsf{false}], n) \\
= \quad & \text{(reorder)} \\
& 9+T(bs +\!\!+ [\mathsf{true}], n) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
& \quad +\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(definition of } T\text{)} \\
& 9+9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 9*(2^{n-|bs+\!\!+[\mathsf{false}]|}-1) \\
& \quad +2^{n-|bs+\!\!+[\mathsf{true}]|+1} + 2^{n-|bs+\!\!+[\mathsf{false}]|+1} \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs+\!\!+[\mathsf{true}]|} \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}] +\!\!+ bs') \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs+\!\!+[\mathsf{false}]|} \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}] +\!\!+ bs') \\
& \quad +\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(simplify)} \\
& 9+9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 9*(2^{n-|bs+\!\!+[\mathsf{false}]|}-1) + 2^{n-|bs|+1} \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs+\!\!+[\mathsf{true}]|} \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}] +\!\!+ bs') \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs+\!\!+[\mathsf{false}]|} \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}] +\!\!+ bs') \\
& \quad +\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(merge sums)} \\
& 9+9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 9*(2^{n-|bs+\!\!+[\mathsf{false}]|}-1) + 2^{n-|bs|+1} \\
& \quad + \left(\sum_{bs' \in \mathsf{Addr}}^{2 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \right) \\
& \quad +\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(rewrite binary sum)} \\
& 9+9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 9*(2^{n-|bs+\!\!+[\mathsf{false}]|}-1) + 2^{n-|bs|+1} \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{2 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le 1} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
= \quad & \text{(merge sums)} \\
& 9+9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 9*(2^{n-|bs+\!\!+[\mathsf{true}]|}-1) + 2^{n-|bs|+1} \\
& \quad + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs')
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \text{(factoring)} \\
&\quad 9 + 2 * 9 * (2^{n-|bs|-1} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
&= \quad \text{(distribute)} \\
&\quad 9 + 9 * (2^{n-|bs|} - 2) + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
&= \quad \text{(distribute)} \\
&\quad 9 + 9 * 2^{n-|bs|} - 18 + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
&= \quad \text{(simplify)} \\
&\quad 9 * 2^{n-|bs|} - 9 + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
&= \quad \text{(factoring)} \\
&\quad 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
&= \quad \text{(definition of } T\text{)} \\
&\quad T(bs, n)
\end{aligned}
$$

Case $1 + |bs| = n$. We obtain the following configuration.

$$
\begin{aligned}
&\longrightarrow^{\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}])} \quad \text{(by Lemma C.4)} \\
&\quad \langle \mathbf{return}\ W \mid \gamma' \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs +\!\!+ [\mathsf{true}], P) \rangle \\
&\text{where} \quad \mathbf{return}\ W = \mathsf{comp}(\tau)(s +\!\!+ [\mathsf{true}]), !b = \mathsf{labs}(\tau)(bs +\!\!+ [\mathsf{true}]), \\
&\qquad \gamma' = \mathsf{env}(\tau)(bs +\!\!+ [\mathsf{true}]), \text{ and } [\![W]\!]\gamma' = b \\
&= \quad \text{(definition of arrive when } 1 + |bs| = n\text{)} \\
&\quad \mathsf{arrive}(bs +\!\!+ [\mathsf{true}], P) \\
&\longrightarrow^{T(bs +\!\!+ [\mathsf{true}], n)} \quad \text{(induction hypothesis)} \\
&\quad \mathsf{depart}(bs +\!\!+ [\mathsf{true}], P) \\
&= \quad \text{(definition of depart when } 1 + |bs| = n\text{)} \\
&\quad \langle \mathbf{return}\ i \mid \gamma \mid \mathsf{residual}(bs +\!\!+ [\mathsf{true}], P) \rangle \\
&\text{where } i = c(bs +\!\!+ [\mathsf{true}]) \le 2^{n-|bs +\!\!+ [\mathsf{true}]|} = 1 \text{ and } \gamma = \mathsf{env}^{\perp}(P) \\
&= \quad \text{(definition of residual and purecont)} \\
&\quad \langle \mathbf{return}\ i \mid \gamma \mid [((\gamma', x_{\mathsf{true}}, \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}) \\
&\qquad\qquad :: \mathsf{purecont}(bs, P), \chi_{id})] \rangle
\end{aligned}
$$

$\longrightarrow$ (M-RetCont)

$\langle \mathbf{let}\, x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\, x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'[x_{\mathsf{true}} \mapsto [\![i]\!]\gamma'] \mid [(\mathsf{purecont}(bs,P), \chi_{id})] \rangle$

$=$ (definition of $[\![-]\!]$ (1 value step))

$\langle \mathbf{let}\, x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\, x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs,P), \chi_{id})] \rangle$

where $\gamma'' = \gamma'[x_{\mathsf{true}} \mapsto i]$

$\longrightarrow$ (M-Let, definition of residual)

$\langle r\ \mathsf{false} \mid \gamma'' \mid \mathsf{residual}(bs +\!\!+ [\mathsf{false}], P) \rangle$

$\longrightarrow$ (M-Resume)

$\langle \mathbf{return}\ \mathsf{false} \mid \gamma'' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs +\!\!+ [\mathsf{false}], P) \rangle$

where $\sigma = \mathsf{pure}(\tau)(bs)$

$\longrightarrow^{\mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}])}$ (by Lemma C.4)

$\langle \mathbf{return}\ W \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs +\!\!+ [\mathsf{false}], P) \rangle$

where $\mathbf{return}\ W = \mathsf{comp}(\tau)(bs +\!\!+ [\mathsf{false}]), !b = \mathsf{labs}(\tau)(bs +\!\!+ [\mathsf{false}]),$
$\gamma = \mathsf{env}(\tau)(bs +\!\!+ [\mathsf{false}]),$ and $[\![W]\!]\gamma = b$

$=$ (definition of arrive when $1 + |bs| = n$)

$\mathsf{arrive}(bs +\!\!+ [\mathsf{false}], P)$

$\longrightarrow^{T(bs +\!\!+ [\mathsf{false}], n)}$ (induction hypothesis)

$\mathsf{depart}(bs +\!\!+ [\mathsf{false}], P)$

$=$ (definition of depart when $1 + |bs| = n$)

$\langle \mathbf{return}\ j \mid \gamma \mid \mathsf{residual}(bs +\!\!+ [\mathsf{false}], P) \rangle$

where $j = c(bs +\!\!+ [\mathsf{false}]) \leq 2^{n - |bs +\!\!+ [\mathsf{false}]|} = 1$ and $\gamma = \mathsf{env}^{\perp}(P)$

$=$ (definition of residual and purecont)

$\langle \mathbf{return}\ j \mid \gamma \mid [((\gamma', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs,P), \chi_{id})] \rangle$

where $\gamma' = \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs,P)$

$\longrightarrow$ (M-RetCont)

$\langle x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs,P), \chi_{id})] \rangle$

where $\gamma'' = \gamma'[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma'] = \gamma'[x_{\mathsf{false}} \mapsto j]$

$\longrightarrow$ (M-Plus)

$\langle \mathbf{return}\ m \mid \gamma'' \mid [(\mathsf{purecont}(bs,P), \chi_{id})] \rangle$

where $m = c(bs +\!\!+ [\mathsf{true}]) + c(bs +\!\!+ [\mathsf{false}]) \leq 2^{n - |bs|}$

$=$ (definition of residual and depart when $|bs| < n$)

$\mathsf{depart}(bs, P)$

Step analysis The total number of machine steps is given by

$$
\begin{aligned}
& 9 + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + T(bs +\!\!+ [\mathsf{true}], n) \\
& \quad + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) + T(bs +\!\!+ [\mathsf{false}], n) \\
= \quad & \text{(reorder)} \\
& 9 + T(bs +\!\!+ [\mathsf{true}], n) + T(bs +\!\!+ [\mathsf{false}], n) \\
& \quad + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(definition of } T \text{ when } |bs| + 1 = n) \\
& 9 + 2 + 2 + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(simplify)} \\
& 9 + 2^2 + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(rewrite } 2 = n - |bs| + 1) \\
& 9 + 2^{n-|bs|+1} + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(multiply by 1)} \\
& 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{true}]) \\
& \qquad\qquad + \mathsf{steps}(\tau)(bs +\!\!+ [\mathsf{false}]) \\
= \quad & \text{(rewrite binary sum)} \\
& 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|} + \sum_{bs' \in \mathsf{Addr}}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(\tau)(bs +\!\!+ bs') \\
= \quad & \text{(definition of } T) \\
& T(bs, n)
\end{aligned}
$$

□

The following theorem is a copy of Theorem 7.11.

Theorem C.6. *For all $n > 0$ and any n-standard predicate P it holds that*

1. *The program* effcount *is a generic count program*

2. *The runtime complexity of* effcount P *is given by the following formula:*

$$\sum_{bs \in \mathsf{Addr}}^{|bs| \le n} \mathsf{steps}(\mathcal{T}(P))(bs) + O(2^n)$$

Proof. The proof begins by direct calculation.

$$
\begin{array}{ll}
& \langle \mathsf{effcount}\, P \mid \emptyset \mid [([],\chi_{id})] \rangle \\
= & \quad \text{(definition of residual)} \\
& \langle \mathsf{effcount}\, P \mid \emptyset \mid \mathsf{residual}([],P) \rangle \\
\longrightarrow & \quad (\text{M-App}, [\![\mathsf{effcount}]\!]\emptyset = (\emptyset, \lambda pred.\cdots)) \\
& \langle \mathbf{handle}\ pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with}\ H_{\mathsf{count}} \mid \gamma \mid \mathsf{residual}([],P) \rangle \\
& \text{where } \gamma = \mathsf{env}^{\perp}(P) \\
\longrightarrow & \quad (\text{M-Handle}) \\
& \langle pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \mid \gamma \mid ([],(\gamma, H_{\mathsf{count}})) :: \mathsf{residual}([],P) \rangle \\
= & \quad \text{(definition of } \chi_{\mathsf{count}}) \\
& \langle pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \mid \gamma \mid ([],\chi_{\mathsf{count}}(P)) :: \mathsf{residual}([],P) \rangle \\
\longrightarrow^{\mathsf{steps}(\tau)([])} & \quad \text{(by Lemma C.4)} \\
& \langle z\ V \mid \gamma' \mid (\sigma,\chi_{\mathsf{count}}(P)) :: \mathsf{residual}([],P) \rangle \\
& \text{where } z\ V = \mathsf{comp}(\tau)(bs), \gamma' = \mathsf{env}(\tau)([])[q \mapsto (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)], \\
& \qquad ?k = \mathsf{labs}(\tau)([]), [\![V]\!]\gamma' = k, \text{ and } \sigma = \mathsf{pure}(\tau)([]) \\
= & \quad \text{(definition of arrive)} \\
& \mathsf{arrive}([],P) \\
\longrightarrow^{T([],n)} & \quad \text{(by Lemma C.5)} \\
& \mathsf{depart}([],P) \\
= & \quad \text{(definition of depart)} \\
& \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}([],P) \rangle \\
& \text{where } \gamma = \mathsf{env}^{\perp}(P) \text{ and } m = c([]) \leq 2^{n-|bs|} = 2^n \\
= & \quad \text{(definition of residual)} \\
& \langle \mathbf{return}\ m \mid \gamma \mid [([],\chi_{id})] \rangle \\
\longrightarrow & \quad (\text{M-Handle} - \text{Ret}, H^{\mathsf{val}}_{id} = \{\mathbf{return}\ x \mapsto \mathbf{return}\ x\}) \\
& \langle \mathbf{return}\ x \mid \emptyset[x \mapsto m] \mid [] \rangle
\end{array}
$$

Analysis The machine yields the value m. By Lemma C.5 it follows that $m \leq 2^{n-|bs|} = 2^{n-|[]|} = 2^n$. Furthermore, the total number of transitions used were

$$
\begin{array}{ll}
& 3 + \mathsf{steps}(\tau)([]) + T([],n) \\
= & \quad \text{(definition of } T) \\
& 3 + \mathsf{steps}(\tau)([]) + 9 * 2^n + 2^{n+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n} \mathsf{steps}(\tau)(bs')
\end{array}
$$

$$
\begin{aligned}
&= \quad \text{(simplify)} \\
&\quad 3 + \mathsf{steps}(\tau)([]) + 9 * 2^n + 2^{n+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n} \mathsf{steps}(\tau)(bs') \\
&= \quad \text{(reorder)} \\
&\quad 3 + \left(\sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n} \mathsf{steps}(\tau)(bs') \right) + \mathsf{steps}(\tau)([]) + 9 * 2^n + 2^{n+1} \\
&= \quad \text{(rewrite as unary sum)} \\
&\quad 3 + \left(\sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n} \mathsf{steps}(\tau)(bs') + \sum_{bs' \in \mathsf{Addr}}^{0 \le |bs'| \le 0} \mathsf{steps}(\tau)(bs') \right) + 9 * 2^n + 2^{n+1} \\
&= \quad \text{(merge sums)} \\
&\quad 3 + \left(\sum_{bs' \in \mathbb{B}^*}^{0 \le |bs'| \le n} \mathsf{steps}(\tau)(bs') \right) + 9 * 2^n + 2^{n+1} \\
&= \quad \text{(definition of } O\text{)} \\
&\quad \left(\sum_{bs' \in \mathbb{B}^*}^{0 \le |bs'| \le n} \mathsf{steps}(\tau)(bs') \right) + O(2^n)
\end{aligned}
$$

□

Appendix D

Berger count

In this appendix I will give a brief presentation of the BergerCount program alluded to in Section 7.4, in order to fill out our overall picture of the relationship between language expressivity and potential program efficiency.

Relation to prior work This appendix is imported from Appendix D of Hillerström et al. [124]. The code snippets in this appendix are based on an implementation of Berger count in SML/NJ written by John Longley. I have transcribed the code snippets, and in certain places tweaked it for presentation.

Berger's original program [19] introduced a remarkable search operator for predicates on *infinite* streams of booleans, and has played an important role in higher-order computability theory [180]. What we wish to highlight here is that if one applies the algorithm to predicates on *finite* boolean vectors, the resulting program, though no longer interesting from a computability perspective, still holds some interest from a complexity standpoint: indeed, it yields what seems to be the best available implementation of generic count within a PCF-style 'functional' language (provided one accepts the use of a primitive for call-by-need evaluation).

Let us consider an adaptation of Berger's search algorithm on finite spaces.

$$\mathsf{bestshot}_n : \mathsf{Predicate}_n \to \mathsf{Point}_n$$
$$\mathsf{bestshot}_n\ pred \stackrel{\text{def}}{=} \mathsf{bestshot}'_n\ pred\ []$$

$$\mathsf{bestshot}'_n : \mathsf{Predicate}_n \to \mathsf{List}_{\mathsf{Bool}} \to \mathsf{Point}_n$$
$$\mathsf{bestshot}'_n\ pred\ start \stackrel{\text{def}}{=} \mathbf{let}\ f \leftarrow \mathsf{memoise}\ (\lambda\langle\rangle.\mathsf{bestshot}''_n\ pred\ start)\ \mathbf{in}$$
$$\mathbf{return}\ (\lambda i.\mathbf{if}\ i < |start|\ \mathbf{then}\ start.i\ \mathbf{else}\ (f\ \langle\rangle).i)$$

$$\begin{aligned}
&\mathsf{bestshot}''_n : \mathsf{Predicate}_n \to \mathsf{List}_{\mathsf{Bool}} \to \mathsf{List}_{\mathsf{Bool}} \\
&\mathsf{bestshot}''_n\ \mathit{pred}\ \mathit{start} \stackrel{\text{def}}{=} \mathbf{if}\ |\mathit{start}| = n\ \mathbf{then\ return}\ \mathit{start} \\
&\qquad\qquad \mathbf{else\ let}\ f \leftarrow \mathsf{bestshot}'_n\ \mathit{pred}\ (\mathsf{append}\ \mathit{start}\ [\mathsf{true}])\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{if}\ \mathit{pred}\ f\ \mathbf{then\ return}\ [f\ 0, \ldots, f\ (n-1)] \\
&\qquad\qquad\quad \mathbf{else}\ \mathsf{bestshot}''_n\ \mathit{pred}\ (\mathsf{append}\ \mathit{start}\ [\mathsf{false}])
\end{aligned}$$

Given any n-standard predicate P the function $\mathsf{bestshot}_n$ returns a point satisfying P if one exists, or dummy point $\lambda i.\mathsf{false}$ if not. It is implemented by via two mutually recursive auxiliary functions whose workings are admittedly hard to elucidate in a few words. The function $\mathsf{bestshot}'_n$ is a generalisation of $\mathsf{bestshot}_n$ that makes a best shot at finding a point π satisfying given predicate and matching some specified list *start* in some initial segment of its components $[\pi(0), \ldots, \pi(i-1)]$. It works 'lazily', drawing its values from *start* wherever possible, and performing an actual search only when required. This actual search is undertaken by $\mathsf{bestshot}''_n$, which proceeds by first searching for a solution that extends the specified list with true; but if no such solution is forthcoming, it settles for false as the next component of the point being constructed. The whole procedure relies on a subtle combination of laziness, recursion and implicit nesting of calls to the provided predicate which means that the search is self-pruning in regions of the binary tree where the predicate only demands some initial segment $q\ 0, \ldots, q\ (i-1)$ of its argument q.

The above program makes use of an operation

$$\mathsf{memoise} : (1 \to \mathsf{List\ Bool}) \to (1 \to \mathsf{List\ Bool})$$

which transforms a given thunk into an equivalent 'memoised' version, i.e. one that caches its value after its first invocation and immediately returns this value on all subsequent invocations. Such an operation may readily be implemented in λ_{s}, or alternatively may simply be added as a primitive in its own right. The latter has the advantage that it preserves the purely 'functional' character of the language, in the sense that every program is observationally equivalent to a λ_{b} program, namely the one obtained by replacing memoise by the identity.

We now show how the above idea may be exploited to yield a generic count program (this development appears to be new).

$$\begin{aligned}
&\mathsf{BergerCount}_n : \mathsf{Predicate}_n \to \mathsf{Nat} \\
&\mathsf{BergerCount}_n\ \mathit{pred} \stackrel{\text{def}}{=} \mathsf{count}'_n\ \mathit{pred}\ []\ 0
\end{aligned}$$

$$\mathsf{count}'_n : \mathsf{Predicate}_n \to \mathsf{List}_{\mathsf{Bool}} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{count}'_n\ pred\ start\ acc \stackrel{\text{def}}{=} \textbf{if } |start| = n \textbf{ then } acc + (\textbf{if } pred\,(\lambda i.start.i) \textbf{ then return } 1 \textbf{ else return } 0) \textbf{ else let } f \leftarrow \mathsf{bestshot}'_n\ pred\ start \textbf{ in } \textbf{if } pred\ f \textbf{ then } \mathsf{count}''_n\ start\ [f\ 0, \ldots, f\ (n-1)]\ acc \textbf{ else return } acc$$

$$\mathsf{count}''_n : \mathsf{Predicate}_n \to \mathsf{List}_{\mathsf{Bool}} \to \mathsf{List}_{\mathsf{Bool}} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{count}''_n\ pred\ start\ leftmost\ acc \stackrel{\text{def}}{=} \textbf{if } |start| = n \textbf{ then } acc + 1 \textbf{ else let } b \leftarrow leftmost.|start| \textbf{ in } \textbf{let } acc' \leftarrow \mathsf{count}''_n\ pred\ (\mathsf{append}\ start\ [b])\ leftmost\ acc \textbf{ in } \textbf{if } b \textbf{ then } \mathsf{count}'_n\ pred\ (\mathsf{append}\ start\ [\mathsf{false}])\ acc' \textbf{ else return } acc'$$

Again, $\mathsf{BergerCount}_n$ is implemented by means of two mutually recursive auxiliary functions. The function count'_n counts the solutions to the provided predicate *pred* that start with the specified list of booleans, adding their number to a previously accumulated total given by *acc*. The function count''_n does the same thing, but exploiting the knowledge that a best shot at the 'leftmost' solution to P within this subtree has already been computed. (We are visualising n-points as forming a binary tree with true to the left of false at each fork.) Thus, count''_n will not re-examine the portion of the subtree to the left of this candidate solution, but rather will start at this solution and work rightward.

This gives rise to an n-count program that can work efficiently on predicates that tend to 'fail fast': more specifically, predicates P that inspect the components of their argument q in order $q\ 0$, $q\ 1$, $q\ 2$, ..., and which are frequently able to return false after inspecting just a small number of these components. Generalising our program from binary to k-ary branching trees, we see that the n-queens problem provides a typical example: most points in the space can be seen *not* to be solutions by inspecting just the first few components. Our experimental results in Section 7.6 attest to the viability of this approach and its overwhelming superiority over the naïve functional method.

By contrast, the above program is *not* able to exploit parts of the tree where our predicate 'succeeds fast', i.e. returns true after seeing just a few components. Unlike the effectful count program of Section 7.3.4, which may sometimes add 2^{n-d} to the count in a single step, the Berger approach can only count solutions one at a time. Thus, supposing P is an n-standard predicate the evaluation of $\mathsf{count}_n\ P$ that returns a natural number c must take time $\Omega(c)$. These observations informally indicate the

likely extent of the efficiency gap between effectful and purely functional computation when it comes to non-n-standard predicates.

Bibliography

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19. ACM, 2003.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. *BRICS Report Series*, 10(14), 2003.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90(5):223–232, 2004.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.*, 342(1):149–172, 2005.

[5] Danel Ahman. *Fibred Computational Effects*. PhD thesis, The University of Edinburgh, Scotland, UK, 2017.

[6] Danel Ahman and Matija Pretnar. Asynchronous effects. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.

[7] Danel Ahman, Amal Ahmed, Sam Lindley, and Andreas Rossberg. Scalable Handling of Effects (Dagstuhl Seminar 21292). *Dagstuhl Reports*, 11(6):54–81, 2021.

[8] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[9] Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In *PLILP*, volume 528 of *LNCS*, pages 1–13. Springer, 1991.

[10] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[11] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *APLAS*, volume 4807 of *LNCS*, pages 239–254. Springer, 2007.

[12] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, Irving Ziller, Robert A. Hughes, and R. Nutt. The FORTRAN automatic

coding system. In *IRE-AIEE-ACM Computer Conference (Western)*, pages 188–198. ACM, 1957.

[13] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[14] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.*, 10(4), 2014.

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

[16] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discret. Math.*, 309(1):1–31, 2009.

[17] Nick Benton and Andrew Kennedy. Monads, effects and transformations. *Electron. Notes Theor. Comput. Sci.*, 26:3–20, 1999.

[18] Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

[19] Ulrich Berger. *Totale Objekte und Mengen in der Bereichstheorie*. PhD thesis, Ludwig Maximillians-Universtität, Munich, 1990.

[20] Bernard Berthomieu and Camille le Moniès de Sagazan. A calculus of tagged types, with applications to process languages. In *Workshop on Types for Program Analysis*, 1995.

[21] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP*, volume 8586 of *LNCS*, pages 257–281. Springer, 2014.

[22] Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1):6, 2007.

[23] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. *BRICS Report Series*, 10(41), 2003.

[24] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Log. Methods Comput. Sci.*, 1(2), 2005.

[25] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. *Inf. Process. Lett.*, 96(1):7–17, 2005.

[26] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL):8:1–8:30, 2018.

[27] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL):6:1–6:28, 2019.

[28] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, 4(POPL):48:1–48:29, 2020.

[29] Richard Bird, Geraint Jones, and Oege de Moor. More haste less speed: lazy versus eager evaluation. *J. Funct. Program.*, 7(5):541–547, 1997.

[30] Richard S. Bird. Functional pearl: A program to solve sudoku. *J. Funct. Program.*, 16(6):671–679, 2006.

[31] Francis Borceux. *Handbook of Categorical Algebra*, volume 1 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

[32] Oliver Bracevac. *Event Correlation with Algebraic Effects - Theory, Design and Implementation*. PhD thesis, Technische Universität Damstadt, Germany, 2019.

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

[34] Jonathan Immanuel Brachthäuser. *Design and Implementation of Effect Handlers for Object-Oriented Programming Languages*. PhD thesis, University of Tübingen, Germany, 2020.

[35] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: extensible algebraic effects in scala (short paper). In *SCALA@SPLASH*, pages 67–72. ACM, 2017.

[36] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect handlers for the masses. *Proc. ACM Program. Lang.*, 2(OOPSLA):111:1–111:27, 2018.

[37] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *J. Funct. Program.*, 30:e8, 2020.

[38] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *J. Funct. Program.*, 30:e8, 2020.

[39] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020.

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

[41] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *PLDI*, pages 99–107. ACM, 1996.

[42] Rod M. Burstall. Writing search algorithms in functional form. In *Machine Intelligence*, volume 3, pages 373–385. Edinburgh University Press, 1968.

[43] Robert Cartwright and Matthias Felleisen. Observable sequentiality and full abstraction. In *POPL*, pages 328–342. ACM Press, 1992.

[44] Paul Chiusano et al. Unison language reference, 2020. Revision cb9a198.

[45] Alonzo Church. A set of postulates for the foundation of logic. In *Annals of Mathematics*, volume 33, pages 346–366, 1932.

[46] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton University Press, USA, 1941.

[47] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3): 313–323, 1999.

[48] William D. Clinger. Proper tail recursion and space efficiency. In *PLDI*, pages 174–185. ACM, 1998.

[49] William D. Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *LISP and Functional Programming*, pages 124–131. ACM, 1988.

[50] William D. Clinger et al. The revised revised report on Scheme or an UnCommon Lisp. Technical Report AIM-848, MIT, August 1985.

[51] Lukas Convent. Enhancing a modular effectful programming language. Master's thesis, School of Informatics, The University of Edinburgh, Scotland, UK, 2017.

[52] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020.

[53] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296. Springer, 2006.

[54] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 3rd edition, 2009.

[55] Robbie Daniels. Efficient generic searches and programming language expressivity. Master's thesis, School of Informatics, the University of Edinburgh, Scotland, August 2016.

[56] Olivier Danvy. Memory allocation and higher-order functions. In *PLDI*, pages 241–252. ACM, 1987.

[57] Olivier Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.

[58] Olivier Danvy. A rational deconstruction of landin's SECD machine. In *IFL*, volume 3474 of *LNCS*, pages 52–71. Springer, 2004.

[59] Olivier Danvy. On evaluation contexts, continuations, and the rest of computation. In *CW*, number CSR-04-1, Birmingham B15 2TT, United Kingdom, 2004.

[60] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. DSc thesis, Aarhus University, Aarhus, Denmark, 2006.

[61] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[62] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[63] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[64] Olivier Danvy and Kevin Millikin. A rational deconstruction of landin's SECD machine with the J operator. *Log. Methods Comput. Sci.*, 4(4), 2008.

[65] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, 2009.

[66] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *PPDP*, pages 162–174. ACM, 2001.

[67] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003.

[68] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. *Inf. Process. Lett.*, 94(5):217–224, 2005.

[69] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009.

[70] Stephen Dolan, Leo White, and Anil Madhavapeddy. Multicore OCaml. OCaml Workshop, 2014.

[71] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. OCaml Workshop, 2015.

[72] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *TFP*, volume 10788 of *LNCS*, pages 98–117. Springer, 2017.

[73] Rémi Douence and Pascal Fradet. The next 700 krivine machines. *High. Order Symb. Comput.*, 20(3):237–255, 2007.

[74] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Comput. Lang.*, 14(2):109–123, 1989.

[75] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007.

[76] Martín Hötzel Escardó. Infinite sets that admit fast exhaustive search. In *LICS*, pages 443–452. IEEE Computer Society, 2007.

[77] David A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, New York, USA, 1995.

[78] Kavon Farvardin and John H. Reppy. From folklore to fact: comparing implementations of stacks and continuations. In *PLDI*, pages 75–90. ACM, 2020.

[79] Matthias Felleisen. *The Calculi of Lambda-nu-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indianapolis, IN, USA, 1987. AAI8727494.

[80] Matthias Felleisen. Reflections on landins's j-operator: A partly historical note. *Comput. Lang.*, 12(3/4):197–207, 1987.

[81] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190. ACM Press, 1988.

[82] Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3):35–75, 1991. Revised version.

[83] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1987.

[84] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. Reasoning with continuations. In *LICS*, pages 131–141. IEEE Computer Society, 1986.

[85] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana 47405, USA, February 1987.

[86] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *LISP and Functional Programming*, pages 52–62. ACM, 1988.

[87] Andrzej Filinski. Representing monads. In *POPL*, pages 446–457. ACM Press, 1994.

[88] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.

[89] Andrzej Filinski. Representing layered monads. In *POPL*, pages 175–188. ACM, 1999.

[90] Andrzej Filinski. Monads in action. In *POPL*, pages 483–494. ACM, 2010.

[91] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

[92] Matthew Flatt and R. Kent Dybvig. Compiler and runtime support for continuation marks. In *PLDI*, pages 45–58. ACM, 2020.

[93] Matthew Flatt and PLT. The Racket reference (version 7.9), November 2020.

[94] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *ICFP*, pages 165–176. ACM, 2007.

[95] Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. Rebuilding racket on chez scheme (experience report). *Proc. ACM Program. Lang.*, 3(ICFP): 78:1–78:15, 2019.

[96] Matthew Fluet et al. MLton documentation, January 2014.

[97] Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4): 81–82, 1990.

[98] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *PACMPL*, 1(ICFP), September 2017.

[99] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.*, 29:e15, 2019.

[100] Simon Fowler. *Typed concurrent functional programming with channels, actors and sessions*. PhD thesis, The University of Edinburgh, Scotland, UK, 2019.

[101] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019.

[102] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *POPL*, pages 245–254. ACM Press, 1985.

[103] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical report 546, Computer Science Department, Indiana University, Bloomington, Indiana 47405, USA, 200.

[104] Daniel P. Friedman, Christopher T Haynes, and Eugene Kohlbecker. Programming with continuations. In Peter Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. ISBN 978-3-642-46490-4.

[105] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *ICFP*, pages 18–27. ACM, 1999.

[106] Bram Geron. *Defined Algebraic Operations*. PhD thesis, University of Birmingham, England, UK, 2019.

[107] Jeremy Gibbons. Unifying theories of programming with monads. In *UTP*, volume 7681 of *LNCS*, pages 23–67. Springer, 2012.

[108] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *ICFP*, pages 2–14. ACM, 2011.

[109] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, France, 1972.

[110] Maria I. Gorinova, Dave Moore, and Matthew D. Hoffman. Automatic reparameterisation of probabilistic programs. In *ICML*, volume 119, pages 3648–3657. PMLR, 2020.

[111] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *FPCA*, pages 12–23. ACM, 1995.

[112] William L. Harrison. The essence of multitasking. In *AMAST*, volume 4019 of *LNCS*, pages 158–172. Springer, 2006.

[113] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *LISP and Functional Programming*, pages 18–24. ACM, 1984.

[114] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.*, 9(4):582–598, 1987.

[115] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.

[116] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP*, pages 128–136. ACM, 1990.

[117] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *PLDI*, pages 66–77. ACM, 1990.

[118] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson III. Subcontinuations. *LISP Symb. Comput.*, 7(1):83–110, 1994.

[119] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016.

[120] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *APLAS*, volume 11275 of *LNCS*, pages 415–435. Springer, 2018.

[121] Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *FSCD*, volume 84 of *LIPIcs*, pages 18:1–18:19, 2017.

[122] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020.

[123] Daniel Hillerström, Sam Lindley, and John Longley. Effects for efficiency: Asymptotic speedup with first-class control. *Proc. ACM Program. Lang.*, 4 (ICFP):100:1–100:29, 2020.

[124] Daniel Hillerström, Sam Lindley, and John Longley. Effects for efficiency: Asymptotic speedup with first-class control. *CoRR*, abs/2007.00605, 2020.

[125] Daniel Hillerström. Handlers for algebraic effects in Links. Master's thesis, School of Informatics, The University of Edinburgh, Scotland, UK, August 2015.

[126] Daniel Hillerström. Compilation of effect handlers and their applications in concurrency. MSc(R) thesis, School of Informatics, The University of Edinburgh, Scotland, UK, 2016.

[127] Daniel Hillerström. Composing UNIX with Effect Handlers: A Case Study in Effect Handler Oriented Programming (extended abstract). ML Workshop, 2021.

[128] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the AMS*, 146:29–60, 1969.

[129] William A. Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[130] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[131] John Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986.

[132] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[133] Graham Hutton and Joel J. Wright. Calculating an exceptional machine. In *Trends in Functional Programming*, volume 5 of *Trends in Functional Programming*, pages 49–64. Intellect, 2004.

[134] Martin Hyland. *Game Semantics*, pages 131–184. Cambridge University Press, 1997.

[135] Roshan P James and Amr Sabry. Yield: Mainstream delimited continuations. In *TPDC*, 2011.

[136] Neil Jones. The expressive power of higher-order types, or, life without CONS. *J. Funct. Program.*, 11:5–94, 2001.

[137] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL*, pages 71–84. ACM Press, 1993.

[138] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language, 1999.

[139] Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *FLOPS*, volume 4989 of *LNCS*, pages 239–254. Springer, 2008.

[140] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage - staging with delimited control. *J. Funct. Program.*, 21(6):617–662, 2011.

[141] Ohad Kammar. *Algebraic theory of type-and-effect systems*. PhD thesis, The University of Edinburgh, Scotland, UK, 2014.

[142] Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*, pages 349–360. ACM, 2012.

[143] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, pages 145–158. ACM, 2013.

[144] Andrew Kennedy. Compiling with continuations, continued. In *ICFP*, pages 177–190. ACM, 2007.

[145] David J. King and Philip Wadler. Combining monads. In *Functional Programming*, Workshops in Computing, pages 134–143. Springer, 1992.

[146] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012.

[147] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Haskell*, pages 94–105. ACM, 2015.

[148] Oleg Kiselyov and Chung-chieh Shan. A substructural type system for delimited continuations. In *TLCA*, volume 4583 of *LNCS*, pages 223–239. Springer, 2007.

[149] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In *CONTEXT*, volume 4635 of *LNCS*, pages 291–302. Springer, 2007.

[150] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *DSL*, volume 5658 of *LNCS*, pages 360–384. Springer, 2009.

[151] Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. ML Workshop, 2016.

[152] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). pages 192–203, 2005.

[153] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *ICFP*, pages 26–37. ACM, 2006.

[154] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

[155] Ikuo Kobori, Yukiyoshi Kameyama, and Oleg Kiselyov. Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In *WoC*, volume 212 of *EPTCS*, pages 36–52, 2015.

[156] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *High. Order Symb. Comput.*, 20(3):199–207, 2007.

[157] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *LISP Symb. Comput.*, 10(3):223–236, 1998.

[158] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 01 1964. ISSN 0010-4620.

[159] Peter J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.

[160] Peter J. Landin. A correspondence between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM*, 8(3):158–167, 1965.

[161] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Reprint of UNIVAC Systems Programming Research report (August 1965).

[162] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *LISP and Functional Programming*, pages 227–238. ACM, 1994.

[163] Daan Leijen. Extensible records with scoped labels. In *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 179–194. Intellect, 2005.

[164] Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, volume 153 of *EPTCS*, pages 100–126, 2014.

[165] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *POPL*, pages 486–499. ACM, 2017.

[166] Daan Leijen. Structured asynchrony with algebraic effects. In *TyDe@ICFP*, pages 16–29. ACM, 2017.

[167] Daan Leijen. Implementing algebraic effects in C - "monads for free in C". In *APLAS*, volume 10695 of *Lecture Notes in Computer Science*, pages 339–363. Springer, 2017.

[168] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[169] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml System Release 4.11: Documentation and user's manual*. INRIA, August 2020.

[170] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.

[171] Bil Lewis, Dan LaLiberte, Richard Stallman, the GNU Manual Group, and et al. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, Boston, MA, USA, 2020. ISBN 1882114744. For Emacs version 27.1.

[172] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, pages 189–199. ACM, 2007.

[173] Sam Lindley and James Cheney. Row-based effect types for database integration. In *TLDI*, pages 91–102. ACM, 2012.

[174] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *POPL*, pages 500–514. ACM, 2017.

[175] John Longley. When is a functional program not a functional program? In *ICFP*, pages 1–7. ACM, 1999.

[176] John Longley. Universal types and what they are good for. In *Domain Theory, Logic and Computation*, pages 25–63. Springer Netherlands, 2003.

[177] John Longley. Some programming languages suggested by game models (extended abstract). In *MFPS*, volume 249, pages 117–134. Elsevier, 2009.

[178] John Longley. The recursion hierarchy for PCF is strict. *Logical Methods in Comput. Sci.*, 14(3:8):1–51, 2018.

[179] John Longley. Bar recursion is not computable via iteration. *Computability*, 8 (2):119–153, 2019.

[180] John Longley and Dag Normann. *Higher-Order Computability*. Theory and Applications of Computability. Springer, 2015.

[181] John Longley and Nicholas Wolverson. Eriskay: a programming language based on game semantics. Presented at GaLoP III, April 2008.

[182] John M. Lucassen. *Types and Effects — Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT, USA, 1987.

[183] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57. ACM Press, 1988.

[184] Žiga Lukšič. *Applications of algebraic effect theories*. PhD thesis, University of Ljubljana, Slovenia, 2020.

[185] Žiga Lukšič and Matija Pretnar. Local algebraic effect theories. *J. Funct. Program.*, 30:e13, 2020.

[186] David MacKenzie et al. *GNU Coreutils*. Free Software Foundation, February 2020. For version 8.32.

[187] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5. Springer-Verlag, New York, 1971.

[188] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

[189] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[190] Marek Materzok and Dariusz Biernacki. A dynamic interpretation of the CPS hierarchy. In *APLAS*, volume 7705 of *LNCS*, pages 296–311. Springer, 2012.

[191] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[192] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[193] Craig McLaughlin. *Relational Reasoning for Effects and Handlers*. PhD thesis, The University of Edinburgh, Scotland, UK, 2020.

[194] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.

[195] Paul-André Melliès. Local states in string diagrams. In *RTA-TLCA*, volume 8560 of *LNCS*, pages 334–348. Springer, 2014.

[196] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In *Logic of Programs*, volume 193 of *LNCS*, pages 219–224. Springer, 1985.

[197] Robin Milner. Processes: A mathematical model of computing agents. In *Studies in Logic and the Foundations of Mathematics*, pages 157–173. Elsevier, 1975.

[198] Robin Milner. Fully abstract models of typed λ-calculi. *Theor. Comput. Sci.*, 4 (1):1–22, 1977.

[199] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[200] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[201] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[202] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, The University of Edinburgh, Scotland, UK, 1990.

[203] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1): 55–92, 1991.

[204] David A. Moon. MACLISP reference manual (revision 0). Project MAC – MIT, April 1974.

[205] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations - A duumvirate of control operators. In *PLILP*, volume 844 of *LNCS*, pages 182–197. Springer, 1994.

[206] J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019.

[207] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999.

[208] Junpei Oishi and Yukiyoshi Kameyama. Staging with control: type-safe multi-stage programming with control operators. In *GPCE*, pages 29–40. ACM, 2017.

[209] Chris Okasaki. Functional pearl: Even higher-order functions for parsing. *J. Funct. Program.*, 8(2):195–199, 1998.

[210] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[211] Nikolaos S. Papspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Greece, 2001.

[212] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[213] Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. Handling local state with global state. In *MPC*, volume 11825 of *LNCS*, pages 18–44. Springer, 2019.

[214] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *ICFP*, pages 216–227. ACM, 2005.

[215] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[216] Nicholas Pippenger. Pure versus impure lisp. In *POPL*, pages 104–109. ACM, 1996.

[217] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *FSCD*, volume 131 of *LIPIcs*, pages 30:1–30:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[218] Ken Pizzini, Paolo Bonzini, Jim Meyering, and Assaf Gordon. *GNU sed, a stream editor*. Free Software Foundation, January 2020. For version 4.8.

[219] Gordon Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[220] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[221] Gordon D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

[222] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Aarhus, Denmark, September 1981.

[223] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[224] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *FoSSaCS*, volume 2030 of *LNCS*, pages 1–24. Springer, 2001.

[225] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

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

[227] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP*, volume 5502 of *LNCS*, pages 80–94. Springer, 2009.

[228] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[229] Leo Poulson. Asynchronous effect handling. Master's thesis, School of Informatics, The University of Edinburgh, Scotland, UK, 2020.

[230] Matija Pretnar. *Logic and handling of algebraic effects*. PhD thesis, The University of Edinburgh, Scotland, UK, 2010.

[231] Matija Pretnar. Inferring algebraic effects. *Log. Methods Comput. Sci.*, 10(3), 2014.

[232] Matija Pretnar. An introduction to algebraic effects and handlers. *Electr. Notes Theor. Comput. Sci.*, 319:19–35, 2015. Invited tutorial paper.

[233] Christian Queinnec. Continuations and web servers. *High. Order Symb. Comput.*, 17(4):277–295, 2004.

[234] Christian Queinnec and Bernard P. Serpette. A dynamic extent control operator for partial continuations. In *POPL*, pages 174–184. ACM Press, 1991.

[235] Eric Steven Raymond. *The Art of UNIX Programming*. Pearson Education, 2003. ISBN 0131429019.

[236] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report RR-1869, INRIA, 1993.

[237] Didier Rémy. *Type Inference for Records in Natural Extension of ML*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.

[238] John C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *LNCS*, pages 408–423. Springer, 1974.

[239] John C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6 (3-4):233–248, 1993.

[240] John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. This paper originally appeared in the Proceedings of the ACM National Conference, volume 2, August, 1972, pages 717–740.

[241] Dennis Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

[242] Amr Hany Saleh. *Efficient Algebraic Effect Handlers*. PhD thesis, KU Leuven, Belgium, 2019.

[243] Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.*, 4 (ICFP):93:1–93:28, 2020.

[244] Dana Scott. A system of functional abstraction. Lectures delivered at University of California, Berkeley, California, USA, 1962/63.

[245] Dana Scott and Christopher Strachey. *Proceedings of the Symposium on Computers and Automata*, 21, 1971.

[246] William Shakespeare. The Tragedy of Hamlet, Prince of Denmark, 1564-1616.

[247] Chung-chieh Shan. Shift to control. ACM SIGPLAN Scheme Workshop, 2004.

[248] Chung-chieh Shan. A static simulation of dynamic delimited control. *High. Order Symb. Comput.*, 20(4):371–401, 2007.

[249] Alex K. Simpson. Lazy functional algorithms for exact real functionals. In *MFCS*, volume 1450 of *LNCS*, pages 456–464. Springer, 1998.

[250] Dorai Sitaram. Handling control. In *PLDI*, pages 147–155. ACM, 1993.

[251] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP Symb. Comput.*, 3(1):67–99, 1990.

[252] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. *CoRR*, abs/2104.00250, 2021.

[253] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robert Bruce Findler, and Jacob Matthews. *Revised6 Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[254] Guy L. Steele. RABBIT: A compiler for SCHEME (a study in compiler optimization). Technical Report TR-474, MIT, Cambridge, Massachusetts, USA, May 1978.

[255] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, University of Oxford, January 1974.

[256] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *High. Order Symb. Comput.*, 13(1/2): 135–152, 2000.

[257] Gerald J. Sussman and Guy L. Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Memo No. 349, MIT, December 1975.

[258] Wouter Swierstra. From mathematics to abstract machine: A formal derivation of an executable krivine machine. In *MSFP*, volume 76 of *EPTCS*, pages 163–177, 2012.

[259] Don Syme, Tomas Petricek, and Dmitry Lomov. The f# asynchronous programming model. In *PADL*, volume 6539 of *LNCS*, pages 175–189. Springer, 2011.

[260] Hayo Thielecke. An introduction to Landin's "a generalization of jumps and labels". *High. Order Symb. Comput.*, 11(2):117–123, 1998.

[261] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *High. Order Symb. Comput.*, 15(2-3):141–160, 2002.

[262] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, pages 188–201. ACM Press, 1994.

[263] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[264] John Ronald Reuel Tolkien. *The lord of the rings: Part 1: The fellowship of the ring*. Allen and Unwin, 1954.

[265] Niki Vazou and Daan Leijen. From monads to effects and back. In *PADL*, volume 9585 of *LNCS*, pages 169–186. Springer, 2016.

[266] Philip Wadler. Theorems for free! In *FPCA*, pages 347–359. ACM, 1989.

[267] Philip Wadler. The essence of functional programming. In *POPL*, pages 1–14. ACM, 1992.

[268] Philip Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992.

[269] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

[270] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.

[271] Mitchell Wand. Continuation-based multiprocessing. In *LISP Conference*, pages 19–28. ACM, 1980.

[272] Mitchell Wand. Complete type inference for simple objects. In *LICS*, pages 37–44. IEEE Computer Society, 1987.

[273] Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019.

[274] Guannan Wei, Oliver Bracevac, Shangyin Tan, and Tiark Rompf. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.*, 4(OOPSLA):164:1–164:33, 2020.

[275] Nicolas Wu and Tom Schrijvers. Fusion for free - efficient algebraic effect handlers. In *MPC*, volume 9129 of *LNCS*, pages 302–322. Springer, 2015.

[276] Ningning Xie and Daan Leijen. Effect handlers in Haskell, evidently. In *Haskell@ICFP*, pages 95–108. ACM, 2020.

[277] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proc. ACM Program. Lang.*, 4(ICFP):99:1–99:29, 2020.

[278] Jeremy Yallop. *Abstraction for web programming*. PhD thesis, The University of Edinburgh, UK, 2010.

[279] Jeremy Yallop. Staged generic programming. *PACMPL*, 1(ICFP):29:1–29:29, 2017.