The Theory and Practice of Programming Languages with Delimited Continuations

Dariusz Biernacki

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

The Theory and Practice of Programming Languages with Delimited Continuations

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
Ph.D. Degree

by
Dariusz Biernacki
October 12, 2005

Abstract

This dissertation presents a study of functional programming languages with first-class delimited continuations. We focus mainly on theoretical and practical aspects of Danvy and Filinski's hierarchy of static delimited-control operators $\texttt{shift}_n$ and $\texttt{reset}_n$, and of Felleisen's dynamic delimited-control operators `control` and `prompt`.

Our study uses the traditional means of specifying semantics of functional languages: higher-order interpreters and abstract machines. We therefore first investigate the essence of interpreters and abstract machines, and we present a systematic and constructive derivation method connecting them. The functional correspondence we propose relates known higher-order interpreters and abstract machines that were invented independently and were believed to be disconnected, and it allows to construct new ones. We treat functional and logic programming languages.

Having established our enabling technology, we turn to the main topic of this dissertation, i.e., delimited continuations. We use continuation-passing style (CPS) as a guide to build an operational foundation for static delimited continuations in the CPS hierarchy: We derive an abstract machine and a reduction semantics for the λ-calculus extended with control operators $\texttt{shift}_n$ and $\texttt{reset}_n$. We also characterize in what sense dynamic-control operators are incompatible with traditional CPS. Additionally, we present new applications of delimited continuations in the CPS hierarchy.

We then formalize and prove the folklore theorem that the static delimited-control operators `shift` and `reset` can be simulated in terms of the dynamic delimited-control operators `control` and `prompt`.

We also show that breadth-first traversal exploits the difference between `shift` and `control`. For the last 15 years, this difference has been repeatedly mentioned in the literature but it has only been illustrated with one-line toy examples. Breadth-first traversal fills this vacuum. We point out how static delimited continuations naturally give rise to the notion of control stack whereas dynamic delimited continuations can account for a notion of 'control queue.'

Finally, we design a 'dynamic continuation-passing style' for dynamic delimited continuations. This dynamic CPS underlies a new abstract machine, an interpreter, a CPS transformation, and a computational monad for dynamic delimited continuations. We also present a new simulation of dynamic delimited continuations in terms of static ones.

Acknowledgements

First of all, I would like to express my deepest gratitude to my Ph.D. advisor Olivier Danvy who has been an extraordinary teacher, mentor, co-author, and friend. I have immensely benefited from his extensive scientific expertise and wise instruction. This thesis could not be written without Olivier's guidance, support and encouragement.

I also thank Peter Mosses for leading my first steps at BRICS and for being a kind and influential advisor and teacher.

I am very grateful to John Reynolds for making our study visit at Carnegie Mellon University possible. I also thank John and Mary for their hospitality during our stay in Pittsburgh.

I have greatly benefited from the courses taught by Olivier Danvy, Ulrich Kohlenbach, Peter Bro Miltersen, Peter Mosses, and John Reynolds at BRICS, as well as by Steve Awodey and Frank Pfenning at CMU.

My studies would not be possible without the excellent working environment at BRICS. My special thanks go to Mogens Nielsen, Uffe Engberg, Karen Kjær Møller, Lene Kjeldsteen, Hanne Friis Jensen, Ingrid Larsen, and Ellen Lindstrøm. I also thank the Danish National Research Foundation for funding my Ph.D. studies at BRICS.

I would like to thank Neil Jones and Yukiyoshi Kameyama for evaluating this dissertation. It is my honor to have them on the Ph.D. committee.

I thank Mads Sig Ager, Małgorzata Biernacka, Olivier Danvy, Jan Midtgaard, and Kevin Millikin for fruitful co-operation on our joint projects and for insightful comments on various drafts of this dissertation. I also thank Julia Lawall for many helpful comments on the articles comprising this thesis.

Many thanks to my friends and colleagues in Århus for the enjoyable time we have had during the last 3 years. Special thanks go to Mads Sig Ager and Jan Midtgaard for their help and encouragement, and for our numerous discussions.

I am grateful to Olaf Bojen for making his home our home. It has been my privilege and a real pleasure to live in Århus for all these years.

Many thanks also to my friends in Lublin whose priceless support accompanied me for the last 3 years. In particular, I would like to thank Łukasz Studziński for his e-mail on August 20, 2002, and for his continued interest and encouragement.

I am deeply indebted to my Mom, Dad, grandmother, and Gosia's mom for their enormous support that has been vital to me. I am especially grateful to my dear Mom whose care and wisdom led me to where I am today.

Finally, I want to thank my wonderful wife Gosia for her love, for her constant support, and for her unwavering faith in me: I would not have done it without You.

Dariusz Biernacki,
Århus, October 12, 2005.

Contents

Part I

Introduction and Background

Chapter 1

Introduction

The theory and practice of programming languages indicate that first-class continuations provide an efficient mechanism for expressing a variety of computational phenomena. First-class continuations enrich a functional language with control effects that otherwise could be only obtained by reformulating the entire program into continuation-passing style (CPS), with an explicit representation of continuations. Control operators for undelimited continuations, as present in modern higher-order functional languages such as Scheme or Standard ML of New Jersey, have become an accepted element in the toolbox of functional programmers. The numerous applications of first-class undelimited continuations include modelling control effects such as non-local exits and backtracking as well as modelling concurrent primitives such as coroutines, lightweight processes, and engines. The theoretical and practical nuances of undelimited continuations appear to be fairly well understood by now, from both the computational and the mathematical perspective.

A class of continuations that go beyond undelimited continuations, usually allowing for more direct solutions, are delimited continuations. Delimited continuations have proven useful in different areas of computer science, such as non-deterministic programming, partial evaluation, concurrency, and mobile computing, to name but a few. Although they have been a topic of active research since they were introduced in the late 1980s, with an upsurge of interest in the last several years, they still seem largely unexplored compared to undelimited continuations.

This dissertation presents a study of programming languages equipped with control operators for delimited continuations. The goal of our work is to fill several remaining gaps in this area, and by doing so, to deepen our understanding of delimited continuations and to identify their new areas of application. To this end, we study a range of aspects of such languages, from theory to practice and implementations. We present and connect different styles of semantics for delimited continuations, discuss the induced program transformations and their role in practical programming, illustrate the theory with meaningful programming examples, and propose new applications and implementations. The enabling technology for our study is based on traditional means of specifying programming languages with control operators, i.e., interpreters and abstract machines. Therefore, we first investigate them and their connections

in a general setting. In particular we are interested in interpreters and abstract machines for functional and logic programming languages. (Incidentally, the logic programming language Prolog also comes with its own control operators, `fail` and `cut`, which we characterize in terms of continuations in Chapter 6.) Our tools to study the operational aspects of delimited continuations, as well as to understand programs using delimited continuations, are well-known program transformations: transformation into continuation-passing style (CPS transformation), together with its left inverse—transformation into direct style (DS transformation), and Reynolds's defunctionalization, together with its left inverse—refunctionalization.

1.1 Functional languages with control operators

The notion of continuation is ubiquitous in the literature on the theory and practice of functional programming. Informally speaking, a continuation represents the "rest of the computation" at a given point of program execution [225]. With this definition, the omnipresence of continuations comes as no surprise—whenever it makes sense to talk about a computation, it also makes sense to talk about the rest of a computation.

Continuations were independently discovered several times in the 1960s and 1970s [195]. They underly continuation-passing style (CPS), where continuations are given an explicit representation, either in an object language (defining a program transformation [100,186,196] and a programming technique [219]) or in a meta language (defining a style of denotational semantics [225] and definitional interpreter [196]). With CPS it is possible to give a semantics to control operators such as `call/cc` (which stands for `call-with-current-continuation`) in Scheme [229] and `callcc` and `throw` in Standard ML of New Jersey [117] as well as to connect programs using such control operators with purely functional programs with an explicit representation of continuations.

Continuations can be classified using two main criteria:

1. Continuations can be either undelimited or delimited. Undelimited continuations represent the entire rest of the computation. Delimited continuations represent a prefix of the rest of the computation.

2. Continuations can be either abortive or composable. Abortive continuations model jumps, i.e., non-local transfer of control, because they are resumed by aborting the current continuation. Composable continuations model non-tail calls, because they are resumed by composing them with the current continuation.

Except for Felleisen's early work [85], only two classes of continuations have been considered in the literature: those that are undelimited and abortive, also called simply continuations, and those that are delimited and composable, also called functional or partial continuations or subcontinuations. Hence, the terms 'undelimited' and 'abortive' as well as 'delimited' and 'composable' are treated as synonyms. In this work, we use the terms 'undelimited' and 'delimited' con-

tinuations, except when we mention Felleisen's work on undelimited composable continuations.

1.1.1 Undelimited continuations

First-class undelimited continuations are made available in a programming language through control operators such as `call/cc`. The operator `call/cc` reifies the current continuation as a first-class value. Applying such a value replaces the then-current continuation with the captured continuation.

Undelimited continuations have found many applications in modelling nontrivial computational phenomena and control abstractions. The classical control operator `call/cc` was used by Haynes to implement backtracking [121], by Haynes et al. to implement coroutines [122], by Wand to implement lightweight processes [247], and by Reppy to design Concurrent ML [192], among other applications.

Other control operators for undelimited continuations that have been thoroughly studied by Felleisen et al. are `abort` and a variant of `call/cc`—the operator $\mathcal{C}$ [89,92]. These control operators were given an operational semantics via abstract machines and reduction semantics (a small-step operational semantics with an explicit representation of evaluation contexts). Felleisen et al. have also developed a calculus for local reasoning about first-class continuations extending Plotkin's λ_v-calculus [186]. Later on, Sabry and Felleisen derived a set of axioms that allows one to reason about first-class continuations in a sound and complete way with respect to continuation-passing style [202].

First-class continuations are relevant not only to computer scientists but also to logicians. As observed by Griffin in his seminal article [109], control operators for undelimited continuations have compelling logical interpretations via the Curry-Howard isomorphism. Whereas the simply-typed λ-calculus corresponds to minimal logic, its extension with Felleisen's $\mathcal{C}$ corresponds to full classical logic. Recently, Ariola et al. have shown that the weaker power of `call/cc` can be explained in terms of logic: adding `call/cc` to the λ-calculus results only in minimal classical logic [11], which is strictly weaker than full classical logic.

The control operator `call/cc` is a successor of other control operators for undelimited continuations: Landin's J-operator, Reynolds's `escape`, and Sussman and Steele's `catch` implemented in the early versions of Scheme. Historically the first control operator was Landin's J-operator [154]. Landin's primary goal was to provide an account of labels and goto statements in Algol 60 [156] using his 'Applicative Expressions' [155]. The semantics of the J-operator was given in terms of the SECD machine [39]: the J-operator reifies the current dump (the component D of the SECD machine) as a first-class value. Applying such a value results in replacing the current components of the machine with the components of the captured dump. Landin observed that not only does the J-operator provide a semantics for jumps in Algol-like languages but it is also capable of expressing nontrivial control behaviors such as backtracking. As later shown by Felleisen, the J-operator and `call/cc` can simulate each other [86].

Reynolds's `escape` [196] was essentially the binder variant of `call/cc` and therefore it is interdefinable with `call/cc`. Its semantics was given by a defini-

tional interpreter written in CPS. The control operator `escape` was inspired by the so-called 'label values' in Reynolds's programming language GEDANKEN, that were used for programming coroutines and backtracking [193].

Sussman and Steele's `catch` [229] is the closest cousin of `call/cc` (in binder form, though). Its semantics coincides with that of `escape` and was given by an interpreter written in MacLISP.

1.1.2 Delimited continuations

Another class of continuations that gained a considerable interest over the years are delimited continuations, originally studied by Felleisen et al. [87, 93], by Johnson and Duggan [133, 134], and by Danvy and Filinski [67]. As already mentioned, there are two major differences between undelimited and delimited continuations:

1. A delimited continuation does not represent the entire rest of the computation but only its prefix, which is determined by a so-called control delimiter.

2. When applied to another value, a delimited continuation does not replace the current continuation but is composed with it.

The way the composition of continuations is realized determines whether a given delimited-control operator is static, i.e., compatible with CPS (e.g., Danvy and Filinski's `shift` and `reset`), or dynamic (e.g., Felleisen's `control` and `prompt`).

1.1.2.1 Static delimited continuations

Static delimited continuations were designed to account for the traditional model of non-deterministic programming based on success and failure continuations [67]. Just as `call/cc` enables one to express in direct style nonlinear uses of continuations in CPS, `shift` and `reset` enable one to express in direct style programs that are written in continuation-composing style (i.e., where continuations are explicit but not all calls are tail calls). The control delimiter `reset` delimits the current continuation. The control operator `shift` reifies the current delimited continuation as a function (it can be viewed as `call-with-current-delimited-continuation`). The semantics of `shift` and `reset` was defined in terms of a continuation semantics and a CPS transformation.

The measure of success of static delimited continuations may be their numerous applications that include non-deterministic programming [33, 67], partial evaluation [17, 60, 61, 80, 110, 159, 228], code generation [235], normalization by evaluation [82], computational monads [96, 98], and mobile computing [227]. Among those, Filinski's work on computational monads deserves special attention [96–98]. Filinski showed that every computational effect expressible in terms of monadic reflection and reification can be simulated by `shift` and `reset`, which in turn can be simulated by `call/cc` and a piece of mutable state. This result suggests that in a sense static delimited continuations are more

fundamental for functional programming than the traditional undelimited continuations (which require the additional presence of mutable state to be equally expressive).

The regular structure of CPS made it possible to study various aspects of static delimited continuations such as axioms for reasoning about programs with `shift` and `reset` [141, 142], types [12, 66, 138–140, 177], monads [144, 240], and partial evaluation [13, 14]. Iterating the CPS transformation also made it possible to build a hierarchy of control operators $\texttt{shift}_n$ and $\texttt{reset}_n$ ($n \geq 1$) that give access to continuations in the CPS hierarchy [67]. Multiple levels of continuations can be used to layer computational effects and to avoid an undesirable interference between them.

1.1.2.2 Dynamic delimited continuations

Dynamic delimited continuations take their roots in Felleisen's undelimited composable continuations [85, 90], which were available through a control operator `control` and were meant to generalize the usual undelimited continuations as available in Scheme. An undelimited composable continuation represents the entire rest of the computation and, when applied, it is composed with the current continuation. The composition of two continuations $\kappa' \circ \kappa$ was assumed to satisfy the condition that the outer continuation κ' should be visible (i.e., accessible by control operators) from the inner continuation κ. Since there is no such operation for the domain of continuation functions, undelimited composable continuations are not compatible with traditional continuation semantics. This assumption also was reflected in the implementation of the composition of continuations which was defined as the concatenation of the contexts they represent.[1]

Despite being more expressive than undelimited abortive continuations, undelimited composable continuations suffered from the same problem, i.e.., unsoundness of the underlying calculus with respect to operational equivalence. Felleisen then realized that adding a control delimiter (called prompt) to the language with undelimited composable continuations, remedies this drawback, leading to a well-behaved calculus with control operators `control` and `prompt` [87]. Intuitively, an application of a control delimiter `prompt` to an expression evaluates this expression as a seperate subprogram, oblivious of its context beyond the control delimiter, and when the value of the subprogram is computed, it is passed to the context.[2]

Later on, Sitaram and Felleisen have shown that adding control delimiters to call-by-value functional languages with first-class undelimited continuations is a necessary condition for the CPS model for such languages to be fully abstract (meaning that if two programs have different denotations in the model, they cannot be observationally equivalent) [214]. Once again, this result suggests that delimited continuations are more powerful than undelimited ones.

The operational semantics of `control` and `prompt` was given in terms of local reduction rules and an abstract machine where the method of compos-

[1] In Chapter 7, we explain why this concatenation is incompatible with CPS.

[2] It therefore mimicks prompts in interactive systems, hence the name 'prompt'.

ing continuations, including the assumption mentioned above,[3] was inherited from undelimited composable continuations. A quasi-continuation semantics for `control` and `prompt` was expressed with algebras of contexts [93], where the meaning of the control operators hinges on algebraic operations for composing continuations. This idea has been recently further explored by Shan [209].

Dynamic delimited continuations were illustrated with examples of system and recursive programming [87], and used in modelling abstractions of control such as coroutines and engines [213].

Since the presentation of the original control operators for delimited continuations in the late 1980's, a great deal of effort has been spent on designing new and more complex delimited-control operators, all of which turned out to be incompatible with CPS. These operators include Sitaram's `run` and `fcontrol` [211], Queinnec and Serpette's `splitter` [191], Dybvig and Hieb's `spawn` [125], Moreau and Queinnec's `marker` and `call/pc` [173], Gunter et al.'s `set` and `cupto` [111], and Nanevski's `mark` and `recall` [178]. They have found applications in models of control [212], concurrency [125, 211], natural languages processing [19, 208], and type-directed partial evaluation [16]. In this work, however, we focus on the original dynamic control operators `control` and `prompt`.

1.1.2.3 Static vs. dynamic

Although a number of articles show that `shift`/`reset`, and `control`/`prompt` are equally expressive and can simulate each other [34, 35, 84, 146, 209], there is a considerable gap between their theoretical foundations as well as between their domains of applications. This dissertation attempts to contrast static and dynamic delimited continuations and to provide some of the missing elements in their theory and practice.

Since static continuations are based on CPS, we use it as a guide to build an operational foundation (abstract machines and reduction semantics) for the CPS hierarchy and to propose new applications of programs that take advantage of the power of CPS in direct style (including higher-levels of the CPS hierarchy). On the other hand, since dynamic continuations are incompatible with CPS, one can only rely on intuitions about control while programming with dynamic delimited continuations. In fact, all the examples that were present in the literature did not use the dynamism of `control` and `prompt`, and could easily be implemented using `shift` and `reset`. We identify certain programming patterns that strictly require the dynamic composition of continuations embodied in `control` and `prompt`, and we study examples using such patterns. We also provide an array of new tools that support understanding of dynamic delimited continuations: a new abstract machine, an underlying higher-order evaluator, an induced dynamic CPS transformation, and a computational monad. Additionally, we consider the mutual simulations of static and dynamic delimited continuations.

[3] If it were not for this assumption, the two continuations could be separated by a control delimiter, and `control` and `prompt` would coincide with `shift` and `reset`.

1.2 Overview and contributions

This dissertation is divided into two parts. Part I is intended to serve as an introduction to the area of functional programming with control operators, in particular to some of the theoretical and practical aspects of delimited continuations. Part II reports on our contributions and consists of a collection of publications as they appeared or will appear. The detailed outline of the rest of the dissertation is as follows.

Part I: Introduction and Background

This part consists of four chapters that provide a background for Part II and put our contributions into perspective.

- **Chapter 2: Programming with Continuations**

  We analyze several motivating examples of programming with continuations. We present a typical illustration of programming with undelimited continuations, both in direct style with the traditional control operator `call/cc`, and in continuation-passing style with different representations of continuations. We also describe the role of the CPS transformation and defunctionalization in connecting different facets of the same functional program and how they provide guidance towards a better understanding of control operators. We then study a classical example of non-deterministic programming in continuation-composing style, in continuation-passing style, and in direct style with `shift`/`reset` and with `control`/`prompt`. We also contrast the non-deterministic programs using delimited continuations with those that use undelimited continuations and mutable state instead. Additionally, we consider an example of programming with hierarchies of delimited continuations.

- **Chapter 3: Semantics of Functional Languages with Control Operators**

  We review the standard methods of defining languages with control operators, i.e., CPS transformations, continuation semantics, definitional interpreters, abstract machines and reduction semantics. We also outline the mechanical methods of connecting these specifications, with a special emphasis on the functional correspondence between higher-order evaluators and abstract machines which is one of our contributions (detailed in Chapters 5 and 6). We also briefly discuss other aspects of languages with control operators such as types and the Curry-Howard isomorphism.

- **Chapter 4: Delimited Continuations**

  We review the theoretical foundations of static and dynamic delimited continuations that we refer to in Part II of the dissertation. We then describe our contributions to the area of delimited continuations presented in Chapters 6-10 and we discuss their role. Finally, we identify challenges for future work.

Part II: Publications

This part consists of seven publications that appeared or will appear in international journals as well as conference and workshop proceedings, and of one technical report.

- **Chapter 5: A Functional Correspondence between Evaluators and Abstract Machines**

  Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8-19. ACM Press, August 2003. [5]

- **Chapter 6: From Interpreter to Logic Engine by Defunctionalization**

  Dariusz Biernacki and Olivier Danvy. From Interpreter to Logic Engine by Defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143-159, Uppsala, Sweden, August 2003. Springer-Verlag. [33]

- **Chapter 7: An Operational Foundation for Delimited Continuations in the CPS Hierarchy**

  Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science*, 1(2:5):1-39, November 2005. [29]

  This chapter subsumes:

  Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM-SIGPLAN Workshop on Continuations (CW'04)*, pages 25-33, Technical report CSR-04-1. Department of Computer Science, Queen Mary's College, January 2004. [28]

- **Chapter 8: A Simple Proof of a Folklore Theorem about Delimited Control**

  Dariusz Biernacki and Olivier Danvy. A Simple Proof of a Folklore Theorem about Delimited Control. *Theoretical Pearl in the Journal of Functional Programming*, 2006. [34]

- **Chapter 9: On the Static and Dynamic Extents of Delimited Continuations**

  Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the Static and Dynamic Extents of Delimited Continuations. *Science of Computer Programming*, 2006. [37]

  This chapter subsumes:

  Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the Dynamic Extent of Delimited Continuations. *Information Processing Letters*, 96(1):7-17, 2005. [36]

- **Chapter 10: A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations**

  Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations. Research Report BRICS RS-05-16, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005. [35]

Chapter 2

Programming with Continuations

In this chapter we analyze several motivating examples of programming with continuations. We start with a typical illustration of programming with undelimited continuations. Our approach, which we also take when presenting programs with delimited continuations in the rest of the dissertation, takes advantage of the fact that programs in direct style correspond to programs in CPS, either higher-order or first-order as can be obtained through Reynolds's defunctionalization [74, 196]. We also give a brief account of CPS transformation and defunctionalization.

We then study a typical example of non-deterministic programming in continuation-composing style, in continuation-passing style, and in direct style with `shift` and `reset`. We also point out the consequences of going beyond CPS and replacing the static control operators `shift` and `reset` by `control` and `prompt`. Additionally, we contrast the non-deterministic programs using delimited continuations with those that use undelimited continuations and mutable state instead. Finally, we consider an example of programming with hierarchies of delimited continuations.

2.1 Continuation-passing style

Continuation-passing style (CPS) is a style of functional programming where functions accept an additional parameter (the continuation), and instead of returning a computed value, call their continuation with this value [217]. Therefore, in CPS programs functions never return and all function calls are tail calls, which means that a value returned by a callee automatically becomes the value returned by a caller.

As an example, let us consider a classical CPS function computing the product of an integer list. The product of an integer list is defined by induction on the structure of the list:

- If the list is empty, the result is the neutral element for multiplication, i.e., `1`.
- If the list is non-empty, the result is computed by multiplying the head of the list by the product of the tail of the list.

Here is a CPS function, written in Scheme, that implements the above algorithm [83, 89, 197]:

```
(define multlist-cps
  (lambda (ls)
    (letrec
      ((walk (lambda (ls k)
               (if (null? ls)
                   (k 1)
                   (walk (cdr ls)
                         (lambda (v) (k (* (car ls) v))))))))
      (walk ls (lambda (v) v)))))
```

The local function `walk` is written in CPS. It is given two parameters: a list of integers and a functional representation of the continuation that determines what to do next. The initial continuation is the identity function reflecting that once the product of the list initially given to `walk` is computed, it becomes the final result. The subsequent continuations, given a value, compute a product of this value and all the previously visited elements of the input list. Function `walk` is defined by induction on the structure of the input list:

- When `walk` is applied to an empty list it computes the product of the entire list by applying its continuation to the intermediate result `1`.
- When `walk` is applied to a non-empty list it computes the product of the tail of the list, with the continuation that when given this product, computes the product of the entire list.

Obviously, the above procedure is not optimal because it continues to traverse the entire list even after one of the integers is `0` (the absorbant element for multiplication), which already determines the final result to be `0`. With CPS it is simple to remedy this situation:

```
(define multlist-optimal-cps
  (lambda (ls)
    (letrec
      ((walk (lambda (ls k)
               (if (null? ls)
                   (k 1)
                   (let ((i (car ls)))
                     (if (= i 0)
                         0
                         (walk (cdr ls)
                               (lambda (v) (k (* i v))))))))))
      (walk ls (lambda (v) v)))))
```

We simply test whether the head of the list is `0` and if so, we discard the rest of the computation encoded in the parameter `k` by not applying `k` and by returning `0` instead. Notice that no multiplications are performed if the list contains `0`.

2.2 First-class continuations: call/cc

However instructive and expressive, programming in CPS can hardly be perceived as a convenient format. One would rather like to have the power of CPS in the more usual direct style. For example, it is simple to write a direct-style counterpart of the function `multlist-cps` from the previous section, where the continuations are implicit:

```
(define multlist
  (lambda (ls)
    (letrec ((walk (lambda (ls)
                     (if (null? ls)
                         1
                         (* (car ls) (walk (cdr ls)))))))
      (walk ls))))
```

In order to program the non-local transfer of control in the function `multlist-optimal-cps` in direct style, however, it is necessary to go beyond pure functional programming and employ a control operator such as `call/cc`.[1] An intuitive semantics of `call/cc` is as follows. When applied to a function of one argument `k`, it constructs a representation of the current continuation and binds it to `k`. When `k` is applied to a value, the then-current continuation is discarded and the value of `k` becomes the current continuation. The direct-style counterpart of `multlist-optimal-cps` reads as follows:

```
(define multlist-optimal
  (lambda (ls)
    (call/cc
      (lambda (stop)
        (letrec ((walk (lambda (ls)
                         (if (null? ls)
                             1
                             (let ((i (car ls)))
                               (if (= i 0)
                                   (stop 0)
                                   (* i (walk (cdr ls))))))))))
          (walk ls))))))
```

Captured continuations can be passed to and returned by functions just as other first-class values. A formal semantics of `call/cc` and other control operators is usually given in terms of a CPS transformation, a continuation semantics, an abstract machine or a reduction semantics (see Chapter 3).

The CPS transformation [100, 186, 196] is a traditional program transformation that turns direct-style programs into CPS ones. In essence, the CPS transformation can be defined as a meta-level procedure that:

1. sequentializes computations,

[1] In ML for instance, we could use the mechanism of exceptions instead. In general, however, there are more involved uses of continuations, e.g., backtracking, that cannot be expressed with exceptions [233].

2. names intermediate results,

3. materializes control-flow in functional parameters called continuations.

For example, CPS-transforming (using the standard call-by-value CPS transformation) `multlist` yields `multlist-cps` and CPS-transforming `multlist-optimal` yields `multlist-optimal-cps`.

The direct-style transformation, on the other hand, is the left inverse of the CPS transformation [59, 72]. For example, transforming `multlist-cps` to direct style yields `multlist` and transforming `multlist-optimal-cps` to direct-style yields `multlist-optimal`.

2.3 First-order representation of continuations

Continuations often come in disguise. For example, the following first-order program also computes the product of a list of integers using a stack (again, we require that no multiplication is performed in case the list contains `0`):

```
(define multlist-optimal-with-stack
  (lambda (ls)
    (letrec
      ((walk (lambda (ls k)
               (if (null? ls)
                   (mul k 1)
                   (let ((i (car ls)))
                     (if (= i 0)
                         0
                         (walk (cdr ls) (cons i k)))))))
       (mul (lambda (k v)
              (if (null? k)
                  v
                  (mul (cdr k) (* (car k) v))))))
      (walk ls '()))))
```

The function `walk` traverses the input list and pushes its elements on the stack `k` (represented as a list). If `walk` encounters `0` it returns it as the final result of the entire computation. If however, `walk` reaches the end of the input list it calls `mul` which performs the actual multiplication of the elements of the stack, using an accumulator `v`.

The function `multlist-optimal-with-stack` is the defunctionalized counterpart of the function `multlist-optimal-cps` [74, 196]. As originally specified by Reynolds, defunctionalization is a program transformation that eliminates higher-order functions. It replaces each function space inhabited by λ-abstractions that are arguments passed to or results produced by higher-order functions, with a first-order data structure and a corresponding interpretive function that dispatches on the data structure. Each λ-abstraction is replaced by the creation of a record in the corresponding data structure. This record identifies the defining expression and holds the values of the free variables of the expression. Each function application is replaced by a call to the corresponding

interpretive function. The interpretive function maps the records of the data structure into the functions they represent.

In our case the only function space that makes the local function `walk` in `multlist-optimal-cps` higher-order is that of the continuation. There are two λ-abstractions defining continuations: one in the initial call to `walk`, with no free variables, and one in the recursive call to `walk`, with two free variables `i` and `k`. We replace them by a creation of a 0-ary tuple `'()` and of a pair `(cons i k)`, respectively (there is no need to attach any additional information to these entities because their structure identifies them uniquely). Thus the data structure representing the continuation takes the form of a list. We define the interpretive function `mul` that, given a list and a value, performs the application of the continuation represented by the list to the value. We also replace the two applications of a continuation to a value (one in the base case and one in the induction case of `walk`) by a call to `mul`. The result is `multlist-optimal-with-stack`.

Defunctionalization has been formalized and proven correct by Banerjee et al. [18], by Nielsen [180], and by Pottier and Gauthier [189]. More examples of defunctionalization can be found in Sections 6.1.1 and 7.2.3, as well as in Danvy and Nielsen's article [74]. The left inverse of defunctionalization is called refunctionalization.

The function `multlist-optimal-with-stack` explains, in terms of a first-order transition system, the control flow in the function `multlist-optimal-cps` that is materialized in continuations. The function `multlist-optimal-cps` in turn, explains in terms of CPS, the behavior of `call/cc` in `multlist-optimal`. CPS transformation and defunctionalization allow one to connect programs that were written independently but are different forms of the same algorithm; they also reveal new programs corresponding to existing ones. In particular, programs with control operators can be obtained as a direct-style image of those written in CPS (tail-recursive programs with defunctionalized continuations are also in CPS, just with a first-order representation of continuations).

2.4 Continuation-composing style

Not all expressions that use continuations model non-tail calls and can be mapped back to direct style using control operators for undelimited continuations. Composing continuations instead of jumping is characteristic of non-deterministic computations. For instance, we can express two standard backtracking primitives `amb` and `fail` in continuation-composing style [67]:

```
(define amb-c
  (lambda (k)
    (begin (k #t) (k #f) "No")))

(define fail-c
  (lambda (k)
    "No"))
```

A non-deterministic choice is implemented by `amb-c`. Given a continuation `k`, the function repeatedly explores all the possible choices by first applying `k`

to `#t`, then after it returns (here the continuations are being composed), by applying `k` to `#f`, and finally by returning `"No"` to signal that there are no more choices. Failures can be handled by `fail-c` which discards the current branch of control. The following function uses `amb-c` and `fail-c` to display all those lists of zeros and ones whose lengths are `n` and whose elements sum to `s`:

```
(define bitseq-c
  (lambda (n s)
    (letrec
      ((bs (lambda (n s k)
             (if (= n 0)
                 (if (= s 0)
                     (k '())
                     (fail-c k))
                 (if (= s 0)
                     (bs (- n 1) s
                         (lambda (vs) (k (cons 0 vs))))
                     (amb-c (lambda (b)
                              (if b
                                  (bs (- n 1) s
                                      (lambda (vs) (k (cons 0 vs))))
                                  (bs (- n 1) (- s 1)
                                      (lambda (vs)
                                        (k (cons 1 vs))))))))))))
      (bs n s display))))
```

The direct-style counterpart of this example is due to Reynolds [197, Section 13.8], who used it to test a general backtracking function written with `call/cc` and assignment that we show later in this section.

In order to make the control flow in `amb-c` and `fail-c` more explicit, we eliminate the non-tail calls by CPS-transforming the two functions:

```
(define amb-2c
  (lambda (k1 k2)
    (k1 #t (lambda (ignore) (k1 #f (lambda (ignore) (k2 "No")))))))

(define fail-2c
  (lambda (k1 k2)
    (k2 "No")))
```

The resulting functions are expressed in extended continuation-passing style (ECPS) [67], where the control flow is materialized in two continuations: a continuation `k1` that plays the role of the traditional success continuation, and a meta-continuation `k2` that plays the role of the traditional failure continuation [167]. One could now CPS transform the function `bitseq-c` and use `amb-2c` and `fail-2c` to obtain the function `bitseq-2c` displaying bit sequences, written in ECPS with the success and failure continuations. The functions `bitseq-2c`, `amb-2c`, and `fail-2c` could then be defunctionalized leading to a first-order program with two stacks: one representing the success continuation (a list of zeros and ones), and the other one representing the failure continuation (a list of success continuations [237]).

2.5 First-class delimited continuations

In their seminal article [67], Danvy and Filinski introduced the control operators `shift` and `reset` to account for the backtracking model of success continuations in continuation-composing style which, via CPS transformation, is equivalent to the traditional success/failure continuation model in continuation-passing style. In this section, we express the backtracking primitives in direct style using `shift` and `reset`. The intuitive semantics of `shift` and `reset` can be explained as follows:

- In an expression `(reset e)`, `reset` delimits the extent of `shift`-operations in `e`. The value of `e` becomes the value of the whole expression.

- In an expression `(shift k e)`, `shift` captures the current continuation up to the nearest dynamically enclosing `reset`, passes this delimited continuation to `k` and delimits the extent of `shift`-operations in `e`. When the captured continuation is applied, it performs its actions and then returns to the point of its invocation with the computed value. The value of `e` becomes the value of the whole expression.

The formal semantics of `shift` and `reset` were given in terms of a meta-continuation semantics and a CPS transformation (for details see Chapter 4).

The backtracking primitives expressed in direct style look as follows:

```
(define amb
  (lambda ()
    (shift k (begin (k #t) (k #f) "No"))))

(define fail
  (lambda ()
    (shift k "No")))
```

CPS-transforming (or more precisely, transforming into continuation-composing style) `amb` and `fail` yields `amb-c` and `fail-c`, respectively.

The function `bitseq-c` mapped back to direct style reads as follows:

```
(define bitseq
  (lambda (n s)
    (letrec ((bs (lambda (n s)
                   (if (= n 0)
                       (if (= s 0)
                           '()
                           (fail))
                       (if (= s 0)
                           (cons 0 (bs (- n 1) s))
                           (if (amb)
                               (cons 0 (bs (- n 1) s))
                               (cons 1 (bs (- n 1) (- s 1)))))))))
      (reset (display (bs n s))))))
```

Again, CPS-transforming `bitseq` yields `bitseq-c`.

The control operators `shift` and `reset` were meant to provide a programmer with the pre-existing means that are intuitively and semantically compatible with CPS. The importance of this fact can be noticed when we compare them with other delimited-control operators. For example, the control operators `control` and `prompt` introduced by Felleisen [87] intuitively have a very similar semantics to `shift` and `reset`. In fact, the semantics of `reset` and `prompt` coincide, whereas `shift` and `control` differ in the way a captured continuation is resumed:

- When a `shift`-captured continuation is resumed, it is separated from the then-current delimited continuation which subsequently cannot be accessed from the resumed continuation. This behavior is imposed by the CPS semantics (see Section 4.1.1).

- When a `control`-captured continuation is resumed, it is sealed to the then-current delimited continuation which subsequently can be accessed from the resumed continuation. This behavior is imposed by the abstract machine semantics, where continuations are composed by concatenation of stack frames (see Section 4.1.2).

In consequence, programs with `control` and `prompt` do not correspond to programs in CPS, and one cannot rely on the traditional notion of continuation while programming with dynamic delimited continuations. Moreover, a vast majority of meaningful programs using `control` and `prompt` are merely simulations of `shift` and `reset`. (In Chapter 9, we present the first examples where using `control` and `prompt` is actually more natural than using `shift` and `reset`.) For example, if in the definitions of `amb`, `fail`, and `bitseq` we replace `shift` and `reset` with `control` and `prompt`, respectively, the resulting function will display answer `"No"`, no matter what input `n` and `s` it is given. The way to fix the problem is dictated by a simulation theorem that we formalize and prove in Chapter 8: it is sufficient to replace, in the definition of `amb`, the applications `(k #t)` and `(k #f)` by `(prompt (k #t))` and `(prompt (k #f))`, respectively. It is one of the goals of this dissertation to explore the space beyond CPS [90] and to provide new tools for understanding and transforming programs that go beyond CPS.

Backtracking can be also obtained with `call/cc` and assignment. Here we present a general function `backtrack-callcc` implementing backtracking as provided by `amb` and `fail` above. This function is due to Reynolds [197, Section 13.8].

```
(define backtrack-callcc
  (lambda (f)
    (let ((cl '()))
      (begin (call/cc
               (lambda (escape)
                 (let ((amb (lambda ()
                              (call/cc
                                (lambda (k)
                                  (begin (set! cl (cons k cl))
                                         #t)))))
                       (fail (lambda () (escape "No"))))
                   (display (f amb fail)))))
             (if (null? cl)
                 "No"
                 (let ((c (car cl)))
                   (begin (set! cl (cdr cl))
                          (c #f))))))))
```

The argument `f` is a function that accepts two arguments: one implementing non-deterministic choice and the other one handling failures. The function `backtrack-callcc`, when applied to such a function, will display all the results returned by this function. The local list `cl` consists of the continuations that represent yet unexplored control branches. The local function `amb` captures the current continuation, stores it in `cl`, and applies it to `#t` (by simply returning `#t`). The local function `fail` discards the current control branch by invoking the continuation `escape`. We can see that the captured continuation `k` plays the role of the success continuation, whereas `cl` is a first-order representation of the failure continuation. For example, applying `backtrack-callcc` to the following function displays `134`:

```
(lambda (amb fail)
  (if (amb) (if (amb) 1 (fail)) (if (amb) 3 4)))
```

We can obtain the same effect, but in a much more concise form, by using `shift` and `reset` instead of `call/cc` and assignment

```
(define backtrack-shift
  (lambda (f)
    (let ((amb (lambda () (shift k (begin (k #t) (k #f) "No"))))
          (fail (lambda () (shift k "No"))))
      (reset (display (f amb fail))))))
```

Macro-expanding Filinski's definitions of `shift` and `reset` in terms of `call/cc` and assignment [96] in `backtrack-shift` yields a semantically equivalent but syntactically different version of `backtrack-callcc`.

2.6 Hierarchies of delimited continuations

In the early studies of delimited continuations, it was soon recognized that a single level of delimited-control operators is not always sufficient. To avoid

situations when two or more uses of a control operator and/or control delimiter interfere with each other in an undesirable way, hierarchical structures of control operators and delimiters were introduced [67, 213].

Let us assume that instead of printing non-deterministically computed results we want to collect them in a list. To this end we need a third backtracking primitive `emit` that, in continuation-composing style, reads as follows [67]:

```
(define emit-2c
  (lambda (v k1 k2)
    (cons v (k1 '() k2))))
```

Given a value `v`, a success continuation `k1`, and a failure continuation `k2`, `emit-2c` invokes `k1` with a dummy argument and `k2`, and emits `v` by prepending it to the result of the application. This way the operation of collecting the successive results do not interfere with the actual backtracking. To make the situation more explicit, let us CPS-transform `emit-2c`:

```
(define emit-3c
  (lambda (v k1 k2 k3)
    (k1 '() k2 (lambda (vs) (k3 (cons v vs))))))
```

Our ultimate goal, however, is to express `emit` in direct style. As an intermediate step we eliminate `k2` in `emit-2c`:

```
(define emit-c
  (lambda (v k1)
    (k1 (shift k (cons v (k '()))))))
```

CPS-transforming `emit-c` yields `emit-2c`. In order to map `emit-c` to direct style, we need to employ a control operator that will capture both success and failure continuation. The control operators that make it possible to abstract control beyond the first level of the CPS hierarchy are `shift2` and `reset2` [67].

```
(define emit
  (lambda (v)
    (shift2 k (cons v (k '())))))
```

CPS-transforming `emit` (using Danvy and Filinski's CPS transformation) yields `emit-c`. Iterating this CPS transformation leads to the CPS hierarchy in which it is possible to define control operators $\texttt{shift}_n$ and $\texttt{reset}_n$ that can abstract control at an arbitrary level $n \geq 1$.

The function that implements backtracking and collects the computed results reads as follows:

```
(define backtrack-shift2
  (lambda (f)
    (let ((amb (lambda () (shift1 k (begin (k #t) (k #f) "No"))))
          (fail (lambda () (shift1 k "No")))
          (emit (lambda (v) (shift2 k (cons v (k '()))))))
      (reset2 (begin (reset1 (emit (f amb fail)))
                     '())))))
```

Independently of the static hierarchy, Sitaram and Felleisen generalized `control` and `prompt` to a dynamic hierarchy and they implemented it in Scheme in terms of `control` and `prompt` [213]. By construction, the dynamic hierarchy is not compatible with CPS. For example, if we replace `shift1`, `reset1`, `shift2`, and `reset2` respectively by `control1`, `prompt1`, `control2`, and `prompt2`, and we add prompts in the definition of `amb`, the resulting function will still perform backtracking but the computed results will be collected in the reverse order. Again, the fix is to replace the application `(k '())` in the definition of `emit` with `prompt2 (k '())`.

A corresponding solution using `call/cc` and assignment can be obtained by maintaining a list `rl` of the computed results, as presented by Reynolds [197, Section 13.8]:

```
(define backtrack-callcc2
  (lambda (f)
    (let ((cl '())
          (rl '()))
      (begin (call/cc
               (lambda (escape)
                 (let* ((amb (lambda ()
                               (call/cc
                                 (lambda (k)
                                   (begin (set! cl (cons k cl))
                                          #t)))))
                        (fail (lambda () (escape "No")))
                        (r (f amb fail)))
                   (set! rl (cons r rl)))))
             (if (null? cl)
                 rl
                 (let ((c (car cl)))
                   (begin (set! cl (cdr cl)) (c #f))))))))
```

2.7 Conclusion

The examples presented in this chapter convey two messages:

- Delimited continuations are more expressive than undelimited continuations, since in most applications the latter require the additional presence of mutable state. This observation finds a confirmation in Filinski's simulation of `shift` and `reset` in terms of `call/cc` and assignment [96]. Programs written with (static) delimited continuations correspond to CPS-based functional encoding, and they rely only on one kind of computational effect, which is capable of simulating other effects [96]. In consequence, programming with (static) delimited continuations is guided by CPS and leads to natural and concise solutions.

- CPS and defunctionalization considerably reinforce one's understanding of control operators, which otherwise is solely based on intuition and testing.

Chapter 3

Semantics of Functional Languages with Control Operators

In this chapter we review the standard methods of defining a semantics for functional languages with control operators. Control operators are traditionally studied in the context of the call-by-value λ-calculus and their semantics is usually defined in terms of a CPS transformation, a continuation semantics, a definitional interpreter, an abstract machine, or a reduction semantics. The goal of this chapter is to review these concepts and set the background for our study of delimited continuations in the second part of the dissertation. To this end, we illustrate various semantic specifications and the relationships between them on a language called ISWIM$_{\mathcal{C}}$ that is a fragment of Landin's ISWIM [157] augmented with the canonical control operator for undelimited continuations—Felleisen's $\mathcal{C}$ [89].

ISWIM is an idealized untyped functional language that embodies the call-by-value evaluation strategy and it forms a natural basis for realistic languages such as Scheme and ML. The subset of ISWIM we treat here is the λ-calculus extended with integers and the successor function. The presented results carry over straightforwardly to functional languages enriched with other constants and basic functions, conditionals, recursion, etc.

The grammar of expressions of ISWIM$_{\mathcal{C}}$ reads as follows (programs are closed expressions):

$$e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{C}x.e$$

The semantics of the control operator $\mathcal{C}$ is similar to that of `call/cc`, though unlike `call/cc`, $\mathcal{C}$ does not duplicate the current continuation.

3.1 CPS transformations

What we today call a CPS transformation was first conceived in the context of imperative languages by van Wijngaarden [236], J. H. Morris [175], and Abdali [2], who described different algorithms for eliminating goto statements in Algol 60. Roughly at the same time, various CPS transformations for functional languages were introduced and studied by Fischer [100], Reynolds [196], and Plotkin [186].

As outlined in Section 2.2, in the context of functional languages, the CPS transformation introduces functional parameters called continuations in order to sequentialize computations and name intermediate results. In effect, all function calls become tail calls, which means that a value returned by the callee is immediately returned by the caller. This phenomenon turns recursive algorithms into tail-recursive, i.e., iterative ones [217].

Another salient feature of programs in CPS is their evaluation-order independence, which was first noticed by Reynolds [196] and subsequently formalized and proved by Plotkin [186]. More precisely, evaluation order is encoded in CPS programs. As a consequence, CPS makes it possible to simulate call by value in call by name and vice versa.

Since the control flow of programs in CPS is made explicit and sequentialized, the CPS transformation has become an accepted intermediate step in compilers for functional languages [10, 102].

A standard call-by-value CPS transformation for ISWIM-like languages, described by Reynolds in his article on definitional interpreters [196], and later formalized by Plotkin [186], is a syntactic operation mapping ISWIM expressions to ISWIM expressions:

$$\begin{aligned}
\overline{x} &= \lambda k.k\,x \\
\overline{\lambda x.e} &= \lambda k.k\,(\lambda x k.\,\overline{e}\,k) \\
\overline{e_0\,e_1} &= \lambda k.\overline{e_0}\,(\lambda v_0.\,\overline{e_1}\,(\lambda v_1.\,v_0\,v_1\,k)) \\
\overline{\ulcorner m \urcorner} &= \lambda k.k\ulcorner m\urcorner \\
\overline{succ\ e} &= \lambda k.\overline{e}\,(\lambda v.\,k\,(succ\ v))
\end{aligned}$$

CPS transformations akin to the above introduce administrative redexes that would not occur if the transformation were performed by hand (e.g., according to Reynolds's recipe). In order to eliminate such administrative redexes, Plotkin devised a so-called colon translation [186] making his CPS transformation consist of two passes. This result was later refined by Danvy and Filinski who proposed a one-pass CPS transformation using the idea of normalization by evaluation [68].

CPS transformation can be used to explicate control operators and to give their semantics in terms of pure functional programs. For example, we can extend the above CPS transformation to account for $\mathcal{C}$, as presented by Felleisen et al. [91]:

$$\overline{\mathcal{C}x.e} = \lambda k.\overline{e}\{(\lambda v k'.\,k\,v)/x\}\,(\lambda v.\,v)$$

where $e\{v/x\}$ is the result of capture-avoiding substitution of v for x in e. The CPS transformation makes the meaning of control operators explicit. $\mathcal{C}$ captures the current continuation as a function, that, when applied, replaces the then-current continuation with the captured continuation.

Following, e.g., Sabry and Felleisen [202], the formal semantics of ISWIM$_\mathcal{C}$ can be obtained through the CPS transformation and an evaluation function $\mathsf{eval}_{\mathsf{DS}}$ for ISWIM (e.g., as defined by Plotkin [186]):

Definition 3.1 *Given a program e, we define*

$$\mathsf{eval}_{\mathsf{CPS}}(e) \stackrel{\text{def}}{=} \mathsf{eval}_{\mathsf{DS}}(\overline{e}\,(\lambda v.v)).$$

It follows from Plotkin's Indifference Theorem [186] that the evaluation function $\mathsf{eval}_{\mathsf{CPS}}$ is independent of the evaluation order of $\mathsf{eval}_{\mathsf{DS}}$.

Control operators are rarely conceived in call-by-name languages, mainly due to the lack of predictability of where control effects may occur in a call-by-name evaluation process. It is, however, possible to use Plotkin's call-by-name CPS transformation for ISWIM [186] extended with Griffin's call-by-name CPS transformation of $\mathcal{C}$ [109] in order to define a call-by-name semantics for ISWIM$_{\mathcal{C}}$ (analogous to $\mathsf{eval}_{\mathsf{CPS}}$). Griffin's CPS transformations of $\mathcal{C}$ were used in the studies of logical contents of control operators, where CPS transformations correspond to embeddings between logics via the Curry-Howard isomorphism [109]. Also out of logical considerations (the Axiom of Choice), Krivine has studied the call-by-name semantics of `call/cc` [149].

The call-by-name CPS transformation can also be useful in deriving call-by-name abstract machines from evaluators, as detailed in Chapter 5.

3.2 Continuation semantics

The traditional denotational semantics of Scott and Strachey [205] has turned out to be insufficient when it comes to describing the behavior of jumps, available, e.g., in imperative languages with labels and goto statements. This observation, as well as an inspiration from the work by Mazurkiewicz on tail functions [164], led Wadsworth and Strachey to a new kind of denotational semantics—continuation semantics [225].

In continuation semantics, the meaning of an expression is defined through a semantic function which takes a new argument called a continuation, and produces the final result of the entire program under consideration. Since the semantic function has access to the continuation of the computation, it can manipulate the current continuation (e.g., discard, replace or duplicate it) and therefore define various control operations such as jumps in Algol 60 and first-class continuations in functional languages.

The call-by-value continuation semantics for ISWIM$_{\mathcal{C}}$ is displayed in Figure 3.1. Here we focus on the evaluation model embodied in the continuation semantics rather than on its domain-theoretic foundations. For example, we elide information about isomorphisms mediating between domains. (A more rigorous presentation of the continuation semantics for ISWIM with control operators can be found in Reynolds's book [197, Section 12.1].)

The semantic function $\mathcal{E}_{\mathcal{C}}$ makes use of an environment that is a function mapping identifiers to values (ρ_{mt} denotes the initial empty environment, whereas $\rho\{x \mapsto v\}$ denotes the environment that maps x to v and is otherwise equal to ρ), and of a continuation that maps values to answers. The domain of values is reflexive and includes integers and functions (representing λ-abstractions and captured continuations). The domain of answers is the lifted domain of values to account for divergence and possible type errors (e.g., if the

- Syntax: $x \in \mathsf{Ide}$
  $e \in \mathsf{Exp}$ $\qquad e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{C}x.e$

- Domains of values and continuations:

$$\begin{aligned} m &\in \mathsf{Int} \\ f &\in \mathsf{Fun} = \mathsf{Val} \to \mathsf{Cont} \to \mathsf{Ans} \\ v &\in \mathsf{Val} = \mathsf{Int} + \mathsf{Fun} \\ \kappa &\in \mathsf{Cont} = \mathsf{Val} \to \mathsf{Ans} \\ \mathsf{Ans} &= \mathsf{Val}_\perp \end{aligned}$$

- Environments: $\rho \in \mathsf{Env} = \mathsf{Ide} \to \mathsf{Val}$

- Semantic function: $\mathcal{E}_\mathcal{C} : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Cont} \to \mathsf{Ans}$

$$\begin{aligned} \mathcal{E}_\mathcal{C}[\![x]\!]\,\rho\,\kappa &= \kappa\,(\rho(x)) \\ \mathcal{E}_\mathcal{C}[\![\lambda x.e]\!]\,\rho\,\kappa &= \kappa\,(\lambda v.\,\lambda\kappa.\,\mathcal{E}_\mathcal{C}[\![e]\!]\,\rho\{x \mapsto v\}\,\kappa) \\ \mathcal{E}_\mathcal{C}[\![e_0\, e_1]\!]\,\rho\,\kappa &= \mathcal{E}_\mathcal{C}[\![e_0]\!]\,\rho\,(\lambda f.\,\mathcal{E}_\mathcal{C}[\![e_1]\!]\,\rho\,(\lambda v.\,f\,v\,\kappa)) \\ \mathcal{E}_\mathcal{C}[\![\ulcorner m \urcorner]\!]\,\rho\,\kappa &= \kappa\, m \\ \mathcal{E}_\mathcal{C}[\![\mathit{succ}\ e]\!]\,\rho\,\kappa &= \mathcal{E}_\mathcal{C}[\![e]\!]\,\rho\,(\lambda m.\,\kappa\,(m+1)) \\ \mathcal{E}_\mathcal{C}[\![\mathcal{C}x.e]\!]\,\rho\,\kappa &= \mathcal{E}_\mathcal{C}[\![e]\!]\,\rho\{x \mapsto \lambda v.\,\lambda\kappa'.\,\kappa\,v\}\,\kappa_0 \end{aligned}$$

Figure 3.1: A call-by-value continuation semantics for ISWIM$_\mathcal{C}$

program contains an expression *succ* e and e is not denoted by an integer). (An alternative approach could assume the domain of answers to be defined as $(\mathsf{Val} + \{\mathit{error}\})_\perp$, where erroneous expressions are denoted by *error* and divergent expressions are denoted by $\perp$.) The initial continuation κ_0 injects its argument into Ans: $\kappa_0 = \lambda v.\mathit{up}(v)$. Obviously, if we considered a typed version of ISWIM$_\mathcal{C}$, all type errors would be detected statically by a typechecker.

The call-by-value CPS transformation of the previous section can be obtained from the continuation semantics as the associated syntax-directed encoding into the term model of the meta-language. For example, it can be shown that

$$\mathcal{E}_\mathcal{C}[\![\overline{e_0\, e_1}]\!]\,\rho \;=\; \lambda\kappa.\,\mathcal{E}_\mathcal{C}[\![\overline{e_0}]\!]\,\rho\,(\lambda v_0.\,\mathcal{E}_\mathcal{C}[\![\overline{e_1}]\!]\,\rho\,(\lambda v_1.\,v_0\,v_1\,\kappa))$$

The relationships between direct and continuation semantics have been studied by Reynolds [194], by Sethi and Tang [207], as well as by Stoy [223]. The formal connection between the continuation semantics for ISWIM$_\mathcal{C}$ and the evaluation function $\mathsf{eval}_{\mathsf{CPS}}$ of the previous section is stated in the following proposition. (For simplicity, the proposition relates only those programs that evaluate to observable values.)

Proposition 3.1 *For any program e and for any integer m,*

$$\mathcal{E}_\mathcal{C}[\![e]\!]\,\rho_{mt}\,\kappa_0 = \mathit{up}(m) \text{ if and only if } \mathsf{eval}_{\mathsf{CPS}}(e) = m.$$

One could define a call-by-name continuation semantics for ISWIM$_{\mathcal{C}}$ that would correspond to the call-by-name CPS transformation in the same way that the call-by-value continuation semantics corresponds to the call-by-value CPS transformation.

Other control operators for undelimited continuations. We can easily extend the continuation semantics of Figure 3.1 to account for other control operators. For example, we can add two clauses defining `call/cc` (noted $\mathcal{K}$) and `abort` (noted $\mathcal{A}$):

$$\begin{aligned}\mathcal{E}_{\mathcal{C}}[\![\mathcal{K}x.e]\!]\,\rho\,\kappa &= \mathcal{E}_{\mathcal{C}}[\![e]\!]\,\rho\{x \mapsto \lambda v\kappa'.\,\kappa\,v\}\,\kappa\\ \mathcal{E}_{\mathcal{C}}[\![\mathcal{A}e]\!]\,\rho\,\kappa &= \mathcal{E}_{\mathcal{C}}[\![e]\!]\,\rho\,\kappa_0\end{aligned}$$

The difference between `call/cc` and $\mathcal{C}$ seems to be minor, but the fact that $\mathcal{C}$ does not duplicate the current continuation makes it more expressive and powerful than `call/cc`. In fact, it is possible to define both `call/cc` and `abort` in terms of $\mathcal{C}$:

$$\begin{aligned}\mathcal{K}x.e &= \mathcal{C}x.x\,e\\ \mathcal{A}e &= \mathcal{C}d.e, \quad \text{where } d \notin FV\,(e)\end{aligned}$$

Conversely, in order to define $\mathcal{C}$, we need to use both `abort` and `call/cc`:

$$\mathcal{C}x.e = \mathcal{K}x.\mathcal{A}e$$

The above equations are sound with respect to the continuation semantics.

Recently, Ariola et al. have explained the difference between `call/cc` and $\mathcal{C}$ in terms of the Curry-Howard isomorphism [12]. It is known that the simply-typed λ-calculus corresponds via the Curry-Howard isomorphism to minimal logic [130]. It is also known since Griffin's discovery [109], that $\mathcal{C}$ corresponds to double-negation elimination ($\neg\neg A \to A$), whereas `call/cc` corresponds to Peirce's law ($((A \to B) \to A) \to A$). Ariola et al. have shown that double-negation elimination entails Peirce's law but not conversely. Consequently, adding them to minimal logic results in full classical and minimal classical logic, respectively. Hence, one can simulate `call/cc` with $\mathcal{C}$ as shown above, but not the other way around. If however, one adds Peirce's law to intuitionistic logic as defined by minimal logic and Ex Falso Quodlibet ($\bot \to A$), the result is full classical logic. Since `abort` corresponds to Ex Falso Quodlibet, the simulation of $\mathcal{C}$ in terms of `call/cc` and `abort` follows.

The studies of type systems for first-class continuations have not been restricted to proof theory. For example, in strongly-typed Standard ML of New Jersey continuations are accessible via the control operators `callcc` and `throw` [117], which satisfy the following signature:

$$\begin{aligned}\texttt{callcc} &: (\alpha\ \texttt{cont} \to \alpha) \to \alpha\\ \texttt{throw} &: \alpha\ \texttt{cont} \to \alpha \to \beta,\end{aligned}$$

where α `cont` is a polymorphic type of α-accepting continuations.

3.3 Definitional interpreters

A direct implementation of a continuation semantics in a functional language gives rise to a definitional interpreter in CPS. Independently of denotational semantics, definitional interpreters are a favorite among functional-language designers and have always been regarded as a fundamental tool for programming-language development. Languages defined by interpreters written in another language include McCarthy's LISP [165], Landin's 'Applicative Expressions' [155], Reynolds's GEDANKEN [193], and Sussman and Steele's Scheme [229].

An interpreter (evaluator) is a program written in a defining or metalanguage that processes a program written in a defined or object language by analyzing an abstract representation of the program text and producing the result (a value or a state) of the execution of the program [103]. Interpreters allow for meta-linguistic abstraction, i.e., establishing new languages, and are indispensable during the phase of developing a programming language. They also facilitate extensions and modifications of existing programming languages and reinforce the understanding of various programming language features (e.g., extending a purely functional language with computational effects [196,218,239]).

Besides defining a language by providing a direct, executable expression of the program semantics, an interpreter provides an implementation of the language. An alternative implementation model consists in using a compiler from the source language to a target language and a hardware or virtual machine for executing the target language. While this approach is usually more efficient and therefore preferable in the realistic realization of a programming language, it is too low-level to be used in the early stages of language development.

Interpreters can be either higher-order or first-order. Higher-order interpreters take advantage of higher-order functions in the metalanguage to express their control-flow and/or expressible and denotable values. (Expressible values are the values that can be the result of the evaluation of an expression, whereas denotable values are the values that can be stored in the environment [222,224].) Such interpreters typically provide a direct implementation of the denotational semantics of a language. First-order interpreters use only first-order features of the metalanguage. They typically provide a direct implementation of an abstract machine [5,103,196].

Like continuation semantics, continuation-based interpreters can be used for describing the computational meaning of control operators. Such interpreters were used by Landin [154] to define the J-operator, by Reynolds [193] and by F. L. Morris [174] to treat GEDANKEN-like value labels and jumps, by Reynolds to define `escape` [196], and by Sussman and Steele to implement `catch` [229].

We present a definitional interpreter for ISWIM$_{\mathcal{C}}$ written in Standard ML that implements the continuation semantics of Section 3.2. First, here is the data type implementing the syntax of the language:

```
type ide = string

datatype exp = VAR of ide | LAM of ide * exp | APP of exp * exp
             | NUM of int | SUCC of exp
             | C of ide * exp
```

Since the continuation semantics makes use of an environment, we assume a structure `Env` that instantiates the signature `ENV` below.

```
signature ENV
= sig
    type 'a env
    val empty : 'a env
    val extend : ide * 'a * 'a env -> 'a env
    val lookup : ide * 'a env -> 'a
  end;
```

We leave the actual implementation of the environment unspecified. Typically, an environment is implemented as a list or as a function. It is also possible to make an interpreter substitution-based with the domain of values included in the syntax of the language. Such interpreters, however, are not compositional (i.e., where the meaning of an expression is a function of the meanings of its subexpressions) which hinders using them for reasoning about evaluation of programs.

The interpreter itself is simply a transliteration of the continuation semantics into ML. Here we present its uncurried version.

```
structure Higher_Order_Interpreter
= struct
    datatype value = INT of int
                   | FUN of value * cont -> answer
    and answer = ANS of value
    withtype continuation  = value -> answer

    (*  eval : exp * value Env.env * continuation -> answer *)
    fun eval (VAR x, env, k)
        = k (Env.lookup (x, env))
      | eval (LAM (x, e), env, k)
        = k (FUN (fn (v, k') => eval (e, Env.extend (x, v, env), k')))
      | eval (APP (e0, e1), env, k)
        = eval (e0, env,
                fn (FUN f) => eval (e1, env,
                                    fn v => f (v, k)))
      | eval (NUM m, env, k)
        = k (INT m)
      | eval (SUCC e, env, k)
        = eval (e, env, fn (INT m) => k (INT (m + 1)))
      | eval (C (x, e), env, k)
        = eval (e, Env.extend (x, FUN (fn (v, k') => k v), env),
                fn v => v)

    (*  evaluate : exp -> answer  *)
    fun evaluate e
        = eval (e, Env.empty, fn v => ANS v)
  end;
```

Evaluation of programs that are denoted by $\perp$ in the continuation semantics either diverge or raise a pattern-matching error signaling a type error.

Since the interpreter is written in CPS, the call-by-value evaluation order of ML is irrelevant—the call-by-value semantics of ISWIM$_{\mathcal{C}}$ is encoded in the structure of the interpreter.

A first-order interpreter for ISWIM$_{\mathcal{C}}$ can be obtained by defunctionalizing the expressible values and the continuations of the higher-order interpreter. The result is essentially Reynolds's Interpreter III [196]:

```
structure First_Order_Interpreter
= struct
    datatype value  = INT of int
                    | FUN of ide * exp * value Env.env
                    | CONT of cont
    and continuation = STOP
                     | ARG of exp * value Env.env * continuation
                     | SUCC of continuation
                     | FUN of value * continuation

    datatype answer = ANS of value

    (*  eval : exp * value Env.env * continuation -> answer  *)
    fun eval (VAR x, env, k)
        = cont (k, Env.lookup (x, env))
      | eval (LAM (x, e), env, k)
        = cont (k, FUN (x, e, env))
      | eval (APP (e0, e1), env, k)
        = eval (e0, env, ARG (e1, env, k))
      | eval (NUM m, env, k)
        = cont (k, INT m)
      | eval (SUCC e, env, k)
        = eval (e, env, SUCC k)
      | eval (C (x, e), env, k)
        = eval (e, Env.extend (x, CONT k, env), STOP)

    (*  cont : continuation * value -> answer  *)
    and cont (STOP, v)
        = ANS v
      | cont (ARG (e, env, k), v)
        = eval (e, env, FUN (v, k))
      | cont (SUCC k, INT m)
        = cont (k, INT (m + 1))
      | cont (FUN (FUN (x, e, env), k), v)
        = eval (e, Env.extend (x, v, env), k)
      | cont (FUN (CONT k, k'), v)
        = cont (k, v)

    (*  evaluate : exp -> answer  *)
    fun evaluate e
        = eval (e, Env.empty, STOP)
  end;
```

This interpreter is a direct ML implementation of the abstract machine presented in the next section.

3.4 Abstract machines

An abstract machine is a first-order deterministic transition system for executing programs, where the transition relation abstractly models the behavior of a hardware machine. Since Landin's SECD machine [155] abstract machines have been ubiquitous in the literature on programming language semantics. Not only have they proven useful in prototyping implementations of programming languages [6,79,115,200], but also they facilitate studies of semantic [34–37, 85, 88, 89, 116, 118, 155, 185, 186] and logical [49, 51, 53, 149, 151, 226] aspects of programming languages. (We distinguish between 'abstract' machines that operate directly on a source language and 'virtual' machines that operate on a compiled code [4, 5].)

The architecture of abstract machines, where continuations are represented as stacks of elementary contexts, makes it possible to define the semantics of control operators. The idea of using an abstract machine as a semantic medium for explicating control operators began with Landin's J-operator whose semantics was given in terms of the SECD machine [39, 154]. The method was later extensively used by Felleisen et al. who introduced the CEK machine which is by now considered the canonical abstract machine for the call-by-value λ-calculus [88, 89]. Different variants of the CEK machine were used to define the semantics of various control operators, including Felleisen's $\mathcal{C}$ and `abort` [89] as well as `control` and `prompt` [87].

Figure 3.2 shows Felleisen's abstract machine for ISWIM$_{\mathcal{C}}$ [89]. This abstract machine extends the CEK machine with the clauses for interpreting $\mathcal{C}$ and applications of a captured context. The machine consists of two kinds of configurations: *eval* and *cont* that determine two sets of transitions. The transitions of the first kind are defined over the structure of expressions and make use of an environment and of contexts representing continuations. The transitions of the second kind are defined over the structure of the contexts and are used whenever an intermediate result has been computed. The expressible values consists of integers, closures representing λ-abstractions and captured contexts representing continuations.

We can define the evaluation function $\mathsf{eval}_{\mathsf{AM}}$ in terms of the abstract machine:

Definition 3.2 *Given a program e, we define*

$$\mathsf{eval}_{\mathsf{AM}}(e) \stackrel{\text{def}}{=} v \text{ if and only if } e \Rightarrow^{+} v.$$

Since the higher-order and first-order interpreters of the previous section implement the continuation semantics and the abstract machine for ISWIM$_{\mathcal{C}}$, respectively, the following proposition is a corollary of the correctness of the functional correspondence between higher-order evaluators and abstract machines introduced in Section 3.6 and detailed in Chapter 5.

Proposition 3.2 *For any program e and for any integer m,*

$$\mathsf{eval}_{\mathsf{AM}}(e) = m \text{ if and only if } \mathcal{E}_{\mathcal{C}}[\![e]\!]\,\rho_{mt}\,\kappa_0 = up(m).$$

- Syntax: $e \quad ::= \quad x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{C}x.e$
- Expressible values (integers, closures and captured contexts):

$$v ::= m \mid [x,\, e,\, \rho] \mid E$$

- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Contexts:

$$E ::= \mathsf{STOP} \mid \mathsf{ARG}\,((e,\, \rho),\ E) \mid \mathsf{SUCC}\,(E) \mid \mathsf{FUN}\,(v,\ E)$$

- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\, \rho_{mt},\, \mathsf{STOP}\rangle_{eval}$
$\langle x,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, \rho(x)\rangle_{cont}$
$\langle \lambda x.e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, [x,\, e,\, \rho]\rangle_{cont}$
$\langle e_0\, e_1,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1,\, \rho),\ E)\rangle_{eval}$
$\langle \ulcorner m \urcorner,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, m\rangle_{cont}$
$\langle \mathit{succ}\ e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{SUCC}\,(E)\rangle_{eval}$
$\langle \mathcal{C}x.e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto E\},\, \mathsf{STOP}\rangle_{eval}$
$\langle \mathsf{ARG}\,((e_1,\, \rho),\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle e_1,\, \rho,\, \mathsf{FUN}\,(v,\ E)\rangle_{eval}$
$\langle \mathsf{SUCC}\,(E),\, m\rangle_{cont}$	$\Rightarrow$	$\langle E,\, m+1\rangle_{cont}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, E\rangle_{eval}$
$\langle \mathsf{FUN}\,(E',\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle E',\, v\rangle_{cont}$
$\langle \mathsf{STOP},\, v\rangle_{cont}$	$\Rightarrow$	v

Figure 3.2: An abstract machine for ISWIM$_{\mathcal{C}}$

3.5 Reduction semantics

A reduction semantics is a small-step operational semantics with an explicit representation of evaluation contexts [89]. In contrast to other methods of defining a language, a reduction semantics is expressed entirely in terms of syntax. Thanks to its simplicity, reduction semantics facilitates reasoning about programs and supports an intuitive understanding of the language. As demonstrated by Felleisen and Friedman [89], a reduction semantics is also a convenient formalism for specifying the semantics of control operators.

A typical description of a reduction semantics consists of the following ingredients:

- *Expressions and values.* Since a reduction semantics is defined purely in

terms of syntax, values that specify the possible outcome of the process of evaluating a program must be included in the syntax. A grammar of expressions and values of ISWIM$_{\mathcal{C}}$ reads as follows:

$$\begin{array}{lcl} e & ::= & x \mid v \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \mathcal{C}x.e \\ v & ::= & \lambda x.e \mid \ulcorner m \urcorner \end{array}$$

We consider only closed expressions and therefore variables are not included in the category of values.

- *Redexes.* Redexes define the shape of expressions that can be contracted. In general, one considers potential redexes and actual redexes. Actual redexes are the redexes that can be contracted according to one of the reduction rules. Potential redexes include actual redexes as well as stuck redexes (stuck redexes correspond to type errors and are denoted by $\perp$ in the continuation semantics). The grammar of the potential redexes in ISWIM$_{\mathcal{C}}$ is as follows:

$$p ::= v_0\, v_1 \mid \mathit{succ}\ v \mid \mathcal{C}x.e$$

and the grammar of the actual redexes is as follows:

$$r ::= (\lambda x.e)\, v \mid \mathit{succ}\ \ulcorner m \urcorner \mid \mathcal{C}x.e$$

- *Evaluation (reduction) contexts.* An evaluation context is a restricted form of a general context (i.e., an expression with a hole) that determines the positions in an expression where a reduction can occur. Evaluation contexts in ISWIM$_{\mathcal{C}}$ are the standard call-by-value contexts of the λ-calculus with constants [89]:

$$E ::= [\,] \mid E\,[[\,]\,e] \mid E\,[\mathit{succ}\,[\,]] \mid E\,[v\,[\,]]$$

An important requirement that makes a reduction semantics deterministic, and therefore practically useful for modeling evaluation, is the following property (which holds for ISWIM$_{\mathcal{C}}$) [85]:

Property 3.1 (Unique decomposition) *For any program e, either e is a value v, or it can be uniquely decomposed into an evaluation context E and a potential redex p, i.e., $e = E\,[p]$.*

- *Reduction rules.* An evaluation process in reduction semantics is driven by expression rewriting rules that rewrite entire programs according to a strategy specified by the evaluation contexts. The reduction rules for ISWIM$_{\mathcal{C}}$ are as follows [109]:

$$\begin{array}{lrcll} (\beta_\lambda) & E\,[(\lambda x.e)\, v] & \rightarrow & E\,[e\{v/x\}] & \\ (\mathit{succ}) & E\,[\mathit{succ}\ \ulcorner m \urcorner] & \rightarrow & E\,[\ulcorner m+1 \urcorner] & \\ (\mathcal{C}_\lambda) & E\,[\mathcal{C}x.e] & \rightarrow & e\{(\lambda y.\mathcal{A}(E[y]))/x\}, & y \notin FV\,(E) \end{array}$$

where $\mathcal{A}e = \mathcal{C}d.e$, $d \notin FV\,(e)$.

The reduction rule $(\mathcal{C}_\lambda)$ is non-standard in that it is context-sensitive, i.e., an evaluation context plays an active role in the contraction of a redex. Such reduction rules are characteristic of languages with control effects. Recently, Biernacka and Danvy have studied context-sensitive reduction semantics with explicit substitutions [30, 31].

- *Evaluation.* Evaluation in a reduction semantics can be defined as repeatedly decomposing a program into a unique context and a redex, contracting the redex (if one of the reduction rules applies), and plugging the contractum into the context. This process may run forever (if the original program diverges), it may get stuck (if a stuck redex is reached), or it may stop with a computed value. This description is captured by the following definition, where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$:

  Definition 3.3 *Given a program e, we define*

  $$\mathsf{eval}_{\mathsf{RS}}(e) \stackrel{\text{def}}{=} v \text{ if and only if } e \rightarrow^* v.$$

The reduction rule $(\mathcal{C}_\lambda)$ reifies a captured context E as the λ-abstraction $\lambda x.\mathcal{A}(E[x])$. Thanks to the presence of $\mathcal{A}$, applying this λ-abstraction results in discarding the current continuation. An alternative formulation of the reduction semantics for ISWIM$_\mathcal{C}$ that, as we shall see in Section 3.7, closely corresponds to the abstract-machine semantics, assumes that the rule $(\mathcal{C}_\lambda)$ is split into two rules:

$$\begin{array}{lrcl} (\mathcal{C}'_\lambda) & E\,[\mathcal{C}x.e] & \rightarrow & e\{E/x\} \\ (\beta_E) & E\,[E'\,v] & \rightarrow & E'\,[v] \end{array}$$

In this reduction semantics, the grammar of values and actual redexes is extended to account for captured contexts:

$$\begin{array}{rcl} v & ::= & \ldots \mid E \\ r & ::= & \ldots \mid E\,v \end{array}$$

The induced evaluation function $\mathsf{eval}'_{\mathsf{RS}}$, generated by the rules (β_λ), $(succ)$, $(\mathcal{C}'_\lambda)$, and (β_E), coincides with $\mathsf{eval}_{\mathsf{RS}}$ modulo the representation of captured contexts. In particular, for any program e and for any integer m,

$$\mathsf{eval}'_{\mathsf{RS}}(e) = \ulcorner m \urcorner \text{ if and only if } \mathsf{eval}_{\mathsf{RS}}(e) = \ulcorner m \urcorner.$$

The equivalence between the modified reduction semantics and the abstract machine semantics has been proved by Felleisen and Friedman's [89] who pioneered a method of extracting reduction rules from an abstract machine [89], as well as of Danvy and Nielsen's refocusing method [75] that allows one to mechanically derive an abstract machine from a reduction semantics. We briefly discuss both methods in Section 3.7.

Proposition 3.3 *For any program e and for any integer m,*

$$\mathsf{eval}'_{\mathsf{RS}}(e) = \ulcorner m \urcorner \textit{ if and only if } \mathsf{eval}_{\mathsf{AM}}(e) = m.$$

Calculi. The reduction rules in a reduction semantics are global in the sense that they are defined on decompositions of whole programs. For the purpose of evaluating programs it is sufficient, but as advocated by Felleisen [85], it is vital to be able to rewrite only fragments of programs and reason about programs locally. Felleisen's motivation was to develop a calculus suitable for algebraic manipulations of programs with control operators in the same manner as the λ-calculus facilitates manipulation of pure functional programs. Following Plotkin's program of designing the λ_v-calculus for call-by-value languages [186] Felleisen developed the $\lambda_{\mathcal{C}}$-calculus for ISWIM$_{\mathcal{C}}$ [85, 89]. Unfortunately, the equations based on the local reduction rules he devised were unsound with respect to the operational equivalence. This shortcoming was later fixed by Felleisen and Hieb [92] at a cost of introducing additional reduction rules. In their revised theory of control, the global reduction rule ($\mathcal{C}_\lambda$) is replaced with the following local reduction rules:

$$\begin{array}{lcll} v\,(\mathcal{C}x.e) & \rightarrow & \mathcal{C}x'.e\{(\lambda y.\mathcal{A}(x'\,(v\,y)))/x\}, & y \notin FV\,(v) \\ (\mathcal{C}x.e)\,e' & \rightarrow & \mathcal{C}x'.e\{(\lambda y.\mathcal{A}(x'\,(y\,e')))/x\}, & y \notin FV\,(e') \\ \mathcal{C}x.e & \rightarrow & \mathcal{C}x'.e\{(\lambda y.\mathcal{A}(x'\,y))/x\} & \\ \mathcal{C}x.\mathcal{C}x'.e & \rightarrow & \mathcal{C}x.e\{(\lambda y.\mathcal{A}y)/x'\} & \end{array}$$

Taking the reflexive, transitive, and compatible (with respect to the evaluation contexts) closure over the local versions of (β_λ) and (*succ*) and over the above rules defines an evaluation function. The equational theory built on top of the local reduction rules is sound with respect to the operational equivalence induced by this evaluation function. Recently, Ariola et al. have proposed an alternative formulation of Felleisen and Hieb's calculus by extending it with a constant *tp* denoting a top-level continuation [11].

3.6 From denotational specification to abstract machine and back

One of the contributions of this dissertation is a two-way functional correspondence between higher-order evaluators and abstract machines for the λ-calculus that we present in Chapter 5. The derivation method we propose consists of closure conversion (defunctionalization of the expressible and denotable values of the evaluator), transformation into continuation-passing style, and defunctionalization of continuations.

The key idea is to turn a higher-order evaluator into a set of tail-recursive functions and to turn these functions into a first-order transition system. As demonstrated in Chapter 2, it is the responsibility of the CPS transformation to make a function tail-recursive, and it is the role of defunctionalization to eliminate higher-order functions. Putting the two program transformations together leads to a systematic, reversible derivation method that connects seemingly unrelated evaluators, and enables one to derive new evaluators.

In Chapter 5 we connect a canonical call-by-name evaluator and Krivine's abstract machine [149], as well as a canonical call-by-value evaluator and Felleisen et al.'s CEK machine [88, 89]. These derivations reveal that Krivine's

abstract machine and the CEK machine correspond to the call-by-name and call-by-value facets of a canonical evaluator for the λ-calculus. The derivation also shows that the contexts of the abstract machines (represented as stacks of elementary contexts) are the defunctionalized continuations of an evaluator in CPS. We also reveal the denotational content (i.e., a compositional evaluator directly implementing a denotational specification) of Hannan and Miller's CLS machine [114] and of Landin's SECD machine [155]. We also discuss possible modifications of the existing abstract machines.

The functional correspondence makes it possible to mechanically derive the abstract machine corresponding to a given evaluator, and also to go from an abstract machine to the underlying evaluator (by using refunctionalization and direct-style transformation). Whereas abstract machines represent a rather low-level implementation model and facilitate reasoning about programming languages and programs, higher-order interpreters are more suitable in the process of designing and developing a programming language. The functional correspondence offers a control over both at the same time. The method was further studied and applied by Ager et al. who derived abstract machines for the call-by-need λ-calculus [6] and for the λ-calculus with computational effects [7].

In Chapter 6, we further study the functional correspondence by applying it to a simple logic programming language, propositional Prolog with cut. To account for backtracking necessary to implement Prolog's nondeterminism we present an interpreter with two layers of continuations: success and failure continuations. This interpreter can be viewed as a non-compositional implementation of the denotational semantics of Prolog with cut presented by de Bruin and de Vink [78]. Since the interpreter is in CPS, we generically parameterize it with the notion of answer, without modifying its shape, and instantiate it with two types of answers that one could require: the first solution and the number of solutions. Defunctionalizing the instantiated interpreters yields two new abstract machines for propositional Prolog. It is possible to use the same technology to derive abstract machines for predicate Prolog, by appropriately modifying the generic interpreter and adding a function finding the most general unifier of two terms [198].

The significance of the obtained results is twofold.

1. We establish an equivalence between the two seemingly unrelated semantics of Prolog, without having to prove it separately.

2. Though the abstract machines we derive should not be perceived as high-performance devices, they may model possible implementations. These results scale up to other success/failure continuation-based interpreters for non-deterministic programming languages such as Icon and Snobol [69].

The higher-order interpreter for ISWIM$_{\mathcal{C}}$ shown in Section 3.3 is already in CPS, and therefore, in order to obtain the first-order interpreter implementing the abstract machine for ISWIM$_{\mathcal{C}}$, we only needed to defunctionalize its expressible values and its continuations. As already pointed out, the first-order evaluator of Section 3.3 implements the abstract machine for ISWIM$_{\mathcal{C}}$ shown in

Section 3.4. The contexts of the abstract machine are the defunctionalized continuations of the interpreter, and the transitions from the configurations *cont* correspond to the function `cont` that interprets them. Since defunctionalization is a meaning-preserving program transformation, Proposition 3.2 follows. In Chapters 7 and 10, we derive new interpreters and abstract machines for functional languages with delimited-control operators.

It is possible to apply the derivation directly at the level of a mathematical metalanguage for denotational semantics. In his book [197, Section 12.4], Reynolds derives a first-order semantics from a continuation semantics for a functional language by defunctionalizing the domains of the functions and continuations. The obtained first-order denotational semantics represents an extension of the CEK abstract machine where the transitions are modeled by continuous functions.

3.7 From abstract machine to reduction semantics and back

Most of the existing abstract machines are built out of semantic components such as semantic values, environments, contexts (control stacks), etc. Such construction seems to make abstract machine semantics unrelated to purely syntactic reduction semantics. However, Felleisen and Friedman [89] have pioneered a constructive method of extracting a reduction semantics from a given abstract machine, and later Danvy and Nielsen [75] have developed a mechanical method of deriving an abstract machine implementing a given reduction semantics.

3.7.1 From an abstract machine to a reduction semantics

Felleisen and Friedman's approach consists in turning an abstract machine into a term-rewriting system by gradually eliminating the semantic components of the abstract machine in favor of syntactic entities [89]. We illustrate the transformation on the abstract machine for ISWIM$_{\mathcal{C}}$ (Figure 3.2). The first step is to eliminate the environment that implements delayed substitutions and to introduce actual substitutions instead. This operation requires that the values of the abstract machines are embedded in the syntax, as they get substituted for free variables in a term. Since the grammar of values includes contexts, they also have to be part of the syntax:

$$\begin{array}{lcl} e & ::= & x \mid v \mid e_0\, e_1 \mid \mathit{succ}\; e \mid \mathcal{C}x.e \\ v & ::= & \ulcorner m \urcorner \mid \lambda x.e \mid E \end{array}$$

The resulting substitution-based abstract machine is an extension of the CK machine [88]. It is a simple exercise to show that the environment-based machine and the substitution-based machine operate in lock-step and induce equivalent evaluation functions [88].

The second step is to convert the semantic representation of the contexts into an equivalent syntactic representation:

$$E ::= [\,] \mid E\,[[\,]\,e] \mid E\,[\mathit{succ}\,[\,]] \mid E\,[v\,[\,]]$$

The resulting abstract machine is a term-rewriting system, where each configuration represents a decomposition of a term into a sub-term and an evaluation context. This abstract machine is an extension of the SCC machine [88] and it is equivalent to the substitution-based abstract machine. The syntactic abstract machine reads as follows:

e	$\Rightarrow$	$\langle e,\, [\,]\rangle_{eval}$
$\langle \lambda x.e,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, \lambda x.e\rangle_{cont}$
$\langle E',\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, E'\rangle_{cont}$
$\langle e_0\, e_1,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, E\,[[\,]\,e_1]\rangle_{eval}$
$\langle \ulcorner m \urcorner,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, \ulcorner m \urcorner\rangle_{cont}$
$\langle \mathit{succ}\ e,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, E\,[\mathit{succ}\,[\,]]\rangle_{eval}$
$\langle \mathcal{C}x.e,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e\{E/x\},\, [\,]\rangle_{eval}$
$\langle E\,[[\,]\,e_1],\, v\rangle_{cont}$	$\Rightarrow$	$\langle e_1,\, E\,[v\,[\,]]\rangle_{eval}$
$\langle E\,[\mathit{succ}\,[\,]],\, \ulcorner m \urcorner\rangle_{cont}$	$\Rightarrow$	$\langle E,\, \ulcorner m+1 \urcorner\rangle_{cont}$
$\langle E\,[\lambda x.e\,[\,]],\, v\rangle_{cont}$	$\Rightarrow$	$\langle e\{v/x\},\, E\rangle_{eval}$
$\langle E\,[E'\,[\,]],\, v\rangle_{cont}$	$\Rightarrow$	$\langle E',\, v\rangle_{cont}$
$\langle [\,],\, v\rangle_{cont}$	$\Rightarrow$	v

The transitions of the abstract machine perform either decomposition or reduction. A transition $\delta \Rightarrow \delta'$ implements decomposition if the configurations δ and δ' represent the same term. Otherwise it implements reduction. Therefore, we can read the reduction rules off the abstract machine. The reduction rules derived from the syntactic abstract machine for ISWIM$_{\mathcal{C}}$ are (β_λ), (succ), $(\mathcal{C}'_\lambda)$, and (β_E), i.e., the reduction rules that constitute the modified reduction semantics of Section 3.5. Putting together the derived reduction rules (that determine the actual redexes), the grammar of terms and values, and the evaluation contexts, we obtain the modified reduction semantics for ISWIM$_{\mathcal{C}}$, presented in Section 3.5 and invented independently of the environment-based abstract machine.

3.7.2 From a reduction semantics to an abstract machine

Recently, Danvy and Nielsen have proposed a mechanical method of deriving an abstract machine corresponding to a reduction semantics satisfying the unique-decomposition property [75]. As described in Section 3.5, the process of evaluating a term in a reduction semantics consists in performing the following actions until a value (or a stuck term) is reached:

1. decomposing the term into a redex and an evaluation context,

2. contracting the redex,

3. plugging the contractum into the evaluation context.

Danvy and Nielsen have observed that the intermediate terms in the composition of the operations of plugging and decomposing can be deforested by fusing the composition into one 'refocus' function. The resulting evaluation function is defined as the reflexive-transitive closure of the operations of refocusing and contracting, and it takes the form of a pre-abstract machine [75]. Compressing its intermediate transitions yields a traditional abstract machine.

Refocusing in the modified reduction semantics for $\text{ISWIM}_{\mathcal{C}}$ yields the syntactic abstract machine of Section 3.7.1. It is then straightforward to turn this abstract machine into the environment-based one. Since the refocusing method has been proven correct [75], Proposition 3.3 follows. Alternatively, as recently demonstrated by Biernacka and Danvy [30, 31], one could start with a reduction semantics based on explicit substitutions [1] and use an enriched refocusing method to directly arrive at the environment-based abstract machine.

3.8 Conclusion

The semantics of programming languages with control operators is usually given in the form of a CPS transformation, a continuation semantics, a definitional interpreter, an abstract machine, or a reduction semantics. All these specifications were introduced and studied separately and have different domains of applications, from reasoning about programs to implementations. The systematic methods of connecting one specification with another that we have reviewed in this chapter make it possible to control different and seemingly disconnected aspects of a programming language, and in particular to view control operators from different angles. In Part II of the dissertation, we give a detailed account of the functional correspondence between evaluators and abstract machines, and we study delimited-control operators using the formalisms presented in this chapter.

Chapter 4

Delimited Continuations

In this chapter we present the theoretical foundations of static and dynamic delimited continuations that are extensively used in Part II of the dissertation. In our presentation, we follow the original articles by Danvy and Filinski on static delimited continuations [66, 67], and by Felleisen on dynamic delimited continuations [87]. We also describe our contributions and possible topics of future work. The related work is treated in Chapters 7-10.

4.1 Foundations

4.1.1 Static delimited continuations

Static delimited-control operators were defined through an extended continuation semantics and through an extended CPS transformation. We first present the CPS-based foundations of `shift` and `reset`, including the extended continuation-passing style and several type systems. We then treat the general case of the hierarchy of control operators $\texttt{shift}_n$ and $\texttt{reset}_n$.

4.1.1.1 Extended CPS

Not every λ-expression is obtainable as the result of the (call-by-value) CPS transformation. Even if the source language includes control operators for undelimited continuations, the resulting expressions are still in CPS, i.e., with no nested function applications. The control operators `shift` and `reset` were meant to exploit some of the unused expressive power in the CPS formalism and to serve as composition and identity for continuation functions, respectively. The language studied in this section is ISWIM augmented with `shift` (noted $\mathcal{S}$) and `reset` (noted $\langle\cdot\rangle$) that we call ISWIM$_{\mathcal{S}}$.

The original definition of `shift` and `reset` [67] was expressed with the use of continuation composition, extending the call-by-value CPS transformation for ISWIM$_{\mathcal{C}}$:

$$\begin{aligned}\overline{\mathcal{S}x.e} &= \lambda k.\overline{e}\{(\lambda v k'.\, k'\,(k\,v))/x\}\,(\lambda v.\, v)\\ \overline{\langle e\rangle} &= \lambda k.k\,(\overline{e}\,(\lambda v.\, v))\end{aligned}$$

Since this definition is expressed in continuation-composing style, it does not correspond to a traditional continuation semantics where continuations model tail calls enforcing a certain evaluation strategy [134]. In order to restore the property of enforcing call-by-value evaluation, Danvy and Filinski CPS-transformed the definition once more. The result is the following extended CPS transformation:

$$\overline{\overline{\mathcal{S}x.e}} = \lambda k_1 k_2.\bar{\bar{e}}\{(\lambda v k_1' k_2'.\, k_1\, v\, (\lambda w.\, k_1'\, w\, k_2'))/x\}\, (\lambda v k_2.\, k_2\, v)\, k_2$$

$$\overline{\overline{\langle e \rangle}} = \lambda k_1 k_2.\bar{\bar{e}}(\lambda v k_2.\, k_2\, v)\, (\lambda v.\, k_1\, v\, k_2)$$

This CPS transformation can be seen as the syntax-directed encoding into the term model of the metalanguage of the evaluation-order-independent meta-continuation semantics shown in Figure 4.1. The style of meta-continuation semantics (or interpreter) and of meta-continuation-passing programming is called extended continuation-passing style (ECPS). In ECPS, the continuation κ represents the current context, i.e., the remaining computation up to the nearest enclosing (dynamically determined[1]) control delimiter, and the meta-continuation γ represents the current meta-context, i.e., the remaining computation beyond the nearest enclosing control delimiter. As illustrated in Section 2.4, this double-continuation model coincides with the traditional success and failure continuations model of backtracking [167].

The meta-continuation γ, intervenes only in the clauses defining `shift` and `reset`, and can be η-reduced in the clauses defining the remaining constructs of the language. The initial continuation is $\lambda v k_2.\, k_2\, v$, and the initial meta-continuation is $\lambda v.\, up(v)$.

The control delimiter $\langle \cdot \rangle$ reinitializes the current delimited continuation. The control operator $\mathcal{S}$ captures the current delimited continuation as a function. When a captured continuation κ is resumed the then-current delimited continuation κ' is 'pushed' on the meta-continuation and κ becomes the current delimited continuation. We observe that κ is not given control over κ'.

4.1.1.2 Typed `shift` and `reset`

CPS induces several static type systems for `shift` and `reset`. The most liberal type-and-effect system that can be read off the CPS for `shift` and `reset` is due to Danvy and Filinski [66]. This type system is defined in the Curry style, using hypothetical judgements of the form

$$\Gamma;\, B \vdash e : A;\, C$$

Such judgements can be interpreted as follows: if e is evaluated in a context represented as a function from A to B, then the type of the result will be C (Γ ranges over the usual typing environments). Therefore, the expression e is allowed to change the result type from B to C. Since CPS-transformed λ-abstractions accept a continuation as an additional parameter, their type has

[1] The aspect of the dynamic scope of control delimiters has been studied by Kameyama [138–140] and by Ariola et al. [12].

- Syntax: $x \in \mathsf{Ide}$
  $e \in \mathsf{Exp} \qquad e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{S}x.e \mid \langle e \rangle$
- Domains of values and continuations:

$$\begin{array}{rrcl} f \in \mathsf{Fun} = & \mathsf{Val} & \to \mathsf{Cont} \to \mathsf{MCont} \to & \mathsf{Ans} \\ \kappa \in \mathsf{Cont} = & \mathsf{Val} & \to \mathsf{MCont} \to & \mathsf{Ans} \\ \gamma \in \mathsf{MCont} = & \mathsf{Val} & \to & \mathsf{Ans} \\ m \in \mathsf{Int} & & & \\ v \in \mathsf{Val} = & \mathsf{Int} + \mathsf{Fun} & & \\ \mathsf{Ans} = & \mathsf{Val}_\perp & & \end{array}$$

- Environments: $\rho \in \mathsf{Env} = \mathsf{Ide} \to \mathsf{Val}$
- Semantic function: $\mathcal{E}_{\mathcal{MC}} : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Cont} \to \mathsf{MCont} \to \mathsf{Ans}$

$$\begin{aligned} \mathcal{E}_{\mathcal{MC}}[\![x]\!]\,\rho\,\kappa\gamma &= \kappa\,(\rho(x))\,\gamma \\ \mathcal{E}_{\mathcal{MC}}[\![\lambda x.e]\!]\,\rho\,\kappa\gamma &= \kappa\,(\lambda v\kappa\gamma.\,\mathcal{E}_{\mathcal{MC}}[\![e]\!]\,\rho\{x \mapsto v\}\,\kappa\,\gamma)\,\gamma \\ \mathcal{E}_{\mathcal{MC}}[\![e_0\, e_1]\!]\,\rho\,\kappa\gamma &= \mathcal{E}_{\mathcal{MC}}[\![e_0]\!]\,\rho\,(\lambda f\gamma.\,\mathcal{E}_{\mathcal{MC}}[\![e_1]\!]\,\rho\,(\lambda v\gamma.\,f\,v\,\kappa\,\gamma)\,\gamma)\,\gamma \\ \mathcal{E}_{\mathcal{MC}}[\![\ulcorner m \urcorner]\!]\,\rho\,\kappa\gamma &= \kappa\,m\,\gamma \\ \mathcal{E}_{\mathcal{C}}[\![\mathit{succ}\ e]\!]\,\rho\,\kappa\gamma &= \mathcal{E}_{\mathcal{MC}}[\![e]\!]\,\rho\,(\lambda m\gamma.\,\kappa\,(m+1)\,\gamma)\,\gamma \\ \mathcal{E}_{\mathcal{MC}}[\![\mathcal{S}x.e]\!]\,\rho\,\kappa\gamma &= \mathcal{E}_{\mathcal{MC}}[\![e]\!]\,\rho\{x \mapsto \lambda v\kappa'\gamma'.\,\kappa\,v\,(\lambda w.\,\kappa'\,w\,\gamma')\}\,(\lambda v\gamma.\,\gamma\,v)\,\gamma \\ \mathcal{E}_{\mathcal{MC}}[\![\langle e \rangle]\!]\,\rho\,\kappa\gamma &= \mathcal{E}_{\mathcal{MC}}[\![e]\!]\,\rho\,(\lambda v\gamma.\,\gamma\,v)(\lambda v.\,\kappa\,v\,\gamma) \end{aligned}$$

Figure 4.1: A meta-continuation semantics for ISWIM$_{\mathcal{S}}$

to take it into account. Hence, a λ-abstraction is assigned a type of the form $A\ {}_C\!\!\to_D B$ that should be understood as a type of the function that, when applied to a value of type A in a context represented as a function from B to C, returns a value of type D. The complete Danvy and Filinski's type system for ISWIM$_{\mathcal{S}}$ is shown in Figure 4.2. In the rule (APP) it is assumed that the evaluation proceeds from left to right (this order of evaluation is encoded in CPS).

Danvy and Filinski's type system enjoys several desirable properties. Obviously, it is compatible with CPS in the sense that if we define CPS transformation on types and typing environments as

$$\begin{array}{rcl} \overline{\mathrm{int}} & = & \mathrm{int} \\ \overline{A\ {}_C\!\!\to_D B} & = & \overline{A} \to (\overline{B} \to \overline{C}) \to \overline{D} \\ \overline{\bullet} & = & \bullet \\ \overline{\Gamma, x : A} & = & \overline{\Gamma}, x : \overline{A} \end{array}$$

we can state the following proposition that validates the type system with respect to the CPS transformation:

Proposition 4.1 *If* $\Gamma;\ B \vdash e : A;\ C$ *then* $\overline{\Gamma} \vdash \overline{e} : (\overline{A} \to \overline{B}) \to \overline{C}$.

- Types: $A, B, C, D, E, F ::= \text{int} \mid A\ {}_C{\to}_D\ B$
- Typing environments: $\Gamma ::= \bullet \mid \Gamma, x : A$
- Inference rules:

$$\frac{}{\Gamma, x : A;\ B \vdash x : A;\ B}(\text{VAR}) \qquad \frac{\Gamma, x : A;\ C \vdash e : B;\ D}{\Gamma;\ E \vdash \lambda x.e : A\ {}_C{\to}_D\ B;\ E}(\text{LAM})$$

$$\frac{\Gamma;\ F \vdash e_0 : A\ {}_C{\to}_E\ B;\ D \qquad \Gamma;\ E \vdash e_1 : A;\ F}{\Gamma;\ C \vdash e_0\, e_1 : B;\ D}(\text{APP})$$

$$\frac{}{\Gamma;\ A \vdash \ulcorner m \urcorner : \text{int};\ A}(\text{INT}) \qquad \frac{\Gamma;\ A \vdash e : \text{int};\ B}{\Gamma;\ A \vdash \mathit{succ}\ e : \text{int};\ B}(\text{SUCC})$$

$$\frac{\Gamma, x : A\ {}_D{\to}_D\ B;\ E \vdash e : E;\ C}{\Gamma;\ B \vdash \mathcal{S}x.e : A;\ C}(\text{SHIFT}) \qquad \frac{\Gamma;\ C \vdash e : C;\ A}{\Gamma;\ B \vdash \langle e \rangle : A;\ B}(\text{RESET})$$

Figure 4.2: Danvy and Filinski's type system for ISWIM$_{\mathcal{S}}$

Danvy and Filinski's type system also ensures that well-typed programs with `shift` and `reset` are well-behaved under evaluation. In order to state any properties of the type system, we need to focus on a concrete instance of evaluation semantics. The kind of semantics that is particularly suitable for studying languages with types is reduction semantics [251]. In Chapter 7, we derive a reduction semantics corresponding to the meta-continuation semantics of Figure 4.1. If we slightly modify this semantics by merging the reduction rules that capture a context and apply a captured context into one rule (cf. Section 3.5), we obtain the following (equivalent) 'folklore' reduction semantics for `shift` and `reset` [141]:

$$\begin{array}{rcll} E\,[(\lambda x.e)\,v] & \to & E\,[e\{v/x\}] & \\ E\,[\mathit{succ}\ \ulcorner m \urcorner] & \to & E\,[\ulcorner m+1 \urcorner] & \\ E\,[\langle F\,[\mathcal{S}x.e] \rangle] & \to & E\,[\langle e\{(\lambda y.\langle F\,[y] \rangle)/x\} \rangle], & y \notin FV\,(F) \\ E\,[\langle v \rangle] & \to & E\,[v] & \end{array}$$

where evaluation contexts E and pure evaluation contexts F are defined as follows:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,[[\,]\,e] \mid E\,[\mathit{succ}\ [\,]] \mid E\,[v\ [\,]] \mid E\,[\langle [\,] \rangle] \\ F & ::= & [\,] \mid F\,[[\,]\,e] \mid F\,[\mathit{succ}\ [\,]] \mid F\,[v\ [\,]] \end{array}$$

Naturally, ISWIM$_{\mathcal{S}}$ satisfies the unique decomposition property, i.e., an expression is either a value (a variable or a λ-abstraction, possibly representing a captured context) or it can be uniquely decomposed as $E\,[p]$, where potential redexes p generalize the actual redexes defined by the left-hand-sides of the reduction rules.

One could venture to make the above reduction rules local, by stripping off the evaluation contexts and replacing the context-sensitive rule for `shift` with several local rules that would capture the context piecemeal. Such local reduction rules are analogous to the ones defining Felleisen's `control` and `prompt` [87] (see Section 4.1.2) and they give rise to equations defining a calculus akin to Felleisen's calculus for `control` and `prompt`. The resulting calculus is sound with respect to CPS, but it is not complete. A sound and complete axiomatization of static delimited continuations that is an extension of the calculus has been found by Kameyama and Hasegawa [142].

Using induction on type derivations one can show that the type system ensures progress and preservation of types under evaluation with the reduction semantics [251], as stated below.

Proposition 4.2 (Progress) *If* $\bullet; B \vdash e : A; C$ *then either* e *is a value or there exists a unique* e' *such that* $e \to e'$.

Proposition 4.3 (Preservation) *If* $\Gamma; B \vdash e : A; C$ *and* $e \to e'$, *then* $\Gamma; B \vdash e' : A; C$.

The two propositions together ensure Milner's slogan "well-typed programs can't go wrong", i.e., a well-typed program evaluates to a value of the same type or it diverges, but it never gets stuck or unexpectedly changes its type.

Danvy and Filinski's type system is maximally expressive at a cost of clarity. Although its type inference procedure can be easily implemented in a suitable system, typically Prolog-like, using it in programming practice can be cumbersome. The way to eliminate some of the effect annotations is to place a restriction on the answer type of contexts (continuations). We obtain a strictly weaker (for an example confirming this claim see Section 7.3.6) but considerably more accessible type system if programs are not allowed to change the type of the result. In other words, if the effect annotations in judgements are identical and therefore one of them can be removed. The restricted type system corresponds to Murthy's development [177] and is shown in Figure 4.3.

It is possible to simplify the types for `shift` and `reset` even further. In fact one can eliminate effect annotations from Murthy's type system by fixing the answer type of all contexts in which control operators can be used to one chosen type T [96, 141, 240].[2] Of course, such a restriction decreases the expressive power of the type system. However, we are not aware of any meaningful or practical examples of programs that type-check in Murthy's type system, but not in the restricted one. For example, Filinski's implementation of `shift` and `reset` in the strongly-typed Standard ML of New Jersey [96] assumes that the answer type is fixed. The simply-typed `shift` and `reset` have the following typing rules (the typing rules for the remaining constructs of ISWIM$_{\mathcal{S}}$ are that of the simply-typed λ-calculus with basic constants and functions):

$$\frac{\Gamma, x : A \to T \vdash e : T}{\Gamma \vdash \mathcal{S}x.e : A} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \langle e \rangle : T}$$

[2]It has been observed that if T is non-atomic, the type system can hide a recursive type, allowing for well-typed programs that do not terminate, even if the language does not contain any constructs for recursion [12].

- Types: $A, B, C, D ::= \text{int} \mid A \to_C B$
- Typing environments: $\Gamma ::= \bullet \mid \Gamma, x : A$
- Inference rules:

$$\frac{}{\Gamma, x : A \vdash x : A;\ B}(\text{VAR}) \qquad \frac{\Gamma, x : A \vdash e : B;\ C}{\Gamma \vdash \lambda x.e : A \to_C B;\ D}(\text{LAM})$$

$$\frac{\Gamma \vdash e_0 : A \to_C B;\ C \qquad \Gamma \vdash e_1 : A;\ C}{\Gamma \vdash e_0\, e_1 : B;\ C}(\text{APP})$$

$$\frac{}{\Gamma \vdash \ulcorner m \urcorner : \text{int};\ A}(\text{INT}) \qquad \frac{\Gamma \vdash e : \text{int};\ A}{\Gamma \vdash \mathit{succ}\ e : \text{int};\ A}(\text{SUCC})$$

$$\frac{\Gamma, x : A \to_C B \vdash e : B;\ B}{\Gamma \vdash \mathcal{S}x.e : A;\ B}(\text{SHIFT}) \qquad \frac{\Gamma \vdash e : A;\ A}{\Gamma \vdash \langle e \rangle : A;\ B}(\text{RESET})$$

Figure 4.3: Murthy's type system for ISWIM$_{\mathcal{S}}$

Both Murthy's and the simple type systems are restricted versions of Danvy and Filinski's type system. As such they are compatible with CPS (Proposition 4.1), and they ensure the progress and preservation properties (Propositions 4.2 and 4.3).

All three type systems have been studied also by Wadler who investigated delimited continuations in the context of a monadic evaluator [240].

4.1.1.3 The CPS hierarchy

Generalizing the concept of the extended continuation-passing style with one meta-continuation to an arbitrary number of layered continuations obtained by repeatedly transforming expressions into CPS leads one to the CPS hierarchy. It is then possible to define control operators $\texttt{shift}_n$ and $\texttt{reset}_n$ that generalize `shift` and `reset`. Given a semantic function $\mathcal{E}_l$ that generalizes $\mathcal{E}_{\mathcal{MC}}$ to l continuations ($l \geq 1$), the semantics of $\texttt{shift}_n$ and $\texttt{reset}_n$ ($1 \leq n < l$) is defined in Figure 4.4, where we present a compact definition with the outer continuations η-reduced. The initial continuations are θ_i.

The role of $\texttt{shift}_n$ is to abstract the first n continuations as a function that, when applied, will resume the captured continuations and will push the then-first n continuations on the $n+1$-continuation. The role of $\texttt{reset}_n$ is to reset the first n continuations by pushing them on the $n+1$-continuation. The remaining constructs of the language are defined by the standard continuation semantics since their interpretation does not depend upon meta-continuations (they can be η-reduced), and are omitted in the figure.

For $l = 1$, we obtain the continuation semantics for ISWIM. For $l = 2$, the semantics defines ISWIM$_{\mathcal{S}}$ and it coincides with $\mathcal{E}_{\mathcal{MC}}$ (`shift` and `reset` are

- Syntax ($1 \leq n < l$):

$$\begin{array}{ll} x \in \mathsf{Ide} & \\ e \in \mathsf{Exp} & e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{S}_n x.e \mid \langle e \rangle_n \end{array}$$

- Domains of values and continuations ($1 \leq i \leq l$, $0 \leq j < l$):

$$\begin{array}{rcl} f & \in & \mathsf{Fun} = \mathsf{Val} \to \mathsf{Ans}_0 \\ \kappa_i & \in & \mathsf{Cont}_i = \mathsf{Val} \to \mathsf{Ans}_i \\ & & \mathsf{Ans}_j = \mathsf{Cont}_{j+1} \to \mathsf{Ans}_{j+1} \\ & & \mathsf{Ans}_l = \mathsf{Ans} \\ & & \mathsf{Ans} = \mathsf{Val}_\perp \\ m & \in & \mathsf{Int} \\ v & \in & \mathsf{Val} = \mathsf{Int} + \mathsf{Fun} \end{array}$$

- Environments: $\rho \in \mathsf{Env} = \mathsf{Ide} \to \mathsf{Val}$

- Semantic function: $\mathcal{E}_l : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Ans}_0$

$$\begin{array}{rcl} \mathcal{E}_l[\![\mathcal{S}_n x.e]\!]\,\rho\,\kappa_1 ... \kappa_n & = & \mathcal{E}_l[\![e]\!]\,\rho\{x \mapsto c\}\,\theta_1 ... \theta_n \\ \mathcal{E}_l[\![\langle e \rangle_n]\!]\,\rho\,\kappa_1 ... \kappa_{n+1} & = & \mathcal{E}_l[\![e]\!]\,\rho\,\theta_1 ... \theta_n (\lambda v.\, \kappa_1\, v\, \kappa_2 ... \kappa_{n+1}) \end{array}$$

$$\begin{array}{c} \text{where } c = \lambda v \kappa'_1 ... \kappa'_{n+1}.\, \kappa_1\, v\, \kappa_2 ... \kappa_n (\lambda w.\, \kappa'_1\, w\, \kappa'_2 ... \kappa'_{n+1}) \\ \text{and } \theta_i = \lambda v \kappa_{i+1}.\, \kappa_{i+1}\, v \text{ for } 1 \leq i < l,\ \theta_l = \lambda v.\, up(v) \end{array}$$

Figure 4.4: A continuation semantics for the hierarchy of control operators $\texttt{shift}_n$ and $\texttt{reset}_n$

aliases for $\texttt{shift}_1$ and $\texttt{reset}_1$). Each $l+1$-semantics defines ISWIM with $\texttt{shift}_n$ and $\texttt{reset}_n$ for all $1 \leq n \leq l$.

Again, the continuation semantics induces an extended CPS transformation, where in order to eliminate operators of level n one has to introduce at least $n+1$ continuations at once to obtain an expression in CPS (the superfluous continuations can be η-reduced):

$$\begin{array}{rcl} \overline{\overline{\mathcal{S}_n x.e}} & = & \lambda k_1 ... k_n . \overline{\overline{e}}\{c/x\}\theta_1 ... \theta_n \\ \overline{\overline{\langle e \rangle_n}} & = & \lambda k_1 ... k_{n+1} . \overline{\overline{e}}\theta_1 ... \theta_n (\lambda v.\, k_1\, v\, k_2 ... k_{n+1}) \end{array}$$

$$\begin{array}{c} \text{where } c = \lambda v k'_1 ... k'_{n+1}.\, k_1\, v\, k_2 ... k_n (\lambda w.\, k'_1\, w\, k'_2 ... k'_{n+1}) \\ \text{and } \theta_i = \lambda v k_{i+1}.\, k_{i+1}\, v \text{ for } 1 \leq i < l,\ \theta_l = \lambda v.\, v \end{array}$$

Recently, Kameyama has generalized the earlier result and derived a concise set of axioms for reasoning about $\texttt{shift}_n$ and $\texttt{reset}_n$ that is sound and complete with respect to the extended CPS [141].

It is also possible to express a CPS transformation that introduces one continuation at a time and converts operators into lower-numbered ones ($n \geq 1$):

$$\overline{\mathcal{S}_{n+1}x.e} = \lambda k.\mathcal{S}_n x'.\overline{e}\{(\lambda v k'.\, k'\, \langle\!\langle x'\, (k\, v)\rangle\!\rangle_n)/x\}\, (\lambda v.\, v)$$
$$\overline{\langle\!\langle e\rangle\!\rangle_{n+1}} = \lambda k.k\, \langle\!\langle \overline{e}\, (\lambda v.\, v)\rangle\!\rangle_n$$

In case $n = 0$, we take the CPS transformation for `shift` and `reset` of the previous section. Iterating this CPS transformation yields the extended CPS transformation.

One could derive type systems for an arbitrary level of the CPS hierarchy in exactly the same way as it was done for the first level. Unfortunately, the type-and-effect systems à la Danvy and Filinski and à la Murthy would be unreasonably cluttered with effect annotations (respectively two or one annotation per level). If, however, the answer type for each level is fixed (a restriction on Murthy's type system), $\texttt{shift}_n$ and $\texttt{reset}_n$ can be given simple types in the exact same way as `shift` and `reset`. This idea has been taken up by Danvy and Yang who implemented the CPS hierarchy in Standard ML [77].

4.1.2 Dynamic delimited continuations

In this section we review the theoretical foundations of dynamic delimited continuations [87]. The language we consider is ISWIM extended with `control` (noted $\mathcal{F}$) and `prompt` (noted #) that we call ISWIM$_{\mathcal{F}}$. The original abstract machine for ISWIM$_{\mathcal{F}}$ is displayed in Figures 4.5 and 4.6. This abstract machine is an extension of the CEK machine with the transitions interpreting control operators, an additional kind of context $\mathsf{MARK}(E)$, and three operations on contexts: $\uparrow$, $\downarrow$, and $\star$. Given a context E, the partial operations $\uparrow$ and $\downarrow$ select its prefix up to the top-most mark and its suffix starting with the top-most mark, respectively. The operation $\star$ concatenates two contexts into one.

An expression $\mathcal{F}x.e$ is interpreted by extending the current environment with the prefix of the current context up to the top-most mark, and by evaluating e in this environment and in the context that is the suffix of the current context starting with the top-most mark. An expression $\#e$ is interpreted by marking the current context and evaluating e in the marked context. An application of a captured context to a value is interpreted by creating a new context that is a result of concatenating the captured context and the current context, and plugging the value into the new context.

The partiality of the operations $\uparrow$ and $\downarrow$ (undefined for STOP) manifests the requirement that programs are closed expressions of the form $\#e$. The dynamism of `control` stems from the operation concatenating contexts.

An alternative, but equivalent, semantics for `control` and `prompt` was given in terms of local (i.e., not context-sensitive) reduction rules:

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \ulcorner m \urcorner \mid \mathit{succ}\ e \mid \mathcal{F}x.e \mid \#e$
- Values: $v ::= m \mid [x,\, e,\, \rho] \mid E$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Contexts:

$$E ::= \mathsf{STOP} \mid \mathsf{ARG}\,((e,\rho),\ E) \mid \mathsf{SUCC}\,(E) \mid \mathsf{FUN}\,(v,\ E) \mid \mathsf{MARK}(E)$$

- Operations on contexts:

$$\begin{array}{rcl}
\mathsf{STOP} \star E' & \stackrel{\text{def}}{=} & E \\
(\mathsf{ARG}\,((e,\rho),\ E)) \star E' & \stackrel{\text{def}}{=} & \mathsf{ARG}\,((e,\rho),\ E \star E') \\
(\mathsf{SUCC}\,(E)) \star E' & \stackrel{\text{def}}{=} & \mathsf{SUCC}\,(E \star E') \\
(\mathsf{FUN}\,(v,\ E)) \star E' & \stackrel{\text{def}}{=} & \mathsf{FUN}\,(v,\ E \star E') \\
 & & \\
\uparrow (\mathsf{ARG}\,((e,\rho),\ E)) & \stackrel{\text{def}}{=} & \mathsf{ARG}\,((e,\rho),\ \uparrow E) \\
\uparrow (\mathsf{SUCC}\,(E)) & \stackrel{\text{def}}{=} & \mathsf{SUCC}\,(\uparrow E) \\
\uparrow (\mathsf{MARK}(E)) & \stackrel{\text{def}}{=} & \mathsf{STOP} \\
\uparrow (\mathsf{FUN}\,(v,\ E)) & \stackrel{\text{def}}{=} & \mathsf{FUN}\,(v,\ \uparrow E) \\
 & & \\
\downarrow (\mathsf{ARG}\,((e,\rho),\ E)) & \stackrel{\text{def}}{=} & \downarrow E \\
\downarrow (\mathsf{SUCC}\,(E)) & \stackrel{\text{def}}{=} & \downarrow E \\
\downarrow (\mathsf{MARK}(E)) & \stackrel{\text{def}}{=} & \mathsf{MARK}(E) \\
\downarrow (\mathsf{FUN}\,(v,\ E)) & \stackrel{\text{def}}{=} & \downarrow E
\end{array}$$

- Initial transition, transition rules, and final transition: see Figure 4.6

Figure 4.5: The original abstract machine for ISWIM$_{\mathcal{F}}$

$$\begin{array}{rcll}
(\lambda x.e)\, v & \to & e\{v/x\} & \\
\mathit{succ}\ \ulcorner m \urcorner & \to & \ulcorner m+1 \urcorner & \\
\#(\mathcal{F}x.e) & \to & \#(e\{(\lambda y.y)/x\}) & \\
(\mathcal{F}x.e)\, e' & \to & \mathcal{F}x'.e\{(\lambda y.x'\,(y\, e'))/x\}, & y \notin FV\,(e') \\
v\,(\mathcal{F}x.e) & \to & \mathcal{F}x'.e\{(\lambda y.x'\,(v\, y))/x\}, & y \notin FV\,(v) \\
\#v & \to & v &
\end{array}$$

Unlike the abstract machine, the reduction semantics generated by the above rules captures contexts piecemeal and represents them as λ-abstractions. It can be shown that an expression in ISWIM$_{\mathcal{F}}$ either is a value (a variable or a λ-abstraction) or it can be uniquely decomposed into a potential redex (actual

- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\, \rho_{mt},\, \mathsf{STOP}\rangle_{eval}$
$\langle x,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, \rho\,(x)\rangle_{cont}$
$\langle \lambda x.e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, [x,\, e,\, \rho]\rangle_{cont}$
$\langle e_0\, e_1,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\ E)\rangle_{eval}$
$\langle \ulcorner m \urcorner,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle E,\, m\rangle_{cont}$
$\langle succ\ e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{SUCC}\,(E)\rangle_{eval}$
$\langle \mathcal{F}x.e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto \uparrow E\},\, \downarrow E\rangle_{eval}$
$\langle \#e,\, \rho,\, E\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{MARK}(E)\rangle_{eval}$
$\langle \mathsf{ARG}\,((e, \rho),\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ E)\rangle_{eval}$
$\langle \mathsf{SUCC}\,(E),\, m\rangle_{cont}$	$\Rightarrow$	$\langle E,\, m+1\rangle_{cont}$
$\langle \mathsf{MARK}(E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle E,\, v\rangle_{cont}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, E\rangle_{eval}$
$\langle \mathsf{FUN}\,(E',\ E),\, v\rangle_{cont}$	$\Rightarrow$	$\langle E' \star E,\, v\rangle_{cont}$
$\langle \mathsf{STOP},\, v\rangle_{cont}$	$\Rightarrow$	v

Figure 4.6: The original abstract machine for ISWIM$_{\mathcal{F}}$, ctd.

redexes are specified by the left-hand-sides of the above reduction rules) and an evaluation context that satisfies the following grammar:

$$E \quad ::= \quad [\,] \mid E\,[[\,]\,e] \mid E\,[succ\,[\,]] \mid E\,[v\,[\,]] \mid E\,[\#[\,]]$$

The evaluation function generated by the above reduction rules and evaluation contexts is equivalent to the evaluation function induced by the abstract machine, in the sense that for any program e, evaluating e with the abstract machine yields an integer if and only if evaluating e with the reduction semantics yields the same integer.

Building the congruence relation on top of the local reduction rules, Felleisen obtained a calculus that is a refinement of his original calculus for undelimited continuations [85, 89] in that it is sound with respect to observational equivalence induced by the abstract machine (or equivalently by the reduction semantics). Later on, Felleisen and Hieb have devised an improved version of the calculus for undelimited continuations [92] that we briefly discussed in Section 3.5.

If one is interested in reduction semantics as a model of evaluation rather than an intermediate step towards a calculus, then it is possible to simplify the local reduction rules by merging the three rules for `control` into one context-sensitive rule (and taking the compatible closure with respect to evaluation

contexts for the remaining rules):

$$E\left[\#(F\left[\mathcal{F}x.e\right])\right] \;\rightarrow\; E\left[\#(e\{(\lambda y.F\left[y\right])/x\})\right], \qquad y \notin FV\left(F\right)$$

where F is a pure evaluation context. The resulting reduction semantics is equivalent to the original one, and therefore also to the abstract machine.

As opposed to static delimited continuations, dynamic delimited continuations cannot be assigned CPS-induced types. Instead, we can adapt the types for `shift` and `reset` to account for `control` and `prompt` by analyzing their reduction rules. For example, the fixed-answer type system is valid (it ensures progress and preservation) also for `control` and `prompt`. We use this fact in Chapter 10, where we present an implementation of `control` and `prompt` in ML.

Sitaram and Felleisen have proposed a Scheme implementation of dynamic delimited-control operators corresponding to `control` and `prompt` [213], and on top of that implementation they have built a hierarchy of dynamic control operators. Since their development is not based on the formal semantics for `control` and `prompt`, it is difficult to see whether the implementation agrees with the specifications reviewed in this section.

4.1.3 Conclusion

Static delimited continuations have been introduced from a denotational, CPS-based standpoint. Dynamic delimited continuations have been introduced from an operational standpoint. When we compare static and dynamic control operators on a common ground, like the context-sensitive reduction semantics for `shift`/`reset` and for `control`/`prompt`, we notice a close resemblance between them. The control delimiters `reset` and `prompt` are essentially aliases of each other. The only difference can be found in the way a captured context is reified in the rules for `shift` and `control`. This seemingly minor difference determines the borderline between the CPS-compatible world and beyond.

Other dynamic control operators that can be defined by simply exploring the combinatorial space of including or not a control delimiter in the right-hand-side of the context-sensitive reduction rule, are `shift0` (noted $\mathcal{S}_0$) and `control0` (noted $\mathcal{F}_0$):

$$\begin{array}{rcl} E\left[\#(F\left[\mathcal{S}_0x.e\right])\right] & \rightarrow & E\left[(e\{(\lambda y.\#F\left[y\right])/x\})\right] \\ E\left[\#(F\left[\mathcal{F}_0x.e\right])\right] & \rightarrow & E\left[(e\{(\lambda y.F\left[y\right])/x\})\right] \end{array}$$

Although we are not aware of any specific applications of these operators, our results for `control`/`prompt` could be straightforwardly adjusted to account for `shift0`/`reset0` and `control0`/`prompt0` [209].

4.2 Contributions

We give a brief description of our contributions to the area of delimited continuations that are presented in Chapters 6-10.

Chapter 6. We consider continuation-based interpreters for propositional Prolog with cut which are examples of continuation-based non-deterministic programming. As such, the interpreters can be mapped to direct style and programmed with control operators for delimited continuations $\texttt{shift}_1$, $\texttt{reset}_1$, $\texttt{shift}_2$, and $\texttt{reset}_2$. The direct-style interpreters provide new applications of programming with delimited-control operators and we present one of them. The results scale to interpreters for other nondeterministic languages.

Chapter 7. We derive an abstract machine for the first level of the CPS hierarchy from an interpreter implementing the meta-continuation semantics $\mathcal{E}_{\mathcal{MC}}$. We then use Felleisen and Friedman's method to extract a reduction semantics for `shift` and `reset` by turning the environment-based abstract machine into a term rewriting system.

We repeat the whole procedure for the more complex case of an arbitrary level l of the CPS hierarchy, obtaining an abstract machine and a reduction semantics corresponding to the semantics $\mathcal{E}_l$.

The abstract machines we obtain can serve as a model for implementations of functional languages with multiple levels of control operators for static delimited continuations. They also make it possible to reason inductively (by induction on the number of transitions) about the evaluation of programs with delimited-control operators.

The reduction semantics we obtain clarify the distinction between the static character of `shift`/`reset` and the dynamic character of `control` and `prompt`. They also make it possible to model evaluation of programs with delimited-control operators purely in terms of syntax. For example, reduction semantics is indispensable when it comes to designing a type system for a language. It is also possible to make the reduction rules local and obtain a confluent calculus for reasoning about programs.

Abstract machines and reduction semantics for the CPS hierarchy have also been considered by Murthy [177]. However, his development was not based on any systematic derivation method connecting his results with the definition of the hierarchy. In contrast, our results presented in Chapter 7 correspond directly to the original specification of $\texttt{shift}_n$ and $\texttt{reset}_n$.

The abstract machine for the first level of the CPS hierarchy uses a context and a meta-context (a stack of contexts) instead of just one context with marks, like the original machine for dynamic delimited continuations shown in this chapter. We modify the machine for `shift` and `reset` to obtain an abstract machine for `control` and `prompt`. This abstract machine is equivalent to Felleisen's machine presented in this section, but it does not require the operations $\uparrow$ and $\downarrow$. It hinges, however, on the third operation—concatenating contexts $\star$. We characterize in what sense this abstract machine falls out of the range of defunctionalization making dynamic delimited continuations incompatible with CPS.

We also present new applications of static delimited continuations: a spectrum of solutions to the classical problem of listing list prefixes [57] and normalization by evaluation for a hierarchical language of units and products. The

problem of listing list prefixes reveals interesting typing issues that we discuss in reference to the type systems presented in this chapter. Finally, the problem of normalization by evaluation illustrates certain efficiency issues and a trade-off between using explicit continuations and accessing continuations with control operators.

Chapter 8. We formalize and prove the folklore theorem that the static delimited-control operators `shift` and `reset` can be simulated in terms of the dynamic delimited-control operators `control` and `prompt`. We base our proof on the two abstract machines derived in Chapter 7.

Chapter 9. We present the first examples that meaningfully exploit the difference between static and dynamic delimited continuations. As we prove in Chapter 8, every program that can be expressed in terms of `shift`/`reset` can be expressed in terms of `control`/`prompt`. Recent discoveries [35, 84, 146, 209] indicate that the opposite is also true. So far, however, all the existing programming examples could be expressed most naturally using `shift`/`reset`, and the ones that were written with `control`/`prompt` merely simulated static continuations (according to the simulation theorem of Chapter 8).

We show that breadth-first tree traversal in direct style and with no auxiliary parameter takes advantage of the dynamic concatenation of stack frames in the semantics of `control` and therefore can be expressed most naturally with dynamic delimited continuations. The examples we consider are depth-first and breadth-first versions of the samefringe problem and Okasaki's functional pearl—tree numbering [182]. The approach scales to other traversal-based algorithms.

We also point out how static delimited continuations naturally give rise to the notion of control stack whereas dynamic delimited continuations can be made to account for a notion of 'control queue.'

Chapter 10. We present a new abstract machine that accounts for dynamic delimited continuations. We prove the correctness of this machine with respect to the definitional abstract machine (of Chapter 7). Unlike the definitional abstract machine, the new abstract machine is in defunctionalized form, which makes it possible to state the corresponding higher-order evaluator and its concomitant CPS transformer. The resulting 'dynamic continuation-passing style' threads a list of trailing delimited continuations and a meta-continuation, i.e., it is a continuation+state-passing style. This style is equivalent to, but simpler than the one recently proposed by Shan [209], and structurally similar to the one recently proposed by Dybvig, Peyton Jones, and Sabry [84].

We illustrate that the new machine is more efficient than the definitional one, and we show that the notion of computation induced by the corresponding evaluator takes the form of a monad. We can, therefore, give the dynamic control operators a parallel treatment to that of Wadler who studied static control operators in a monadic setting [240].

We also present examples of using dynamic CPS transformation in programming practice, and show how it supports the understanding of dynamic control operators.

Finally, we derive a new simulation of dynamic delimited continuations in terms of static ones. Together with the converse simulation proved in Chapter 8, this result demonstrates that static and dynamic delimited continuations are equally expressive.

Dynamic CPS also makes it possible to implement dynamic delimited continuations in ML or Scheme using first-class continuations and two storage cells.

The presented results can be easily adapted to account for the dynamic hierarchy of control operators $\texttt{control}_n/\texttt{prompt}_n$ that has been informally described by Sitaram and Felleisen [213], as well as for other dynamic control operators `shift0`/`reset0` and `control0`/`prompt0` [209] (and their hierarchies).

4.3 Future work

We briefly describe some of the possible topics for future work.

4.3.1 Back to direct style

Transformation to direct style is a program transformation that aims at mapping continuation-passing programs back to direct style. The direct-style transformation for pure functional languages has been studied by Danvy [59] who observed that terms in CPS use continuation parameters linearly according to a stack-like discipline. Exploiting this observation Danvy has devised a one-pass direct-style transformation that is the left inverse of Plotkin's call-by-value CPS transformation. This direct-style transformation has been later conservatively extended by Danvy and Lawall to account for first-class undelimited continuations [72]. First-class continuations break the stack-like discipline of continuations and therefore require a more advanced syntactic analysis.

Direct-style transformations have found their place in the programmer's toolbox, facilitating mechanical and meaning preserving translations between programs written in CPS and direct style. They prove particularly useful when it comes to writing and manipulating programs using control operators. Instead of working in direct style, one can perform one's tasks in CPS and mechanically transform the results to direct style.

Since programs using static control operators `shift` and `reset` correspond to programs in continuation-composing style, it would be highly desirable to be able to mechanically transform such terms into their direct-style counterparts. Composing such a transformation with Danvy's direct-style transformation would yield a transformation from extended CPS to direct style. Although this task appears to be nontrivial, as it is difficult to characterize a discipline according to which continuations in continuation-composing style can be used, finding the direct-style transformation for delimited continuations seems a worthwhile goal.

4.3.2 Curry-Howard isomorphism

Finding a logical interpretation of delimited continuations has been a long-standing and nontrivial challenge. Whereas undelimited continuations correspond via the Curry-Howard isomorphism to classical logic (Section 3.2), the logical content of delimited continuations is still unknown. The main complication is introduced by control delimiters which have a dynamic scope and require suitable effect annotations. This issue was first identified by Kameyama who studied a variant of static delimited continuations with static scope [138, 140].

Recently, Ariola et al. have considerably advanced the study of the logical interpretation of delimited continuations by further exploring the dynamic nature of control delimiters [12]. They have shown that static delimited continuations can be embedded in a calculus with a single dynamic variable $\widehat{tp}$ denoting the top-level continuation (this approach extends their previous results for undelimited continuations, where the top-level continuation was represented by a single constant tp). This calculus is then given three static semantics that strictly correspond to the three type (and effect) systems presented in Section 4.1.1. Finally, the most expressive calculus, based on Danvy and Filinski's type system, is embedded in a calculus that via the Curry-Howard isomorphism corresponds to classical subtractive logic [52]. Subtraction $A - B$ is a connective dual to implication, classically equivalent to $A \wedge \neg B$, and it serves Ariola et al. to pair information about a term (A) and about an evaluation context ($\neg B$).

While this work is a clear step forward in the process of searching for a logical content of delimited continuations, nobody knows whether subtractive logic is the ultimate answer:

1. The embedding defined by Ariola et al. uses only a fragment of subtractive logic and even for this fragment reductions in the source calculus are not directly mapped to reductions in the target calculus. Does this mean that subtractive logic is too general? Could the embedding be refined? What is the computational content of the unused fragment of subtractive logic?

2. Although the considered control operators are CPS compatible, CPS is ignored in the development. Would the result be different if the approach were based on CPS?

It would be interesting to see whether CPS can help us answer the above questions and build logical foundations for static delimited continuations, as it helped us build their operational foundations, and whether it may lead to a refinement of Ariola et al.'s results and/or to a different, perhaps more precise, logical interpretation of static delimited continuations. Having obtained satisfactory results for static delimited continuations, we could then either adapt them to dynamic delimited continuations or take advantage of a similar approach using dynamic CPS.

4.3.3 Categorical semantics

A different purely theoretical question that is connected to the problem of logical content is how to assign a sound categorical semantics for delimited continuations. There has been a number of articles devoted to categorical interpretations of CPS and of first-class undelimited continuations [95, 128, 129, 183, 206, 226, 231], but none concerning delimited continuations.

Among many others, there is Hofmann's work on a sound and complete axiomatization of call-by-value control operators (Felleisen's $\mathcal{C}$ and $\mathcal{A}$) for undelimited continuations, with respect to a class of models in (nearly) cartesian closed categories [128]. It would be instructive to investigate whether Hofmann's approach can be extended or adjusted to account for extended CPS and static delimited continuations, and if so, how the obtained axioms would relate to Hasegawa and Kameyama's untyped axioms. Another possible angle of approaching the task could be to attempt to follow Streicher and Reus's results [226] of interpreting a call-by-name version of Felleisen's $\mathcal{C}$ in the so-called category of negated domains [152].

Most of the proposed categorical interpretations of undelimited continuations take advantage of the fact that undelimited continuations correspond to classical logic via the Curry-Howard isomorphism. This observation suggests that if the logical interpretation of delimited continuations were known, we would be in much better position to design a suitable categorical semantics.

Part II

Publications

Chapter 5

A Functional Correspondence between Evaluators and Abstract Machines

with Mads Sig Ager, Olivier Danvy and Jan Midtgaard [5]

Abstract

We bridge the gap between functional evaluators and abstract machines for the λ-calculus, using closure conversion, transformation into continuation-passing style, and defunctionalization.

We illustrate this approach by deriving Krivine's abstract machine from an ordinary call-by-name evaluator and by deriving an ordinary call-by-value evaluator from Felleisen et al.'s CEK machine. The first derivation is strikingly simpler than what can be found in the literature. The second one is new. Together, they show that Krivine's abstract machine and the CEK machine correspond to the call-by-name and call-by-value facets of an ordinary evaluator for the λ-calculus.

We then reveal the denotational content of Hannan and Miller's CLS machine and of Landin's SECD machine. We formally compare the corresponding evaluators and we illustrate some degrees of freedom in the design spaces of evaluators and of abstract machines for the λ-calculus with computational effects.

Finally, we consider the Categorical Abstract Machine and the extent to which it is more of a virtual machine than an abstract machine.

5.1 Introduction and related work

In Hannan and Miller's words [114, Section 7], there are fundamental differences between denotational definitions and definitions of abstract machines. While a functional programmer tends to be familiar with denotational definitions [221], he typically wonders about the following issues:

Design: How does one design an abstract machine? How were existing abstract machines, starting with Landin's SECD machine, designed? How does one make variants of an existing abstract machine? How does one extend an

existing abstract machine to a bigger source language? How does one go about designing a new abstract machine? How does one relate two abstract machines?

Correctness: How does one prove the correctness of an abstract machine? Assuming it implements a reduction strategy, should one prove that each of its transitions implements a part of this strategy? Or should one characterize it in reference to a given evaluator, or to another abstract machine?

A variety of answers to these questions can be found in the literature. Landin invented the SECD machine as an implementation model for functional languages [155], and Plotkin proved its equivalence to an evaluation function [186, Section 2]. Krivine discovered an abstract machine from a logical standpoint [148], and Crégut proved its correctness in reference to a reduction strategy; he also generalized it from weak to strong normalization [50]. Curien discovered the Categorical Abstract Machine from a categorical standpoint [49, 54]. Felleisen et al. invented the CEK machine from an operational standpoint [88, 89, 102]. Hannan and Miller discovered the CLS machine from a proof-theoretical standpoint [114]. Many people derived, invented, or (re-)discovered Krivine's machine. Many others proposed modifications of existing machines. And recently, Rose presented a method to construct abstract machines from reduction rules [199], while Hardin, Maranget, and Pagano presented a method to extract the reduction strategy of a machine by extracting axioms from its transitions and structural rules from its architecture [115].

In this article, we propose one constructive answer to all the questions above. We present a correspondence between functional evaluators and abstract machines based on a two-way derivation consisting of closure conversion, transformation into continuation-passing style (CPS), and defunctionalization. This two-way derivation lets us connect each of the machines above with an evaluator, and makes it possible to echo variations in the evaluator into variations in the abstract machine, and vice versa. The evaluator clarifies the reduction strategy of the corresponding machine. The abstract machine makes the evaluation steps explicit in a transition system.

Some machines operate on λ-terms directly whereas others operate on compiled λ-terms expressed with an instruction set. Accordingly, we distinguish between *abstract* machines and *virtual* machines in the sense that virtual machines have an instruction set and abstract machines do not; instead, abstract machines directly operate on source terms and do not need a compiler from source terms to instructions. (Grégoire and Leroy make the same point when they talk about a *compiled* implementation of strong reduction [108].)

Prerequisites: ML, abstract machines, λ-interpreters, CPS transformation, defunctionalization, and closure conversion. We use ML as a meta-language, and we assume a basic familiarity with Standard ML and reasoning about ML programs. Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language. The warnings could be avoided with an option type or with an explicit exception, at the

price of readability and direct relation to the usual mathematical specifications of abstract machines.

It would be helpful to the reader to know at least one of the machines considered in the rest of this article, be it Krivine's machine, the CEK machine, the CLS machine, the SECD machine, or the Categorical Abstract Machine. It would also be helpful to have already seen a λ-interpreter written in a functional language [103, 196, 220, 239]. In particular, we make use of Strachey's notions of expressible values, i.e., the values obtained by evaluating an expression, and denotable values, i.e., the values denoted by identifiers [224].

We make use of the CPS transformation [68,202]: a term is CPS-transformed by naming all its intermediate results, sequentializing their computation, and introducing continuations. Plotkin was the first to establish the correctness of the CPS transformation [186].

We also make use of Reynolds's defunctionalization [196]: defunctionalizing a program amounts to replacing each of its function spaces by a data type and an apply function; the data type enumerates all the function abstractions that may give rise to inhabitants of this function space in this program [74]. Nielsen, Banerjee, Heintze, and Riecke have established the correctness of defunctionalization [18, 180].

A particular case of defunctionalization is closure conversion: in an evaluator, closure conversion amounts to replacing each of the function spaces in expressible and denotable values by a tuple, and inlining the corresponding apply function.

We would like to stress that all the concepts used here are elementary ones, and that the significance of this article is the one-fits-all derivation between evaluators and abstract machines.

Overview: The rest of this article is organized as follows. We first consider a call-by-name and a call-by-value evaluator, and we present the corresponding machines, which are Krivine's machine and the CEK machine. We then turn to the CLS machine and the SECD machine, and we present the corresponding evaluators. We finally consider the Categorical Abstract Machine. For simplicity, we do not cover laziness and sharing, but they come for free by threading a heap of updateable thunks in a call-by-name evaluator [6].

5.2 Call-by-name, call-by-value, and the λ-calculus

We first go from a call-by-name evaluator to Krivine's abstract machine (Section 5.2.1) and then from the CEK machine to a call-by-value evaluator (Section 5.2.2). Krivine's abstract machine operates on de Bruijn-encoded λ-terms, and the CEK machine operates on λ-terms with names. Starting from the corresponding evaluators, it is simple to construct a version of Krivine's abstract machine that operates on λ-terms with names, and a version of the CEK machine that operates on de Bruijn-encoded λ-terms (Section 5.2.3).

The derivation steps consist of closure conversion, transformation into continuation-passing style, and defunctionalization of continuations. Closure converting expressible and denotable values makes the evaluator first order. CPS

transforming the evaluator makes its control flow manifest as a continuation. Defunctionalizing the continuation materializes the control flow as a first-order data structure. The result is a transition function, i.e., an abstract machine.

5.2.1 From a call-by-name evaluator to Krivine's machine

Krivine's abstract machine [50] operates on de Bruijn-encoded λ-terms. In this representation, identifiers are represented by their lexical offset, as traditional since Algol 60 [246].

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
```

Programs are closed terms.

5.2.1.1 A higher-order and compositional call-by-name evaluator

Our starting point is the canonical call-by-name evaluator for the λ-calculus [220, 222]. This evaluator is compositional in the sense of denotational semantics [204, 222, 250] and higher order (`Eval0.eval`). It is compositional because it solely defines the meaning of each term as a composition of the meaning of its parts. It is higher order because the data types `Eval0.denval` and `Eval0.expval` contain functions: denotable values (`denval`) are thunks and expressible values (`expval`) are functions. An environment is represented as a list of denotable values. A program is evaluated in an empty environment (`Eval0.main`).

```
structure Eval0
= struct
    datatype denval = THUNK of unit -> expval
         and expval = FUNCT of denval -> expval

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = let val (THUNK thunk) = List.nth (e, n)
          in thunk ()
          end
      | eval (ABS t, e)
        = FUNCT (fn v => eval (t, v :: e))
      | eval (APP (t0, t1), e)
        = let val (FUNCT f) = eval (t0, e)
          in f (THUNK (fn () => eval (t1, e)))
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

An identifier denotes a thunk. Evaluating an identifier amounts to forcing this thunk. Evaluating an abstraction yields a function. Evaluating an application requires the evaluation of the sub-expression in position of function; the intermediate result is a function, which is applied to a thunk.

5.2.1.2 From higher-order functions to closures

We now closure-convert the evaluator of Section 5.2.1.1.

In `Eval0`, the function spaces in the data types of denotable and expressible values are only inhabited by instances of the λ-abstractions `fn v => eval (t, v :: e)` in the meaning of abstractions, and `fn () => eval (t1, e)` in the meaning of applications. Each of these λ-abstractions has two free variables: a term and an environment. We defunctionalize these function spaces into closures [74, 155, 196], and we inline the corresponding apply functions.

```
structure Eval1
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list

    (*  eval : term * denval list -> expval  *)
    fun eval (IND n, e)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e')
          end
      | eval (ABS t, e)
        = FUNCT (t, e)
      | eval (APP (t0, t1), e)
        = let val (FUNCT (t, e')) = eval (t0, e)
          in eval (t, (THUNK (t1, e)) :: e')
          end

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

The definition of an abstraction is now `Eval1.FUNCT (t, e)` instead of `fn v => Eval0.eval (t, v :: e)`, and its use is now `Eval1.eval (t, (Eval1.THUNK (t1, e)) :: e')` instead of `f (Eval0.THUNK (fn () => Eval0.eval (t1, e)))`. Similarly, the definition of a thunk is now `Eval1.THUNK (t1, e)` instead of `Eval0.THUNK (fn () => Eval0.eval (t1, e))` and its use is `Eval1.eval (t, e')` instead of `thunk ()`.

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 5.1 (full correctness)
For any program `p : term`, `Eval0.main p` *and* `Eval1.main p` *either both diverge or yield expressible values that are related by closure conversion.*

5.2.1.3 CPS transformation

We transform `Eval1.eval` into continuation-passing style.[1] Doing so makes it tail recursive.

```
structure Eval2
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list

    (*  eval : term * denval list * (expval -> 'a)  *)
    (*         -> 'a                                 *)
    fun eval (IND n, e, k)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', k)
          end
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn (FUNCT (t, e'))
                       => eval (t,
                                (THUNK (t1, e)) :: e',
                                k))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

The following proposition is a corollary of the correctness of the CPS transformation.

Proposition 5.2 (full correctness)
For any program `p : term`, `Eval1.main p` *and* `Eval2.main p` *either both diverge or yield expressible values that are structurally equal.*

5.2.1.4 Defunctionalizing the continuations

The function space of the continuation is inhabited by instances of two λ-abstractions: the initial one in the definition of `Eval2.main`, with no free variables, and one in the meaning of an application, with three free variables. To defunctionalize the continuation, we thus define a data type `cont` with two summands and the corresponding `apply_cont` function to interpret these summands.

[1] Since programs are closed, applying `List.nth` cannot fail and therefore it denotes a total function. We thus keep it in direct style [71].

```
structure Eval3
= struct
    datatype denval = THUNK of term * denval list
         and expval = FUNCT of term * denval list
         and   cont = CONT0
                    | CONT1 of term * denval list * cont

    (*  eval : term * denval list * cont -> expval  *)
    fun eval (IND n, e, k)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', k)
          end
      | eval (ABS t, e, k)
        = apply_cont (k, FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT1 (t1, e, k))
    and apply_cont (CONT0, v)
        = v
      | apply_cont (CONT1 (t1, e, k), FUNCT (t, e'))
        = eval (t, (THUNK (t1, e)) :: e', k)

    (*  main : term -> expval  *)
     fun main t
         = eval (t, nil, CONT0)
  end
```

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 5.3 (full correctness)
For any program `p : term`*,* `Eval2.main p` *and* `Eval3.main p` *either both diverge or yield expressible values that are structurally equal.*

We identify that `cont` is a stack of thunks, and that the transitions are those of Krivine's abstract machine.

5.2.1.5 Krivine's abstract machine

To obtain the canonical definition of Krivine's abstract machine, we abandon the distinction between denotable and expressible values and we use thunks instead, we represent the defunctionalized continuation as a list of thunks instead of a data type, and we inline `apply_cont`.

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t,\, e]$
- Environments: $e ::= \mathit{nil} \mid v :: e$
- Stacks of expressible values: $s ::= \mathit{nil} \mid v :: s$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t,\, \mathit{nil},\, \mathit{nil}\rangle$
$\langle n,\, e,\, s\rangle$	$\Rightarrow$	$\langle t,\, e',\, s\rangle,\ \mathit{where}\ [t,\, e'] = \mathit{nth}(e,\, n)$
$\langle \lambda t,\, e,\, [t',\, e'] :: s\rangle$	$\Rightarrow$	$\langle t,\, [t',\, e'] :: e,\, s\rangle$
$\langle t_0\, t_1,\, e,\, s\rangle$	$\Rightarrow$	$\langle t_0,\, e,\, [t_1,\, e] :: s\rangle$
$\langle \lambda t,\, e,\, \mathit{nil}\rangle$	$\Rightarrow$	$[t,\, e]$

Figure 5.1: Krivine's abstract machine

```
structure Eval4
= struct
    datatype thunk = THUNK of term * thunk list

    (*  eval : term * thunk list * thunk list  *)
    (*      -> term * thunk list               *)
    fun eval (IND n, e, s)
        = let val (THUNK (t, e')) = List.nth (e, n)
          in eval (t, e', s)
          end
      | eval (ABS t, e, nil)
        = (ABS t, e)
      | eval (ABS t, e, (t', e') :: s)
        = eval (t, (THUNK (t', e')) :: e, s)
      | eval (APP (t0, t1), e, s)
        = eval (t0, e, (t1, e) :: s)

     (*  main : term -> term * thunk list  *)
     fun main t
         = eval (t, nil, nil)
  end
```

The following proposition is straightforward to prove.

Proposition 5.4 (full correctness)
For any program `p : term`, `Eval3.main p` *and* `Eval4.main p` *either both diverge or yield expressible values that are structurally equal.*

- Source syntax: $t ::= w \mid t_0\, t_1$
  $w ::= x \mid \lambda x.t$
- Expressible values (closures): $v ::= [x,\, t,\, e]$
- Environments: $e ::= e_{mt} \mid e[x \mapsto v]$
- Evaluation contexts: $k ::= \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k)$
- Initial transition, transition rules (two kinds), and final transition:

t	$\Rightarrow$	$\langle t,\, e_{mt},\, \texttt{stop}\rangle_{eval}$
$\langle w,\, e,\, k\rangle_{eval}$	$\Rightarrow$	$\langle k,\, \gamma(w, e)\rangle_{cont}$
$\langle t_0\, t_1,\, e,\, k\rangle_{eval}$	$\Rightarrow$	$\langle t_0,\, e,\, \texttt{arg}(t_1, e, k)\rangle_{eval}$
$\langle \texttt{arg}(t_1, e, k),\, v\rangle_{cont}$	$\Rightarrow$	$\langle t_1,\, e,\, \texttt{fun}(v, k)\rangle_{eval}$
$\langle \texttt{fun}([x,\, t,\, e], k),\, v\rangle_{cont}$	$\Rightarrow$	$\langle t,\, e[x \mapsto v],\, k\rangle_{eval}$
$\langle \texttt{stop},\, v\rangle_{cont}$	$\Rightarrow$	v

where $\gamma(x, e) = e(x)$
$\gamma(\lambda x.t, e) = [x,\, t,\, e]$

Figure 5.2: The CEK abstract machine

For comparison with `Eval4`, the canonical definition of Krivine's abstract machine [50, 112, 148] is shown in Figure 5.1. Variables n are represented by their de Bruijn index, and the abstract machine operates on triples consisting of a term, an environment, and a stack of expressible values.

Each line in the canonical definition matches a clause in `Eval4`. We conclude that Krivine's abstract machine can be seen as a defunctionalized, CPS-transformed, and closure-converted version of the standard call-by-name evaluator for the λ-calculus. This evaluator evidently implements Hardin, Maranget, and Pagano's K strategy [115, Section 3].

5.2.2 From the CEK machine to a call-by-value evaluator

The CEK machine [88,89,102] operates on λ-terms with names and distinguishes between values and computations in their syntax (i.e., it distinguishes trivial and serious terms, in Reynolds's words [196]).

```
datatype term = VALUE of value
              | COMP of comp
    and value = VAR of string    (* name *)
              | LAM of string * term
    and  comp = APP of term * term
```

Programs are closed terms.

5.2.2.1 The CEK abstract machine

Our starting point is displayed in Figure 5.2 [102, Figure 2, page 239]. Variables x are represented by their name, and the abstract machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a term, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and an expressible value. Environments are extended in the `fun`-transition, and consulted in γ. The empty environment is denoted by e_{mt}.

This specification is straightforward to program in ML:

```
signature ENV
= sig
    type 'a env
    val mt : 'a env
    val lookup : 'a env * string -> 'a
    val extend : string * 'a * 'a env -> 'a env
  end
```

Environments are represented as a structure `Env : ENV` containing a representation of the empty environment `mt`, an operation `lookup` to retrieve the value bound to a name in an environment, and an operation `extend` to extend an environment with a binding.

```
structure Eval0
= struct
    datatype expval = CLOSURE of string * term * expval Env.env
    datatype ev_context
             = STOP
             | ARG of term * expval Env.env * ev_context
             | FUN of expval * ev_context

    (*  eval : term * expval Env.env * ev_context  *)
    (*         -> expval                            *)
    fun eval (VALUE v, e, k)
        = continue (k, eval_value (v, e))
      | eval (COMP (APP (t0, t1)), e, k)
        = eval (t0, e, ARG (t1, e, k))
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (x, t, e)
    and continue (STOP, w)
        = w
      | continue (ARG (t1, e, k), w)
        = eval (t1, e, FUN (w, k))
      | continue (FUN (CLOSURE (x, t, e), k), w)
        = eval (t, Env.extend (x, w, e), k)
```

```
    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt, STOP)
  end
```

5.2.2.2 Refunctionalizing the evaluation contexts into continuations

We identify that the data type `ev_context` and the function `continue` are a defunctionalized representation. The corresponding higher-order evaluator reads as follows. As can be observed, it is in continuation-passing style.

```
structure Eval1
= struct
    datatype expval = CLOSURE of string * term * expval Env.env

    (*  eval : term * expval Env.env * (expval -> 'a)  *)
    (*         -> 'a                                    *)
    fun eval (VALUE v, e, k)
        = k (eval_value (v, e))
      | eval (COMP (APP (t0, t1)), e, k)
        = eval (t0, e,
                fn (CLOSURE (x, t, e'))
                => eval (t1, e,
                         fn w
                         => eval (t, Env.extend (x, w, e'), k)))
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (x, t, e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt, fn w => w)
  end
```

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 5.5 (full correctness)
For any program `p : term`, `Eval0.main p` *and* `Eval1.main p` *either both diverge or yield expressible values that are structurally equal.*

5.2.2.3 Back to direct style

CPS-transforming the following direct-style evaluator yields the evaluator of Section 5.2.2.2 [59].

```
structure Eval2
= struct
    datatype expval = CLOSURE of string * term * expval Env.env
```

```
    (*  eval : term * expval Env.env -> expval  *)
    fun eval (VALUE v, e)
        = eval_value (v, e)
      | eval (COMP (APP (t0, t1)), e)
        = let val (CLOSURE (x, t, e')) = eval (t0, e)
              val w = eval (t1, e)
          in eval (t, Env.extend (x, w, e'))
          end
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (x, t, e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt)
  end
```

The following proposition is a corollary of the correctness of the direct-style transformation.

Proposition 5.6 (full correctness)
For any program `p : term`, `Eval1.main p` *and* `Eval2.main p` *either both diverge or yield expressible values that are structurally equal.*

5.2.2.4 From closures to higher-order functions

We observe that the closures, in `Eval2`, are defunctionalized representations with an apply function inlined. The corresponding higher-order evaluator reads as follows.

```
structure Eval3
= struct
    datatype expval = CLOSURE of expval -> expval

    (*  eval : term * expval Env.env -> expval  *)
    fun eval (VALUE v, e)
        = eval_value (v, e)
      | eval (COMP (APP (t0, t1)), e)
        = let val (CLOSURE f) = eval (t0, e)
              val w = eval (t1, e)
          in f w
          end
    and eval_value (VAR x, e)
        = Env.lookup (e, x)
      | eval_value (LAM (x, t), e)
        = CLOSURE (fn w => eval (t, Env.extend (x, w, e)))

    (*  main : term -> expval  *)
    fun main t
        = eval (t, Env.mt)
  end
```

The following proposition is a corollary of the correctness of defunctionalization.

Proposition 5.7 (full correctness)
For any program `p : term`*,* `Eval2.main p` *and* `Eval3.main p` *either both diverge or yield expressible values that are related by closure conversion.*

5.2.2.5 A higher-order and compositional call-by-value evaluator

The result in `Eval3` is a call-by-value evaluator that is compositional and higher-order. This call-by-value evaluator is the canonical one for the λ-calculus [196, 220,222]. We conclude that the CEK machine can be seen as a defunctionalized, CPS-transformed, and closure-converted version of the standard call-by-value evaluator for λ-terms.

5.2.3 Variants of Krivine's machine and of the CEK machine

It is easy to construct a variant of Krivine's abstract machine for λ-terms with names, by starting from a call-by-name evaluator for λ-terms with names. Similarly, it is easy to construct a variant of the CEK machine for λ-terms with de Bruijn indices, by starting from a call-by-value evaluator for λ-terms with indices. It is equally easy to start from a call-by-value evaluator for λ-terms with de Bruijn indices and no distinction between values and computations; the resulting abstract machine coincides with Hankin's eager machine [112, Section 8.1.2].

Abstract machines processing λ-terms with de Bruijn indices often resolve indices with transitions:

$$\begin{array}{rcl} \langle 0,\, v :: e,\, s\rangle & \Rightarrow & v :: s \\ \langle n+1,\, v :: e,\, s\rangle & \Rightarrow & \langle n,\, e,\, s\rangle \end{array}$$

Compared to the evaluator of Section 5.2.1.1, the evaluator corresponding to this machine has `List.nth` inlined and is not compositional:

```
fun eval (IND 0, denval :: e, s)
    = ... denval ...
  | eval (IND n, denval :: e, s)
    = eval (IND (n - 1), e, s)
  | ...
```

5.2.4 Conclusion

We have shown that Krivine's abstract machine and the CEK abstract machine are counterparts of canonical evaluators for call-by-name and for call-by-value λ-terms, respectively. The derivation of Krivine's machine is strikingly simpler than what can be found in the literature. That the CEK machine can be derived is, to the best of our knowledge, new. That these two machines are two sides of the same coin is also new. We have not explored any other aspect of this call-by-name/call-by-value duality [55].

Using substitutions instead of environments or inlining one of the standard computational monads (state, continuations, etc. [239]) in the call-by-value evaluator yields variants of the CEK machine that have been documented in the literature [88, Chapter 8]. For example, inlining the state monad in a monadic evaluator yields a state-passing evaluator. The corresponding abstract machine has one more component to represent the state. In general, inlining monads provides a generic recipe to construct arbitrarily many new abstract machines. It does not seem as straightforward, however, to construct a "monadic abstract machine" and then to inline a monad; we are currently studying the issue.

On another note, one can consider an evaluator for strictness-annotated λ-terms—represented either with names or with indices, and with or without distinction between values and computations. One is then led to an abstract machine that generalizes Krivine's machine and the CEK machine [70].

Finally, it is straightforward to extend Krivine's machine and the CEK machine to bigger source languages (with literals, primitive operations, conditional expressions, block structure, recursion, etc.), by starting from evaluators for these bigger languages. For example, all the abstract machines in "The essence of compiling with continuations" [102] are defunctionalized continuation-passing evaluators, i.e., interpreters.

In the rest of this article, we illustrate further the correspondence between evaluators and abstract machines.

5.3 The CLS abstract machine

The CLS abstract machine, displayed in Figure 5.3, is due to Hannan and Miller [114]. Variables n are represented by their de Bruijn index, and the abstract machine operates on triples consisting of a list of directives, a stack of environments, and a stack of expressible values.

5.3.1 The CLS machine

Hannan and Miller's specification is straightforward to program in ML:

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
```

Programs are closed terms.

```
structure Eval0
= struct
    datatype directive = TERM of term
                       | AP
    datatype env = ENV of expval list
      and expval = CLOSURE of term * env
```

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Lists of directives: $c ::= \mathit{nil} \mid t :: c \mid \mathtt{ap} :: c$
- Expressible values (closures): $v ::= [t,\, e]$
- Environments: $e ::= \mathit{nil} \mid v :: e$
- Stacks of environments: $l ::= \mathit{nil} \mid v :: l$
- Stacks of expressible values: $s ::= \mathit{nil} \mid v :: s$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t :: \mathit{nil},\, \mathit{nil} :: \mathit{nil},\, \mathit{nil}\rangle$
$\langle \lambda t :: c,\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, l,\, [t,\, e] :: s\rangle$
$\langle (t_0\, t_1) :: c,\, e :: l,\, s\rangle$	$\Rightarrow$	$\langle t_0 :: t_1 :: \mathtt{ap} :: c,\, e :: e :: l,\, s\rangle$
$\langle 0 :: c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle c,\, l,\, v :: s\rangle$
$\langle n+1 :: c,\, (v :: e) :: l,\, s\rangle$	$\Rightarrow$	$\langle n :: c,\, e :: l,\, s\rangle$
$\langle \mathtt{ap} :: c,\, l,\, v :: [t,\, e] :: s\rangle$	$\Rightarrow$	$\langle t :: c,\, (v :: e) :: l,\, s\rangle$
$\langle \mathit{nil},\, \mathit{nil},\, v :: s\rangle$	$\Rightarrow$	v

Figure 5.3: The CLS abstract machine

```
  (*  run : directive list * env list * expval list  *)
  (*       -> expval                                  *)
  fun run (nil, nil, v :: s)
      = v
    | run ((TERM (IND 0)) :: c, (ENV (v :: e)) :: l, s)
      = run (c, l, v :: s)
    | run ((TERM (IND n)) :: c, (ENV (v :: e)) :: l, s)
      = run ((TERM (IND (n - 1))) :: c, (ENV e) :: l, s)
    | run ((TERM (ABS t)) :: c, e :: l, s)
      = run (c, l, (CLOSURE (t, e)) :: s)
    | run ((TERM (APP (t0, t1))) :: c, e :: l, s)
      = run ((TERM t0) :: (TERM t1) :: AP :: c, e :: e :: l, s)
    | run (AP :: c, l, v :: (CLOSURE (t, ENV e)) :: s)
      = run ((TERM t) :: c, (ENV (v :: e)) :: l, s)

  (*  main : term -> expval *)
  fun main t
      = run ((TERM t) :: nil, (ENV nil) :: nil, nil)
end
```

5.3.2 A disentangled definition of the CLS machine

In the definition of Section 5.3.1, all the possible transitions are meshed together in one recursive function, `run`. Instead, let us factor `run` into several mutually recursive functions, each of them with one induction variable.

In this disentangled definition,

- `run_c` interprets the list of control directives, i.e., it specifies which transition to take if the list is empty, starts with a term, or starts with an apply directive. If the list is empty, the computation terminates. If the list starts with a term, `run_t` is called, caching the term in the first parameter. If the list starts with an apply directive, `run_a` is called.
- `run_t` interprets the top term in the list of control directives.
- `run_a` interprets the top value in the current stack.

The disentangled definition reads as follows:

```
structure Eval1
= struct
    datatype directive = TERM of term
                       | AP
    datatype env = ENV of expval list
      and expval = CLOSURE of term * env

    (*  run_c : directive list * env list * expval list  *)
    (*          -> expval                                 *)
    fun run_c (nil, nil, v :: s)
        = v
      | run_c ((TERM t) :: c, l, s)
        = run_t (t, c, l, s)
      | run_c (AP :: c, l, s)
        = run_a (c, l, s)
    and run_t (IND 0, c, (ENV (v :: e)) :: l, s)
        = run_c (c, l, v :: s)
      | run_t (IND n, c, (ENV (v :: e)) :: l, s)
        = run_t (IND (n - 1), c, (ENV e) :: l, s)
      | run_t (ABS t, c, e :: l, s)
        = run_c (c, l, (CLOSURE (t, e)) :: s)
      | run_t (APP (t0, t1), c, e :: l, s)
        = run_t (t0, (TERM t1) :: AP :: c, e :: e :: l, s)
    and run_a (c, l, v :: (CLOSURE (t, ENV e)) :: s)
        = run_t (t, c, (ENV (v :: e)) :: l, s)

    (*  main : term -> expval  *)
    fun main t
        = run_t (t, nil, (ENV nil) :: nil, nil)
  end
```

Proposition 5.8 (full correctness)
For any program `p : term`, `Eval0.main p` *and* `Eval1.main p` *either both diverge or yield expressible values that are structurally equal.*

Proof. By fold-unfold [40]. The invariants are as follows. For any `t : term`, `e : expval list`, and `s : expval list`, in each of the following pairs either both expressions diverge or yield expressible values that are structurally equal:

```
   Eval1.run_c (c, l, s)  and  Eval0.run (c, l, s),
Eval1.run_t (t, c, l, s)  and  Eval0.run ((TERM t) :: c, l, s),
   Eval1.run_a (c, l, s)  and  Eval0.run (AP :: c, l, s).
```

□

5.3.3 The evaluator corresponding to the CLS machine

In the disentangled definition of Section 5.3.2, there are three possible ways to construct a list of control directives (nil, cons'ing a term, and cons'ing an apply directive). We could specify these constructions as a data type rather than as a list. Such a data type, together with `run_c`, is in the image of defunctionalization (`run_c` is the apply functions of the data type). The corresponding higher-order evaluator is in continuation-passing style. Transforming it back to direct style yields the following evaluator:

```
structure Eval3
= struct
    datatype env = ENV of expval list
      and expval = CLOSURE of term * env

    (*  run_t : term * env list * expval list  *)
    (*              -> env list * expval list  *)
    fun run_t (IND 0, (ENV (v :: e)) :: l, s)
        = (l, v :: s)
      | run_t (IND n, (ENV (v :: e)) :: l, s)
        = run_t (IND (n - 1), (ENV e) :: l, s)
      | run_t (ABS t, e :: l, s)
        = (l, (CLOSURE (t, e)) :: s)
      | run_t (APP (t0, t1), e :: l, s)
        = let val (l, s) = run_t (t0, e :: e :: l, s)
              val (l, s) = run_t (t1, l, s)
          in run_a (l, s)
          end
    and run_a (l, v :: (CLOSURE (t, ENV e)) :: s)
        = run_t (t, (ENV (v :: e)) :: l, s)

    (*  main : term -> expval *)
    fun main t
        = let val (nil, v :: s)
                  = run_t (t, (ENV nil) :: nil, nil)
          in v
          end
end
```

The following proposition is a corollary of the correctness of defunctionalization and of the CPS transformation.

Proposition 5.9 (full correctness)
For any program `p : term`, `Eval1.main p` *and* `Eval3.main p` *either both diverge or yield expressible values that are structurally equal.*

- Source syntax: $t ::= x \mid \lambda x.t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [x,\, t,\, e]$
- Stacks of expressible values: $s ::= \mathit{nil} \mid v :: s$
- Environments: $e ::= e_{mt} \mid e[x \mapsto v]$
- Lists of directives: $c ::= \mathit{nil} \mid t :: c \mid \mathtt{ap} :: c$
- Dumps: $d ::= \mathit{nil} \mid \langle s,\, e,\, c\rangle :: d$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle \mathit{nil},\, e_{mt},\, t :: \mathit{nil},\, \mathit{nil}\rangle$
$\langle s,\, e,\, x :: c,\, d\rangle$	$\Rightarrow$	$\langle e(x) :: s,\, e,\, c,\, d\rangle$
$\langle s,\, e,\, (\lambda x.t) :: c,\, d\rangle$	$\Rightarrow$	$\langle [x,\, t,\, e] :: s,\, e,\, c,\, d\rangle$
$\langle s,\, e,\, (t_0\, t_1) :: c,\, d\rangle$	$\Rightarrow$	$\langle s,\, e,\, t_1 :: t_0 :: \mathtt{ap} :: c,\, d\rangle$
$\langle [x,\, t,\, e'] :: v :: s,\, e,\, \mathtt{ap} :: c,\, d\rangle$	$\Rightarrow$	$\langle \mathit{nil},\, e'[x \mapsto v],\, t :: \mathit{nil},\, d'\rangle$, *where* $d' = (s, e, c) :: d$
$\langle v :: s,\, e,\, \mathit{nil},\, (s', e', c') :: d\rangle$	$\Rightarrow$	$\langle v :: s',\, e',\, c',\, d\rangle$
$\langle v :: s,\, e,\, \mathit{nil},\, \mathit{nil}\rangle$	$\Rightarrow$	v

Figure 5.4: The SECD abstract machine

As in Section 5.2, this evaluator can be made compositional by refunctionalizing the closures into higher-order functions and by factoring the resolution of de Bruijn indices into an auxiliary lookup function.

We conclude that the evaluation model embodied in the CLS machine is a call-by-value interpreter threading a stack of environments and a stack of intermediate results with a caller-save strategy (witness the duplication of environments on the stack in the meaning of applications) and with a left-to-right evaluation of sub-terms. In particular, the meaning of a term is a partial endofunction over a stack of environments and a stack of intermediate results.

5.4 The SECD abstract machine

The SECD abstract machine, displayed in figure 5.4, is due to Landin [155]. Variables x are represented by their name, and the abstract machine operates on quadruples consisting of a stack of expressible values, an environment, a list of directives, and a dump. Environments are consulted in the first transition rule, and extended in the fourth. The empty environment is denoted by e_{mt}.

5.4.1 The SECD machine

Landin's specification is straightforward to program in ML. Programs are closed terms. Environments are as in Section 5.2.2.

```
datatype term = VAR of string    (* name *)
              | LAM of string * term
              | APP of term * term

structure Eval0
= struct
    datatype directive = TERM of term
                       | AP

    datatype value = CLOSURE of string * term * value Env.env

    fun run (v :: nil, e', nil, nil)
        = v
      | run (s, e, (TERM (VAR x)) :: c, d)
        = run ((Env.lookup (e, x)) :: s, e, c, d)
      | run (s, e, (TERM (LAM (x, t))) :: c, d)
        = run ((CLOSURE (x, t, e)) :: s, e, c, d)
      | run (s, e, (TERM (APP (t0, t1))) :: c, d)
        = run (s, e, (TERM t1) :: (TERM t0) :: AP :: c, d)
      | run ((CLOSURE (x, t, e')) :: v :: s, e, AP :: c, d)
        = run (nil, Env.extend (x, v, e'), (TERM t) :: nil,
               (s, e, c) :: d)
      | run (v :: nil, e', nil, (s, e, c) :: d)
        = run (v :: s, e, c, d)

    (*  main : term -> value  *)
    fun main t
        = run (nil, Env.mt, (TERM t) :: nil, nil)
  end
```

5.4.2 A disentangled definition of the SECD machine

As in the CLS machine, in the definition of Section 5.4.1, all the possible transitions are meshed together in one recursive function, `run`. Instead, we can factor `run` into several mutually recursive functions, each of them with one induction variable. These mutually recursive functions are in defunctionalized form: the one processing the dump is an apply function for the data type representing the dump (a list of stacks, environments, and lists of directives), and the one processing the control is an apply function for the data type representing the control (a list of directives). The corresponding higher-order evaluator is in continuation-passing style with two nested continuations and one control delimiter, `reset` [68, 96]. The delimiter resets the control continuation when evaluating the body of a λ-abstraction. (More detail is available in a technical report [65].)

5.4.3 The evaluator corresponding to the SECD machine

The direct-style version of the evaluator from Section 5.4.2 reads as follows:

```
structure Eval4
= struct
    datatype value = CLOSURE of string * term * value Env.env

    (*  eval : term * value list * value Env.env  *)
    (*             -> value list * value Env.env  *)
    fun eval (VAR x, s, e)
        = ((Env.lookup (x, e)) :: s, e)
      | eval (LAM (x, t), s, e)
        = ((CLOSURE (x, t, e)) :: s, e)
      | eval (APP (t0, t1), s, e)
        = let val (s, e) = eval (t1, s, e)
              val (s, e) = eval (t0, s, e)
          in apply (s, e)
          end
    and apply ((CLOSURE (x, t, e')) :: v :: s, e)
        = let val (v :: nil, _)
                  = reset (fn ()
                           => eval (t, nil, Env.extend (x, v, e')))
          in (v :: s, e)
          end

    (*  main : term -> value  *)
    fun main t
        = let val (v :: nil, _)
                  = reset (fn () => eval (t, nil, Env.mt))
          in v
          end
  end
```

The following proposition is a corollary of the correctness of defunctionalization and of the CPS transformation.

Proposition 5.10 (full correctness)
For any program `p : term`, `Eval0.main p` *and* `Eval4.main p` *either both diverge or yield expressible values that are structurally equal.*

As in Sections 5.2 and 5.3, this evaluator can be made compositional by refunctionalizing the closures into higher-order functions.

We conclude that the evaluation model embodied in the SECD machine is a call-by-value interpreter threading a stack of intermediate results and an environment with a callee-save strategy (witness the dynamic passage of environments in the meaning of applications), a right-to-left evaluation of sub-terms, and a control delimiter. In particular, the meaning of a term is a partial endofunction over a stack of intermediate results and an environment. Furthermore, this evaluator evidently implements Hardin, Maranget, and Pagano's L strategy, i.e., right-to-left call by value, without us having to "guess" its inference rules [115, Section 4].

The denotational content of the SECD machine puts a new light on it. For example, its separation between a control register and a dump register is explained by the control delimiter in the evaluator (`reset` in `Eval4.eval`).[2] Removing this control delimiter gives rise to an abstract machine with a single stack component for control—not by a clever change in the machine itself, but by a straightforward simplification in the corresponding evaluator.

5.5 Variants of the CLS machine and of the SECD machine

It is straightforward to construct a variant of the CLS machine for λ-terms with names, by starting from an evaluator for λ-term with names. Similarly, it is straightforward to construct a variant of the SECD machine for λ-terms with de Bruijn indices, by starting from an evaluator for λ-term with indices. In the same vein, it is simple to construct call-by-name versions of the CLS machine and of the SECD machine, by starting from call-by-name evaluators. It is also simple to construct a properly tail recursive version of the SECD machine, and to extend the CLS machine and the SECD machine to bigger source languages, by extending the corresponding evaluator.

5.6 The Categorical Abstract Machine

What is the difference between an abstract machine and a virtual machine? Elsewhere [4], we propose to distinguish them based on the notion of instruction set: A virtual machine has an instruction set whereas an abstract machine does not. Therefore, an abstract machine directly operates on a λ-term, but a virtual machine operates on a compiled representation of a λ-term, expressed using an instruction set. (This distinction can be found elsewhere in the literature [108].)

The Categorical Abstract Machine [49], for example, has an instruction set—categorical combinators—and therefore (despite its name) it is a virtual machine, not an abstract machine. In contrast, Krivine's machine, the CEK machine, the CLS machine, and the SECD machine are all abstract machines, not virtual machines, since they directly operate on λ-terms. In this section, we present the abstract machine corresponding to the Categorical Abstract Machine (CAM). We start from the evaluation model embodied in the CAM [4].

5.6.1 The evaluator corresponding to the CAM

The evaluation model embodied in the CAM is an interpreter threading a stack with its top element cached in a register, representing environments as expressible values (namely nested pairs linked as lists), with a caller-save strategy (witness the duplication of the register on the stack in the meaning of applications below), and with a left-to-right evaluation of sub-terms. In particular, the meaning of a term is a partial endofunction over the register and the stack. This evaluator reads as follows:

[2] A rough definition of `reset` is `fun reset t = t ()`.
A more accurate definition, however, falls out of the scope of this article [68, 96].

```
datatype term = IND of int (* de Bruijn index *)
              | ABS of term
              | APP of term * term
              | NIL
              | CONS of term * term
              | CAR of term
              | CDR of term
```

Programs are closed terms.

```
structure Eval0
= struct
    datatype expval
             = NULL
             | PAIR of expval * expval
             | CLOSURE of expval * (expval * expval list
                                    -> expval * expval list)

    (*  access : int * expval * expval list  *)
    (*              -> expval * expval list  *)
    fun access (0, PAIR (v1, v2), s)
        = (v2, s)
      | access (n, PAIR (v1, v2), s)
        = access (n - 1, v1, s)

    (*  eval : term * expval * expval list  *)
    (*             -> expval * expval list  *)
    fun eval (IND n, v, s)
        = access (n, v, s)
      | eval (ABS t, v, s)
        = (CLOSURE (v, fn (v, s) => eval (t, v, s)), s)
      | eval (APP (t0, t1), v, s)
        = let val (v, v' :: s) = eval (t0, v, v :: s)
              val (v', (CLOSURE (v, f)) :: s) = eval (t1, v', v :: s)
          in f (PAIR (v, v'), s)
          end
      | eval (NIL, v, s)
        = (NULL, s)
      | eval (CONS (t1, t2), v, s)
        = let val (v, v' :: s) = eval (t1, v, v :: s)
              val (v, v' :: s) = eval (t2, v', v :: s)
          in (PAIR (v', v), s)
          end
      | eval (CAR t, v, s)
        = let val (PAIR (v1, v2), s) = eval (t, v, s)
          in (v1, s)
          end
      | eval (CDR t, v, s)
        = let val (PAIR (v1, v2), s) = eval (t, v, s)
          in (v2, s)
          end
```

```
    (*  main : term -> expval  *)
    fun main t
        = let val (v, nil) = eval (t, NULL, nil)
          in v
          end
end
```

This evaluator evidently implements Hardin, Maranget, and Pagano's X strategy [115, Section 6].

5.6.2 The abstract machine corresponding to the CAM

As in Sections 5.2, 5.3, and 5.4, we can closure-convert the evaluator of Section 5.6.1 by defunctionalizing its expressible values, transform it into continuation-passing style, and defunctionalize its continuations. The resulting abstract machine is shown in Figure 5.5.
Variables n are represented by their de Bruijn index, and the abstract machine consists of two mutually recursive transition functions. The first transition function operates on quadruples consisting of a term, an expressible value, a stack of expressible values, and an evaluation context. The second transition function operates on triples consisting of an evaluation context, an expressible value, and a stack of expressible values.

This abstract machine embodies the evaluation model of the CAM. Naturally, more intuitive names could be chosen instead of `CONT0`, `CONT1`, etc.

5.7 Conclusion and issues

We have presented a constructive correspondence between functional evaluators and abstract machines. This correspondence builds on off-the-shelf program transformations: closure conversion, CPS transformation, defunctionalization, and inlining.[3] We have shown how to reconstruct known machines (Krivine's machine, the CEK machine, the CLS machine, and the SECD machine) and how to construct new ones. Conversely, we have revealed the denotational content of known abstract machines. We have shown that Krivine's abstract machine and the CEK machine correspond to canonical evaluators for the λ-calculus. We have also shown that they are dual of each other since they correspond to call-by-name and call-by-value evaluators in the same direct style.

In terms of denotational semantics [169, 204], Krivine's machine and the CEK machine correspond to a standard semantics, whereas the CLS machine and the SECD machine correspond to a stack semantics of the λ-calculus. Finally, we have exhibited the abstract machine corresponding to the CAM, which puts the reader in a new position to answer the recurrent question as to whether the CLS machine is closer to the CAM or to the SECD machine.

Since this article was written, we have studied the correspondence between functional evaluators and abstract machines for call by need [6] and for Proposi-

[3] Indeed the push-enter twist of Krivine's machine is obtained by inlining `apply_cont` in Section 5.2.1.5.

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1 \mid \texttt{nil} \mid (\texttt{cons}\, t_1\, t_2) \mid (\texttt{car}\, t) \mid (\texttt{cdr}\, t)$
- Expressible values (unit value, pairs, and closures):

$$v ::= \texttt{null} \mid (v_1,\, v_2) \mid [v,\, t]$$

- Stacks of expressible values: $s ::= \mathit{nil} \mid v :: s$
- Evaluation contexts: $k ::= \texttt{CONT0} \mid \texttt{CONT1}(t,\, k) \mid \texttt{CONT2}(k) \mid \texttt{CONT3}(t,\, k) \mid \texttt{CONT4}(k) \mid \texttt{CONT5}(k) \mid \texttt{CONT6}(k)$
- Initial transition, transition rules (two kinds), and final transition:

t	$\Rightarrow$	$\langle t,\, \texttt{null},\, \mathit{nil},\, \texttt{CONT0}\rangle_{eval}$
$\langle n,\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle k,\, \gamma(n, v),\, s\rangle_{cont}$
$\langle \lambda t,\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle k,\, [v,\, t],\, s\rangle_{cont}$
$\langle \texttt{nil},\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle k,\, \texttt{null},\, s\rangle_{cont}$
$\langle t_0\, t_1,\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle t_0,\, v,\, v :: s,\, \texttt{CONT1}(t_1,\, k)\rangle_{eval}$
$\langle (\texttt{cons}\, t_1\, t_2),\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle t_1,\, v,\, v :: s,\, \texttt{CONT3}(t_2,\, k)\rangle_{eval}$
$\langle (\texttt{car}\, t),\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle t,\, v,\, s,\, \texttt{CONT5}(k)\rangle_{eval}$
$\langle (\texttt{cdr}\, t),\, v,\, s,\, k\rangle_{eval}$	$\Rightarrow$	$\langle t,\, v,\, s,\, \texttt{CONT6}(k)\rangle_{eval}$
$\langle \texttt{CONT1}(t,\, k),\, v,\, v' :: s\rangle_{cont}$	$\Rightarrow$	$\langle t,\, v',\, v :: s,\, \texttt{CONT2}(k)\rangle_{eval}$
$\langle \texttt{CONT2}(k),\, v',\, [v,\, t] :: s\rangle_{cont}$	$\Rightarrow$	$\langle t,\, (v,\, v'),\, s,\, k\rangle_{eval}$
$\langle \texttt{CONT3}(t_1,\, k),\, v,\, v' :: s\rangle_{cont}$	$\Rightarrow$	$\langle t_1,\, v',\, v :: s,\, \texttt{CONT4}(k)\rangle_{eval}$
$\langle \texttt{CONT4}(k),\, v,\, v' :: s\rangle_{cont}$	$\Rightarrow$	$\langle k,\, (v',\, v),\, s\rangle_{cont}$
$\langle \texttt{CONT5}(k),\, (v_1,\, v_2),\, s\rangle_{cont}$	$\Rightarrow$	$\langle k,\, v_1,\, s\rangle_{cont}$
$\langle \texttt{CONT6}(k),\, (v_1,\, v_2),\, s\rangle_{cont}$	$\Rightarrow$	$\langle k,\, v_2,\, s\rangle_{cont}$
$\langle \texttt{CONT0},\, v,\, \mathit{nil}\rangle_{cont}$	$\Rightarrow$	v

where $\gamma(0, (v_1,\, v_2)) = v_2$
$\gamma(n, (v_1,\, v_2)) = \gamma(n-1, v_1)$

Figure 5.5: The abstract machine corresponding to the CAM

tional Prolog [33]. In both cases, we derived sensible machines out of canonical evaluators.

It seems to us that this correspondence between functional evaluators and abstract machines builds a reliable bridge between denotational definitions and definitions of abstract machines. On the one hand, it allows one to identify the denotational content of an abstract machine in the form of a functional interpreter. On the other hand, it gives one a precise and generic recipe to construct arbitrarily many new variants of abstract machines (e.g., with substitutions or environments, or with stacks) or of arbitrarily many new abstract machines,

starting from an evaluator with any given computational monad [172].

Acknowledgments: We are grateful to Małgorzata Biernacka, Julia Lawall, and Henning Korsholm Rohde for timely comments. Thanks are also due to the anonymous reviewers.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

Chapter 6

From Interpreter to Logic Engine by Defunctionalization

with Olivier Danvy [33]

Abstract

Starting from a continuation-based interpreter for a simple logic programming language, propositional Prolog with cut, we derive the corresponding logic engine in the form of an abstract machine. The derivation originates in previous work (our article at PPDP 2003) where it was applied to the lambda-calculus. The key transformation here is Reynolds's defunctionalization that transforms a tail-recursive, continuation-passing interpreter into a transition system, i.e., an abstract machine. Similar denotational and operational semantics were studied by de Bruin and de Vink (their article at TAPSOFT 1989), and we compare their study with our derivation. Additionally, we present a direct-style interpreter of propositional Prolog expressed with control operators for delimited continuations.

6.1 Introduction

In previous work [5], we presented a derivation from interpreter to abstract machine that makes it possible to connect known λ-calculus interpreters to known abstract machines for the λ-calculus, as well as to discover new ones. The goal of this work is to test this derivation on a programming language other than the λ-calculus. Our pick here is a simple logic programming language, propositional Prolog with cut (Section 6.2). We present its abstract syntax, informal semantics, and computational model, which we base on success and failure continuations (Section 6.3). We then specify an interpreter for propositional Prolog in a generic and parameterized way that leads us to a logic engine. This logic engine is a transition system that we obtain by defunctionalizing the success and failure continuations (Section 6.4). We also present and analyze a direct-style interpreter for propositional Prolog (Appendix 6.A).

The abstract machines we consider are models of computation rather than devices for high performance, and the transformations we consider are changes

of representation rather than optimizations.

Prerequisites: We expect a passing familiarity with the notions of success and failure continuations as well as with Standard ML and its module language.

As for defunctionalization, it originates in Reynolds's seminal article on definitional interpreters for higher-order programming languages [196]. The point of defunctionalization is to transform a higher-order program into a first-order program by replacing its function types by sum types. Before defunctionalization, the inhabitants of each function type are instances of anonymous lambda-abstractions. Defunctionalizing a program amounts to enumerating these lambda-abstractions in a sum type: each function introduction (i.e., lambda-abstraction) is replaced by the corresponding constructor holding the values of the free variables of this lambda-abstraction, and each function elimination (i.e., application) is replaced by a case dispatch. After defunctionalization, the inhabitants of each function type are represented by elements of a corresponding sum type.

Danvy and Nielsen's study of defunctionalization contains many examples [74], but to make the present article self-contained, let us consider two concrete cases.

6.1.1 A simple example of defunctionalization

The following (trivial) program is higher-order because of the auxiliary function `aux`, which is passed a function of type `int -> int` as argument:

```
(*  aux : int * (int -> int) -> int  *)
fun aux (x, f)
    = (f 10) + (f x)

(*  main : int * int * int -> int  *)
fun main (a, b, c)
    = (aux (a, fn x => x + b)) * (aux (c, fn x => x * x))
```

The inhabitants of the function space `int -> int` are instances of the two anonymous lambda-abstractions declared in `main`, `fn x => x + b` and `fn x => x * x`. The first one has one free variable (`b`, of type `int`), and the second one is closed, i.e., it has no free variables.

To defunctionalize this program, we enumerate these lambda-abstractions in a sum type `lam`, and we define the corresponding apply function to interpret each of the summands:

```
datatype lam = LAM1 of int
             | LAM2

(*  apply_lam : lam * int -> int  *)
fun apply_lam (LAM1 b, x)
    = x + b
  | apply_lam (LAM2, x)
    = x * x
```

In the defunctionalized program, each lambda-abstraction is replaced by the corresponding constructor, and each application is replaced by a call to the apply function:

```
(*  aux : int * lam -> int  *)
fun aux (x, f)
    = (apply_lam (f, 10)) + (apply_lam (f, x))

(*  main : int * int * int -> int  *)
fun main (a, b, c)
    = (aux (a, LAM1 b)) * (aux (c, LAM2))
```

The resulting program is first order.

6.1.2 A more advanced example: the factorial function

Let us defunctionalize the following continuation-passing version of the factorial function:

```
(*  fac_c : int * (int -> 'a) -> 'a  *)
fun fac_c (0, k)
    = k 1
  | fac_c (n, k)
    = fac_c (n - 1, fn v => k (n * v))

(*  main : int -> int  *)
fun main n
    = fac_c (n, fn v => v)
```

We consider the whole program (i.e., both `main` and `fac_c`). Therefore the polymorphic type `'a`, i.e., the domain of answers, is instantiated to `int`. The candidate function space for defunctionalization is that of the continuation, `int -> int`. Its inhabitants are instances of two lambda-abstractions: the initial continuation in `main` with no free variables, and the intermediate continuation in the induction case of `fac_c` with two free variables: `n` and `k`. The corresponding data type has therefore two constructors:

```
datatype cont = CONT0
              | CONT1 of int * cont

(*  apply_cont : cont * int -> int  *)
fun apply_cont (CONT0, v)
    = v
  | apply_cont (CONT1 (n, k), v)
    = apply_cont (k, n * v)
```

Correspondingly, the apply function associated to the data type interprets each of these constructors according to the initial continuation and the intermediate continuation.

We observe that `cont` is isomorphic to the data type of lists of integers. We therefore adopt this simpler representation of defunctionalized continuations:

```
type cont = int list

(*  apply_cont : cont * int -> int  *)
fun apply_cont (nil, v)
    = v
  | apply_cont (n :: k, v)
    = apply_cont (k, n * v)
```

In the defunctionalized program, the continuations are replaced by the constructors, and the applications of the continuations are replaced by a call to `apply_cont`:

```
(*  fac_c : int * cont -> int  *)
fun fac_c (0, k)
    = apply_cont (k, 1)
  | fac_c (n, k)
    = fac_c (n - 1, n :: k)

(*  main : int -> int  *)
fun main n
    = fac_c (n, nil)
```

The resulting program is first-order, all its calls are tail calls, and all computations in the actual parameters are elementary. It is therefore a transition system in the sense of automata and formal languages [162]. Both `main` and `fac_c`, together with their actual parameters, form configurations and their ML definitions specify a transition relation, as expressed in the following table. The top transition specifies the initial state and the bottom transition specifies the terminating configurations. The machine consists of two mutually recursive transition functions; the first one operates over pairs of integers, and the second one operates over a stack of integers and an integer:

n	$\Rightarrow$	$\langle n,\, nil\rangle_{fac}$
$\langle 0,\, k\rangle_{fac}$	$\Rightarrow$	$\langle k,\, 1\rangle_{app}$
$\langle n,\, k\rangle_{fac}$	$\Rightarrow$	$\langle n-1,\, n :: k\rangle_{fac}$
$\langle n :: k,\, v\rangle_{app}$	$\Rightarrow$	$\langle k,\, n \times v\rangle_{app}$
$\langle nil,\, v\rangle_{app}$	$\Rightarrow$	v

Accordingly, the result of defunctionalizing a continuation-passing interpreter is also a transition system, i.e., an abstract machine in the sense of automata and formal languages [162]. We used this property in our work on the λ-calculus [5], and we use it here for propositional Prolog.

6.2 Propositional Prolog

The abstract syntax of propositional Prolog reads as follows:

```
structure Source
= struct
    type ide = string
    datatype atom = IDE of ide
                  | OR of goal * goal
                  | CUT
                  | FAIL
    withtype goal = atom list
    type clause = ide * goal
    datatype program = PROGRAM of clause list
    datatype top_level_goal = GOAL of goal
  end
```

A program consists of a list of clauses. A clause consists of an identifier (the head of the clause) and a goal (the body of the clause). A goal is a list of atoms; an empty list represents the logical value 'true' and a non-empty list of atoms represents their conjunction. Each atom is either an identifier, the disjunction of two goals, the cut operator, or the fail operator.

The intuitive semantics of the language is standard. Given a Prolog program and a goal, we try to verify whether the goal follows from the program in the sense of propositional logic, i.e., in terms of logic programming, whether the SLD-resolution algorithm for this goal and this program stops with the empty clause. If it does, then the answer is positive; if it stops with one or more subgoals still waiting resolution, then the answer is negative. Here the unification algorithm consists in looking up the clause with a specified head in the program.

An atom can be a disjunction of two goals, and therefore if a chosen body does not lead to the positive answer, the other disjunct is tried, using backtracking. Backtracking can also be used to find all possible solutions in the resolution tree, which in case of propositional Prolog amounts to counting the positive answers. Two operators provide additional control over the traversal of the resolution tree: the cut operator removes some of the potential paths and the fail operator makes the current goal unsatisfiable, which triggers backtracking.

6.3 A generic interpreter for propositional Prolog

To account for the backtracking necessary to implement resolution, we use success and failure continuations [69]. A failure continuation is a parameterless function (i.e., a thunk) yielding a final answer. A success continuation maps a failure continuation to a final answer. The initial success continuation is applied if a solution has been found. The initial failure continuation is applied if no solution has been found. In addition, to account for the cut operator, we pass a cut continuation, i.e., a cached failure continuation. As usual with continuations, the domain of answers is left unspecified.

6.3.1 A generic notion of answers and results

We specify answers with an ML signature. The type of answers comes together with an initial success continuation and an initial failure continuation. The signature also declares a type of results and an extraction function mapping a (generic) answer to a (specific) result.

```
signature ANSWER
= sig
    type answer
    val sc_init : (unit -> answer) -> answer
    val fc_init : unit -> answer

    type result
    val extract : answer -> result
  end
```

6.3.2 Specific answers and results

We consider two kinds of answers: the first solution, if any, and the total number of solutions.

The first solution: This notion of answer is the simplest to define. Both `answer` and `result` are defined as the type of booleans and `extract` is the identity function. The initial success continuation ignores the failure continuation and yields `true`, whereas the initial failure continuation yields `false`.

```
structure Answer_first : ANSWER
= struct
    type answer = bool
    fun sc_init fc = true
    fun fc_init () = false

    type result = bool
    fun extract a = a
  end
```

The number of solutions: This notion of answer is more delicate. One could be tempted to define `answer` as the type of integers, but the resulting implementation would no longer be tail recursive.[1] Instead, we use an extra layer of continuations: We define `answer` as the type of functions from integers to integers, `result` as the type of integers, and `extract` as a function triggering the whole resolution by applying an answer to the initial count, 0. The initial success continuation takes note of an intermediate success by incrementing the current count and activating the failure continuation. The initial failure continuation is passed the final count and returns it.

[1] In "`fun sc_init fc = 1 + (fc ())`", the call to `fc` is not a tail call.

```
structure Answer_how_many : ANSWER
= struct
    type answer = int -> int
    fun sc_init fc = (fn m => fc () (m+1))
    fun fc_init () = (fn m => m)

    type result = int
    fun extract a = a 0
  end
```

6.3.3 The generic interpreter, semi-compositionally

We define a generic interpreter for propositional Prolog, displayed in Figure 6.1, as a recursive descent over the source syntax, parameterized by a notion of answers, and implementing the following signature:

```
signature INTERPRETER
= sig
    type result
    val main : Source.top_level_goal * Source.program -> result
  end
```

In `run_goal`, an empty list of atoms is interpreted as 'true', and accordingly, the success continuation is activated. A non-empty list of atoms is sequentially interpreted by `run_seq` by extending the success continuation; this interpretation singles out the last atom in a properly tail-recursive manner. An identifier is interpreted either by failing if it is not the head of any clause in the program, or by resolving the corresponding goal with the cut continuation replaced with the current failure continuation. The function `lookup` searching for a clause with a given head reads as follows:

```
(*  lookup : Source.ide * Source.clause list -> Source.goal option  *)
fun lookup (i, p)
    = let fun walk nil
              = NONE
            | walk ((i', g) :: p)
              = if i = i'
                then SOME g
                else walk p
      in walk p
      end
```

A disjunction of two goals is interpreted by extending the failure continuation. The cut operator is interpreted by replacing the failure continuation with the cut continuation. The fail operator is interpreted as 'false', and accordingly, the failure continuation is activated.

This interpreter is not compositional (in the sense of denotational semantics) because `g`, in the interpretation of identifiers, does not denote a proper subpart of the denotation of `l`. The interpreter, however, is semi-compositional in Jones's sense [135, 136], i.e., `g` denotes a proper subpart of the source program. (To make the interpreter compositional, one can follow the tradition

```
functor mkInterpreter (structure A : ANSWER) : INTERPRETER
= struct
    open Source
    type result = A.result
    type fcont = unit -> A.answer
    type scont = fcont -> A.answer
    type ccont = fcont

    (*  run_goal : goal * clause list * scont * fcont * ccont
                  -> A.answer  *)
    fun run_goal (nil, p, sc, fc, cc)
        = sc fc
      | run_goal (a :: g, p, sc, fc, cc)
        = run_seq (a, g, p, sc, fc, cc)

    (*  run_seq : atom * goal * clause list * scont * fcont
                * ccont -> A.answer  *)
    and run_seq (a, nil, p, sc, fc, cc)
        = run_atom (a, p, sc, fc, cc)
      | run_seq (a, a' :: g, p, sc, fc, cc)
        = run_atom (a, p,
                    fn fc' => run_seq (a', g, p, sc, fc', cc),
                    fc, cc)

    (*  run_atom : atom * clause list * scont * fcont * ccont
                  -> A.answer  *)
    and run_atom (IDE i, p, sc, fc, cc)
        = (case lookup (i, p)
             of NONE => fc ()
              | (SOME g) => run_goal (g, p, sc, fc, fc))
      | run_atom (OR (g1, g2), p, sc, fc, cc)
        = run_goal (g1, p, sc,
                    fn () => run_goal (g2, p, sc, fc, cc), cc)
      | run_atom (CUT, p, sc, fc, cc)
        = sc cc
      | run_atom (FAIL, p, sc, fc, cc)
        = fc ()

    (*  main : top_level_goal * program -> A.result  *)
    fun main (GOAL g, PROGRAM p)
        = let val a = run_goal (g, p, A.sc_init, A.fc_init,
                                A.fc_init)
          in A.extract a
          end
  end
```

Figure 6.1: A generic interpreter for propositional Prolog

of denotational semantics and use an environment mapping an identifier to a

function that either evaluates the goal denoted by the identifier or calls the failure continuation. The environment is threaded in the interpreter instead of the program. The resulting ML interpreter represents the valuation function of a denotational semantics of propositional Prolog.)

6.3.4 Specific interpreters

6.3.4.1 A specific interpreter computing the first solution:

A specific interpreter computing the first solution, if any, is obtained by instantiating `mkInterpreter` with the corresponding notion of answers:

```
structure Prolog_first
  = mkInterpreter (structure A = Answer_first)
```

6.3.4.2 A specific interpreter computing the number of solutions:

A specific interpreter computing the number of solutions is also obtained by instantiating `mkInterpreter` with the corresponding notion of answers:

```
structure Prolog_how_many
  = mkInterpreter (structure A = Answer_how_many)
```

Appendix 6.A contains a direct-style counterpart of the interpreter (uncurried and without cut) computing the number of solutions.

6.4 Two abstract machines for propositional Prolog

We successively consider each of the specific Prolog interpreters of Section 6.3.4 and we defunctionalize their continuations. As already illustrated in Section 6.1 with the factorial program, in each case, the result is an abstract machine. Indeed the interpreters are in continuation-passing style, and thus:

- all their calls are tail calls, and therefore they can run iteratively; and
- all their subcomputations (i.e., the computation of their actual parameters) are elementary.

In both cases the types of the defunctionalized success and failure continuations read as follows:

```
datatype scont = SCONT0
               | SCONT1 of atom * goal * clause list * scont * ccont
     and fcont = FCONT0
               | FCONT1 of goal * clause list * scont * fcont * ccont
withtype ccont = fcont
```

As in Section 6.1.2, since both data types are isomorphic to the data type of lists, we represent them as such when presenting the abstract machines.

- Control stacks: $sc ::= nil \mid (a,\ g,\ p,\ cc) :: sc$
  $fc ::= nil \mid (g,\ p,\ sc,\ cc) :: fc$
  $cc ::= fc$

- Initial transition, transition rules and final transitions:

$\langle g,\ p\rangle$	$\Rightarrow$	$\langle g,\ p,\ nil,\ nil,\ nil\rangle_{goal}$
$\langle nil,\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc\rangle_{goal}$	$\Rightarrow$	$\langle a,\ g,\ p',\ sc,\ fc,\ cc'\rangle_{seq}$
$\langle a :: g,\ p,\ sc,\ fc,\ cc\rangle_{goal}$	$\Rightarrow$	$\langle a,\ g,\ p,\ sc,\ fc,\ cc\rangle_{seq}$
$\langle a,\ nil,\ p,\ sc,\ fc,\ cc\rangle_{seq}$	$\Rightarrow$	$\langle a,\ p,\ sc,\ fc,\ cc\rangle_{atom}$
$\langle a,\ a' :: g,\ p,\ sc,\ fc,\ cc\rangle_{seq}$	$\Rightarrow$	$\langle a,\ p,\ (a',\ g,\ p,\ cc) :: sc,\ fc,\ cc\rangle_{atom}$
$\langle \texttt{IDE}\ i,\ p,\ sc,\ fc,\ cc\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p,\ sc,\ fc,\ fc\rangle_{goal}$ *if lookup* $(i,\ p)$ *succeeds with* g
$\langle \texttt{IDE}\ i,\ p,\ sc,\ (g,\ p,\ sc',\ cc') :: fc,\ cc\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p',\ sc',\ fc,\ cc'\rangle_{goal}$ *if lookup* $(i,\ p)$ *fails*
$\langle \texttt{OR}\,(g_1,\ g_2),\ p,\ sc,\ fc,\ cc\rangle_{atom}$	$\Rightarrow$	$\langle g_1,\ p,\ sc,\ (g_2,\ p,\ sc,\ cc) :: fc,\ cc\rangle_{goal}$
$\langle \texttt{CUT},\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc\rangle_{atom}$	$\Rightarrow$	$\langle a,\ g,\ p',\ sc,\ fc,\ cc'\rangle_{seq}$
$\langle \texttt{FAIL},\ p,\ sc,\ (g,\ p',\ sc',\ cc') :: fc,\ cc\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p',\ sc',\ fc,\ cc'\rangle_{goal}$
$\langle nil,\ p,\ nil,\ fc,\ cc\rangle_{goal}$	$\Rightarrow$	*true*
$\langle \texttt{IDE}\ i,\ p,\ sc,\ nil,\ cc\rangle_{atom}$	$\Rightarrow$	*false*, *if lookup* $(i,\ p)$ *fails*
$\langle \texttt{FAIL},\ p,\ sc,\ nil,\ cc\rangle_{atom}$	$\Rightarrow$	*false*
$\langle \texttt{CUT},\ p,\ nil,\ fc,\ cc\rangle_{atom}$	$\Rightarrow$	*true*

Figure 6.2: An abstract machine computing the first solution

- Control stacks: $sc ::= nil \mid (a,\ g,\ p,\ cc) :: sc$
  $fc ::= nil \mid (g,\ p,\ sc,\ cc) :: fc$
  $cc ::= fc$

- Initial transition, transition rules and final transitions:

$\langle g,\ p\rangle$	$\Rightarrow$	$\langle g,\ p,\ nil,\ nil,\ nil,\ 0\rangle_{goal}$
$\langle nil,\ p,\ nil,\ (g,\ p',\ sc,\ cc') :: fc,\ cc,\ m\rangle_{goal}$	$\Rightarrow$	$\langle g,\ p',\ sc,\ fc,\ cc',\ m+1\rangle_{goal}$
$\langle nil,\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc,\ m\rangle_{goal}$	$\Rightarrow$	$\langle a,\ g,\ p',\ sc,\ fc,\ cc',\ m\rangle_{seq}$
$\langle a :: g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{goal}$	$\Rightarrow$	$\langle a,\ g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq}$
$\langle a,\ nil,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq}$	$\Rightarrow$	$\langle a,\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom}$
$\langle a,\ a' :: g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq}$	$\Rightarrow$	$\langle a,\ p,\ (a',\ g,\ p,\ cc) :: sc,\ fc,\ cc,\ m\rangle_{atom}$
$\langle \texttt{IDE}\ i,\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p,\ sc,\ fc,\ fc,\ m\rangle_{goal}$ *if lookup* $(i,\ p)$ *succeeds with* g
$\langle \texttt{IDE}\ i,\ p,\ sc,\ (g,\ p,\ sc',\ cc') :: fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p',\ sc',\ fc,\ cc',\ m\rangle_{goal}$ *if lookup* $(i,\ p)$ *fails*
$\langle \texttt{OR}\,(g_1,\ g_2),\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle g_1,\ p,\ sc,\ (g_2,\ p,\ sc,\ cc) :: fc,\ cc,\ m\rangle_{goal}$
$\langle \texttt{CUT},\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle a,\ g,\ p',\ sc,\ fc,\ cc',\ m\rangle_{seq}$
$\langle \texttt{CUT},\ p,\ nil,\ (g,\ p',\ sc,\ cc') :: fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p',\ sc,\ fc,\ cc',\ m+1\rangle_{goal}$
$\langle \texttt{FAIL},\ p,\ sc,\ (g,\ p',\ sc',\ cc') :: fc,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$\langle g,\ p',\ sc',\ fc,\ cc',\ m\rangle_{goal}$
$\langle nil,\ p,\ nil,\ nil,\ cc,\ m\rangle_{goal}$	$\Rightarrow$	$m+1$
$\langle \texttt{FAIL},\ p,\ sc,\ nil,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	m
$\langle \texttt{CUT},\ p,\ nil,\ nil,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	$m+1$
$\langle \texttt{IDE}\ i,\ p,\ sc,\ nil,\ cc,\ m\rangle_{atom}$	$\Rightarrow$	m, *if lookup* $(i,\ p)$ *fails*

Figure 6.3: An abstract machine computing the number of solutions

The first solution: The abstract machine is defined as the transition system shown in Figure 6.2. The top part specifies the initial state and the bottom part specifies the terminating configurations. The machine consists of three mutually recursive transition functions, two of which operate over a quintuple and one over a six-element tuple. The quintuple consists of the goal, the program, the (defunctionalized) success continuation, the (defunctionalized) failure continuation and the cut continuation (a register caching a previous failure continuation). The six-element tuple additionally has the first atom of the goal as its first element.

The number of solutions: This abstract machine is displayed in Figure 6.3 and is similar to the previous one, but operates over a six- and seven-element tuples. The extra component is the counter.

Both machines are deterministic because they were derived from (deterministic) functions.

6.5 Related work and conclusion

In previous work [5, 6, 65], we presented a derivation from interpreter to abstract machine, and we were curious to see it applied to something else than a functional programming language. The present paper reports its application to a logic programming language, propositional Prolog. In its entirety, the derivation consists of closure conversion, transformation into continuation-passing style (CPS), and defunctionalization. Closure conversion ensures that any higher-order values are made first-order.[2] The CPS transformation makes the flow of control of the interpreter manifest as a continuation. Defunctionalization materializes the flow of control as a first-order data structure. In the present case, propositional Prolog is a first-order language and the interpreter we consider is already in continuation-passing style (cf. Appendix 6.A). Therefore the derivation reduces to defunctionalization. The result is a simple logic engine, i.e., mutually recursive and first-order transition functions. It was derived, not invented, and so, for example, its two stacks arise as defunctionalized continuations. Similarly, it is properly tail recursive since the interpreter is already properly tail recursive.

Since the correctness of defunctionalization has been established [18, 180], the correctness of the logic engine is a corollary of the correctness of the original interpreter.

Prolog has both been specified and formalized functionally. For example, Carlsson has shown how to implement Prolog in a functional language [41]. Continuation-based semantics of Prolog have been studied by de Bruin and de Vink [78] as well as by Nicholson and Foo [179]. Our closest related work is de Bruin and de Vink's continuation semantics for Prolog with cut:

[2] Closures, for example, are used to implement higher-order logic programming [45].

- de Bruin and de Vink present a denotational semantics with success and failure continuations; their semantics is (of course) compositional, and comparable to the compositional interpreter outlined in Section 6.3.3. The only difference is that their success continuations expect both a failure continuation and a cut continuation, whereas our success continuations expect only a failure continuation. Analyzing the control flow of the corresponding interpreter, we have observed that the cut continuation is the same at the definition point and at the use point of a success continuation. Therefore, there is actually no need to pass cut continuations to success continuations.

- de Bruin and de Vink also present an operational semantics, and prove it equivalent to their denotational semantics. In contrast, we defunctionalized the interpreter corresponding to a denotational semantics into an interpreter corresponding to an operational semantics. We also "refunctionalized" the interpreter corresponding to de Bruin and de Vink's operational semantics, and we observed that in the resulting interpreter (which corresponds to a denotational semantics), success continuations are not passed cut continuations.

Designing abstract machines is a favorite among functional programmers [79]. Unsurprisingly, this is also the case among logic programmers, for example, with Warren's abstract machine [8], which incidentally is more of a device for high performance than a model of computation. Just as unsurprisingly, functional programmers use functional programming languages as their metalanguage and logic programmers use logic programming languages as their metalanguage. For example, Kursawe showed how to "invent" Prolog machines out of logic-programming considerations [150]. The goal of our work here was more modest: we simply aimed to test an interpreter-to-abstract-machine derivation that works well for the λ-calculus. The logic engine we obtained is basic but plausible. Its chief illustrative virtue is to show that the representation of a denotational semantics can be mechanically defunctionalized into the representation of an operational semantics (and, actually, vice versa). It also shows that proper tail recursion and the two control stacks did not need to be invented—they were already present in the original interpreter.

An alternative to deriving an abstract machine from an interpreter is to factor this interpreter into a compiler and a virtual machine, using, e.g., Wand's combinator-based compiler derivation [244], Jørring and Scherlis's staging transformations [137], Hannan's pass-separation approach [113], or more generally the binding-time separation techniques of partial evaluation [136, 171]. We are currently experimenting with a such a factorization to stage our Prolog interpreter into a byte-code compiler and a virtual machine executing this byte code [4].

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Jan Midtgaard, and the anonymous referees for their comments.

This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

```
structure Prolog_how_many_DS : INTERPRETER
= struct
    open Source
    type result = int

    (*  run_goal : goal * clause list * int -> int  *)
    fun run_goal (nil, p, m)
        = m
      | run_goal (a :: g, p, m)
        = run_seq (a, g, p, m)

    (*  run_seq : atom * goal * clause list * int -> int  *)
    and run_seq (a, nil, p, m)
        = run_atom (a, p, m)
      | run_seq (a, a' :: g, p, m)
        = let val m' = run_atom (a, p, m)
          in run_seq (a', g, p, m')
          end

    (*  run_atom : atom * clause list * int -> int  *)
    and run_atom (FAIL, p, m)
        = shift (fn sc => m)
      | run_atom (IDE i, p, m)
        = (case lookup (i, p)
             of NONE => shift (fn sc => m)
              | (SOME g) => run_goal (g, p, m))
      | run_atom (OR (g1, g2), p, m)
        = shift (fn sc => let val m' = reset (fn () =>
                                        sc (run_goal (g1, p, m)))
                          in sc (run_goal (g2, p, m'))
                          end)

    (*  main : top_level_goal * program -> int  *)
    fun main (GOAL g, PROGRAM p)
        = reset (fn () => let val m = run_goal (g, p, 0)
                          in m + 1
                          end)
  end
```

Figure 6.4: A direct-style interpreter for propositional Prolog

6.A A direct-style interpreter for Prolog

The interpreter of Section 6.3 is in continuation-passing style to account for the backtracking necessary to implement resolution. Therefore, our derivation method which in its entirety consists of three steps—closure conversion, CPS transformation, and defunctionalization [5]—was reduced to only the last step. Less natural, but making the derivation closer to its original specification, would be to start it with an interpreter in direct style. The failure continuation could be eliminated by transforming the interpreter into direct style [59]. The success continuation, however, would remain. Because it is used

non tail-recursively in the clause for disjunctions, it is what is technically called a *delimited* continuation (in contrast to the usual unlimited continuations of denotational semantics [225]). Transforming the interpreter into direct style requires control operators for delimited continuations that are compatible with continuation-passing style, e.g., shift and reset [67, 68, 96].

Figure 6.4 presents such a direct-style interpreter for Propositional Prolog without cut, counting the number of solutions. CPS-transforming this interpreter once makes the success continuation appear. CPS-transforming the result makes the failure continuation appear, and yields the interpreter of Section 6.3.4.2 (minus cut). Defunctionalizing this interpreter yields the abstract machine of Figure 6.3 (minus cut).

The reset control operator delimits control. Any subsequent use of the shift control operator will capture a delimited continuation that can be composed; this delimited continuation is the success continuation. Conjunction, in `run_seq`, is implemented by function composition. Failure, in `run_atom`, is implemented by capturing the current success continuation and not applying it. Disjunction, in `run_atom`, is implemented by capturing the current success continuation and applying it twice. This interpreter is properly tail recursive, which is achieved by the two functions `run_goal` and `run_seq` that single out the last atom in a goal.

The interpreter is a new example of nondeterministic programming in direct style with control operators for the first level of the CPS hierarchy [29, 67]. In order to interpret the cut operator we would have to use the control operators of the second level, shift_2 and reset_2.

Chapter 7

An Operational Foundation for Delimited Continuations in the CPS Hierarchy

with Małgorzata Biernacka and Olivier Danvy [29]

Abstract

We present an abstract machine and a reduction semantics for the λ-calculus extended with control operators that give access to delimited continuations in the CPS hierarchy. The abstract machine is derived from an evaluator in continuation-passing style (CPS); the reduction semantics (i.e., a small-step operational semantics with an explicit representation of evaluation contexts) is constructed from the abstract machine; and the control operators are the shift and reset family. At level n of the CPS hierarchy, programs can use the control operators shift_i and reset_i for $1 \leq i \leq n$, the evaluator has $n+1$ layers of continuations, the abstract machine has $n+1$ layers of control stacks, and the reduction semantics has $n+1$ layers of evaluation contexts.

We also present new applications of delimited continuations in the CPS hierarchy: finding list prefixes and normalization by evaluation for a hierarchical language of units and products.

7.1 Introduction

The studies of delimited continuations can be classified in two groups: those that use continuation-passing style (CPS) and those that rely on operational intuitions about control instead. Of the latter, there is a large number proposing a variety of control operators [12, 87, 90, 93, 111, 125, 173, 191, 213, 240] which have found applications in models of control, concurrency, and type-directed partial evaluation [16, 125, 214]. Of the former, there is the work revolving around the family of control operators shift and reset [66–68, 77, 96, 98, 141, 142, 177, 240] which have found applications in non-deterministic programming, code generation, partial evaluation, normalization by evaluation, computational monads, and mobile computing [13, 14, 17, 33, 60, 61, 80, 82, 99, 106, 110, 123, 143, 146, 159, 209, 227, 228, 235].

The original motivation for shift and reset was a continuation-based programming pattern involving several layers of continuations. The original specification of these operators relied both on a repeated CPS transformation and on an evaluator with several layers of continuations (as is obtained by repeatedly transforming a direct-style evaluator into continuation-passing style). Only subsequently have shift and reset been specified operationally, by developing operational analogues of a continuation semantics and of the CPS transformation [77].

The goal of our work here is to establish a new operational foundation for delimited continuations, using CPS as a guideline. To this end, we start with the original evaluator for shift_1 and reset_1. This evaluator uses two layers of continuations: a continuation and a meta-continuation. We then defunctionalize it into an abstract machine [5] and we construct the corresponding reduction semantics [85], as pioneered by Felleisen and Friedman [89]. The development scales to shift_n and reset_n. It is reusable for any control operators that are compatible with CPS, i.e., that can be characterized with a (possibly iterated) CPS translation or with a continuation-based evaluator. It also pinpoints where operational intuitions go beyond CPS.

This article is structured as follows. In Section 7.2, we review the enabling technology of our work: Reynolds's defunctionalization, the observation that a defunctionalized CPS program implements an abstract machine, and the observation that Felleisen's evaluation contexts are the defunctionalized continuations of a continuation-passing evaluator; we demonstrate this enabling technology on a simple example, arithmetic expressions. In Section 7.3, we illustrate the use of shift and reset with the classic example of finding list prefixes, using an ML-like programming language. In Section 7.4, we then present our main result: starting from the original evaluator for shift and reset, we defunctionalize it into an abstract machine; we analyze this abstract machine and construct the corresponding reduction semantics. In Section 7.5, we extend this result to the CPS hierarchy. In Section 7.6, we illustrate the CPS hierarchy with a class of normalization functions for a hierarchical language of units and products.

7.2 From evaluator to reduction semantics for arithmetic expressions

We demonstrate the derivation from an evaluator to a reduction semantics. The derivation consists of the following steps:

1. we start from an evaluator for a given language; if it is in direct style, we CPS-transform it;

2. we defunctionalize the CPS evaluator, obtaining a value-based abstract machine;

3. we modify the abstract machine to make it term-based instead of value-based; in particular, if the evaluator uses an environment, then so does

- Values: $\mathsf{val} \ni v ::= m$
- Evaluation function: $\mathsf{eval} : \mathsf{exp} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner) & = & m \\ \mathsf{eval}\,(e_1 + e_2) & = & \mathsf{eval}\,(e_1) + \mathsf{eval}\,(e_2) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\,(e) \quad = \quad \mathsf{eval}\,(e)$$

Figure 7.1: A direct-style evaluator for arithmetic expressions

the corresponding value-based abstract machine, and in that case, making the machine term-based leads us to use substitutions rather than an environment;

4. we analyze the transitions of the term-based abstract machine to identify the evaluation strategy it implements and the set of reductions it performs; the result is a reduction semantics.

The first two steps are based on previous work on a functional correspondence between evaluators and abstract machines [5–7, 33, 65], which itself is based on Reynolds's seminal work on definitional interpreters [196]. The last two steps follow the lines of Felleisen and Friedman's original work on a reduction semantics for the call-by-value λ-calculus extended with control operators [89]. The last step has been studied further by Hardin, Maranget, and Pagano [115] in the context of explicit substitutions and by Biernacka, Danvy, and Nielsen [30, 31, 75].

In the rest of this section, our running example is the language of arithmetic expressions, formed using natural numbers (the values) and additions (the computations):

$$\mathsf{exp} \ni e ::= \ulcorner m \urcorner \mid e_1 + e_2$$

7.2.1 The starting point: an evaluator in direct style

We define an evaluation function for arithmetic expressions by structural induction on their syntax. The resulting direct-style evaluator is displayed in Figure 7.1.

7.2.2 CPS transformation

We CPS-transform the evaluator by naming intermediate results, sequentializing their computation, and introducing an extra functional parameter, the continuation [68, 186, 217]. The resulting continuation-passing evaluator is displayed in Figure 7.2.

- Values: $\mathsf{val} \ni v ::= m$
- Continuations: $\mathsf{cont} = \mathsf{val} \to \mathsf{val}$
- Evaluation function: $\mathsf{eval} : \mathsf{exp} \times \mathsf{cont} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner,\, k) & = & k\, m \\ \mathsf{eval}\,(e_1 + e_2,\, k) & = & \mathsf{eval}\,(e_1,\, \lambda m_1.\, \mathsf{eval}\,(e_2,\, \lambda m_2.\, k\,(m_1 + m_2))) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\,(e) \quad = \quad \mathsf{eval}\,(e,\, \lambda v.\, v)$$

Figure 7.2: A continuation-passing evaluator for arithmetic expressions

- Values: $\mathsf{val} \ni v ::= m$
- Defunctionalized continuations:

$$\mathsf{cont} \ni k ::= [\,] \mid \mathsf{ADD}_2\,(e,\, k) \mid \mathsf{ADD}_1\,(v,\, k)$$

- Functions $\mathsf{eval} : \mathsf{exp} \times \mathsf{cont} \to \mathsf{val}$
  $\mathsf{apply\_cont} : \mathsf{cont} \times \mathsf{val} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner,\, k) & = & \mathsf{apply\_cont}\,(k,\, m) \\ \mathsf{eval}\,(e_1 + e_2,\, k) & = & \mathsf{eval}\,(e_1,\, \mathsf{ADD}_2\,(e_2,\, k)) \end{array}$$

$$\begin{array}{rcl} \mathsf{apply\_cont}\,([\,],\, v) & = & v \\ \mathsf{apply\_cont}\,(\mathsf{ADD}_2\,(e_2,\, k),\, v_1) & = & \mathsf{eval}\,(e_2,\, \mathsf{ADD}_1\,(v_1,\, k)) \\ \mathsf{apply\_cont}\,(\mathsf{ADD}_1\,(m_1,\, k),\, m_2) & = & \mathsf{apply\_cont}\,(k,\, m_1 + m_2) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\,(e) \quad = \quad \mathsf{eval}\,(e,\, [\,])$$

Figure 7.3: A defunctionalized continuation-passing evaluator for arithmetic expressions

7.2.3 Defunctionalization

The generalization of closure conversion [155] to defunctionalization is due to Reynolds [196]. The goal is to represent a functional value with a first-order data structure. The means is to partition the function space into a first-order sum where each summand corresponds to a lambda-abstraction in the program. In a defunctionalized program, function introduction is thus represented as an injection, and function elimination as a call to a first-order apply function implementing a case dispatch. In an ML-like functional language, sums

- Values: $v ::= m$
- Evaluation contexts: $C ::= [\,] \mid \mathsf{ADD}_2\ (e,\ C) \mid \mathsf{ADD}_1\ (v,\ C)$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\ [\,]\rangle_{eval}$
$\langle \ulcorner m \urcorner,\ C\rangle_{eval}$	$\Rightarrow$	$\langle C,\ m\rangle_{app}$
$\langle e_1 + e_2,\ C\rangle_{eval}$	$\Rightarrow$	$\langle e_1,\ \mathsf{ADD}_2\ (e_2,\ C)\rangle_{eval}$
$\langle \mathsf{ADD}_2\ (e_2,\ C),\ v_1\rangle_{app}$	$\Rightarrow$	$\langle e_2,\ \mathsf{ADD}_1\ (v_1,\ C)\rangle_{eval}$
$\langle \mathsf{ADD}_1\ (m_1,\ C),\ m_2\rangle_{app}$	$\Rightarrow$	$\langle C,\ m_1 + m_2\rangle_{app}$
$\langle [\,],\ v\rangle_{app}$	$\Rightarrow$	v

Figure 7.4: A value-based abstract machine for evaluating arithmetic expressions

are represented as data types, injections as data-type constructors, and apply functions are defined by case over the corresponding data types [74].

Here, we defunctionalize the continuation of the continuation-passing evaluator in Figure 7.2. We thus need to define a first-order algebraic data type and its apply function. To this end, we enumerate the lambda-abstractions that give rise to the inhabitants of this function space; there are three: the initial continuation in evaluate and the two continuations in eval. The initial continuation is closed, and therefore the corresponding algebraic constructor is nullary. The two other continuations have two free variables, and therefore the corresponding constructors are binary. As for the apply function, it interprets the algebraic constructors. The resulting defunctionalized evaluator is displayed in Figure 7.3.

7.2.4 Abstract machines as defunctionalized continuation-passing programs

Elsewhere [5,65], we have observed that a defunctionalized continuation-passing program implements an abstract machine: each configuration is the name of a function together with its arguments, and each function clause represents a transition. (As a corollary, we have also observed that the defunctionalized continuation of an evaluator forms what is known as an 'evaluation context' [64, 74, 89].)

Indeed Plotkin's Indifference Theorem [186] states that continuation-passing programs are independent of their evaluation order. In Reynolds's words [196], all the subterms in applications are 'trivial'; and in Moggi's words [172], these subterms are values and not computations. Furthermore, continuation-passing programs are tail recursive [217]. Therefore, since in a continuation-passing program all calls are tail calls and all subcomputations are elementary, a defunc-

- Expressions and values: $e ::= v \mid e_1 \texttt{ + } e_2$
  $v ::= \ulcorner m \urcorner$
- Evaluation contexts: $C ::= [\,] \mid \mathsf{ADD}_2\ (e,\ C) \mid \mathsf{ADD}_1\ (v,\ C)$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\ [\,] \rangle_{eval}$
$\langle \ulcorner m \urcorner,\ C \rangle_{eval}$	$\Rightarrow$	$\langle C,\ \ulcorner m \urcorner \rangle_{app}$
$\langle e_1 \texttt{ + } e_2,\ C \rangle_{eval}$	$\Rightarrow$	$\langle e_1,\ \mathsf{ADD}_2\ (e_2,\ C) \rangle_{eval}$
$\langle \mathsf{ADD}_2\ (e_2,\ C),\ v_1 \rangle_{app}$	$\Rightarrow$	$\langle e_2,\ \mathsf{ADD}_1\ (v_1,\ C) \rangle_{eval}$
$\langle \mathsf{ADD}_1\ (\ulcorner m_1 \urcorner,\ C),\ \ulcorner m_2 \urcorner \rangle_{app}$	$\Rightarrow$	$\langle C,\ \ulcorner m_1 + m_2 \urcorner \rangle_{app}$
$\langle [\,],\ v \rangle_{app}$	$\Rightarrow$	v

Figure 7.5: A term-based abstract machine for processing arithmetic expressions

tionalized continuation-passing program implements a transition system [187], i.e., an abstract machine.

We thus reformat Figure 7.3 into Figure 7.4. The correctness of the abstract machine with respect to the initial evaluator follows from the correctness of CPS transformation and of defunctionalization.

7.2.5 From value-based abstract machine to term-based abstract machine

We observe that the domain of expressible values in Figure 7.4 can be embedded in the syntactic domain of expressions. We therefore adapt the abstract machine to work on terms rather than on values. The result is displayed in Figure 7.5; it is a syntactic theory [85].

7.2.6 From term-based abstract machine to reduction semantics

The method of deriving a reduction semantics from an abstract machine was introduced by Felleisen and Friedman [89] to give a reduction semantics for control operators. Let us demonstrate it.

We analyze the transitions of the abstract machine in Figure 7.5. The second component of *eval*-transitions—the stack representing "the rest of the computation"—has already been identified as the evaluation context of the currently processed expression. We thus read a configuration $\langle e,\ C \rangle_{eval}$ as a decomposition of some expression into a sub-expression e and an evaluation context C.

Next, we identify the reduction and decomposition rules in the transitions of

the machine. Since a configuration can be read as a decomposition, we compare the left-hand side and the right-hand side of each transition. If they represent the same expression, then the given transition defines a decomposition (i.e., it searches for the next redex according to some evaluation strategy); otherwise we have found a redex. Moreover, reading the decomposition rules from right to left defines a 'plug' function that reconstructs an expression from its decomposition.

Here the decomposition function as read off the abstract machine is total. In general, however, it may be undefined for stuck terms; one can then extend it straightforwardly into a total function that decomposes a term into a context and a *potential redex*, i.e., an *actual redex* (as read off the machine), or a *stuck redex*.

In this simple example there is only one reduction rule. This rule performs the addition of natural numbers:

$$\text{(add)} \quad C\,[\ulcorner m_1 \urcorner + \ulcorner m_2 \urcorner] \;\rightarrow\; C\,[\ulcorner m_1 + m_2 \urcorner]$$

The remaining transitions decompose an expression according to the left-to-right strategy.

7.2.7 From reduction semantics to term-based abstract machine

In Section 7.2.6, we have constructed the reduction semantics corresponding to the abstract machine of Figure 7.5, as pioneered by Felleisen and Friedman [88, 89]. Over the last few years [30, 31, 63, 75], Biernacka, Danvy, and Nielsen have studied the converse transformation and systematized the construction of an abstract machine from a reduction semantics. The main idea is to short-cut the decompose-contract-plug loop, in the definition of evaluation as the transitive closure of one-step reduction, into a refocus-contract loop. The refocus function is constructed as an efficient (i.e., deforested) composition of plug and decompose that maps a term and a context either to a value or to a redex and a context. The result is a 'pre-abstract machine' computing the transitive closure of the refocus function. This pre-abstract machine can then be simplified into an eval/apply abstract machine.

It is simple to verify that using refocusing, one can go from the reduction semantics of Section 7.2.6 to the eval/apply abstract machine of Figure 7.5.

7.2.8 Summary and conclusion

We have demonstrated how to derive an abstract machine out of an evaluator, and how to construct the corresponding reduction semantics out of this abstract machine. In Section 7.4, we apply this derivation and this construction to the first level of the CPS hierarchy, and in Section 7.5, we apply them to an arbitrary level of the CPS hierarchy. But first, let us illustrate how to program with delimited continuations.

7.3 Programming with delimited continuations

We present two examples of programming with delimited continuations. Given a list xs and a predicate p, we want

1. to find the first prefix of xs whose last element satisfies p, and

2. to find all such prefixes of xs.

For example, given the predicate $\lambda m.m > 2$ and the list $[0, 3, 1, 4, 2, 5]$, the first prefix is $[0, 3]$ and the list of all the prefixes is $[[0, 3], [0, 3, 1, 4], [0, 3, 1, 4, 2, 5]]$.

In Section 7.3.1, we start with a simple solution that uses a first-order accumulator. This simple solution is in defunctionalized form. In Section 7.3.2, we present its higher-order counterpart, which uses a functional accumulator. This functional accumulator acts as a delimited continuation. In Section 7.3.3, we present its direct-style counterpart (which uses shift and reset) and in Section 7.3.4, we present its continuation-passing counterpart (which uses two layers of continuations). In Section 7.3.5, we introduce the CPS hierarchy informally. We then mention a typing issue in Section 7.3.6 and review related work in Section 7.3.7.

7.3.1 Finding prefixes by accumulating lists

A simple solution is to accumulate the prefix of the given list in reverse order while traversing this list and testing each of its elements:

- if no element satisfies the predicate, there is no prefix and the result is the empty list;

- otherwise, the prefix is the reverse of the accumulator.

$$
\begin{array}{lcl}
\mathit{find\_first\_prefix\_a}\,(p,\ xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\ a) \\
& & \qquad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\ a) \\
& & \qquad = \mathit{let}\ a' = x :: a \\
& & \qquad\ \ \mathit{in}\ \mathit{if}\ p\,x \\
& & \qquad\qquad \mathit{then}\ \mathit{reverse}\,(a',\ \mathit{nil}) \\
& & \qquad\qquad \mathit{else}\ \mathit{visit}\,(xs,\ a') \\
& & \ \ \mathit{and}\ \mathit{reverse}\,(\mathit{nil},\ xs) \\
& & \qquad = xs \\
& & \quad \mid \mathit{reverse}\,(x :: a,\ xs) \\
& & \qquad = \mathit{reverse}\,(a,\ x :: xs) \\
& & \mathit{in}\ \mathit{visit}\,(xs,\ \mathit{nil})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{find\_all\_prefixes\_a}\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\,a) \\
& & \quad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\,a) \\
& & \quad = \mathit{let}\ a' = x :: a \\
& & \quad\ \ \mathit{in}\ \mathit{if}\ p\,x \\
& & \quad\quad \mathit{then}\,(\mathit{reverse}\,(a',\,\mathit{nil})) \\
& & \quad\quad\quad :: (\mathit{visit}\,(xs,\,a')) \\
& & \quad\quad \mathit{else}\ \mathit{visit}\,(xs,\,a') \\
& & \mathit{and}\ \mathit{reverse}\,(\mathit{nil},\,xs) \\
& & \quad = xs \\
& & \quad \mid \mathit{reverse}\,(x :: a,\,xs) \\
& & \quad = \mathit{reverse}\,(a,\,x :: xs) \\
& & \mathit{in}\ \mathit{visit}\,(xs,\,\mathit{nil})
\end{array}
$$

To find the first prefix, one stops as soon as a satisfactory list element is found. To list all the prefixes, one continues the traversal, adding the current prefix to the list of the remaining prefixes.

We observe that the two solutions are in defunctionalized form [74,196]: the accumulator has the data type of a defunctionalized function and *reverse* is its apply function. We present its higher-order counterpart next [132].

7.3.2 Finding prefixes by accumulating list constructors

Instead of accumulating the prefix in reverse order while traversing the given list, we accumulate a function constructing the prefix:

- if no element satisfies the predicate, the result is the empty list;
- otherwise, we apply the functional accumulator to construct the prefix.

$$
\begin{array}{lcl}
\mathit{find\_first\_prefix\_c}_1\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\,k) \\
& & \quad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\,k) \\
& & \quad = \mathit{let}\ k' = \lambda vs.k\,(x :: vs) \\
& & \quad\ \ \mathit{in}\ \mathit{if}\ p\,x \\
& & \quad\quad \mathit{then}\ k'\,\mathit{nil} \\
& & \quad\quad \mathit{else}\ \mathit{visit}\,(xs,\,k') \\
& & \mathit{in}\ \mathit{visit}\,(xs,\,\lambda vs.vs)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{find\_all\_prefixes\_c}_1\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\,k) \\
& & \qquad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\,k) \\
& & \qquad = \mathit{let}\ k' = \lambda vs.k\,(x :: vs) \\
& & \qquad\ \ \mathit{in}\ \mathit{if}\ p\,x \\
& & \qquad\qquad \mathit{then}\ (k'\ \mathit{nil}) :: (\mathit{visit}\,(xs,\,k')) \\
& & \qquad\qquad \mathit{else}\ \mathit{visit}\,(xs,\,k') \\
& & \mathit{in}\ \mathit{visit}\,(xs,\,\lambda vs.vs)
\end{array}
$$

To find the first prefix, one applies the functional accumulator as soon as a satisfactory list element is found. To list all such prefixes, one continues the traversal, adding the current prefix to the list of the remaining prefixes.

Defunctionalizing these two definitions yields the two definitions of Section 7.3.1.

The functional accumulator is a delimited continuation:

- In *find_first_prefix_c*$_1$, *visit* is written in CPS since all calls are tail calls and all sub-computations are elementary. The continuation is initialized in the initial call to *visit*, discarded in the base case, extended in the induction case, and used if a satisfactory prefix is found.

- In *find_all_prefixes_c*$_1$, *visit* is almost written in CPS except that the continuation is composed if a satisfactory prefix is found: it is used twice—once where it is applied to the empty list to construct a prefix, and once in the visit of the rest of the list to construct a list of prefixes; this prefix is then prepended to this list of prefixes.

These continuation-based programming patterns (initializing a continuation, not using it, or using it more than once as if it were a composable function) have motivated the control operators shift and reset [67, 68]. Using them, in the next section, we write *visit* in direct style.

7.3.3 Finding prefixes in direct style

The two following local functions are the direct-style counterpart of the two local functions in Section 7.3.2:

$$
\begin{array}{lcl}
\mathit{find\_first\_prefix\_c}_0\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
& & \qquad = \mathcal{S}k.\mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs) \\
& & \qquad = x :: (\mathit{if}\ p\,x\ \mathit{then}\ \mathit{nil}\ \mathit{else}\ \mathit{visit}\ xs) \\
& & \mathit{in}\ \langle \mathit{visit}\ xs \rangle
\end{array}
$$

$$\begin{array}{lcl}
\mathit{find_all_prefixes_c}_0\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
& & \quad = \mathcal{S}k.\mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs) \\
& & \quad = x :: \mathit{if}\ p\,x \\
& & \qquad \mathit{then}\ \mathcal{S}k'.\langle(k'\ \mathit{nil})\rangle :: \langle k'\,(\mathit{visit}\ xs)\rangle \\
& & \qquad \mathit{else}\ \mathit{visit}\ xs \\
& & \mathit{in}\ \langle \mathit{visit}\ xs\rangle
\end{array}$$

In both cases, *visit* is in direct style, i.e., it is not passed any continuation. The initial calls to *visit* are enclosed in the control delimiter reset (noted $\langle\cdot\rangle$ for conciseness). In the base cases, the current (delimited) continuation is captured with the control operator shift (noted $\mathcal{S}$), which has the effect of emptying the (delimited) context; this captured continuation is bound to an identifier k, which is not used; *nil* is then returned in the emptied context. In the induction case of *find_all_prefixes_c*$_0$, if the predicate is satisfied, *visit* captures the current continuation and applies it twice—once to the empty list to construct a prefix, and once to the result of visiting the rest of the list to construct a list of prefixes; this prefix is then prepended to the list of prefixes.

CPS-transforming these two local functions yields the two definitions of Section 7.3.2 [68].

7.3.4 Finding prefixes in continuation-passing style

The two following local functions are the continuation-passing counterpart of the two local functions in Section 7.3.2:

$$\begin{array}{lcl}
\mathit{find_first_prefix_c}_2\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\,k_1,\,k_2) \\
& & \quad = k_2\ \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\,k_1,\,k_2) \\
& & \quad = \mathit{let}\ k_1' = \lambda(vs,\,k_2').k_1\,(x :: vs,\,k_2') \\
& & \qquad \mathit{in}\ \mathit{if}\ p\,x \\
& & \qquad\quad \mathit{then}\ k_1'\,(\mathit{nil},\,k_2) \\
& & \qquad\quad \mathit{else}\ \mathit{visit}\,(xs,\,k_1',\,k_2) \\
& & \mathit{in}\ \mathit{visit}\,(xs,\,\lambda(vs,\,k_2).k_2\ vs,\,\lambda vs.vs)
\end{array}$$

$$\begin{array}{lcl}
\mathit{find_all_prefixes_c}_2\,(p,\,xs) & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\,k_1,\,k_2) \\
& & \quad = k_2\ \mathit{nil} \\
& & \quad \mid \mathit{visit}\,(x :: xs,\,k_1,\,k_2) \\
& & \quad = \mathit{let}\ k_1' = \lambda(vs,\,k_2').k_1\,(x :: vs,\,k_2') \\
& & \qquad \mathit{in}\ \mathit{if}\ p\,x \\
& & \qquad\quad \mathit{then}\ k_1'\,(\mathit{nil},\,\lambda vs.\mathit{visit}\ (xs,\,k_1', \\
& & \qquad\qquad\qquad\qquad \lambda vss.k_2\,(vs :: vss)) \\
& & \qquad\quad \mathit{else}\ \mathit{visit}\,(xs,\,k_1',\,k_2) \\
& & \mathit{in}\ \mathit{visit}\,(xs,\,\lambda(vs,\,k_2).k_2\ vs,\,\lambda vss.vss)
\end{array}$$

CPS-transforming the two local functions of Section 7.3.2 adds another layer of continuations and restores the syntactic characterization of all calls being tail calls and all sub-computations being elementary.

7.3.5 The CPS hierarchy

If k_2 were used non-tail recursively in a variant of the examples of Section 7.3.4, we could CPS-transform the definitions one more time, adding one more layer of continuations and restoring the syntactic characterization of all calls being tail calls and all sub-computations being elementary. We could also map this definition back to direct style, eliminating k_2 but accessing it with shift. If the result were mapped back to direct style one more time, k_2 would then be accessed with a new control operator, shift_2, and k_1 would be accessed with shift (or more precisely with shift_1).

All in all, successive CPS-transformations induce a CPS hierarchy [67, 77], and abstracting control up to each successive layer is achieved with successive pairs of control operators shift and reset—reset to initialize the continuation up to a level, and shift to capture a delimited continuation up to this level. Each pair of control operators is indexed by the corresponding level in the hierarchy. Applying a captured continuation packages all the current layers on the next layer and restores the captured layers. When a captured continuation completes, the packaged layers are put back into place and the computation proceeds. (This informal description is made precise in Section 7.4.)

7.3.6 A note about typing

The type of *find_all_prefixes_c*$_1$, in Section 7.3.2, is

$$(\alpha \to \mathit{bool}) \times \alpha\ \mathit{list} \to \alpha\ \mathit{list}\ \mathit{list}$$

and the type of its local function *visit* is

$$\alpha\ \mathit{list} \times (\alpha\ \mathit{list} \to \alpha\ \mathit{list}) \to \alpha\ \mathit{list}\ \mathit{list}.$$

In this example, the co-domain of the continuation is not the same as the co-domain of *visit*.

Thus *find_all_prefixes_c*$_0$ provides a simple and meaningful example where Filinski's typing of shift [96] does not fit, since it must be used at type

$$((\beta \to \mathit{ans}) \to \mathit{ans}) \to \beta$$

for a given type *ans*, i.e., the answer type of the continuation and the type of the computation must be the same. In other words, control effects are not allowed to change the types of the contexts. Due to a similar restriction on the type of shift, the example does not fit either in Murthy's pseudo-classical type system for the CPS hierarchy [177] and in Wadler's most general monadic type system [240, Section 3.4]. It however fits in Danvy and Filinski's original type system [66] which Ariola, Herbelin, and Sabry have recently embedded in classical subtractive logic [12].

7.3.7 Related work

The example considered in this section builds on the simpler function that unconditionally lists the successive prefixes of a given list. This simpler function is a traditional example of delimited continuations [57, 212]:

- In the Lisp Pointers [57], Danvy presents three versions of this function: a typed continuation-passing version (corresponding to Section 7.3.2), one with delimited control (corresponding to Section 7.3.3), and one in assembly language.

- In his PhD thesis [212, Section 6.3], Sitaram presents two versions of this function: one with an accumulator (corresponding to Section 7.3.1) and one with delimited control (corresponding to Section 7.3.3).

In Section 7.3.2, we have shown that the continuation-passing version mediates the version with an accumulator and the version with delimited control since defunctionalizing the continuation-passing version yields one and mapping it back to direct style yields the other.

7.3.8 Summary and conclusion

We have illustrated delimited continuations with the classic example of finding list prefixes, using CPS as a guideline. Direct-style programs using shift and reset can be CPS-transformed into continuation-passing programs where some calls may not be tail calls and some sub-computations may not be elementary. One more CPS transformation establishes this syntactic property with a second layer of continuations. Further CPS transformations provide the extra layers of continuation that are characteristic of the CPS hierarchy.

In the next section, we specify the λ-calculus extended with shift and reset.

7.4 From evaluator to reduction semantics for delimited continuations

We derive a reduction semantics for the call-by-value λ-calculus extended with shift and reset, using the method demonstrated in Section 7.2. First, we transform an evaluator into an environment-based abstract machine. Then we eliminate the environment from this abstract machine, making it substitution-based. Finally, we read all the components of a reduction semantics off the substitution-based abstract machine.

Terms consist of integer literals, variables, λ-abstractions, function applications, applications of the successor function, reset expressions, and shift expressions:

$$e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle e \rangle \mid \mathcal{S}k.e$$

Programs are closed terms.

This source language is a subset of the language used in the examples of Section 7.3. Adding the remaining constructs is a straightforward exercise and does not contribute to our point here.

- Terms: $\mathsf{exp} \ni e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle e \rangle \mid \mathcal{S}k.e$
- Values: $\mathsf{val} \ni v ::= m \mid f$
- Answers, meta-continuations, continuations and functions:

$$\begin{array}{rcl} \mathsf{ans} & = & \mathsf{val} \\ k_2 \in \mathsf{cont}_2 & = & \mathsf{val} \to \mathsf{ans} \\ k_1 \in \mathsf{cont}_1 & = & \mathsf{val} \times \mathsf{cont}_2 \to \mathsf{ans} \\ f \in \mathsf{fun} & = & \mathsf{val} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans} \end{array}$$

- Initial continuation and meta-continuation: $\theta_1 = \lambda(v,\, k_2).\, k_2\ v$, $\theta_2 = \lambda v.\, v$
- Environments: $\mathsf{env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{exp} \times \mathsf{env} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans}$

$$\begin{array}{rcl} \mathsf{eval}\ (\ulcorner m \urcorner,\, \rho,\, k_1,\, k_2) & = & k_1\ (m,\, k_2) \\ \mathsf{eval}\ (x,\, \rho,\, k_1,\, k_2) & = & k_1\ (\rho(x),\, k_2) \\ \mathsf{eval}\ (\lambda x.e,\, \rho,\, k_1,\, k_2) & = & k_1\ (\lambda(v,\, k_1',\, k_2').\, \mathsf{eval}\ (e,\, \rho\{x \mapsto v\},\, k_1',\, k_2'),\, k_2) \\ \mathsf{eval}\ (e_0\, e_1,\, \rho,\, k_1,\, k_2) & = & \mathsf{eval}\ (e_0,\, \rho, \\ & & \lambda(f,\, k_2').\, \mathsf{eval}\ (e_1,\, \rho,\, \lambda(v,\, k_2'').\, f\ (v,\, k_1,\, k_2''),\, k_2'),\, k_2) \\ \mathsf{eval}\ (\mathit{succ}\ e,\, \rho,\, k_1,\, k_2) & = & \mathsf{eval}\ (e,\, \rho,\, \lambda(m,\, k_2').\, k_1\ (m+1,\, k_2'),\, k_2) \\ \mathsf{eval}\ (\langle e \rangle,\, \rho,\, k_1,\, k_2) & = & \mathsf{eval}\ (e,\, \rho,\, \theta_1,\, \lambda v.\, k_1\ (v,\, k_2)) \\ \mathsf{eval}\ (\mathcal{S}k.e,\, \rho,\, k_1,\, k_2) & = & \mathsf{eval}\ (e,\, \rho\{k \mapsto c\},\, \theta_1,\, k_2) \\ & & \text{where } c = \lambda(v,\, k_1',\, k_2').\, k_1\ (v,\, \lambda v'.\, k_1'\ (v',\, k_2')) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\,(e) \quad = \quad \mathsf{eval}\ (e,\, \rho_{mt},\, \theta_1,\, \theta_2)$$

Figure 7.6: An environment-based evaluator for the first level of the CPS hierarchy

7.4.1 An environment-based evaluator

Figure 7.6 displays an evaluator for the language of the first level of the CPS hierarchy. This evaluation function represents the original call-by-value semantics of the λ-calculus with shift and reset [67], augmented with integer literals and applications of the successor function. It is defined by structural induction over the syntax of terms, and it makes use of an environment ρ, a continuation k_1, and a meta-continuation k_2.

The evaluation of a terminating program that does not get stuck (i.e., a program where no ill-formed applications occur in the course of evaluation) yields either an integer, a function representing a λ-abstraction, or a captured continuation. Both evaluate and eval are partial functions to account for non-

terminating or stuck programs. The environment stores previously computed values of the free variables of the term under evaluation.

The meta-continuation intervenes to interpret reset expressions and to apply captured continuations. Otherwise, it is passively threaded through the evaluation of literals, variables, λ-abstractions, function applications, and applications of the successor function. (If it were not for shift and reset, and if eval were curried, k_2 could be eta-reduced and the evaluator would be in ordinary continuation-passing style.)

The reset control operator is used to delimit control. A reset expression $\langle e \rangle$ is interpreted by evaluating e with the initial continuation and a meta-continuation on which the current continuation has been "pushed." (Indeed, and as will be shown in Section 7.4.2, defunctionalizing the meta-continuation yields the data type of a stack [74].)

The shift control operator is used to abstract (delimited) control. A shift expression $\mathcal{S}k.e$ is interpreted by capturing the current continuation, binding it to k, and evaluating e in an environment extended with k and with a continuation reset to the initial continuation. Applying a captured continuation is achieved by "pushing" the current continuation on the meta-continuation and applying the captured continuation to the new meta-continuation. Resuming a continuation is achieved by reactivating the "pushed" continuation with the corresponding meta-continuation.

7.4.2 An environment-based abstract machine

The evaluator displayed in Figure 7.6 is already in continuation-passing style. Therefore, we only need to defunctionalize its expressible values and its continuations to obtain an abstract machine. This abstract machine is displayed in Figure 7.7.

The abstract machine consists of three sets of transitions: *eval* for interpreting terms, $cont_1$ for interpreting the defunctionalized continuations (i.e., the evaluation contexts),[1] and $cont_2$ for interpreting the defunctionalized meta-continuations (i.e., the meta-contexts).[2] The set of possible values includes integers, closures and captured contexts. In the original evaluator, the latter two were represented as higher-order functions, but defunctionalizing expressible values of the evaluator has led them to be distinguished.

This eval/apply/meta-apply abstract machine is an extension of the CEK machine [89], which is an eval/apply machine, with the meta-context C_2 and its two transitions, and the two transitions for shift and reset. C_2 intervenes to process reset expressions and to apply captured continuations. Otherwise, it is passively threaded through the processing of literals, variables, λ-abstractions,

[1] The grammar of evaluation contexts in Figure 7.7 is isomorphic to the grammar of evaluation contexts in the standard inside-out notation:

$$C_1 \quad ::= \quad [\,] \mid C_1[[\,]\ (e, \rho)] \mid C_1[\mathit{succ}\ [\,]] \mid C_1[v\ [\,]]$$

[2] To build on Peyton Jones's terminology [161], this abstract machine is therefore in 'eval/apply/meta-apply' form.

- Terms: $e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
- Values (integers, closures, and captured continuations):

$$v ::= m \mid [x,\, e,\, \rho] \mid C_1$$

- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation contexts:

$$C_1 ::= [\,] \mid \mathsf{ARG}\,((e, \rho),\ C_1) \mid \mathsf{SUCC}\,(C_1) \mid \mathsf{FUN}\,(v,\ C_1)$$

- Meta-contexts: $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\, \rho_{mt},\, [\,],\, \bullet \rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, m,\, C_2 \rangle_{\mathit{cont}_1}$
$\langle x,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, \rho\,(x),\, C_2 \rangle_{\mathit{cont}_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2 \rangle_{\mathit{cont}_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\ C_1),\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ e,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{SUCC}\,(C_1),\, C_2 \rangle_{\mathit{eval}}$
$\langle \langle\!\langle e \rangle\!\rangle,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e,\, \rho,\, [\,],\, C_2 \cdot C_1 \rangle_{\mathit{eval}}$
$\langle \mathcal{S}k.e,\, \rho,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e,\, \rho\{k \mapsto C_1\},\, [\,],\, C_2 \rangle_{\mathit{eval}}$
$\langle [\,],\, v,\, C_2 \rangle_{\mathit{cont}_1}$	$\Rightarrow$	$\langle C_2,\, v \rangle_{\mathit{cont}_2}$
$\langle \mathsf{ARG}\,((e, \rho),\ C_1),\, v,\, C_2 \rangle_{\mathit{cont}_1}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathsf{SUCC}\,(C_1),\, m,\, C_2 \rangle_{\mathit{cont}_1}$	$\Rightarrow$	$\langle C_1,\, m+1,\, C_2 \rangle_{\mathit{cont}_1}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2 \rangle_{\mathit{cont}_1}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2 \rangle_{\mathit{cont}_1}$	$\Rightarrow$	$\langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{\mathit{cont}_1}$
$\langle C_2 \cdot C_1,\, v \rangle_{\mathit{cont}_2}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2 \rangle_{\mathit{cont}_1}$
$\langle \bullet,\, v \rangle_{\mathit{cont}_2}$	$\Rightarrow$	v

Figure 7.7: An environment-based abstract machine for the first level of the CPS hierarchy

function applications, and applications of the successor function. (If it were not for shift and reset, C_2 and its transitions could be omitted and the abstract machine would reduce to the CEK machine.)

Given an environment ρ, a context C_1, and a meta-context C_2, a reset expression $\langle\!\langle e \rangle\!\rangle$ is processed by evaluating e with the same environment ρ, the empty context $[\,]$, and a meta-context where C_1 has been pushed on C_2.

Given an environment ρ, a context C_1, and a meta-context C_2, a shift

- Terms and values: $e ::= v \mid x \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
  $v ::= \ulcorner m \urcorner \mid \lambda x.e \mid C_1$
- Evaluation contexts: $C_1 ::= [\,] \mid \mathsf{ARG}\,(e,\ C_1) \mid \mathsf{SUCC}\,(C_1) \mid \mathsf{FUN}\,(v,\ C_1)$
- Meta-contexts: $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\ [\,],\ \bullet \rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\ \ulcorner m \urcorner,\ C_2 \rangle_{\mathit{cont_1}}$
$\langle \lambda x.e,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\ \lambda x.e,\ C_2 \rangle_{\mathit{cont_1}}$
$\langle C_1',\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\ C_1',\ C_2 \rangle_{\mathit{cont_1}}$
$\langle e_0\, e_1,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e_0,\ \mathsf{ARG}\,(e_1,\ C_1),\ C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ e,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e,\ \mathsf{SUCC}\,(C_1),\ C_2 \rangle_{\mathit{eval}}$
$\langle \langle\!\langle e \rangle\!\rangle,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e,\ [\,],\ C_2 \cdot C_1 \rangle_{\mathit{eval}}$
$\langle \mathcal{S}k.e,\ C_1,\ C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle e\{C_1/k\},\ [\,],\ C_2 \rangle_{\mathit{eval}}$
$\langle [\,],\ v,\ C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_2,\ v \rangle_{\mathit{cont_2}}$
$\langle \mathsf{ARG}\,(e,\ C_1),\ v,\ C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle e,\ \mathsf{FUN}\,(v,\ C_1),\ C_2 \rangle_{\mathit{eval}}$
$\langle \mathsf{SUCC}\,(C_1),\ \ulcorner m \urcorner,\ C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1,\ \ulcorner m+1 \urcorner,\ C_2 \rangle_{\mathit{cont_1}}$
$\langle \mathsf{FUN}\,(\lambda x.e,\ C_1),\ v,\ C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle e\{v/x\},\ C_1,\ C_2 \rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1',\ v,\ C_2 \cdot C_1 \rangle_{\mathit{cont_1}}$
$\langle C_2 \cdot C_1,\ v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	$\langle C_1,\ v,\ C_2 \rangle_{\mathit{cont_1}}$
$\langle \bullet,\ v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	v

Figure 7.8: A substitution-based abstract machine for the first level of the CPS hierarchy

expression $\mathcal{S}k.e$ is processed by evaluating e with an extension of ρ where k denotes C_1, the empty context $[\,]$, and a meta-context C_2. Applying a captured context C_1' is achieved by pushing the current context C_1 on the current meta-context C_2 and continuing with C_1'. Resuming a context C_1 is achieved by popping it off the meta-context $C_2 \cdot C_1$ and continuing with C_1.

The correctness of the abstract machine with respect to the evaluator is a consequence of the correctness of defunctionalization. In order to express it formally, we define a partial function $\mathsf{eval}^{\mathsf{e}}$ mapping a term e to a value v whenever the environment-based machine, started with e, stops with v. The following theorem states this correctness by relating observable results:

Theorem 7.1 *For any program e and any integer value m, $\mathsf{evaluate}\,(e) = m$ if and only if $\mathsf{eval}^{\mathsf{e}}\,(e) = m$.*

Proof. The theorem follows directly from the correctness of defunctionaliza-

tion [18, 180]. □

The environment-based abstract machine can serve both as a foundation for implementing functional languages with control operators for delimited continuations and as a stepping stone in theoretical studies of shift and reset. In the rest of this section, we use it to construct a reduction semantics of shift and reset.

7.4.3 A substitution-based abstract machine

The environment-based abstract machine of Figure 7.7, on which we want to base our development, makes a distinction between terms and values. Since a reduction semantics is specified by purely syntactic operations (it gives meaning to terms by specifying their rewriting strategy and an appropriate notion of reduction, and is indeed also referred to as 'syntactic theory'), we need to embed the domain of values back into the syntax. To this end we transform the environment-based abstract machine into the substitution-based abstract machine displayed in Figure 7.8. The transformation is standard, except that we also need to embed evaluation contexts in the syntax; hence the substitution-based machine operates on terms where "quoted" (in the sense of Lisp) contexts can occur. (If it were not for shift and reset, C_2 and its transitions could be omitted and the abstract machine would reduce to the CK machine [89].)

We write $e\{v/x\}$ to denote the result of the usual capture-avoiding substitution of the value v for x in e.

Formally, the relationship between the two machines is expressed with the following simulation theorem, where evaluation with the substitution-based abstract machine is captured by the partial function $\mathsf{eval}^{\mathsf{s}}$, defined analogously to $\mathsf{eval}^{\mathsf{e}}$.

Theorem 7.2 *For any program e, either both $\mathsf{eval}^{\mathsf{s}}(e)$ and $\mathsf{eval}^{\mathsf{e}}(e)$ are undefined, or there exist values v, v' such that $\mathsf{eval}^{\mathsf{s}}(e) = v$, $\mathsf{eval}^{\mathsf{e}}(e) = v'$ and $\mathcal{T}(v') = v$. The function $\mathcal{T}$ relates a semantic value with its syntactic representation and is defined as follows:*[3]

$$
\begin{array}{rcl}
\mathcal{T}(m) & = & \ulcorner m \urcorner \\
\mathcal{T}([x,\, e,\, \rho]) & = & \lambda x.e\{\mathcal{T}(\rho(x_1))/x_1\} \ldots \{\mathcal{T}(\rho(x_n))/x_n\}, \\
 & & \textit{where } FV\,(\lambda x.\, e) = \{x_1, \ldots, x_n\} \\
\mathcal{T}([\,]) & = & [\,] \\
\mathcal{T}(\mathsf{ARG}\,((e, \rho),\, C_1)) & = & \mathsf{ARG}\,(e\{\mathcal{T}(\rho(x_1))/x_1\} \ldots \{\mathcal{T}(\rho(x_n))/x_n\},\ \mathcal{T}(C_1)), \\
 & & \textit{where } FV\,(e) = \{x_1, \ldots, x_n\} \\
\mathcal{T}(\mathsf{FUN}\,(v,\, C_1)) & = & \mathsf{FUN}\,(\mathcal{T}(v),\ \mathcal{T}(C_1)) \\
\mathcal{T}(\mathsf{SUCC}(C_1)) & = & \mathsf{SUCC}(\mathcal{T}(C_1))
\end{array}
$$

Proof. We extend the translation function $\mathcal{T}$ to meta-contexts and configurations, in the expected way, e.g.,

$$\mathcal{T}(\langle e,\, \rho,\, C_1,\, C_2\rangle_{eval}) = \langle e\{\mathcal{T}(\rho(x_1))/x_1\} \ldots \{\mathcal{T}(\rho(x_n))/x_n\},\, \mathcal{T}(C_1),\, \mathcal{T}(C_2)\rangle_{eval}$$

where $FV\,(e) = \{x_1, \ldots, x_n\}$

[3] $\mathcal{T}$ is a generalization of Plotkin's function Real [186].

Then it is straightforward to show that the two abstract machines operate in lock step with respect to the translation. Hence, for any program e, both machines diverge or they both stop (after the same number of transitions) with the values v and $\mathcal{T}(v)$, respectively. □

We now proceed to analyze the transitions of the machine displayed in Figure 7.8. We can think of a configuration $\langle e, C_1, C_2\rangle_{eval}$ as the following decomposition of the initial term into a meta-context C_2, a context C_1, and an intermediate term e:

$$C_2 \,\#\, C_1[e]$$

where # separates the context and the meta-context. Each transition performs either a reduction, or a decomposition in search of the next redex. Let us recall that a decomposition is performed when both sides of a transition are partitions of the same term; in that case, depending on the structure of the decomposition $C_2 \,\#\, C_1[e]$, a subpart of the term is chosen to be evaluated next, and the contexts are updated accordingly. We also observe that *eval*-transitions follow the structure of e, $cont_1$-transitions follow the structure of C_1 when the term has been reduced to a value, and $cont_2$-transitions follow the structure of C_2 when a value in the empty context has been reached.

Next we specify all the components of the reduction semantics based on the analysis of the abstract machine.

7.4.4 A reduction semantics

A reduction semantics provides a reduction relation on expressions by defining values, evaluation contexts, and redexes [85, 88, 89, 252]. In the present case,

- the values are already specified in the (substitution-based) abstract machine:

$$v ::= \ulcorner m \urcorner \mid \lambda x.e \mid C_1$$

- the evaluation contexts and meta-contexts are already specified in the abstract machine, as the data-type part of defunctionalized continuations;

$$C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,(e,\ C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{SUCC}(C_1)$$
$$C_2 ::= \bullet \mid C_2 \cdot C_1$$

- we can read the redexes off the transitions of the abstract machine:

$$r ::= succ\ \ulcorner m \urcorner \mid (\lambda x.e)\, v \mid \mathcal{S}k.e \mid C_1'\, v \mid \langle v \rangle$$

Based on the distinction between decomposition and reduction, we single out the following reduction rules from the transitions of the machine:

$$\begin{array}{lrcl}
(\delta) & C_2 \,\#\, C_1[succ\ \ulcorner m \urcorner] & \rightarrow & C_2 \,\#\, C_1[\ulcorner m+1 \urcorner] \\
(\beta_\lambda) & C_2 \,\#\, C_1[(\lambda x.e)\, v] & \rightarrow & C_2 \,\#\, C_1[e\{v/x\}] \\
(\mathcal{S}_\lambda) & C_2 \,\#\, C_1[\mathcal{S}k.e] & \rightarrow & C_2 \,\#\, [e\{C_1/k\}] \\
(\beta_{ctx}) & C_2 \,\#\, C_1[C_1'\, v] & \rightarrow & C_2 \cdot C_1 \,\#\, C_1'[v] \\
(\text{Reset}) & C_2 \,\#\, C_1[\langle v \rangle] & \rightarrow & C_2 \,\#\, C_1[v]
\end{array}$$

(β_λ) is the usual call-by-value β-reduction; we have renamed it to indicate that the applied term is a λ-abstraction, since we can also apply a captured context, as in (β_{ctx}). ($\mathcal{S}_\lambda$) is plausibly symmetric to (β_λ) — it can be seen as an application of the abstraction $\lambda k.e$ to the current context. Moreover, (β_{ctx}) can be seen as performing both a reduction and a decomposition: it is a reduction because an application of a context with a hole to a value is reduced to the value plugged into the hole; and it is a decomposition because it changes the meta-context, as if the application were enclosed in a reset. Finally, (Reset) makes it possible to pass the boundary of a context when the term inside this context has been reduced to a value.

The β_{ctx}-rule and the $\mathcal{S}_\lambda$-rule give a justification for representing a captured context C_1 as a term $\lambda x.\langle C_1[x]\rangle$, as found in other studies of shift and reset [141, 142, 177]. In particular, the need for delimiting the captured context is a consequence of the β_{ctx}-rule.

Finally, we can read the decomposition function off the transitions of the abstract machine:

$$\begin{array}{rcl}
\mathit{decompose}(e) & = & \mathit{decompose}'\,(e,\,[\,],\,\bullet) \\
\mathit{decompose}'\,(e_0\,e_1,\,C_1,\,C_2) & = & \mathit{decompose}'\,(e_0,\,\mathsf{ARG}\,(e_1,\,C_1),\,C_2) \\
\mathit{decompose}'\,(\mathit{succ}\;e,\,C_1,\,C_2) & = & \mathit{decompose}'\,(e,\,\mathsf{SUCC}(C_1),\,C_2) \\
\mathit{decompose}'\,(\langle e\rangle,\,C_1,\,C_2) & = & \mathit{decompose}'\,(e,\,[\,],\,C_2\cdot C_1) \\
\mathit{decompose}'\,(v,\,\mathsf{ARG}\,(e,\,C_1),\,C_2) & = & \mathit{decompose}'\,(e,\,\mathsf{FUN}\,(v,\,C_1),\,C_2)
\end{array}$$

In the remaining cases either a value or a redex has been found:

$$\begin{array}{rcl}
\mathit{decompose}'\,(v,\,[\,],\,\bullet) & = & \bullet\;\#\;[v] \\
\mathit{decompose}'\,(v,\,[\,],\,C_2\cdot C_1) & = & C_2\;\#\;C_1[\langle v\rangle] \\
\mathit{decompose}'\,(\mathcal{S}k.e,\,C_1,\,C_2) & = & C_2\;\#\;C_1[\mathcal{S}k.e] \\
\mathit{decompose}'\,(v,\,\mathsf{FUN}\,((\lambda x.e),\,C_1),\,C_2) & = & C_2\;\#\;C_1[(\lambda x.e)\,v] \\
\mathit{decompose}'\,(v,\,\mathsf{FUN}\,(C_1',\,C_1),\,C_2) & = & C_2\;\#\;C_1[C_1'\,v] \\
\mathit{decompose}'\,(\ulcorner m\urcorner,\,\mathsf{SUCC}\,(C_1),\,C_2) & = & C_2\;\#\;C_1[\mathit{succ}\;\ulcorner m\urcorner]
\end{array}$$

An inverse of the *decompose* function, traditionally called *plug*, reconstructs a term from its decomposition:

$$\begin{array}{rcl}
\mathit{plug}\,(\bullet\;\#\;[e]) & = & e \\
\mathit{plug}\,(C_2\cdot C_1\;\#\;[e]) & = & \mathit{plug}\,(C_2\;\#\;C_1[\langle e\rangle]) \\
\mathit{plug}\,(C_2\;\#\;(\mathsf{ARG}\,(e',\,C_1))[e]) & = & \mathit{plug}\,(C_2\;\#\;C_1[e\,e']) \\
\mathit{plug}\,(C_2\;\#\;(\mathsf{FUN}\,(v,\,C_1))[e]) & = & \mathit{plug}\,(C_2\;\#\;C_1[v\,e]) \\
\mathit{plug}\,(C_2\;\#\;(\mathsf{SUCC}\,(C_1))[e]) & = & \mathit{plug}\,(C_2\;\#\;C_1[\mathit{succ}\;e])
\end{array}$$

In order to talk about unique decomposition, we need to define the set of potential redexes (i.e., the disjoint union of actual redexes and stuck redexes). The grammar of potential redexes reads as follows:

$$p ::= \mathit{succ}\;v \mid v_0\,v_1 \mid \mathcal{S}k.e \mid \langle v\rangle$$

Lemma 7.1 (Unique decomposition) *A program e is either a value v or there exist a unique context C_1, a unique meta-context C_2 and a potential redex*

p such that $e = plug\,(C_2 \mathbin{\#} C_1[p])$. *In the former case* $decompose(e) = \bullet \mathbin{\#} [v]$ *and in the latter case either* $decompose(e) = C_2 \mathbin{\#} C_1[p]$ *if* p *is an actual redex, or* $decompose(e)$ *is undefined.*

Proof. The first part follows by induction on the structure of e. The second part follows from the equation $decompose(plug\,(C_2 \mathbin{\#} C_1[r])) = C_2 \mathbin{\#} C_1[r]$ which holds for all C_2, C_1 and r. □

It is evident that evaluating a program either using the derived reduction rules or using the substitution-based abstract machine yields the same result.

Theorem 7.3 *For any program* e *and any value* v, $\mathsf{eval}^{\mathsf{s}}\,(e) = v$ *if and only if* $e \rightarrow^* v$, *where* $\rightarrow^*$ *is the reflexive, transitive closure of the one-step reduction defined by the relation* $\rightarrow$.

Proof. When evaluating with the abstract machine, each contraction is followed by decomposing the contractum in the current context and meta-context. When evaluating with the reduction rules, however, each contraction is followed by plugging the contractum and decomposing the resulting term. Therefore, the theorem follows from the equation

$$decompose'\,(e,\, C_1,\, C_2) = decompose(plug\,(C_2 \mathbin{\#} C_1[e]))$$

which holds for any C_2, C_1 and e. □

We have verified that using refocusing [31,75], one can go from this reduction semantics to the abstract machine of Figure 7.8.

7.4.5 Beyond CPS

Alternatively to using the meta-context to compose delimited continuations, as in Figure 7.7, we could compose them by concatenating their representation [93]. Such a concatenation function is defined as follows:

$$\begin{array}{rcl}
[\,] \star C_1' & = & C_1' \\
(\mathsf{ARG}\,((e,\rho),\ C_1)) \star C_1' & = & \mathsf{ARG}\,((e,\rho),\ C_1 \star C_1') \\
(\mathsf{SUCC}\,(C_1)) \star C_1' & = & \mathsf{SUCC}\,(C_1 \star C_1') \\
(\mathsf{FUN}\,(v,\ C_1)) \star C_1' & = & \mathsf{FUN}\,(v,\ C_1 \star C_1')
\end{array}$$

(The second clause would read $(\mathsf{ARG}\,(e,\ C_1)) \star C_1' = \mathsf{ARG}\,(e,\ C_1 \star C_1')$ for the contexts of Figure 7.8.)

Then, in Figures 7.7 and 7.8, we could replace the transition

$$\boxed{\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{cont_1}}$$

by the following one:

$$\boxed{\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1' \star C_1,\, v,\, C_2 \rangle_{cont_1}}$$

This replacement changes the control effect of shift to that of Felleisen et al.'s $\mathcal{F}$ operator [87]. Furthermore, the modified abstract machine is in structural correspondence with Felleisen et al.'s abstract machine for $\mathcal{F}$ and # [87, 93].

This representation of control (as a list of 'stack frames') and this implementation of composing delimited continuations (by concatenating these lists) are at the heart of virtually all non-CPS-based accounts of delimited control. However, the modified environment-based abstract machine does not correspond to a defunctionalized continuation-passing evaluator because it is not in the range of defunctionalization [74] since the first-order representation of functions should have a single point of consumption. Here, the constructors of contexts are not solely consumed by the *cont₁* transitions of the abstract machine as in Figures 7.7 and 7.8, but also by $\star$. Therefore, the abstract machine that uses $\star$ is not in the range of Reynolds's defunctionalization and it thus does not immediately correspond to a higher-order, continuation-passing evaluator. In that sense, control operators using $\star$ go beyond CPS.

Elsewhere [35], we have rephrased the modified abstract machine to put it in defunctionalized form, and we have exhibited the corresponding higher-order evaluator and the corresponding 'dynamic' continuation-passing style. This dynamic CPS is not just plain CPS but is a form of continuation+state-passing style where the threaded state is a list of intermediate delimited continuations. Unexpectedly, it is also in structural correspondence with the architecture for delimited control recently proposed by Dybvig, Peyton Jones, and Sabry on other operational grounds [84].

7.4.6 Static vs. dynamic delimited continuations

Irrespectively of any new dynamic CPS and any new architecture for delimited control, there seems to be remarkably few examples that actually illustrate the expressive power of dynamic delimited continuations. We have recently presented one, breadth-first traversal [36], and we present another one below.

The two following functions traverse a given list and return another list. The recursive call to *visit* is abstracted into a delimited continuation, which is applied to the tail of the list:

$$
\begin{array}{lcl}
\mathit{foo}\ \mathit{xs} & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
& & \qquad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\ (x :: \mathit{xs}) \\
& & \qquad = \mathit{visit} \\
& & \qquad\quad (\mathcal{S} k.x :: (k\ \mathit{xs})) \\
& & \mathit{in}\ \langle \mathit{visit}\ \mathit{xs} \rangle
\end{array}
\qquad\qquad
\begin{array}{lcl}
\mathit{bar}\ \mathit{xs} & \stackrel{\text{def}}{=} & \mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
& & \qquad = \mathit{nil} \\
& & \quad \mid \mathit{visit}\ (x :: \mathit{xs}) \\
& & \qquad = \mathit{visit} \\
& & \qquad\quad (\mathcal{F} k.x :: (k\ \mathit{xs})) \\
& & \mathit{in}\ \langle \mathit{visit}\ \mathit{xs} \rangle
\end{array}
$$

On the left, *foo* uses $\mathcal{S}$ and on the right, *bar* uses $\mathcal{F}$; for the rest, the two definitions are identical. Given an input list, *foo* copies it and *bar* reverses it.

To explain this difference and to account for the extended source language, we need to expand the grammar of evaluation contexts, e.g., with a production to account for calls to the list constructor:

$$C_1 ::= [\,] \mid \mathsf{ARG}\,(e,\ C_1) \mid \mathsf{SUCC}(C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{CONS}(v, C_1) \mid \ldots$$

Similarly, we need to expand the definition of concatenation as follows:

$$(\mathsf{CONS}(v, C_1)) \star C_1' \quad = \quad \mathsf{CONS}(v, C_1 \star C_1')$$

Here is a trace of the two computations in the form of the calls to and returns from *visit* for the input list $1 :: 2 :: nil$:

foo: Every time the captured continuation is resumed, its representation is kept separate from the current context. The meta-context therefore grows whereas the captured context solely consists of $\mathsf{FUN}\,(visit,\ [\,])$ throughout (writing *visit* in the context for simplicity):

$$\begin{aligned}
C_2 \ \#\ & C_1[\langle visit\ (1 :: 2 :: nil)\rangle] \\
C_2 \cdot C_1 \ \#\ & [visit\ (1 :: 2 :: nil)] \\
C_2 \cdot C_1 \cdot (\mathsf{CONS}(1, [\,])) \ \#\ & [visit\ (2 :: nil)] \\
C_2 \cdot C_1 \cdot (\mathsf{CONS}(1, [\,])) \cdot (\mathsf{CONS}(2, [\,])) \ \#\ & [visit\ nil] \\
C_2 \cdot C_1 \cdot (\mathsf{CONS}(1, [\,])) \cdot (\mathsf{CONS}(2, [\,])) \ \#\ & [nil] \\
C_2 \cdot C_1 \cdot (\mathsf{CONS}(1, [\,])) \ \#\ & [2 :: nil] \\
C_2 \cdot C_1 \ \#\ & [1 :: 2 :: nil] \\
C_2 \ \#\ & C_1[1 :: 2 :: nil]
\end{aligned}$$

bar: Every time the captured continuation is resumed, its representation is concatenated to the current context. The meta-context therefore remains the same whereas the context changes dynamically. The first captured context is $\mathsf{FUN}\,(visit,\ [\,])$; concatenating it to $\mathsf{CONS}(1, [\,])$ yields $\mathsf{CONS}(1, \mathsf{FUN}\,(visit,\ [\,]))$, which is the second captured context:

$$\begin{aligned}
C_2 \ \#\ & C_1[\langle visit\ (1 :: 2 :: nil)\rangle] \\
C_2 \cdot C_1 \ \#\ & [visit\ (1 :: 2 :: nil)] \\
C_2 \cdot C_1 \ \#\ & (\mathsf{CONS}(1, [\,]))[visit\ (2 :: nil)] \\
C_2 \cdot C_1 \ \#\ & (\mathsf{CONS}(2, \mathsf{CONS}(1, [\,])))[visit\ nil] \\
C_2 \cdot C_1 \ \#\ & (\mathsf{CONS}(2, \mathsf{CONS}(1, [\,])))[nil] \\
C_2 \cdot C_1 \ \#\ & (\mathsf{CONS}(2, [\,]))[1 :: nil] \\
C_2 \cdot C_1 \ \#\ & [2 :: 1 :: nil] \\
C_2 \ \#\ & C_1[2 :: 1 :: nil]
\end{aligned}$$

7.4.7 Summary and conclusion

We have presented the original evaluator for the λ-calculus with shift and reset; this evaluator uses two layers of continuations. From this call-by-value evaluator we have derived two abstract machines, an environment-based one and a substitution-based one; each of these machines uses two layers of evaluation contexts. Based on the substitution-based machine we have constructed a reduction semantics for the λ-calculus with shift and reset; this reduction semantics, by construction, is sound with respect to CPS. Finally, we have pointed out the difference between the static and dynamic delimited control operators at the level of the abstract machine and we have presented a simple but meaningful example illustrating their differing behavior.

- Terms ($1 \leq i \leq n$): $\mathsf{exp} \ni e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle e \rangle_i \mid \mathcal{S}_i k.e$
- Values: $\mathsf{val} \ni v ::= m \mid f$
- Answers, continuations and functions ($1 \leq i \leq n$):

$$\begin{array}{rcll} & \mathsf{ans} & = & \mathsf{val} \\ k_{n+1} \in & \mathsf{cont}_{n+1} & = & \mathsf{val} \rightarrow \mathsf{ans} \\ k_i \in & \mathsf{cont}_i & = & \mathsf{val} \times \mathsf{cont}_{i+1} \times \ldots \times \mathsf{cont}_{n+1} \rightarrow \mathsf{ans} \\ f \in & \mathsf{fun} & = & \mathsf{val} \times \mathsf{cont}_1 \times \ldots \times \mathsf{cont}_{n+1} \rightarrow \mathsf{ans} \end{array}$$

- Initial continuations ($1 \leq i \leq n$):

$$\begin{array}{rcl} \theta_i & = & \lambda(v,\, k_{i+1},\, k_{i+2},\, \ldots,\, k_{n+1}).\, k_{i+1}\ (v,\, k_{i+2},\, \ldots,\, k_{n+1}) \\ \theta_{n+1} & = & \lambda v.\, v \end{array}$$

- Environments: $\mathsf{env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: see Figure 7.10

Figure 7.9: An environment-based evaluator for the CPS hierarchy at level n

7.5 From evaluator to reduction semantics for the CPS hierarchy

We construct a reduction semantics for the call-by-value λ-calculus extended with shift_n and reset_n. As in Section 7.4, we go from an evaluator to an environment-based abstract machine, and from a substitution-based abstract machine to a reduction semantics. Because of the regularity of CPS, the results can be generalized from level 1 to higher levels without repeating the actual construction, based only on the original specification of the hierarchy [67]. In particular, the proofs of the theorems generalize straightforwardly from level 1.

7.5.1 An environment-based evaluator

At the nth level of the hierarchy, the language is extended with operators shift_i and reset_i for all i such that $1 \leq i \leq n$. The evaluator for this language is shown in Figures 7.9 and 7.10. If $n = 1$, it coincides with the evaluator displayed in Figure 7.6.

The evaluator uses $n + 1$ layers of continuations. In the five first clauses (literal, variable, λ-abstraction, function application, and application of the successor function), the continuations $k_2, \ldots, k_{n+1}$ are passive: if the evaluator were curried, they could be eta-reduced. In the clauses defining shift_i and reset_i, the continuations $k_{i+2}, \ldots, k_{n+1}$ are also passive. Each pair of control operators is indexed by the corresponding level in the hierarchy: reset_i is used to "push" each successive continuation up to level i onto level $i + 1$ and to

- Evaluation function ($1 \leq i \leq n$): eval_n : $\mathsf{exp} \times \mathsf{env} \times \mathsf{cont}_1 \times \ldots \times \mathsf{cont}_{n+1} \to \mathsf{ans}$

$$
\begin{array}{rcl}
\mathsf{eval}_n\,(\ulcorner m \urcorner,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\,(m,\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(x,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\,(\rho(x),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(\lambda x.e,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\,(\lambda(v,\, k_1',\, k_2',\, \ldots,\, k_{n+1}').\,\mathsf{eval}_n\,(e,\, \rho\{x \mapsto v\},\, k_1',\, k_2',\, \ldots,\, k_{n+1}'),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(e_0\,e_1,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\,(e_0,\, \rho, \\
 & & \quad \lambda(f,\, k_2',\, \ldots,\, k_{n+1}').\,\mathsf{eval}_n\,(e_1,\, \rho, \\
 & & \quad\quad \lambda(v,\, k_2'',\, \ldots,\, k_{n+1}'').\,f\,(v,\, k_1,\, k_2'',\, \ldots,\, k_{n+1}''), \\
 & & \quad\quad k_2',\, \ldots,\, k_{n+1}'), \\
 & & \quad k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(\mathit{succ}\ e,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\,(e,\, \rho,\, \lambda(m,\, k_2',\, \ldots,\, k_{n+1}').\,k_1\,(m+1,\, k_2',\, \ldots,\, k_{n+1}'),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(\langle e \rangle_i,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\,(e,\, \rho,\, \theta_1,\, \ldots,\, \theta_i,\, \lambda(v,\, k_{i+2}',\, \ldots,\, k_{n+1}').\,k_1\,(v,\, k_2,\, \ldots,\, k_{i+1},\, k_{i+2}',\, \ldots,\, k_{n+1}'),\, k_{i+2},\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\,(\mathcal{S}_i k.e,\, \rho,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\,(e,\, \rho\{k \mapsto c_i\},\, \theta_1,\, \ldots,\, \theta_i,\, k_{i+1},\, \ldots,\, k_{n+1})
\end{array}
$$

where $c_i = \lambda(v,\, k_1',\, \ldots,\, k_{n+1}').\,k_1\,(v,\, k_2,\, \ldots,\, k_i,\, \lambda(v',\, k_{i+2}'',\, \ldots,\, k_{n+1}'').\,k_1'\,(v',\, k_2',\, \ldots,\, k_{i+1}',\, k_{i+2}'',\, \ldots,\, k_{n+1}''),\, k_{i+2}',\, \ldots,\, k_{n+1}')$

- Main function: $\mathsf{evaluate}_n$: $\mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}_n\,(e) \quad = \quad \mathsf{eval}_n\,(e,\, \rho_{mt},\, \theta_1,\, \ldots,\, \theta_n,\, \theta_{n+1})$$

Figure 7.10: An environment-based evaluator for the CPS hierarchy at level n, ctd.

- Terms $(1 \leq i \leq n)$: $e ::= \ulcorner m \urcorner \mid x \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle e \rangle_i \mid \mathcal{S}_i k.e$
- Values $(1 \leq i \leq n)$: $v ::= m \mid [x,\, e,\, \rho] \mid C_i$
- Evaluation contexts $(2 \leq i \leq n+1)$:

$$C_1 ::= [\,] \mid \mathsf{ARG}\,((e, \rho),\ C_1) \mid \mathsf{SUCC}(C_1) \mid \mathsf{FUN}\,(v,\ C_1)$$
$$C_i ::= [\,] \mid C_i \cdot C_{i-1}$$

- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Initial transition, transition rules, and final transition: see Figure 7.12

Figure 7.11: An environment-based abstract machine for the CPS hierarchy at level n

reinitialize them with $\theta_1, \ldots, \theta_i$, which are the successive CPS counterparts of the identity function; shift_i is used to abstract control up to level i into a delimited continuation and to reinitialize the successive continuations up to level i with $\theta_1, \ldots, \theta_i$.

Applying a delimited continuation that was abstracted up to level i "pushes" each successive continuation up to level i onto level $i+1$ and restores the successive continuations that were captured in a delimited continuation. When such a delimited continuation completes, and when an expression delimited by reset_i completes, the successive continuations that were pushed onto level $i+1$ are "popped" back into place and the computation proceeds.

7.5.2 An environment-based abstract machine

Defunctionalizing the evaluator of Figures 7.9 and 7.10 yields the environment-based abstract machine displayed in Figures 7.11 and 7.12. If $n = 1$, it coincides with the abstract machine displayed in Figure 7.7.

The abstract machine consists of $n+2$ sets of transitions: *eval* for interpreting terms and $\mathit{cont}_1, \ldots, \mathit{cont}_{n+1}$ for interpreting the successive defunctionalized continuations. The set of possible values includes integers, closures and captured contexts.

This abstract machine is an extension of the abstract machine displayed in Figure 7.7 with $n+1$ contexts instead of 2 and the corresponding transitions for shift_i and reset_i. Each meta_{i+1}-context intervenes to process reset_i expressions and to apply captured continuations. Otherwise, the successive contexts are passively threaded to process literals, variables, λ-abstractions, function applications, and applications of the successor function.

Given an environment ρ and a series of successive contexts, a reset_i expression $\langle e \rangle_i$ is processed by evaluating e with the same environment ρ, i empty contexts, and a meta_{i+1}-context over which all the intermediate contexts have been pushed on.

- Initial transition, transition rules, and final transition ($1 \leq i \leq n,\ 2 \leq j \leq n$):

e	$\Rightarrow$	$\langle e,\, \rho_{mt},\, [\,],\, [\,],\, \ldots,\, [\,] \rangle_{eval}$
$\langle \ulcorner m \urcorner,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, m,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle x,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \rho\,(x),\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\ C_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle succ\ e,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{SUCC}(k_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \langle e \rangle_i,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, [\,],\, \ldots,\, [\,],\, C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathcal{S}_i k.e,\, \rho,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{k \mapsto C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)\},\, [\,],\, \ldots,\, [\,],\, C_{i+1},\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle [\,],\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v,\, C_3,\, \ldots,\, C_{n+1} \rangle_{cont_2}$
$\langle \mathsf{ARG}\,((e, \rho),\ k_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathsf{SUCC}(C_1),\, m,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C_1,\, m+1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ k_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathsf{FUN}\,(C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots),\ C_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C'_1,\, v,\, C'_2,\, \ldots,\, C'_i,\, C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle [\,],\, v,\, C_{j+1},\, \ldots,\, C_{n+1} \rangle_{cont_j}$	$\Rightarrow$	$\langle C_{j+1},\, v,\, C_{j+2},\, \ldots,\, C_{n+1} \rangle_{cont_{j+1}}$
$\langle C_j \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v,\, C_{j+1},\, \ldots,\, C_{n+1} \rangle_{cont_j}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle C_{n+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v \rangle_{cont_{n+1}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle [\,],\, v \rangle_{cont_{n+1}}$	$\Rightarrow$	v

Figure 7.12: An environment-based abstract machine for the CPS hierarchy at level n, ctd.

- Terms and values ($1 \leq i \leq n$): $e ::= v \mid x \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle e \rangle_i \mid \mathcal{S}_i k.e$
  $v ::= \ulcorner m \urcorner \mid \lambda x.e \mid C_i$
- Evaluation contexts ($2 \leq i \leq n+1$):

$$C_1 ::= [\,] \mid \mathsf{ARG}\,(e,\ C_1) \mid \mathsf{SUCC}(C_1) \mid \mathsf{FUN}\,(v,\ C_1)$$
$$C_i ::= [\,] \mid C_i \cdot C_{i-1}$$

- Initial transition, transition rules, and final transition: see Figure 7.14

Figure 7.13: A substitution-based abstract machine for the CPS hierarchy at level n

Given an environment ρ and a series of successive contexts, a shift expression $\mathcal{S}_i k.e$ is processed by evaluating e with an extension of ρ where k denotes a composition of the i surrounding contexts, i empty contexts, and the remaining outer contexts. Applying a captured context is achieved by pushing all the current contexts on the next outer context, restoring the composition of the captured contexts, and continuing with them. Resuming a composition of captured contexts is achieved by popping them off the next outer context and continuing with them.

In order to relate the resulting abstract machine to the evaluator, we define a partial function $\mathsf{eval}^{\mathsf{e}}_n$ mapping a term e to a value v whenever the machine for level n, started with e, stops with v. The correctness of the machine with respect to the evaluator is ensured by the following theorem:

Theorem 7.4 *For any program e and any integer value m, $\mathsf{evaluate}_n\,(e) = m$ if and only if $\mathsf{eval}^{\mathsf{e}}_n\,(e) = m$.*

7.5.3 A substitution-based abstract machine

In the same fashion as in Section 7.4.3, we construct the substitution-based abstract machine corresponding to the environment-based abstract machine of Section 7.5.2. The result is displayed in Figures 7.13 and 7.14. If $n = 1$, it coincides with the abstract machine displayed in Figure 7.8.

The nth level contains $n+1$ evaluation contexts and each context C_i can be viewed as a stack of non-empty contexts C_{i-1}. Terms are decomposed as

$$C_{n+1}\ \#_n\ C_n\ \#_{n-1}\ C_{n-1}\ \#_{n-2}\ \cdots\ \#_2\ C_2\ \#_1\ C_1[e],$$

where each $\#_i$ represents a context delimiter of level i. All the control operators that occur at the jth level (with $j < n$) of the hierarchy do not use the contexts $j+2, \ldots, n+1$. The functions *decompose* and its inverse *plug* can be read off the machine, as for level 1.

The transitions of the machine for level j are "embedded" in the machine for level $j+1$; the extra components are threaded but not used.

- Initial transition, transition rules, and final transition ($1 \leq i \leq n,\ 2 \leq j \leq n$):

e	$\Rightarrow$	$\langle e,\, [\,],\, [\,],\, \ldots,\, [\,] \rangle_{eval}$
$\langle \ulcorner m \urcorner,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle \lambda x.e,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \lambda x.e,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle C'_i,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, C'_i,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle e_0\, e_1,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \mathsf{ARG}\,((e_1,\, \rho),\, C_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle succ\ e,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e,\, \mathsf{SUCC}(C_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \langle\!\langle e \rangle\!\rangle_i,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e,\, [\,],\, \ldots,\, [\,],\, C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathcal{S}_i k.e,\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$	$\Rightarrow$	$\langle e\{C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)/k\},\, [\,],\, \ldots,\, [\,],\, C_{i+1},\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle [\,],\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v,\, C_3,\, \ldots,\, C_{n+1} \rangle_{cont_2}$
$\langle \mathsf{ARG}\,(e,\ k_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \mathsf{FUN}\,(v,\ C_1),\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathsf{SUCC}(C_1),\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C_1,\, \ulcorner m+1 \urcorner,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle \mathsf{FUN}\,((\lambda x.e),\ k_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle e\{v/x\},\, C_1,\, C_2,\, \ldots,\, C_{n+1} \rangle_{eval}$
$\langle \mathsf{FUN}\,(C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots),\ C_1),\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$	$\Rightarrow$	$\langle C'_1,\, v,\, C'_2,\, \ldots,\, C'_i,\, C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle [\,],\, v,\, C_{j+1},\, \ldots,\, C_{n+1} \rangle_{cont_j}$	$\Rightarrow$	$\langle C_{j+1},\, v,\, C_{j+2},\, \ldots,\, C_{n+1} \rangle_{cont_{j+1}}$
$\langle C_j \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v,\, C_{j+1},\, \ldots,\, C_{n+1} \rangle_{cont_j}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle C_{n+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v \rangle_{cont_{n+1}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1} \rangle_{cont_1}$
$\langle [\,],\, v \rangle_{cont_{n+1}}$	$\Rightarrow$	v

Figure 7.14: A substitution-based abstract machine for the the CPS hierarchy at level n, ctd.

We define a partial function $\mathsf{eval}^{\mathsf{s}}_n$ capturing the evaluation by the substitution-based abstract machine for an arbitrary level n, analogously to the definition of $\mathsf{eval}^{\mathsf{e}}_n$. Now we can relate evaluation with the environment-based and the substitution-based abstract machines for level n.

Theorem 7.5 *For any program e, either both $\mathsf{eval}^{\mathsf{s}}_n(e)$ and $\mathsf{eval}^{\mathsf{e}}_n(e)$ are undefined, or there exist values v, v' such that $\mathsf{eval}^{\mathsf{s}}_n(e) = v$, $\mathsf{eval}^{\mathsf{e}}_n(e) = v'$ and $\mathcal{I}_n(v') = v$.*

The definition of $\mathcal{I}_n$ extends that of $\mathcal{I}$ from Theorem 7.2 in such a way that it is homomorphic for all the contexts C_i, with $2 \leq i \leq n$.

7.5.4 A reduction semantics

Along the same lines as in Section 7.4.4, we construct the reduction semantics for the CPS hierarchy based on the abstract machine of Figures 7.13 and 7.14. For an arbitrary level n we obtain the following set of reduction rules, for all $1 \leq i \leq n$; they define the actual redexes:

$$(\delta) \qquad C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[\mathit{succ}\ \ulcorner m \urcorner] \to_n C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[\ulcorner m+1 \urcorner]$$

$$(\beta_\lambda) \qquad C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[(\lambda x.e)\, v] \to_n C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[e\{v/x\}]$$

$$(\mathcal{S}^i_\lambda) \qquad \begin{array}{l} C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[\mathcal{S}_i k.e] \to_n \\ C_{n+1}\ \#_n\ \cdots\ \#_{i+1}\ C_{i+1}\ \#_i\ [\,] \ldots\ \#_1\ [e\{C_i \cdot (\ldots (C_2 \cdot C_1)\ldots)/k\}] \end{array}$$

$$(\beta^i_{ctx}) \qquad \begin{array}{l} C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[C'_i \cdot (\ldots (C'_2 \cdot C'_1)\ldots)\, v] \to_n \\ C_{n+1}\ \#_n\ \cdots\ \#_{i+1}\ C_{i+1} \cdot (\ldots (C_2 \cdot C_1)\ldots)\ \#_i\ C'_i\ \#_{i-1}\ \cdots\ \#_1\ C'_1[v] \end{array}$$

$$(\mathrm{Reset}^i) \qquad C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[\langle v \rangle_i] \to_n C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[v]$$

Each level contains all the reductions from lower levels, and these reductions are compatible with additional layers of evaluation contexts. In particular, at level 0 there are only δ- and β_λ-reductions.

The values and evaluation contexts are already specified in the abstract machine. Moreover, the potential redexes are defined according to the following grammar:

$$p_n ::= \mathit{succ}\ v \mid v_0\, v_1 \mid \mathcal{S}_i k.e \mid \langle v \rangle_i \quad (1 \leq i \leq n)$$

Lemma 7.2 (Unique decomposition for level n) *A program e is either a value or there exists a unique sequence of contexts $C_1, \ldots, C_{n+1}$ and a potential redex p_n such that $e = \mathit{plug}\,(C_{n+1}\ \#_n\ \cdots\ \#_1\ C_1[p_n])$.*

Evaluating a term using either the derived reduction rules or the substitution-based abstract machine from Section 7.5.3 yields the same result:

Theorem 7.6 *For any program e and any value v, $\mathsf{eval}^{\mathsf{s}}_n(e) = v$ if and only if $e \to^*_n v$, where $\to^*_n$ is the reflexive, transitive closure of $\to_n$.*

As in Section 7.4.4, using refocusing, one can go from a given reduction semantics of Section 7.5.4 into a pre-abstract machine and the corresponding eval/apply abstract machine of Figures 7.13 and 7.14.

7.5.5 Beyond CPS

As in Section 7.4.5, one can define a family of concatenation functions over contexts and use it to implement composable continuations in the CPS hierarchy, giving rise to a family of control operators $\mathcal{F}_n$ and $\#_n$. Again the modified environment-based abstract machine does not immediately correspond to a defunctionalized continuation-passing evaluator. Such control operators go beyond traditional CPS.

7.5.6 Static vs. dynamic delimited continuations

As in Section 7.4.6, one can illustrate the difference between static and dynamic delimited continuations in the CPS hierarchy. For example, replacing shift_2 and reset_2 respectively by $\mathcal{F}_2$ and $\#_2$ in Danvy and Filinski's version of Abelson and Sussman's generator of triples [67, Section 3] yields a list in reverse order.[4]

7.5.7 Summary and conclusion

We have generalized the results presented in Section 7.4 from level 1 to the whole CPS hierarchy of control operators shift_n and reset_n. Starting from the original evaluator for the λ-calculus with shift_n and reset_n that uses $n+1$ layers of continuations, we have derived two abstract machines, an environment-based one and a substitution-based one; each of these machines use $n+1$ layers of evaluation contexts. Based on the substitution-based machine we have obtained a reduction semantics for the λ-calculus extended with shift_n and reset_n which, by construction, is sound with respect to CPS.

7.6 Programming in the CPS hierarchy

To finish, we present new examples of programming in the CPS hierarchy. The examples are normalization functions. In Sections 7.6.1 and 7.6.2, we first describe normalization by evaluation and we present the simple example of the free monoid. In Section 7.6.3, we present a function mapping a proposition into its disjunctive normal form; this normalization function uses delimited continuations. In Section 7.6.4, we generalize the normalization functions of Sections 7.6.2 and 7.6.3 to a hierarchical language of units and products, and we express the corresponding normalization function in the CPS hierarchy.

7.6.1 Normalization by evaluation

Normalization by evaluation is a 'reduction-free' approach to normalizing terms. Instead of reducing a term to its normal form, one evaluates this term into a

[4] Thanks are due to an anonymous reviewer for pointing this out.

non-standard model and reifies its denotation into its normal form [82]:

$$\begin{aligned} \mathit{eval} &: \mathit{term} \to \mathit{value} \\ \mathit{reify} &: \mathit{value} \to \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} &: \mathit{term} \to \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} &= \mathit{reify} \circ \mathit{eval} \end{aligned}$$

Normalization by evaluation has been developed in intuitionistic type theory [48, 163], proof theory [24, 25], category theory [9], and partial evaluation [60, 61], where it has emerged as a new field of application for delimited continuations [17, 61, 82, 99, 110, 123, 228].

7.6.2 The free monoid

A source term in the free monoid is either a variable, the unit element, or the product of two terms:

$$\mathit{term} \ni t ::= x \mid \varepsilon \mid t \star t'$$

The product is associative and the unit element is neutral. These properties justify the following conversion rules:

$$\begin{aligned} t \star (t' \star t'') &\leftrightarrow (t \star t') \star t'' \\ t \star \varepsilon &\leftrightarrow t \\ \varepsilon \star t &\leftrightarrow t \end{aligned}$$

We aim (for example) for list-like flat normal forms:

$$\mathit{term}^{\mathrm{nf}} \ni \widehat{t} ::= \varepsilon^{\mathrm{nf}} \mid x \star^{\mathrm{nf}} \widehat{t}$$

In a reduction-based approach to normalization, one would orient the conversion rules into reduction rules and one would apply these reduction rules until a normal form is obtained:

$$\begin{aligned} t \star (t' \star t'') &\leftarrow (t \star t') \star t'' \\ \varepsilon \star t &\to t \end{aligned}$$

In a reduction-free approach to normalization, one defines a normalization function as the composition of a non-standard evaluation function and a reification function. Let us state such a normalization function.

The non-standard domain of values is the transformer

$$\mathit{value} = \mathit{term}^{\mathrm{nf}} \to \mathit{term}^{\mathrm{nf}}.$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{aligned} \mathit{eval}\, x &= \lambda t.x \star^{\mathrm{nf}} t \\ \mathit{eval}\, \varepsilon &= \lambda t.t \\ \mathit{eval}\, (t \star t') &= (\mathit{eval}\, t) \circ (\mathit{eval}\, t') \\ \mathit{reify}\, v &= v\, \varepsilon^{\mathrm{nf}} \\ \mathit{normalize}\, t &= \mathit{reify}\, (\mathit{eval}\, t) \end{aligned}$$

In effect, *eval* is a homomorphism from the source monoid to the monoid of transformers (unit is mapped to unit and products are mapped to products) and the normalization function hinges on the built-in associativity of function composition. Beylin, Dybjer, Coquand, and Kinoshita have studied its theoretical content [26, 48, 145]. From a (functional) programming standpoint, the reduction-based approach amounts to flattening a tree iteratively by reordering it, and the reduction-free approach amounts to flattening a tree with an accumulator.

7.6.3 A language of propositions

A source term, i.e., a proposition, is either a variable, a literal (true or false), a conjunction, or a disjunction:

$$term \ni t ::= x \mid true \mid t \wedge t' \mid false \mid t \vee t'$$

Conjunction and disjunction are associative and distribute over each other; *true* is neutral for conjunction and absorbant for disjunction; and *false* is neutral for disjunction and absorbant for conjunction.

We aim (for example) for list-like disjunctive normal forms:

$$\begin{array}{rcl} term^{\mathrm{nf}} & \ni & \widehat{t} ::= d \\ term_{\mathrm{d}}^{\mathrm{nf}} & \ni & d ::= false^{\mathrm{nf}} \mid c \vee^{\mathrm{nf}} d \\ term_{\mathrm{c}}^{\mathrm{nf}} & \ni & c ::= true^{\mathrm{nf}} \mid x \wedge^{\mathrm{nf}} c \end{array}$$

Our normalization function is the result of composing a non-standard evaluation function and a reification function. We state them below without proof.

Given the domains of transformers

$$\begin{array}{rcl} F_1 & = & term_{\mathrm{c}}^{\mathrm{nf}} \to term_{\mathrm{c}}^{\mathrm{nf}} \\ F_2 & = & term_{\mathrm{d}}^{\mathrm{nf}} \to term_{\mathrm{d}}^{\mathrm{nf}} \end{array}$$

the non-standard domain of values is ans_1, where

$$\begin{array}{rcl} ans_2 & = & F_2 \\ ans_1 & = & (F_1 \to ans_2) \to ans_2. \end{array}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{array}{rcl} eval_0\ x\ k\ d & = & k\,(\lambda c.x \wedge^{\mathrm{nf}} c)\,d \\ eval_0\ true\ k\ d & = & k\,(\lambda c.c)\,d \\ eval_0\,(t \wedge t')\,k\,d & = & eval_0\ t\,(\lambda f_1.eval_0\ t'\,(\lambda f_1'.k\,(f_1 \circ f_1')))\,d \\ eval_0\ false\ k\ d & = & d \\ eval_0\,(t \vee t')\,k\,d & = & eval_0\ t\ k\,(eval_0\ t'\ k\ d) \end{array}$$

$$reify_0\ v = v\,(\lambda f_1.\lambda d.(f_1\ true^{\mathrm{nf}}) \vee^{\mathrm{nf}} d)\ false^{\mathrm{nf}}$$

$$normalize\ t = reify_0\,(eval_0\ t)$$

This normalization function uses a continuation k, an accumulator d to flatten disjunctions, and another one c to flatten conjunctions. The continuation is

delimited: the three first clauses of $eval_0$ are in CPS; in the fourth, k is discarded (accounting for the fact that *false* is absorbant for conjunction); and in the last, k is duplicated and used in non-tail position (achieving the distribution of conjunctions over disjunctions). The continuation and the accumulators are initialized in the definition of $reify_0$.

Uncurrying the continuation and mapping $eval_0$ and $reify_0$ back to direct style yield the following definition, which lives at level 1 of the CPS hierarchy:

$$\begin{array}{rcl}
eval_1\ x\ d & = & (\lambda c.x \wedge^{\mathrm{nf}} c,\ d) \\
eval_1\ true\ d & = & (\lambda c.c,\ d) \\
eval_1\ (t \wedge t')\ d & = & let\ (f_1,\ d) = eval_1\ t\ d \\
 & & \quad in\ let\ (f_1',\ d) = eval_1\ t'\ d \\
 & & \quad\quad in\ (f_1 \circ f_1',\ d) \\
eval_1\ false\ d & = & \mathcal{S}k.d \\
eval_1\ (t \vee t')\ d & = & \mathcal{S}k.k\,(eval_1\ t\ \langle k\,(eval_1\ t'\ d)\rangle) \\
 & & \\
reify_1\ v & = & \langle let\ (f_1,\ d) = v\ false^{\mathrm{nf}} \\
 & & \quad in\ (f_1\ true^{\mathrm{nf}}) \vee^{\mathrm{nf}} d\rangle \\
normalize\ t & = & reify_1\ (eval_1\ t)
\end{array}$$

The three first clauses of $eval_1$ are in direct style; the two others abstract control with shift. In the fourth clause, the context is discarded; and in the last clause, the context is duplicated and composed. The context and the accumulators are initialized in the definition of $reify_1$.

This direct-style version makes it even more clear than the CPS version that the accumulator for the disjunctions in normal form is a threaded state. A continuation-based, state-based version (or better, a monad-based one) can therefore be written—but it is out of scope here.

7.6.4 A hierarchical language of units and products

We consider a generalization of propositional logic where a source term is either a variable, a unit in a hierarchy of units, or a product in a hierarchy of products:

$$\begin{array}{rcl}
term & \ni & t ::= x \mid \varepsilon_i \mid t \star_i t' \\
 & & \quad where\ 1 \leq i \leq n.
\end{array}$$

All the products are associative. All units are neutral for products with the same index.

The free monoid: The language corresponds to that of the free monoid if $n = 1$, as in Section 7.6.2.

Boolean logic: The language corresponds to that of propositions if $n = 2$, as in Section 7.6.3: ε_1 is *true*, $\star_1$ is $\wedge$, ε_2 is *false*, and $\star_2$ is $\vee$.

Multi-valued logic: In general, for each $n > 2$ we can consider a suitable n-valued logic [107]; for example, in case $n = 4$, the language corresponds to that of Belnap's bilattice $\mathcal{FOUR}$ [22]. It is also possible to modify the normalization function to work for less regular logical structures (e.g., other bilattices).

Monads: In general, the language corresponds to that of layered monads [172]: each unit element is the unit of the corresponding monad, and each product is the 'bind' of the corresponding monad. In practice, layered monads are collapsed into one for programming consumption [98], but prior to this collapse, all the individual monad operations coexist in the computational soup.

In the remainder of this section, we assume that all the products, besides being associative, distribute over each other, and that all units, besides being neutral for products with the same index, are absorbant for products with other indices. We aim (for example) for a generalization of disjunctive normal forms:

$$\begin{array}{lcl} term^{\mathrm{nf}} & \ni & \widehat{t} \ ::= t_n \\ term_n^{\mathrm{nf}} & \ni & t_n ::= \varepsilon_n^{\mathrm{nf}} \mid t_{n-1} \star_n^{\mathrm{nf}} t_n \\ & & \vdots \\ term_1^{\mathrm{nf}} & \ni & t_1 ::= \varepsilon_1^{\mathrm{nf}} \mid t_0 \star_1^{\mathrm{nf}} t_1 \\ term_0^{\mathrm{nf}} & \ni & t_0 ::= x \end{array}$$

For presentational reasons, in the remainder of this section we arbitrarily fix n to be 5.

Our normalization function is the result of composing a non-standard evaluation function and a reification function. We state them below without proof. Given the domains of transformers

$$\begin{array}{lcl} F_1 & = & term_1^{\mathrm{nf}} \ \to \ term_1^{\mathrm{nf}} \\ F_2 & = & term_2^{\mathrm{nf}} \ \to \ term_2^{\mathrm{nf}} \\ F_3 & = & term_3^{\mathrm{nf}} \ \to \ term_3^{\mathrm{nf}} \\ F_4 & = & term_4^{\mathrm{nf}} \ \to \ term_4^{\mathrm{nf}} \\ F_5 & = & term_5^{\mathrm{nf}} \ \to \ term_5^{\mathrm{nf}} \end{array}$$

the non-standard domain of values is ans_1, where

$$\begin{array}{lcl} ans_5 & = & F_5 \\ ans_4 & = & (F_4 \ \to \ ans_5) \ \to \ ans_5 \\ ans_3 & = & (F_3 \ \to \ ans_4) \ \to \ ans_4 \\ ans_2 & = & (F_2 \ \to \ ans_3) \ \to \ ans_3 \\ ans_1 & = & (F_1 \ \to \ ans_2) \ \to \ ans_2. \end{array}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{array}{rcl} eval_0 \, x \, k_1 \, k_2 \, k_3 \, k_4 \, t_5 & = & k_1 \, (\lambda t_1 . x \star_1^{\mathrm{nf}} t_1) \, k_2 \, k_3 \, k_4 \, t_5 \\ eval_0 \, \varepsilon_1 \, k_1 \, k_2 \, k_3 \, k_4 \, t_5 & = & k_1 \, (\lambda t_1 . t_1) \, k_2 \, k_3 \, k_4 \, t_5 \\ eval_0 \, (t \star_1 t') \, k_1 \, k_2 \, k_3 \, k_4 \, t_5 & = & eval_0 \, t \, (\lambda f_1 . eval_0 \, t' \, (\lambda f_1' . k_1 \, (f_1 \circ f_1'))) \, k_2 \, k_3 \, k_4 \, t_5 \\ eval_0 \, \varepsilon_2 \, k_1 \, k_2 \, k_3 \, k_4 \, t_5 & = & k_2 \, (\lambda t_2 . t_2) \, k_3 \, k_4 \, t_5 \\ eval_0 \, (t \star_2 t') \, k_1 \, k_2 \, k_3 \, k_4 \, t_5 & = & eval_0 \, t \, k_1 \\ & & (\lambda f_2 . eval_0 \, t' \, k_1 \, (\lambda f_2' . k_2 \, (f_2 \circ f_2'))) k_3 k_4 t_5 \end{array}$$

$$
\begin{array}{rcl}
eval_0\,\varepsilon_3\,k_1\,k_2\,k_3\,k_4\,t_5 & = & k_3\,(\lambda t_3.t_3)\,k_4\,t_5 \\
eval_0\,(t \star_3 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & eval_0\,t\,k_1\,k_2 \\
 & & \quad (\lambda f_3.eval_0\,t'\,k_1\,k_2\,(\lambda f_3'.k_3\,(f_3 \circ f_3')))k_4 t_5 \\
eval_0\,\varepsilon_4\,k_1\,k_2\,k_3\,k_4\,t_5 & = & k_4\,(\lambda t_4.t_4)\,t_5 \\
eval_0\,(t \star_4 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & eval_0\,t\,k_1\,k_2 k_3 \\
 & & \quad (\lambda f_4.eval_0\,t'\,k_1\,k_2\,k_3\,(\lambda f_4'.k_4\,(f_4 \circ f_4')))t_5 \\
eval_0\,\varepsilon_5\,k_1\,k_2\,k_3\,k_4\,t_5 & = & t_5 \\
eval_0\,(t \star_5 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & eval_0\,t\,k_1\,k_2\,k_3\,k_4\,(eval_0\,t'\,k_1\,k_2\,k_3\,k_4\,t_5)
\end{array}
$$

$$
\begin{array}{rcl}
reify_0\,v & = & v\,(\lambda f_1.\lambda k_2.k_2\,(\lambda t_2.(f_1\,\varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2)) \\
 & & \quad (\lambda f_2.\lambda k_3.k_3\,(\lambda t_3.(f_2\,\varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3)) \\
 & & \quad (\lambda f_3.\lambda k_4.k_4\,(\lambda t_4.(f_3\,\varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4)) \\
 & & \quad (\lambda f_4.\lambda t_5.(f_4\,\varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5) \\
 & & \quad \varepsilon_5
\end{array}
$$

$$
normalize\,t \;=\; reify_0\,(eval_0\,t)
$$

This normalization function uses four delimited continuations k_1, k_2, k_3, k_4 and five accumulators t_1, t_2, t_3, t_4, t_5 to flatten each of the successive products. In the clause of each ε_i, the continuations $k_1, \ldots, k_{i-1}$ are discarded, accounting for the fact that ε_i is absorbant for $\star_1, \ldots, \star_{i-1}$, and the identity function is passed to k_i, accounting for the fact that ε_i is neutral for $\star_i$. In the clause of each $\star_{i+1}$, the continuations $k_1, \ldots, k_i$ are duplicated and used in non-tail position, achieving the distribution of $\star_{i+1}$ over $\star_1, \ldots, \star_i$. The continuations and the accumulators are initialized in the definition of $reify_0$.

This normalization function lives at level 0 of the CPS hierarchy, but we can express it at a higher level using shift and reset. For example, uncurrying k_3 and k_4 and mapping $eval_0$ and $reify_0$ back to direct style twice yield the following intermediate definition, which lives at level 2:

$$
\begin{array}{rcl}
eval_2\,x\,k_1\,k_2\,t_5 & = & k_1\,(\lambda t_1.x \star_1^{\mathrm{nf}} t_1)\,k_2\,t_5 \\
eval_2\,\varepsilon_1\,k_1\,k_2\,t_5 & = & k_1\,(\lambda t_1.t_1)\,k_2\,t_5 \\
eval_2\,(t \star_1 t')\,k_1\,k_2\,t_5 & = & eval_2\,t\,(\lambda f_1.eval_2\,t'\,(\lambda f_1'.k_1\,(f_1 \circ f_1')))\,k_2\,t_5 \\
eval_2\,\varepsilon_2\,k_1\,k_2\,t_5 & = & k_2\,(\lambda t_2.t_2)\,t_5 \\
eval_2\,(t \star_2 t')\,k_1\,k_2\,t_5 & = & eval_2\,t\,k_1\,(\lambda f_2.eval_2\,t'\,k_1\,(\lambda f_2'.k_2\,(f_2 \circ f_2')))\,t_5 \\
eval_2\,\varepsilon_3\,k_1\,k_2\,t_5 & = & (\lambda t_3.t_3,\, t_5) \\
eval_2\,(t \star_3 t')\,k_1\,k_2\,t_5 & = & let\ (f_3,\, t_5) = eval_2\,t\,k_1\,k_2\,t_5 \\
 & & \quad in\ let\ (f_3',\, t_5) = eval_2\,t'\,k_1\,k_2\,t_5 \\
 & & \quad\quad in\ (f_3 \circ f_3',\, t_5) \\
eval_2\,\varepsilon_4\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.(\lambda t_4.t_4,\, t_5) \\
eval_2\,(t \star_4 t')\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.let\ (f_4,\, t_5) = \langle k_3\,(eval_2\,t\,k_1\,k_2\,t_5)\rangle_1 \\
 & & \quad\quad in\ let\ (f_4',\, t_5) = \langle k_3\,(eval_2\,t'\,k_1\,k_2\,t_5)\rangle_1 \\
 & & \quad\quad\quad in\ (f_4 \circ f_4',\, t_5) \\
eval_2\,\varepsilon_5\,k_1\,k_2\,t_5 & = & \mathcal{S}_2 k_4.t_5 \\
eval_2\,(t \star_5 t')\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.\mathcal{S}_2 k_4.let\ t_5 = \langle k_4\,\langle k_3\,(eval_2\,t'\,k_1\,k_2\,t_5)\rangle_1\rangle_2 \\
 & & \quad\quad\quad in\ \langle k_4\,\langle k_3\,(eval_2\,t\,k_1\,k_2\,t_5)\rangle_1\rangle_2
\end{array}
$$

$$
\begin{aligned}
\mathit{reify}_2\, v \;=\; & \langle \mathit{let}\ (f_4,\, t_5) = \langle \mathit{let}\ (f_3,\, t_5) \\
& \qquad = v\, (\lambda f_1.\lambda k_2.k_2\, (\lambda t_2.(f_1\, \varepsilon_1^{\mathrm{nf}})\, \star_2^{\mathrm{nf}}\, t_2)) \\
& \qquad\quad (\lambda f_2.\lambda t_3.(f_2\, \varepsilon_2^{\mathrm{nf}})\, \star_3^{\mathrm{nf}}\, t_3) \\
& \qquad\quad \varepsilon_5 \\
& \qquad \mathit{in}\ (\lambda f_4.(f_3\, \varepsilon_3^{\mathrm{nf}})\, \star_4^{\mathrm{nf}}\, t_4,\, t_5)\rangle_1 \\
& \mathit{in}\ (f_4\, \varepsilon_4^{\mathrm{nf}})\, \star_5^{\mathrm{nf}}\, t_5\rangle_2
\end{aligned}
$$

$$
\mathit{normalize}\, t \;=\; \mathit{reify}_2\, (\mathit{eval}_2\, t)
$$

Whereas eval_0 had four layered continuations, eval_2 has only two layered continuations since it has been mapped back to direct style twice. Where eval_0 accesses k_3 as one of its parameters, eval_2 abstracts the first layer of control with shift_1, and where eval_0 accesses k_4 as one of its parameters, eval_2 abstracts the first and the second layer of control with shift_2.

Uncurrying k_1 and k_2 and mapping eval_2 and reify_2 back to direct style twice yield the following direct-style definition, which lives at level 4 of the CPS hierarchy:

$$
\begin{aligned}
\mathit{eval}_4\, x\, t_5 \;=\; & (\lambda t_1.x\, \star_1^{\mathrm{nf}}\, t_1,\, t_5) \\
\mathit{eval}_4\, \varepsilon_1\, t_5 \;=\; & (\lambda t_1.t_1,\, t_5) \\
\mathit{eval}_4\, (t \star_1 t')\, t_5 \;=\; & \mathit{let}\ (f_1,\, t_5) = \mathit{eval}_4\, t\, t_5 \\
& \mathit{in}\ \mathit{let}\ (f_1',\, t_5) = \mathit{eval}_4\, t'\, t_5 \\
& \quad \mathit{in}\ (f_1 \circ f_1',\, t_5) \\
\mathit{eval}_4\, \varepsilon_2\, t_5 \;=\; & \mathcal{S}_1 k_1.(\lambda t_2.t_2,\, t_5) \\
\mathit{eval}_4\, (t \star_2 t')\, t_5 \;=\; & \mathcal{S}_1 k_1.\mathit{let}\ (f_2,\, t_5) = \langle k_1\, (\mathit{eval}_4\, t\, t_5)\rangle_1 \\
& \quad \mathit{in}\ \mathit{let}\ (f_2',\, t_5) = \langle k_1\, (\mathit{eval}_4\, t'\, t_5)\rangle_1 \\
& \quad\quad \mathit{in}\ (f_2 \circ f_2',\, t_5) \\
\mathit{eval}_4\, \varepsilon_3\, t_5 \;=\; & \mathcal{S}_2 k_2.(\lambda t_3.t_3,\, t_5) \\
\mathit{eval}_4\, (t \star_3 t')\, t_5 \;=\; & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathit{let}\ (f_3,\, t_5) = \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t\, t_5)\rangle_1\rangle_2 \\
& \quad \mathit{in}\ \mathit{let}\ (f_3',\, t_5) = \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t'\, t_5)\rangle_1\rangle_2 \\
& \quad\quad \mathit{in}\ (f_3 \circ f_3',\, t_5) \\
\mathit{eval}_4\, \varepsilon_4\, t_5 \;=\; & \mathcal{S}_3 k_3.(\lambda t_4.t_4,\, t_5) \\
\mathit{eval}_4\, (t \star_4 t')\, t_5 \;=\; & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathit{let}\ (f_4,\, t_5) = \langle k_3\, \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t\, t_5)\rangle_1\rangle_2\rangle_3 \\
& \quad \mathit{in}\ \mathit{let}\ (f_4',\, t_5) \\
& \quad\quad = \langle k_3\, \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t'\, t_5)\rangle_1\rangle_2\rangle_3 \\
& \quad\quad \mathit{in}\ (f_4 \circ f_4',\, t_5) \\
\mathit{eval}_4\, \varepsilon_5\, t_5 \;=\; & \mathcal{S}_4 k_4.t_5 \\
\mathit{eval}_4\, (t \star_5 t')\, t_5 \;=\; & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathcal{S}_4 k_4.\mathit{let}\ t_5 \\
& \qquad = \langle k_4\, \langle k_3\, \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t'\, t_5)\rangle_1\rangle_2\rangle_3\rangle_4 \\
& \qquad \mathit{in}\ \langle k_4\, \langle k_3\, \langle k_2\, \langle k_1\, (\mathit{eval}_4\, t\, t_5)\rangle_1\rangle_2\rangle_3\rangle_4
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reify}_4\, v \;=\; & \langle \mathit{let}\ (f_4,\, t_5) = \langle \mathit{let}\ (f_3,\, t_5) = \langle \mathit{let}\ (f_2,\, t_5) \\
& \qquad = \langle \mathit{let}\ (f_1,\, t_5) = v\, \varepsilon_5 \\
& \qquad\quad \mathit{in}\ (\lambda f_2.(f_1\, \varepsilon_1^{\mathrm{nf}})\, \star_2^{\mathrm{nf}}\, t_2,\, t_5)\rangle_1 \\
& \qquad \mathit{in}\ (\lambda f_3.(f_2\, \varepsilon_2^{\mathrm{nf}})\, \star_3^{\mathrm{nf}}\, t_3,\, t_5)\rangle_2 \\
& \quad \mathit{in}\ (\lambda f_4.(f_3\, \varepsilon_3^{\mathrm{nf}})\, \star_4^{\mathrm{nf}}\, t_4,\, t_5)\rangle_3 \\
& \mathit{in}\ (f_4\, \varepsilon_4^{\mathrm{nf}})\, \star_5^{\mathrm{nf}}\, t_5\rangle_4
\end{aligned}
$$

$$
\mathit{normalize}\, t \;=\; \mathit{reify}_4\, (\mathit{eval}_4\, t)
$$

Whereas $eval_2$ had two layered continuations, $eval_4$ has none since it has been mapped back to direct style twice. Where $eval_2$ accesses k_1 as one of its parameters, $eval_4$ abstracts the first layer of control with shift_1, and where $eval_2$ accesses k_2 as one of its parameters, $eval_4$ abstracts the first and the second layer of control with shift_2. Where $eval_2$ uses reset_1 and shift_1, $eval_4$ uses reset_3 and shift_3, and where $eval_2$ uses reset_2 and shift_2, $eval_4$ uses reset_4 and shift_4.

7.6.5 A note about efficiency

We have implemented all the definitions of Section 7.6.4 as well as the intermediate versions $eval_1$ and $eval_3$ in ML [77]. We have also implemented hierarchical normalization functions for other values than 5.

For high products (i.e., in Section 7.6.4, for source terms using $\star_3$ and $\star_4$), the normalization function living at level 0 of the CPS hierarchy is the most efficient one. On the other hand, for low products (i.e., in Section 7.6.4, for source terms using $\star_1$ and $\star_2$), the normalization functions living at a higher level of the CPS hierarchy are the most efficient ones. These relative efficiencies are explained in terms of resources:

- Accessing to a continuation as an explicit parameter is more efficient than accessing to it through a control operator.

- On the other hand, the restriction of $eval_4$ to source terms that only use ε_1 and $\star_1$ is in direct style, whereas the corresponding restrictions of $eval_2$ and $eval_0$ pass a number of extra parameters. These extra parameters penalize performance.

The better performance of programs in the CPS hierarchy has already been reported for level 1 in the context of continuation-based partial evaluation [159], and it has been reported for a similar "pay as you go" reason: a program that abstracts control relatively rarely is run more efficiently in direct style with a control operator rather than in continuation-passing style.

7.6.6 Summary and conclusion

We have illustrated the CPS hierarchy with an application of normalization by evaluation that naturally involves successive layers of continuations and that demonstrates the expressive power of shift_n and reset_n.

The application also suggests alternative control operators that would fit better its continuation-based programming pattern. For example, instead of representing a delimited continuation as a function and apply it as such, we could represent it as a continuation and apply it with a 'throw' operator as in MacLisp and Standard ML of New Jersey. For another example, instead of throwing a value to a continuation, we could specify the continuation of a computation, e.g., with a $\mathit{reflect}_i$ special form. For a third example, instead of abstracting control up to a layer n, we could give access to each of the successive

layers up to n, e.g., with a $\mathcal{L}_n$ operator. Then instead of

$$\begin{array}{rcl} eval_4\,(t \star_4 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.let\,(f_4,\,t_5) \\ & & \quad = \langle k_3\,\langle k_2\,\langle k_1\,(eval_4\,t\,t_5)\rangle_1\rangle_2\rangle_3 \\ & & \quad in\ let\,(f_4',\,t_5) \\ & & \qquad = \langle k_3\,\langle k_2\,\langle k_1\,(eval_4\,t'\,t_5)\rangle_1\rangle_2\rangle_3 \\ & & \qquad in\ (f_4 \circ f_4',\,t_5) \end{array}$$

one could write

$$\begin{array}{rcl} eval_4\,(t \star_4 t')\,t_5 & = & \mathcal{L}_3\,(k_1,\,k_2,\,k_3).let\,(f_4',\,t_5) \\ & & \quad = reflect_3\,(eval_4\,t\,t_5,\,k_1,\,k_2,\,k_3) \\ & & \quad in\ let\,(f_4',\,t_5) \\ & & \qquad = reflect_3\,(eval_4\,t'\,t_5,\,k_1,\,k_2,\,k_3) \\ & & \qquad in\ (f_4 \circ f_4',\,t_5). \end{array}$$

Such alternative control operators can be more convenient to use, while being compatible with CPS.

7.7 Conclusion and issues

We have used CPS as a guideline to establish an operational foundation for delimited continuations. Starting from a call-by-value evaluator for λ-terms with shift and reset, we have mechanically derived the corresponding abstract machine. From this abstract machine, it is straightforward to obtain a reduction semantics of delimited control that, by construction, is compatible with CPS—both for one-step reduction and for evaluation. These results can also be established without the guideline of CPS, but less easily.

The whole approach generalizes straightforwardly to account for the shift_n and reset_n family of delimited-control operators and more generally for any control operators that are compatible with CPS. These results would be non-trivial to establish without the guideline of CPS.

Defunctionalization provides a key for connecting continuation-passing style and operational intuitions about control. Indeed most of the time, control stacks and evaluation contexts are the defunctionalized continuations of an evaluator. Defunctionalization also provides a key for identifying where operational intuitions about control go beyond CPS (see Section 7.4.5).

We do not know whether CPS is the ultimate answer, but the present work shows yet another example of its usefulness. It is like nothing can go wrong with CPS.

Acknowledgments: We are grateful to Mads Sig Ager, Julia Lawall, Jan Midtgaard, and the referees of CW'04 and of LMCS for their comments. The third author would also like to thank Samuel Lindley for our joint initial study of the normalization functions of Section 7.6.

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-02-0474 (for the two first authors) and Grant no. 21-03-0545 (for the third author).

Chapter 8

A Simple Proof of a Folklore Theorem about Delimited Control

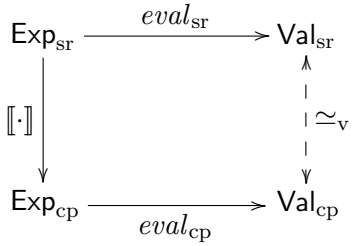
with Olivier Danvy [34]

Abstract

We formalize and prove the folklore theorem that the static delimited-control operators `shift` and `reset` can be simulated in terms of the dynamic delimited-control operators `control` and `prompt`. The proof is based on small-step operational semantics.

8.1 Introduction

In the recent upsurge of interest in delimited continuations [12, 84, 106, 146, 209] it appears to be taken for granted that dynamic delimited continuations can simulate static delimited continuations by delimiting the context of their resumption. And indeed this property has been mentioned early in the literature about delimited continuations [67, Section 5]. We are, however, not aware of any proof of this folklore theorem, and our goal here is to provide such a proof. To this end, we present two abstract machines—one for static delimited continuations as provided by the control operators `shift` and `reset` [67] and inducing a partial evaluation function $eval_{\mathrm{sr}}$, and one for dynamic delimited continuations as provided by the control operators `control` and `prompt` [93] and inducing a partial evaluation function $eval_{\mathrm{cp}}$—and one compositional mapping $[\![\cdot]\!]$ from programs using `shift` and `reset` to programs using `control` and `prompt`. We then prove that the following diagram commutes:



where the value equivalence $\simeq_{\mathrm{v}}$, for ground values, is defined as equality.

8.2 The formalization

Figures 8.1 and 8.2 display two abstract machines, one for the λ-calculus extended with `shift` and `reset`, and one for the λ-calculus extended with `control` and `prompt`. These two machines only differ in the application of captured contexts (which represent delimited continuations in the course of executing source programs).

For simplicity, in the source syntax, we distinguish between λ-bound variables (x) and `shift`- or `control`-bound variables (k). We use the same meta-variables (e, m, i, x, k, v, ρ, C_1 and C_2) ranging over the components of the two abstract machines whenever it does not lead to ambiguity. Programs are closed terms.

8.2.1 A definitional abstract machine for `shift` and `reset`

In our earlier work [29], we derived a definitional abstract machine for `shift` and `reset` by defunctionalizing the continuation and meta-continuation of Danvy and Filinski's definitional evaluator [67]. This definitional abstract machine is displayed in Figure 8.1; it is a straightforward extension of Felleisen et al.'s CEK machine [89] with a meta-context. The source language is the untyped λ-calculus extended with integers, the successor function, `shift` (noted $\mathcal{S}$), and `reset` (noted $\langle\cdot\rangle$). The machine is an extension of the CEK machine because when given a program that does not use `shift` and `reset`, it operates in lock step with the CEK machine. When delimiting control with `reset`, the machine pushes the current context on the current meta-context, and proceeds with an empty context. When abstracting control with `shift`, the machine captures the current context and proceeds with an empty context. When applying a captured context, the machine pushes the current context on the current meta-context, and proceeds with the captured context.

Definition 8.1 *The partial evaluation function $eval_{\mathrm{sr}}$ mapping programs to values is defined as follows:*

$$eval_{\mathrm{sr}}\,(e) = v \text{ if and only if } \langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval} \Rightarrow_{\mathrm{sr}}^{+} \langle \mathsf{nil},\, v\rangle_{cont_2}.$$

We could define the function $eval_{\mathrm{sr}}$ in terms of the initial and final transition, but they play only an administrative role, i.e., to load an input term to the machine and to unload the computed value from the machine.

8.2.2 A definitional abstract machine for `control` and `prompt`

In our earlier work [29], we also showed how to modify the abstract machine for `shift` and `reset` to obtain a definitional abstract machine for `control` and `prompt` [87,93]. This abstract machine is displayed in Figure 8.2. The source language is the λ-calculus extended with integers, the successor function, `control`

- Terms and identifiers: $e ::= \ulcorner m \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
  $i ::= x \mid k$
- Values (integers, closures, and captured contexts):

$$v ::= m \mid [x,\, e,\, \rho] \mid C_1$$

- Environments: $\rho ::= \rho_{mt} \mid \rho\{i \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\ C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{SUCC}\,(C_1)$
- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1,\, m,\, C_2\rangle_{\mathit{cont}_1}$
$\langle i,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1,\, \rho(i),\, C_2\rangle_{\mathit{cont}_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{\mathit{cont}_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\ C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho,\, \mathsf{SUCC}\,(C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \langle\!\langle e \rangle\!\rangle,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{\mathit{eval}}$
$\langle \mathcal{S}k.e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{END},\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_2,\, v\rangle_{\mathit{cont}_2}$
$\langle \mathsf{ARG}\,((e, \rho),\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\mathrm{sr}}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1',\, v,\, C_1 :: C_2\rangle_{\mathit{cont}_1}$
$\langle \mathsf{SUCC}\,(C_1),\, m,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1,\, m+1,\, C_2\rangle_{\mathit{cont}_1}$
$\langle C_1 :: C_2,\, v\rangle_{\mathit{cont}_2}$	$\Rightarrow_{\mathrm{sr}}$	$\langle C_1,\, v,\, C_2\rangle_{\mathit{cont}_1}$
$\langle \mathsf{nil},\, v\rangle_{\mathit{cont}_2}$	$\Rightarrow_{\mathrm{sr}}$	v

Figure 8.1: A definitional abstract machine for `shift` and `reset`

(noted $\mathcal{F}$) and `prompt` (noted #). The machine is an extension of the CEK machine because when given a program that does not use `control` and `prompt`, it operates in lock step with the CEK machine. When delimiting control with `prompt`, the machine pushes the current context on the current meta-context, and proceeds with an empty context. When abstracting control with `control`, the machine captures the current context and proceeds with an empty context. When applying a captured context, the machine concatenates the captured context to the current context and proceeds with the resulting context.

- Terms and identifiers: $e ::= \ulcorner m \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid \mathit{succ}\ e \mid \#e \mid \mathcal{F}k.e$
  $i ::= x \mid k$
- Values (integers, closures, and captured contexts):

$$v ::= m \mid [x,\, e,\, \rho] \mid C_1$$

- Environments: $\rho ::= \rho_{mt} \mid \rho\{i \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\ C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{SUCC}\,(C_1)$
- Concatenation of contexts:

$$\begin{array}{rcl} \mathsf{END} \star C_1' & \stackrel{\text{def}}{=} & C_1' \\ (\mathsf{ARG}\,((e, \rho),\ C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{ARG}\,((e, \rho),\ C_1 \star C_1') \\ (\mathsf{FUN}\,(v,\ C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{FUN}\,(v,\ C_1 \star C_1') \\ (\mathsf{SUCC}\,(C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{SUCC}\,(C_1 \star C_1') \end{array}$$

- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle C_1,\, m,\, C_2\rangle_{\mathit{cont}_1}$
$\langle i,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle C_1,\, \rho(i),\, C_2\rangle_{\mathit{cont}_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{\mathit{cont}_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\ C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho,\, \mathsf{SUCC}\,(C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \#e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{\mathit{eval}}$
$\langle \mathcal{F}k.e,\, \rho,\, C_1,\, C_2\rangle_{\mathit{eval}}$	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{END},\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\text{cp}}$	$\langle C_2,\, v\rangle_{\mathit{cont}_2}$
$\langle \mathsf{ARG}\,((e, \rho),\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\text{cp}}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{\mathit{eval}}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\text{cp}}$	$\langle C_1' \star C_1,\, v,\, C_2\rangle_{\mathit{cont}_1}$
$\langle \mathsf{SUCC}\,(C_1),\, m,\, C_2\rangle_{\mathit{cont}_1}$	$\Rightarrow_{\text{cp}}$	$\langle C_1,\, m+1,\, C_2\rangle_{\mathit{cont}_1}$
$\langle C_1 :: C_2,\, v\rangle_{\mathit{cont}_2}$	$\Rightarrow_{\text{cp}}$	$\langle C_1,\, v,\, C_2\rangle_{\mathit{cont}_1}$
$\langle \mathsf{nil},\, v\rangle_{\mathit{cont}_2}$	$\Rightarrow_{\text{cp}}$	v

Figure 8.2: A definitional abstract machine for `control` and `prompt`

Definition 8.2 *The partial evaluation function* $\mathit{eval}_{\text{cp}}$ *mapping programs to values is defined as follows:*

$$\mathit{eval}_{\text{cp}}\,(e) = v \text{ if and only if } \langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{\mathit{eval}} \Rightarrow_{\text{cp}}^{+} \langle \mathsf{nil},\, v\rangle_{\mathit{cont}_2}.$$

8.2.3 Static vs. dynamic delimited continuations

In Figure 8.1, `shift` and `reset` are said to be *static* because the application of a delimited continuation (represented as a captured context) does not depend on the current context. It is implemented by pushing the current context on the stack of contexts and installing the captured context as the new current context, as shown by the following transition:

$$\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{cont_1} \ \Rightarrow_{\mathrm{sr}}\ \langle C_1',\ v,\ C_1 :: C_2 \rangle_{cont_1}$$

A subsequent `shift` operation will therefore capture the remainder of the reinstated context, statically.

In Figure 8.2, `control` and `prompt` are said to be *dynamic* because the application of a delimited continuation (also represented as a captured context) depends on the current context. It is implemented by concatenating the captured context to the current context, as shown by the following transition:

$$\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{cont_1} \ \Rightarrow_{\mathrm{cp}}\ \langle C_1' \star C_1,\ v,\ C_2 \rangle_{cont_1}$$

A subsequent `control` operation will therefore capture the remainder of the reinstated context together with the then-current context, dynamically.

The two abstract machines differ only in this single transition. Because of this single transition, programs using `shift` and `reset` are compatible with the traditional notion of continuation-passing style [29, 67, 186] whereas programs using `control` and `prompt` give rise to a more complex notion of continuation-passing style that threads a dynamic state [35, 84, 209]. This difference in the semantics of `shift` and `control` also induces distinct computational behaviors, as illustrated in the following example.

Copying vs. reversing a list: Using `call-with-current-delimited-continuation` (instead of `shift` or `control`) and `delimit-continuation` (instead of `reset` or `prompt`), let us consider the following function that traverses a given list and returns another list [29, Sec. 4.5]; this function is written in the syntax of Scheme [143]:

```
(define traverse
  (lambda (xs)
    (letrec ([visit
              (lambda (xs)
                (if (null? xs)
                    '()
                    (visit (call-with-current-delimited-continuation
                             (lambda (k)
                               (cons (car xs) (k (cdr xs))))))))])
      (delimit-continuation
        (lambda ()
          (visit xs))))))
```

- The function copies its input list if `shift` and `reset` are used instead of `call-with- current-delimited-continuation` and `delimit-continuation`.

The reason why is that reinstating a `shift`-abstracted context keeps it distinct from the current context. Here, `shift` successively abstracts a delimited context that solely consists of the call to `visit`. Intuitively, this delimited context reads as follows:

```
(lambda (v)
  (delimit-continuation
    (lambda ()
      (visit v))))
```

- The function reverses its input list if `control` and `prompt` are used instead of `call-with-current-delimited-continuation` and `delimit-continuation`. The reason why is that reinstating a `control`-abstracted context grafts it to the current context. Here, `control` successively abstracts a context that consists of the call to `visit` followed by the construction of a reversed prefix of the input list. Intuitively, when the input list is `(1 2 3)`, the successive contexts read as follows:

```
(lambda (v) (visit v))
(lambda (v) (cons 1 (visit v))
(lambda (v) (cons 2 (cons 1 (visit v))))
```

Programming folklore. *To obtain the effect of* `shift` *and* `reset` *using* `control` *and* `prompt`*, one should replace every occurrence of a shift-bound variable k by its η-expanded and delimited version $\lambda x.\#(k\,x)$. (As a β_{v}-optimization, every application of k to a trivial expression e (typically a value) can be replaced by $\#(k\,e)$.)*

And indeed, replacing

```
(cons (car xs) (k (cdr xs)))
```

by

```
(cons (car xs) (delimit-continuation
                 (lambda ()
                   (k (cdr xs)))))
```

in the definition of `traverse` above makes it copy its input list, no matter whether `shift` and `reset` or `control` and `prompt` are used.

We formalize the replacement above with the following compositional translation from the language with `shift` and `reset` to the language with `control` and `prompt`.

Definition 8.3 *The translation* $[\![\cdot]\!]$ *is defined as follows:*

$$\begin{array}{rcl}
[\![\ulcorner m \urcorner]\!] & = & \ulcorner m \urcorner \\
[\![x]\!] & = & x \\
[\![k]\!] & = & \lambda x.\#(k\,x), \text{ where } x \text{ is fresh} \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\
[\![e_0\,e_1]\!] & = & [\![e_0]\!]\,[\![e_1]\!] \\
[\![\langle e \rangle]\!] & = & \#[\![e]\!] \\
[\![\mathcal{S}k.e]\!] & = & \mathcal{F}k.[\![e]\!]
\end{array}$$

In the next section, we prove that for any program e, $eval_{\mathrm{sr}}\,(e)$ and $eval_{\mathrm{cp}}\,([\![e]\!])$ are equivalent (in the sense of Definition 8.5 below) and, in particular, equal for ground values.

8.3 The folklore theorem and its formal proof

We first define an auxiliary abstract machine for `control` and `prompt` that implements the application of an η-expanded and delimited continuation in one step. By construction, this auxiliary abstract machine is equivalent to the definitional one of Figure 8.2. We then show that the auxiliary machine operates in lock step with the definitional abstract machine of Figure 8.1. To this end, we define a family of relations between the abstract machine for `shift` and `reset` and the auxiliary abstract machine. The folklore theorem follows.

8.3.1 An auxiliary abstract machine for `control` and `prompt`

Definition 8.4 *The auxiliary abstract machine for* `control` *and* `prompt` *is defined as follows:*

1. *All the components, including configurations* δ*, of the auxiliary abstract machine are identical to the components of the definitional abstract machine of Figure 8.2.*

2. *The transitions of the auxiliary abstract machine, denoted* $\delta \Rightarrow_{\mathrm{aux}} \delta'$*, are defined as follows:*

   - *if* $\delta = \langle \mathsf{FUN}\,([x,\, \#(k\,x),\, \rho],\, C_1),\, v,\, C_2 \rangle_{cont_1}$
     then $\delta' = \langle C_1',\, v,\, C_1 :: C_2 \rangle_{cont_1}$*, where* $C_1' = \rho(k)$*;*
   - *otherwise,* δ' *is the configuration such that* $\delta \Rightarrow_{\mathrm{cp}} \delta'$*, if it exists.*

3. *The partial evaluation function* $eval_{\mathrm{aux}}$ *is defined in the usual way:* $eval_{\mathrm{aux}}\,(e) = v$ *if and only if* $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil} \rangle_{eval} \Rightarrow^{+}_{\mathrm{aux}} \langle \mathsf{nil},\, v \rangle_{cont_2}$*.*

The following lemma shows that the definitional abstract machine for `control` and `prompt` simulates the single step of the auxiliary abstract machine in several steps.

Lemma 8.1 *For all v, C_1, C_1' and C_2,*

$$\langle \mathsf{FUN}\,([x,\, \#(k\,x),\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1} \Rightarrow_{\mathrm{cp}}^{+} \langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1},$$

where $C_1' = \rho(k)$.

Proof. From the definition of the abstract machine for `control` and `prompt` in Figure 8.2:

$$\begin{aligned}
\langle \mathsf{FUN}\,([x,\, \#(k\,x),\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1} &\Rightarrow_{\mathrm{cp}}\\
\langle \#(k\,x),\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval} &\Rightarrow_{\mathrm{cp}}\\
\langle k\,x,\, \rho\{x \mapsto v\},\, \mathsf{END},\, C_1 :: C_2\rangle_{eval} &\Rightarrow_{\mathrm{cp}}\\
\langle k,\, \rho\{x \mapsto v\},\, \mathsf{ARG}\,((x, \rho\{x \mapsto v\}),\, \mathsf{END}),\, C_1 :: C_2\rangle_{eval} &\Rightarrow_{\mathrm{cp}}\\
\langle \mathsf{ARG}\,((x, \rho\{x \mapsto v\}),\, \mathsf{END}),\, C_1',\, C_1 :: C_2\rangle_{cont_1} &\Rightarrow_{\mathrm{cp}}\\
\langle x,\, \rho\{x \mapsto v\},\, \mathsf{FUN}\,(C_1',\, \mathsf{END}),\, C_1 :: C_2\rangle_{eval} &\Rightarrow_{\mathrm{cp}}\\
\langle \mathsf{FUN}\,(C_1',\, \mathsf{END}),\, v,\, C_1 :: C_2\rangle_{cont_1} &\Rightarrow_{\mathrm{cp}}\\
\langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}
\end{aligned}$$

□

Proposition 8.1 *For any program e and for any value v, $eval_{\mathrm{cp}}\,(e) = v$ if and only if $eval_{\mathrm{aux}}\,(e) = v$.*

Proof. Follows directly from Definition 8.4 and Lemma 8.1. □

8.3.2 A family of relations

We now define a family of relations between the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt`. To distinguish between the two machines, as a diacritical convention [169], we annotate the components of the machine for `shift` and `reset` with a hat.

Definition 8.5 *The relations between the components of the abstract machine for* `shift` *and* `reset` *and the auxiliary abstract machine for* `control` *and* `prompt` *are defined as follows:*

1. *Terms: $\widehat{e} \simeq_{\mathrm{e}} e$ iff $[\![\widehat{e}]\!] = e$*
2. *Values:*
   (a) *$\widehat{m} \simeq_{\mathrm{v}} m$ iff $\widehat{m} = m$*
   (b) *$[\widehat{x},\, \widehat{e},\, \widehat{\rho}] \simeq_{\mathrm{v}} [x,\, e,\, \rho]$ iff $\widehat{x} = x$, $\widehat{e} \simeq_{\mathrm{e}} e$ and $\widehat{\rho} \simeq_{\mathrm{env}} \rho$*
   (c) *$\widehat{C_1} \simeq_{\mathrm{v}} [x,\, \#(k\,x),\, \rho]$ iff $\widehat{C_1} \simeq_{\mathrm{c}} \rho(k)$*
3. *Environments:*
   (a) *$\widehat{\rho_{mt}} \simeq_{\mathrm{env}} \rho_{mt}$*
   (b) *$\widehat{\rho}\{x \mapsto \widehat{v}\} \simeq_{\mathrm{env}} \rho\{x \mapsto v\}$ iff $\widehat{v} \simeq_{\mathrm{v}} v$ and $\widehat{\rho} \setminus \{x\} \simeq_{\mathrm{env}} \rho \setminus \{x\}$, where $\rho \setminus \{i\}$ denotes the restriction of ρ to its domain excluding i*
   (c) *$\widehat{\rho}\{k \mapsto \widehat{C_1}\} \simeq_{\mathrm{env}} \rho\{k \mapsto C_1\}$ iff $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ and $\widehat{\rho} \setminus \{k\} \simeq_{\mathrm{env}} \rho \setminus \{k\}$*

4. *Contexts:*
   (a) $\widehat{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END}$
   (b) $\widehat{\mathsf{ARG}}\,((\widehat{e}, \widehat{\rho}),\ \widehat{C_1}) \simeq_{\mathrm{c}} \mathsf{ARG}\,((e, \rho),\ C_1)$ *iff* $\widehat{e} \simeq_{\mathrm{e}} e$, $\widehat{\rho} \simeq_{\mathrm{env}} \rho$, *and* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$
   (c) $\widehat{\mathsf{FUN}}\,(\widehat{v},\ \widehat{C_1}) \simeq_{\mathrm{c}} \mathsf{FUN}\,(v,\ C_1)$ *iff* $\widehat{v} \simeq_{\mathrm{v}} v$ *and* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$
   (d) $\widehat{\mathsf{SUCC}}\,(\widehat{C_1}) \simeq_{\mathrm{c}} \mathsf{SUCC}\,(C_1)$ *iff* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$

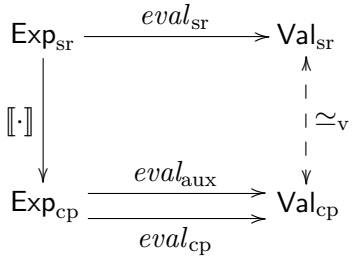
5. *Meta-contexts:*
   (a) $\widehat{\mathsf{nil}} \simeq_{\mathrm{mc}} \mathsf{nil}$
   (b) $\widehat{C_1} :: \widehat{C_2} \simeq_{\mathrm{mc}} C_1 :: C_2$ *iff* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$

6. *Configurations:*
   (a) $\langle \widehat{e}, \widehat{\rho}, \widehat{C_1}, \widehat{C_2} \rangle_{\widehat{eval}} \simeq \langle e, \rho, C_1, C_2 \rangle_{eval}$ *iff*
   $\widehat{e} \simeq_{\mathrm{e}} e$, $\widehat{\rho} \simeq_{\mathrm{env}} \rho$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1$, *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$
   (b) $\langle \widehat{C_1}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}} \simeq \langle C_1, v, C_2 \rangle_{cont_1}$ *iff*
   $\widehat{C_1} \simeq_{\mathrm{c}} C_1$, $\widehat{v} \simeq_{\mathrm{v}} v$, *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$
   (c) $\langle \widehat{C_2}, \widehat{v} \rangle_{\widehat{cont_2}} \simeq \langle C_2, v \rangle_{cont_2}$ *iff*
   $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$ *and* $\widehat{v} \simeq_{\mathrm{v}} v$

The relations are intended to capture the equivalence of the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt` when run on a term $\widehat{e}$ and on its translation $[\![\widehat{e}]\!]$, respectively. Most of the cases are homomorphic on the structure of a component. The critical cases are: (1)—a formalization of the programming folklore formulated in Section 8.2.3, and (2)(*c*)—a formalization of the fact that a `control`-abstracted continuation is applied by concatenating its representation to the current context whereas when a `shift`-abstracted continuation is applied, its representation is kept separate from the current context.

8.3.3 The formal proof

We first show that indeed, running the abstract machine for `shift` and `reset` on a program $\widehat{e}$ and running the auxiliary abstract machine for `control` and `prompt` on a program $[\![\widehat{e}]\!]$ yield results that are equivalent in the sense of the above relations. Then by Proposition 8.1, we obtain the equivalence result of the abstract machine for `shift` and `reset` and the definitional abstract machine for `control` and `prompt`, as summarized in the following diagram:



More precisely, we show that the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt` operate in lock-step with respect to the relations. To this end we need to prove the following lemmas.

Lemma 8.2 *For all configurations $\widehat{\delta}$, δ, $\widehat{\delta'}$ and δ', if $\widehat{\delta} \simeq \delta$ then*

$$\widehat{\delta} \Rightarrow_{\mathrm{sr}} \widehat{\delta'} \text{ if and only if } \delta \Rightarrow_{\mathrm{aux}} \delta' \text{ and } \widehat{\delta'} \simeq \delta'.$$

Proof. By case inspection of $\widehat{\delta} \simeq \delta$. All cases follow directly from the definition of the relation $\simeq$ and the definitions of the abstract machines. We present two crucial cases:

Case: $\widehat{\delta} = \langle k, \widehat{\rho}, \widehat{C_1}, \widehat{C_2} \rangle_{\widehat{eval}}$ and $\delta = \langle \lambda x.\#(k\,x), \rho, C_1, C_2 \rangle_{eval}$.
From the definition of the abstract machine for `shift` and `reset`, $\widehat{\delta} \Rightarrow_{\mathrm{sr}} \widehat{\delta'}$, where $\widehat{\delta'} = \langle \widehat{C_1}, \widehat{\rho}(k), \widehat{C_2} \rangle_{\widehat{cont_1}}$.
From the definition of the auxiliary abstract machine for `control` and `prompt`, $\delta \Rightarrow_{\mathrm{aux}} \delta'$, where $\delta' = \langle C_1, [x, \#(k\,x), \rho], C_2 \rangle_{cont_1}$.
By assumption, $\widehat{\rho}(k) \simeq_{\mathrm{c}} \rho(k)$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ and $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widehat{\delta'} \simeq \delta'$.

Case: $\widehat{\delta} = \langle \widehat{\mathsf{FUN}}\,(\widehat{C_1}', \widehat{C_1}), \widehat{v}, \widehat{C_2} \rangle_{\widehat{eval}}$
and $\delta = \langle \mathsf{FUN}\,([x, \#(k\,x), \rho], C_1), v, C_2 \rangle_{eval}$.
From the definition of the abstract machine for `shift` and `reset`, $\widehat{\delta} \Rightarrow_{\mathrm{sr}} \widehat{\delta'}$, where $\widehat{\delta'} = \langle \widehat{C_1}', \widehat{v}, \widehat{C_1} :: \widehat{C_2} \rangle_{\widehat{cont_1}}$.
From the definition of the auxiliary abstract machine for `control` and `prompt`, $\delta \Rightarrow_{\mathrm{aux}} \delta'$, where $\delta' = \langle C_1', v, C_1 :: C_2 \rangle_{cont_1}$, and $C_1' = \rho(k)$.
By assumption, $\widehat{C_1}' \simeq_{\mathrm{c}} C_1'$, $\widehat{v} \simeq_{\mathrm{v}} v$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ and $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widehat{\delta'} \simeq \delta'$.

□

Lemma 8.3 *For all configurations $\widehat{\delta}$, δ, $\widehat{\delta'}$ and δ', and for any $n \geq 1$, if $\widehat{\delta} \simeq \delta$ then*

$$\widehat{\delta} \Rightarrow_{\mathrm{sr}} \widehat{\delta'} \text{ if and only if } \delta \Rightarrow_{\mathrm{aux}} \delta' \text{ and } \widehat{\delta'} \simeq \delta'.$$

Proof. By induction on n, using Lemma 8.2. □

We are now in position to prove the formal statement of the equivalence between the two abstract machines:

Proposition 8.2 *For any program $\widehat{e}$, either both $eval_{\mathrm{sr}}(\widehat{e})$ and $eval_{\mathrm{aux}}(\llbracket \widehat{e} \rrbracket)$ are undefined or there exist values $\widehat{v}$ and v, such that $eval_{\mathrm{sr}}(\widehat{e}) = \widehat{v}$, $eval_{\mathrm{aux}}(\llbracket \widehat{e} \rrbracket) = v$, and $\widehat{v} \simeq_{\mathrm{v}} v$.*

Proof. Since the initial configurations $\langle \widehat{e}, \widehat{\rho_{mt}}, \widehat{\mathsf{END}}, \widehat{\mathsf{nil}} \rangle_{\widehat{eval}}$ and $\langle \llbracket \widehat{e} \rrbracket, \rho_{mt}, \mathsf{END}, \mathsf{nil} \rangle_{eval}$ are in the relation $\simeq$, then by Lemma 8.3 both abstract machines reach their final configurations $\langle \widehat{\mathsf{nil}}, \widehat{v} \rangle_{\widehat{cont_2}}$ and $\langle \mathsf{nil}, v \rangle_{cont_2}$ after the same number of transitions and with $\widehat{v} \simeq_{\mathrm{v}} v$, or both diverge. □

Theorem 8.1 *For any program $\widehat{e}$, either both $eval_{\mathrm{sr}}(\widehat{e})$ and $eval_{\mathrm{cp}}(\llbracket\widehat{e}\rrbracket)$ are undefined or there exist values $\widehat{v}$ and v, such that $eval_{\mathrm{sr}}(\widehat{e}) = \widehat{v}$, $eval_{\mathrm{cp}}(\llbracket\widehat{e}\rrbracket) = v$, and $\widehat{v} \simeq_{\mathrm{v}} v$.*

Proof. Follows directly from Proposition 8.1 and Proposition 8.2. □

Corollary 8.1 (Folklore) *For any program $\widehat{e}$, and for any integer m, $eval_{\mathrm{sr}}(\widehat{e}) = m$ if and only if $eval_{\mathrm{cp}}(\llbracket\widehat{e}\rrbracket) = m$.*

Extending the source language with more syntactic constructs (other ground values and primitive operations, conditional expressions, recursive definitions, etc.) is straightforward. It is equally simple to extend the proof.

Our simple proof is based on the original (operational) specification of static and dynamic delimited continuations. An alternative proof could be based, e.g., on equational reasoning [87, 142].

8.4 Conclusion

We have formalized and proved that the dynamic delimited-control operators `control` and `prompt` can simulate the static delimited-control operators `shift` and `reset` by delimiting the context of the resumption of captured continuations. Several converse simulations have been presented recently [35, 146, 209]. These converse simulations are considerably more involved than the present one, and have not been formalized and proved yet.

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Julia Lawall, Kevin Millikin, and Kristian Støvring for their comments. Special thanks to the anonymous reviewers for an insightful e-mail exchange. This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

Chapter 9

On the Static and Dynamic Extents of Delimited Continuations

with Olivier Danvy and Chung-chieh Shan [37]

Abstract

We show that breadth-first traversal exploits the difference between the static delimited-control operator `shift` (alias $\mathcal{S}$) and the dynamic delimited-control operator `control` (alias $\mathcal{F}$). For the last 15 years, this difference has been repeatedly mentioned in the literature but it has only been illustrated with one-line toy examples. Breadth-first traversal fills this vacuum.

We also point out where static delimited continuations naturally give rise to the notion of control stack whereas dynamic delimited continuations can be made to account for a notion of 'control queue.'

9.1 Introduction

To distinguish between the static extent and the dynamic extent of delimited continuations, we first need to review the notions of continuation and of delimited continuation.

9.1.1 Background

Continuation-passing style (CPS) is a time-honored and logic-based format for functional programs where all intermediate results are named, all calls are tail calls, and programs are evaluation-order independent [109, 168, 186, 196, 217]. While this format has been an active topic of study [23, 27, 32, 85, 97, 105, 119, 153, 158, 176, 181, 188, 201, 210, 212, 231], it also has been felt as a straightjacket both from a semantics point of view [85, 87, 90, 93, 133, 134, 214] and from a programming point of view [56, 58, 66, 67], where one would like to relax the tail-call constraint and compose continuations.

In direct style, continuations are accessed with a variety of control operators such as Landin's `J` [154], Reynolds's `escape` [196], Scheme's `call/cc` [46,143], and Standard ML of New Jersey's `callcc` and `throw` [81]. These control operators

give access to the current continuation as a first-class value. Activating such a first-class continuation has the effect of resuming the computation at the point where this continuation was captured; the then-current continuation is *abandoned*. Such first-class continuations *do not return to the point of their activation*—they model jumps, i.e., tail calls [217, 225].

In direct style, delimited continuations are also accessed with control operators such as Felleisen et al.'s `control` (alias $\mathcal{F}$) [85, 90, 93, 214] and Danvy and Filinski's `shift` (alias $\mathcal{S}$) [66–68]. These control operators also give access to the current continuation as a first-class value; activating such a first-class continuation also has the effect of resuming the computation at the point where this continuation was captured; the then-current continuation, however, *is then resumed*. Such first-class continuations *return to the point of their activation*—they model non-tail calls.

For a first-class continuation to return to the point of its activation, one must declare its point of completion, since this point is no longer at the very end of the overall computation, as with traditional, undelimited first-class continuations. In direct style, this declaration is achieved with a new kind of operator, due to Felleisen [85, 87]: a control delimiter. The control delimiter corresponding to `control` is called `prompt` (alias #). The control delimiter corresponding to `shift` is called `reset` (alias $\langle\cdot\rangle$) and its continuation-passing counterpart is a classical backtracking idiom in functional programming [3, 41, 44, 167, 203, 229], one that is currently enjoying a renewal of interest [33, 69, 127, 147, 216, 237, 249]. Other, more advanced, delimited-control operators exist [84, 111, 125, 173, 178, 191]; we return to them in the conclusion.

In the present work, we focus on `shift` and `control`.

9.1.2 Overview

In Section 9.2, we present an environment-based abstract machine that specifies the behaviors of `shift` and `control`, and we show how the extent of a `shift`-abstracted delimited continuation is static whereas that of a `control`-abstracted delimited continuation is dynamic. We show how `shift` can be trivially simulated in terms of `control` and `prompt`, which is a well-known result [34], and we review recently discovered simulations of control and prompt in terms of shift and reset [35, 146, 209]. In Section 9.3, we present a roadmap of Sections 9.4 and 9.5, where we show how the static extent of a delimited continuation is compatible with a control stack and depth-first traversal, and how the dynamic extent of a delimited continuation can be made to account for a 'control queue' and breadth-first traversal.

Prerequisites and preliminaries: Besides some awareness of CPS and the CPS transformation [68, 186, 217], we assume a passing familiarity with defunctionalization [74, 196].

Our programming language of discourse is Standard ML [170]. In the following sections, we will make use of the notational equivalence of expressions such as

```
x1 :: x2 :: xs
(x1 :: x2 :: nil) @ xs
[x1, x2] @ xs
```

where `::` denotes infix list construction and `@` denotes infix list concatenation. In an environment where `x1` denotes `1`, `x2` denotes `2`, and `xs` denotes `[3, 4, 5]`, each of the three expressions above evaluates to `[1, 2, 3, 4, 5]`.

9.2 An operational characterization

In our previous work [29], we derived an environment-based abstract machine for the λ-calculus with `shift` and `reset` by defunctionalizing the corresponding definitional interpreter [67]. We use this abstract machine to explain the static extent of the delimited continuations abstracted by `shift` and the dynamic extent of the delimited continuations abstracted by `control`.

9.2.1 An abstract machine for `shift` and `reset`

The abstract machine is displayed in Figure 9.1; `reset` is noted $\langle\cdot\rangle$ and `shift` is noted $\mathcal{S}$. The set of possible values consists of closures and captured contexts. The machine extends Felleisen et al.'s CEK machine [89] with a meta-context C_2, the two transitions for $\langle\cdot\rangle$ and $\mathcal{S}$, and the transition for applying a captured context to a value in an evaluation context and a meta-context. Intuitively, an evaluation context represents the rest of the computation up to the nearest enclosing delimiter, and a meta-context represents all of the remaining computation [64].

Given a term e, the machine is initialized in an *eval*-state with an empty environment ρ_{mt}, an empty context END, and an empty meta-context nil. The transitions out of an *eval*-state are defined by cases on its first component:

- a variable x is looked up in the current environment and the machine switches to a $cont_1$-state;
- an abstraction $\lambda x.e$ is evaluated into a closure $[x, e, \rho]$ and the machine switches to a $cont_1$-state;
- an application $e_0\, e_1$ is processed by pushing e_1 and the environment onto the context and switching to a new *eval*-state to process e_0;
- a reset-expression $\langle e\rangle$ is processed by pushing the current context on the current meta-context and switching to a new *eval*-state to process e in an empty context, as an intermediate computation;
- a shift-expression $\mathcal{S}k.e$ is processed by capturing the context C_1 and binding it to k, and switching to a new *eval*-state to process e in an empty context.

The transitions of a $cont_1$-state are defined by cases on its first component:

- an empty context END specifies that an intermediate computation is completed; it is processed by switching to a $cont_2$-state;

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
- Values (closures and captured continuations): $v ::= [x,\, e,\, \rho] \mid C_1$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\, C_1) \mid \mathsf{FUN}\,(v,\, C_1)$
- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \rho\,(x),\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\, C_1),\, C_2\rangle_{eval}$
$\langle \langle\!\langle e \rangle\!\rangle,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{eval}$
$\langle \mathcal{S}k.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{eval}$
$\langle \mathsf{END},\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle \mathsf{ARG}\,((e, \rho),\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\, C_1),\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,(C_1',\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}$
$\langle C_1 :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow$	v

Figure 9.1: A call-by-value abstract machine for the λ-calculus extended with `shift` ($\mathcal{S}$) and `reset` ($\langle\!\langle\cdot\rangle\!\rangle$)

- a context $\mathsf{ARG}\,((e, \rho),\, C_1)$ specifies the evaluation of an argument; it is processed by switching to an *eval*-state to process e in a new context;
- a context $\mathsf{FUN}\,([x,\, e,\, \rho],\, C_1)$ specifies the application of a closure; it is processed by switching to an *eval*-state to process the term e with an extension of the environment ρ;
- a context $\mathsf{FUN}\,(C_1',\, C_1)$ specifies the application of a captured context; it is processed by pushing C_1 on top of the meta-context and switching to a new $cont_1$-state to process C_1'.

The transitions of a $cont_2$-state are defined by cases on its first component:

- an empty meta-context nil specifies that the overall computation is completed; it is processed as a final transition;

- a non-empty meta-context specifies that the overall computation is not completed; $C_1 :: C_2$ is processed by switching to a $cont_1$-state to process C_1.

All in all, this abstract machine is a straight defunctionalized continuation-passing evaluator [29, 67].

9.2.2 An abstract machine for `control` and `prompt`

Unlike `shift` and `reset`, whose definition is based on CPS, `control` and `prompt` are specified by representing delimited continuations as a list of stack frames and their composition as the concatenation of these representations [85, 93]. Such a concatenation function $\star$ is defined as follows:

$$\begin{array}{rcl} \mathsf{END} \star C_1' & = & C_1' \\ (\mathsf{ARG}\,((e,\rho),\ C_1)) \star C_1' & = & \mathsf{ARG}\,((e,\rho),\ C_1 \star C_1') \\ (\mathsf{FUN}\,(v,\ C_1)) \star C_1' & = & \mathsf{FUN}\,(v,\ C_1 \star C_1') \end{array}$$

It is then simple to modify the abstract machine to compose delimited continuations by concatenating their representation: in Figure 9.1, one merely replaces the transition that applies a captured context C_1' by pushing the current context C_1 onto the meta-context C_2, i.e.,

$$\boxed{\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1',\ v,\ C_1 :: C_2 \rangle_{cont_1}}$$

with a transition that applies a captured context C_1' by concatenating it with the current context C_1:

$$\boxed{\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1' \star C_1,\ v,\ C_2 \rangle_{cont_1}}$$

This change gives $\mathcal{S}$ (alias `shift`) the behavior of $\mathcal{F}$ (alias `control`). In contrast, $\langle\cdot\rangle$ (alias `reset`) and # (alias `prompt`) have the same definition. The rest of the machine does not change.

In our previous work [29, Section 4.5], we have pointed out that the dynamic behavior of `control` is incompatible with CPS because the modified abstract machine no longer corresponds to a defunctionalized continuation-passing evaluator [74]. Indeed `shift` is static, whereas `control` is dynamic, in the following sense:

- `shift` captures a delimited continuation in a representation C_1 that, when applied, remains distinct from the current context C_1'. Consequently, the current context C_1' *cannot* be accessed from C_1 by another use of `shift`. (An analogy: in a statically scoped programming language, the environment of an application remains distinct from the environment of the applied function. A non-local variable in the function refers to the environment of its definition. Consequently, the environment of a function application cannot be accessed before the function completes.)

- `control` captures a delimited continuation in a representation C_1 that, when applied, grafts itself to the current context C_1'. Consequently, the current context C_1' *can* be accessed from C_1 by another use of `control`. (An analogy: in a dynamically scoped programming language, the environment of an application is extended with the environment of the applied function. A non-local variable in the function refers to the environment of its application. Consequently, the environment of a function application can be accessed before the function completes.)

This difference of extent can be observed with delimited continuations that, when applied, capture the current continuation [31, Section 5] [66, Section 6.1] [68, Section 5.3] [93, Section 4]. A `control`-abstracted delimited continuation dynamically captures the current continuation, above and beyond its point of activation, whereas a `shift`-abstracted delimited continuation statically captures the current continuation up to its point of activation.

9.2.3 Simulating `shift` in terms of `control` and `prompt`

It is simple to obtain the effect of `shift` using `control`: for each captured continuation k, every occurrence of $k\,v$ should be replaced by $\#(k\,v)$ when v is a value, and every other occurrence of k should be replaced with $\lambda x.\#(k\,x)$. (In ML, for each captured continuation `k`, every occurrence of `k v` should be replaced by `prompt (fn () => k v)` when `v` denotes a value, and every other occurrence of `k` should be replaced with `fn x => prompt (fn () => k x)`.)

This way, when k (i.e., some context C_1') is applied, the context of its application is always `END` and it is a consequence of the definition of $\star$ that $C_1' \star \mathsf{END} = C_1'$. The two first authors have recently given a formal proof of the correctness of this simulation [34].

9.2.4 Simulating `control` in terms of `shift` and `reset`

Recently it has been shown that `control` and `prompt` can be expressed in terms of `shift` and `reset`, which unexpectedly proves that `shift` is actually as expressive as `control`.

- In his previous article [209], Shan presented a simulation that is based on his observation that dynamic continuations are recursive. His simulation keeps (as a piece of mutable state) the context in which a `control`-captured delimited continuation is applied. This simulation is untyped and implemented in Scheme.

- In their recent article [35], Biernacki, Danvy, and Millikin presented a new simulation that is based on a 'Dynamic Continuation-Passing Style' (DCPS) for dynamic delimited continuations. Their idea is to use a trail of continuations to represent the context in which a `control`-captured delimited continuation is applied, and to compose continuations by concatenating such trails of continuations. This simulation is typed and implemented in ML.

- In his recent article [146], Kiselyov proposed a new simulation that is based on trampolining. In order to let a `control`-captured continuation access the context where it is applied, he reifies such an access in a sum type interpreted by `prompt`. This simulation is untyped and implemented in Scheme.

Concomitant with each solution is a CPS transformation for `control` and `prompt` that conservatively extends the usual call-by-value CPS transformation for the λ-calculus, with the requirement that continuations be recursive (or more precisely, that their answer type be higher-order and recursive).

In Appendix B, we present Shan's implementation of `control` and `prompt` in Standard ML of New Jersey [209]. This implementation is based on Filinski's implementation of `shift` and `reset` in SML [96], which we present in Appendix A. Filinski's implementation takes the form of a functor mapping the type of intermediate answers to a structure containing an instance of `shift` and `reset` at that type:

```
signature SHIFT_AND_RESET
= sig
    type intermediate_answer
    val shift : (('a -> intermediate_answer) -> intermediate_answer)
                -> 'a
    val reset : (unit -> intermediate_answer) -> intermediate_answer
  end
```

Likewise, our implementation takes the form of a functor mapping the type of intermediate answers to a structure containing an instance of `control` and `prompt` at that type:

```
signature CONTROL_AND_PROMPT
= sig
    type intermediate_answer
    val control : (('a -> intermediate_answer) -> intermediate_answer)
                  -> 'a
    val prompt : (unit -> intermediate_answer) -> intermediate_answer
  end
```

9.2.5 Three examples in ML

Using the implementation of `shift` and `reset` (Appendix 9.A), and of `control` and `prompt` (Appendix 9.B), we present three simple examples illustrating the difference between `shift` and `control`. Let us fix the type of intermediate answers to be `int`:

```
local structure SR = Shift_and_Reset
                       (type intermediate_answer = int)
in val shift = SR.shift
   val reset = SR.reset
end
```

```
local structure CP = Control_and_Prompt
                       (type intermediate_answer = int)
in val control = CP.control
   val prompt = CP.prompt
end
```

The following ML expression

```
reset
 (fn () => shift (fn k => 10 + (k 100))
           + shift (fn k' => 1))
```

evaluates to `11`, whereas (replacing `reset` by `prompt` and `shift` by `control`)

```
prompt
 (fn () => control (fn k => 10 + (k 100))
           + control (fn k' => 1))
```

evaluates to `1` and (delimiting the application of `k` with `prompt`)

```
prompt
 (fn () => control (fn k => 10 + prompt (fn () => k 100))
           + control (fn k' => 1))
```

evaluates to `11`.

In the first case, `shift (fn k => 10 + (k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + shift (fn k' => 1)`. When `k` is applied, the expression `shift (fn k' => 1)` is evaluated in a context that could be represented functionally as `fn v => 100 + v` and in a meta-context that could be represented as `(fn v => 10 + v) :: nil`; this context is captured and discarded, and the intermediate answer is `1`; this intermediate answer is plugged into the top context from the meta-context, i.e., `fn v => 10 + v` is applied to `1`; the next intermediate answer is `11`; and it is the final answer since the meta-context is empty.

In the second case, `control (fn k => 10 + (k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + control (fn k' => 1)`. When `k` is applied, the expression `control (fn k' => 1)` is evaluated in a context that results from composing `fn v => 10 + v` and `fn v => 100 + v` (and therefore could be represented functionally as `fn v => 10 + (100 + v)`), and in a meta-context which is empty; this context is captured and discarded, and the intermediate answer is `1`; and it is the final answer since the meta-context is empty.

In the third case, `control (fn k => 10 + prompt (fn () => k 100))` is evaluated with a continuation that could be written functionally as `fn v => v + control (fn k' => 1)`. When `k` is applied, the expression `control (fn k' => 1)` is evaluated in a context that results from composing `fn v => v` and `fn v => 100 + v` (and therefore could be represented functionally as `fn v => 100 + v`), and in a meta-context which could be represented as `(fn v => 10 + v) :: nil`; this context is captured and discarded, and the intermediate answer is `1`; this intermediate answer is plugged into the top context from the meta-context,

i.e., `fn v => 10 + v` is applied to `1`; the next intermediate answer is `11`; and it is the final answer since the meta-context is empty.

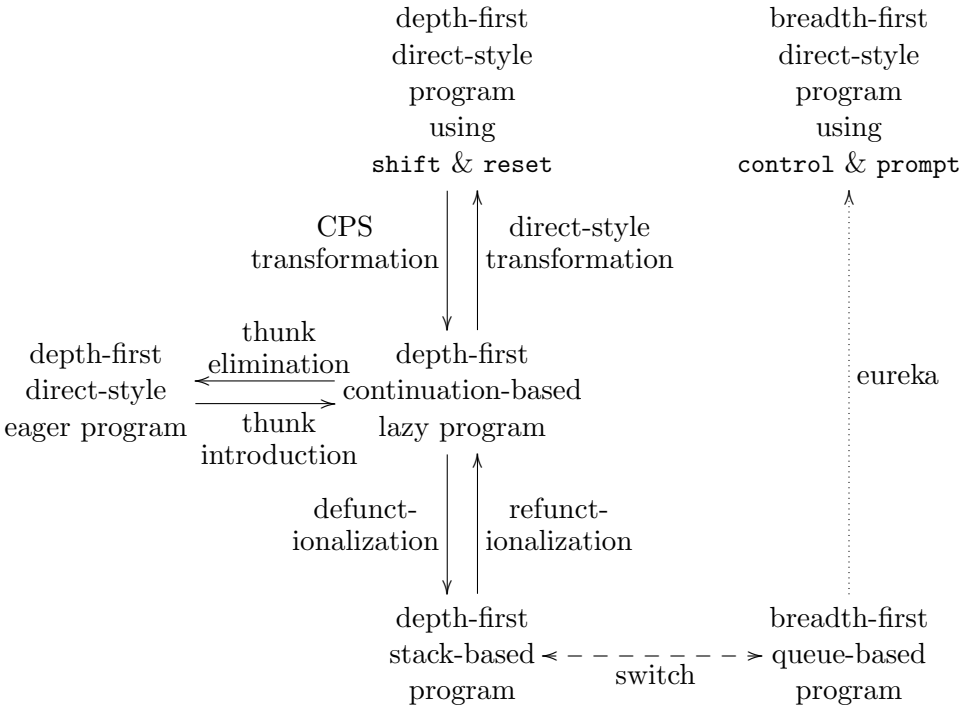
The CPS counterpart of the first ML expression above reads as follows:

```
let val k = fn v => let val k' = fn v' => v + v'
                    in 1
                    end
in 10 + (k 100)
end
```

No such simple functional encoding exists for the second and third ML expressions above [35].

9.3 Programming with delimited continuations

In Section 9.4, we present an array of solutions to the traditional samefringe example and to its breadth-first counterpart. In Section 9.5, we present an array of solutions to Okasaki's breadth-first numbering pearl and to its depth-first counterpart. In both sections, the presentation is structured according to the following diagram:



- Our starting point here is a direct-style eager program (left side of the diagram). We can make this program lazy by using thunks, i.e., functions of type `unit -> 'a` (center of the diagram).
- We can then defunctionalize the thunks in the lazy program, obtaining a stack-based program (bottom center of the diagram).

- Alternatively, we can view the type `unit -> 'a` not as a functional device to implement laziness but as a delimited continuation. The lazy program is then, in actuality, a continuation-based one, and one that is the CPS counterpart of a direct-style program using `shift` and `reset` (top center of the diagram).
- The stack-based program (bottom center of the diagram) implements a depth-first traversal. Replacing the stack with a queue yields a program implementing a breadth-first traversal (bottom right of the diagram).
- By analogy with the rest of the diagram, we infer the direct-style program using `control` and `prompt` (top right of the diagram) from this queue-based program.

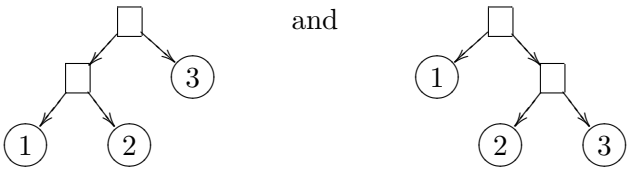
The three nodes in the center of the diagram—the CPS program, its direct-style counterpart, and its defunctionalized counterpart—follow the transformational tradition established in Reynolds's and Wand's seminal articles about continuations [196,243]. In particular the 'data-structure continuation' [243, page 179] of the depth-first program is a stack. By analogy, the data-structure continuation of the breadth-first program is a queue. We conjecture that the queue-based program could be mechanically obtained from the direct-style one by some kind of 'abstract CPS transformation' [93, 190], but fleshing out this conjecture falls out of the scope of the present article [35].

9.4 The samefringe problem

We present a spectrum of solutions to the traditional depth-first samefringe problem and its breadth-first counterpart. We work on Lisp-like binary trees of integers (S-expressions):

```
datatype tree = LEAF of int
              | NODE of tree * tree
```

The samefringe problem is traditionally stated as follows. Given two trees of integers, one wants to know whether they have the same sequence of leaves when read from left to right. For example, the two trees `NODE (NODE (LEAF 1, LEAF 2), LEAF 3)` and `NODE (LEAF 1, NODE (LEAF 2, LEAF 3))` have the same fringe `[1, 2, 3]` (representing it as a list) even though they are shaped differently:



Computing a fringe is done by traversing a tree depth-first and from left to right.

By analogy, we also address the breadth-first counterpart of the samefringe problem. Given two trees of integers, we want to know whether they have the

same fringe when traversed in left-to-right breadth-first order. For example, the breadth-first fringe of the left tree just above is `[3, 1, 2]` and that of the right tree just above is `[1, 2, 3]`.

We express the samefringe function generically by abstracting the representation of sequences of leaves with a data type `sequence` and a notion of computation (to compute the next element in a sequence):

```
signature GENERATOR
= sig
    type 'a computation
    datatype sequence = END
                      | NEXT of int * sequence computation

    val make_sequence : tree -> sequence
    val compute : sequence computation -> sequence
  end
```

The following functor maps a representation of sequences of leaves to a structure containing the samefringe function. Given two trees, `same_fringe` maps them into two sequences of integers (with `make_sequence`) and iteratively traverses these sequences with an auxiliary `loop` function. This function stops as soon as one of the two sequences is exhausted or two differing leaves are found:

```
functor make_Same_Fringe (structure G : GENERATOR)
= struct
    (*  same_fringe : tree * tree -> bool  *)
    fun same_fringe (t1, t2)
        = let (*  loop : G.sequence * G.sequence -> bool  *)
              fun loop (G.END, G.END)
                  = true
                | loop (G.NEXT (i1, a1), G.NEXT (i2, a2))
                  = i1 = i2
                    andalso loop (G.compute a1, G.compute a2)
                | loop _
                  = false
          in loop (G.make_sequence t1, G.make_sequence t2)
          end
  end
```

In the remainder of this section, we review a variety of generators.

9.4.1 Depth first

9.4.1.1 An eager traversal

The simplest solution is to represent sequences as a data type isomorphic to that of lists. To this end, we define `make_sequence` as an accumulator-based flatten function:

```
structure Eager_generator : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = 'a

    (*  visit : tree * sequence computation -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = visit (t1, visit (t2, a))

    fun make_sequence t
        = visit (t, END)

    fun compute value
        = value
  end
```

In this solution, the sequence of leaves is built eagerly and therefore completely before any comparison takes place. This choice is known to be inefficient because if two leaves differ, the remaining two sequences are not used and therefore did not need to be built.

9.4.1.2 A lazy traversal

A more efficient solution—and indeed a traditional motivation for lazy evaluation [104, 124]—is to construct the sequences lazily and to traverse them on demand. In the following generator, the data type `sequence` implements lazy sequences; the construction of the rest of the lazy sequence is delayed with a thunk of type `unit -> sequence`; and `make_sequence` is defined as an accumulator-based flatten function:

```
structure Lazy_generator : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    (*  visit : tree * sequence computation -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = visit (t1, fn () => visit (t2, a))

    fun make_sequence t
        = visit (t, fn () => END)

    fun compute thunk
        = thunk ()
  end
```

Unlike in the eager solution, the construction of the sequence in `Lazy_generator` and the comparisons in `same_fringe` are interleaved. This choice is known to be more efficient because if two leaves differ, the remaining two sequences are not built at all.

9.4.1.3 A continuation-based traversal

Alternatively to viewing the thunk of type `unit -> sequence`, in the lazy traversal of Section 9.4.1.2, as a functional device to implement laziness, we can view it as a delimited continuation that is initialized in the initial call to `visit` in `make_sequence`, extended in the induction case of `visit`, captured in the base case of `visit`, and resumed in `compute`. From that viewpoint, the lazy traversal is also a continuation-based one.

9.4.1.4 A direct-style traversal with `shift` and `reset`

In direct style, the delimited continuation `a` of Section 9.4.1.3 is initialized with the control delimiter `reset`, extended by functional sequencing, captured by the delimited-control operator `shift`, and resumed by function application.

Using Filinski's functor `Shift_and_Reset` defined in Appendix 9.A, one can therefore define the lazy generator in direct style as follows:

```
structure Lazy_generator_with_shift_and_reset : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    local structure SR = Shift_and_Reset
                           (type intermediate_answer = sequence)
    in val shift = SR.shift
       val reset = SR.reset
    end

    (*  visit : tree -> unit  *)
    fun visit (LEAF i)
        = shift (fn a => NEXT (i, a))
      | visit (NODE (t1, t2))
        = let val () = visit t1
          in visit t2
          end

    fun make_sequence t
        = reset (fn () => let val () = visit t
                          in END
                          end)

    fun compute thunk
        = thunk ()
  end
```

CPS-transforming `visit` and `make_sequence` yields the definitions of Section 9.4.1.2.

The key points of this CPS transformation are as follows:

- the clause

```
visit (NODE (t1, t2))
= let val () = visit t1
  in visit t2
  end
```

  is transformed into:

```
visit (NODE (t1, t2), a)
= visit (t1, fn () => visit (t2, a))
```

- the clause

```
visit (LEAF i)
= shift (fn a => NEXT (i, a))
```

  is transformed into:

```
visit (LEAF i, a)
= NEXT (i, a)
```

- and the expression

```
reset (fn () => let val () = visit t
                in END
                end)
```

  is transformed into:

```
visit (t, fn () => END)
```

9.4.1.5 A stack-based traversal

Alternatively to writing the lazy solution in direct style, we can defunctionalize its computation (which has type `sequence computation`, i.e., `unit -> sequence`) and obtain a first-order solution [74, 196]. The inhabitants of the function space `unit -> sequence` are instances of the function abstractions in the initial call to `visit` (i.e., `fn () => END`) and in the induction case of `visit` (i.e., `fn () => visit (t2, a)`). We therefore represent this function space by (1) a sum corresponding to these two possibilities, and (2) the corresponding apply function, `continue`, to interpret each of the summands. We represent this sum with an ML data type, which is recursive because of the recursive call to `visit`. This data type is isomorphic to that of a list of subtrees, which we use for simplicity in the code below. The result is essentially McCarthy's solution [166]:

```
structure Lazy_generator_stack_based : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = tree list

    (*  visit : tree * tree list -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = visit (t1, t2 :: a)
    (*  continue : tree list * unit -> sequence  *)
    and continue (nil, ())
        = END
      | continue (t :: a, ())
        = visit (t, a)

    fun make_sequence t
        = visit (t, nil)

    fun compute a
        = continue (a, ())
  end
```

This solution traverses a given tree incrementally by keeping a stack of its subtrees. To make this point more explicit, and as a stepping stone towards breadth-first traversal, let us fold the definition of `continue` in the induction case of `visit` so that `visit` always calls `continue`:

```
      | visit (NODE (t1, t2), a)
        = continue (t1 :: t2 :: a, ())
```

(Unfolding the call to `continue` gives back the definition above.)

We now clearly have a stack-based definition of depth-first traversal, and furthermore we have shown that this stack corresponds to the continuation of a function implementing a recursive descent. (Such a stack is referred to as a 'data-structure continuation' in the literature [243, page 179].)

9.4.2 Breadth first

9.4.2.1 A queue-based traversal

Replacing the (last-in, first-out) stack, in the definition of Section 9.4.1.5, by a (first-in, first-out) queue yields a definition that implements breadth-first, rather than depth-first, traversal:

```
structure Lazy_generator_queue_based : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = tree list

    (*  visit : tree * tree list -> sequence  *)
    fun visit (LEAF i, a)
        = NEXT (i, a)
      | visit (NODE (t1, t2), a)
        = continue (a @ [t1, t2], ())

    (*  continue : tree list * unit -> sequence  *)
    and continue (nil, ())
        = END
      | continue (t :: a, ())
        = visit (t, a)

    fun make_sequence t
        = visit (t, nil)

    fun compute a
        = continue (a, ())
  end
```

In contrast to Section 9.4.1.5, where the clause for nodes was (essentially) concatenating the two subtrees in front of the list of subtrees:

```
      | visit (NODE (t1, t2), a)
        = continue ([t1, t2] @ a, ())   (* then *)
```

the clause for nodes is concatenating the two subtrees in the back of the list of subtrees:

```
      | visit (NODE (t1, t2), a)
        = continue (a @ [t1, t2], ())   (* now *)
```

Nothing else changes in the definition of the generator. In particular, subtrees are still removed from the front of the list of subtrees by `continue`. With this last-in, first-out policy, the generator yields a sequence in breadth-first order.

Because the `::`-constructors of the list of subtrees are not solely consumed by `continue` but also by `@`, this definition *is not in the range of defunctionalization* [74]. Therefore, even though `visit` is tail-recursive and constructs a data structure that is interpreted in `continue`, it does not correspond to a continuation-passing function. And indeed, traversing an inductive data structure breadth-first does not mesh well with compositional recursive descent: how would one write a breadth-first traversal with a fold function?

9.4.2.2 A direct-style traversal with `control` and `prompt`

The critical operation in the definition of `visit`, in Section 9.4.2.1, is the enqueuing of the subtrees `t1` and `t2` to the current queue `a`, which is achieved by the list concatenation `a @ [t1, t2]`. We observe that this concatenation matches the concatenation of stack frames in the specification of `control` in Section 9.2.2.

Therefore—and this is a eureka step—one can write `visit` in direct style using `control` and `prompt`. To this end, we represent both queues `a` and `[t1, t2]` as dynamic delimited continuations in such a way that their composition represents the concatenation of `a` and `[t1, t2]`. The direct-style traversal reads as follows:

```
structure Lazy_generator_with_control_and_prompt : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    local structure CP = Control_and_Prompt
                           (type intermediate_answer = sequence)
    in val control = CP.control
       val prompt = CP.prompt
    end

    (*  visit : tree -> unit  *)
    fun visit (LEAF i)
        = control (fn a => NEXT (i, a))
      | visit (NODE (t1, t2))
        = control (fn a => let val END = a ()
                               val () = visit t1
                               val () = visit t2
                           in END
                           end)

    fun make_sequence t
        = prompt (fn () => let val () = visit t
                           in END
                           end)

    fun compute a = prompt (fn () => a ())
  end
```

In the induction case, the current delimited continuation (representing the current control queue) is captured, bound to `a`, and applied to `()`. The implicit continuation of this application visits `t1` and then `t2`, and therefore represents the queue `[t1, t2]`. Applying `a` seals it to the implicit continuation so that any continuation captured by a subsequent recursive call to `visit` in `a` captures both the rest of `a` and the traversal of `t1` and `t2`, i.e., the rest of the new control queue.

9.4.3 Summary and conclusion

We first have presented a spectrum of solutions to the traditional depth-first samefringe problem. Except for the defunctionalized ones, all the solutions are compositional in the sense of denotational semantics (i.e., visiting each subtree is defined as the composition of visiting its own subtrees). The one using `shift` and `reset` is new. We believe that connecting the lazy solution with McCarthy's stack-based solution by defunctionalization is new as well.

By replacing the stack with a queue in the stack-based program, we have then obtained a solution to the breadth-first counterpart of the samefringe problem. Viewing this queue as a 'data-structure continuation,' we have observed that the operations upon it correspond to the operations induced by the composition of a dynamic delimited continuation and the current (delimited) continuation. We have then written this program compositionally and in direct style using `control` and `prompt`.

In the induction clause of `visit` in Section 9.4.2.2, if we returned *after* visiting `t1` and `t2` instead of before,

```
| visit (NODE (t1, t2))
  = control (fn a => let val () = visit t1
                         val () = visit t2
                     in a ()
                     end)
```

we would obtain depth-first traversal. This modified clause can be simplified into

```
| visit (NODE (t1, t2))
  = let val () = visit t1
    in visit t2
    end
```

which coincides with the corresponding clause in Section 9.4.1.4. The resulting pattern of use of `control` and `prompt` in the modified definition is the traditional one used to simulate `shift` and `reset` [34].

It is therefore simple to program depth-first traversal with `control` and `prompt`. But conversely, obtaining a breadth-first traversal using `shift` and `reset` would require a far less simple encoding of `control` and `prompt` in terms of `shift` and `reset`, such as those discussed in Section 9.2.4.

9.5 Numbering a tree

We now turn to Okasaki's problem of numbering a tree in breadth-first order with successive numbers [182]. We express it in direct style with `control` and `prompt`, and we then outline its depth-first counterpart. Okasaki considers fully-labeled binary trees:

```
datatype tree = LEAF of int
              | NODE of tree * int * tree
```

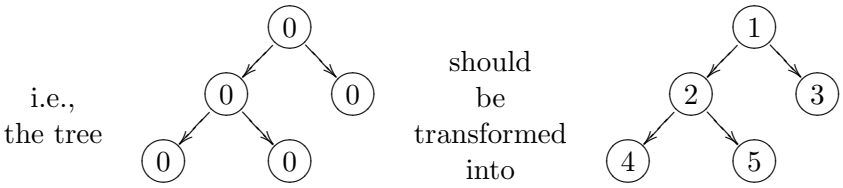
9.5.1 Breadth-first numbering

Given a tree T containing $|T|$ labels, we want to create a new tree of the same shape, but with the values in the nodes and leaves replaced by the numbers $1 \ldots |T|$ in breadth-first order. For example, the tree

```
NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)
```

contains 5 labels and should be transformed into

```
NODE (NODE (LEAF 4, 2, LEAF 5), 1, LEAF 3)
```



9.5.1.1 A queue-based traversal

In his solution [182], Okasaki relabels a tree by mapping it recursively into a first-in, first-out list of subtrees at call time and constructing the result at return time by reading this queue. To this end, he needs an auxiliary function

```
last_two_and_before : int list -> int list * int * int
```

such that applying it to the list `[xn, ..., x3, x2, x1]` yields the triple `([xn, ..., x3], x2, x1)`.

Okasaki's solution reads as follows:

```
(*  breadth_first_label : tree -> tree  *)
fun breadth_first_label t
    = let (*  visit : tree * int * tree list -> tree list  *)
          fun visit (LEAF _, i, k)
              = (LEAF i) :: (continue (k, i+1))
            | visit (NODE (t1, _, t2), i, k)
              = let val (rest, t1', t2')
                        = last_two_and_before
                            (continue (k @ [t1, t2], i+1))
                in (NODE (t1', i, t2')) :: rest
                end
          (*  continue : tree list * int -> tree list  *)
          and continue (nil, _)
              = nil
            | continue (t :: k, i)
              = visit (t, i, k)
      in last (visit (t, 1, nil))
      end
```

where `last` is a function mapping a non-empty list to its last element.

The above algorithm uses two queues of trees:

- the input queue, with function `visit` processing its front element, and with function `continue` processing its tail, and
- the output backwards queue, which is enqueued in both clauses of function `visit`, and which is dequeued by functions `last_two_and_before` and `last`.

9.5.1.2 A direct-style traversal with `control` and `prompt`

As in Section 9.4.2.2, we observe that the concatenation, in the definition of `visit` just above, matches the concatenation of stack frames in the specification of `control` in Section 9.2.2. One can therefore write the above function in direct style, using `control` and `prompt`. However, the solution requires a change of representation of the intermediate answer type of delimited continuations, i.e., the output queue, from `tree list` to `tree list * int` in order to unify the type `int` of the threaded index and the type `tree list` of the computation.

The direct-style breadth-first numbering program reads as follows:

```
local structure CP = Control_and_Prompt
                     (type intermediate_answer = tree list * int)
in val control = CP.control
   val prompt = CP.prompt
end

(*  breadth_first_label' : tree -> tree  *)
fun breadth_first_label' t
    = let (*  visit : tree * int -> int  *)
          fun visit (LEAF _, i)
              = control
                 (fn k =>
                   let val (ts, i') = prompt (fn () => k (i+1))
                   in ((LEAF i) :: ts, i')
                   end)
            | visit (NODE (t1, _, t2), i)
              = control
                 (fn k =>
                   let val (ts, i')
                           = prompt
                              (fn () =>
                                let val (nil, i1) = k (i+1)
                                    val i2 = visit (t1, i1)
                                    val i3 = visit (t2, i2)
                                in (nil, i3)
                                end)
                       val (rest, t1', t2')
                           = last_two_and_before ts
                   in ((NODE (t1', i, t2')) :: rest, i')
                   end)
      in last (#1 (prompt (fn () => let val i = visit (t, 1)
                                    in (nil, i)
                                    end)))
      end
```

Again, the queuing effect is obtained in the induction case, where the current delimited continuation (of `visit`) is captured, bound to `k`, and applied to the increased index `i+1`. The implicit continuation of this application visits `t1` and then `t2`. Applying `k` seals it to the implicit continuation so that any continuation captured by an ulterior recursive call to `visit` in `k` captures both the rest of `k` and the visit of `t1` and `t2`.

In the program above, before the last leaf in the tree is visited, the intermediate results represent the current value of the index. After the last leaf in the tree is visited, the intermediate results represent the current output queue. Therefore, we need to fix the intermediate answer type to `tree list * int` so that the intermediate results are represented as pairs, where, depending on the stage of the computation, one of the components contains significant information. Before the last leaf in the tree is visited, the significant information (i.e., the index) is contained only in the second component, and the first component is irrelevant and always equal to `nil`. After the last leaf in the tree is visited, the significant information (i.e., the output queue) is contained only in the first component, and the second component is irrelevant and always equal to $|T|+1$ (where T is the input tree and $|T|$ is the number of its labels).

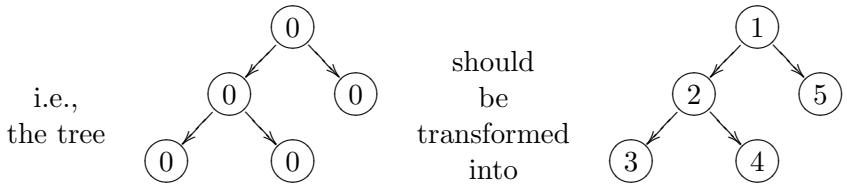
9.5.2 Depth-first numbering

We now turn to the depth-first counterpart of Okasaki's pearl, and present a spectrum of solutions to the problem of depth-first tree numbering. Given a tree T containing $|T|$ labels, we want to create a new tree of the same shape, but with the values in the nodes and leaves replaced by the numbers $1 \ldots |T|$ in depth-first order. For example, the tree

```
NODE (NODE (LEAF 0, 0, LEAF 0), 0, LEAF 0)
```

should be transformed into

```
NODE (NODE (LEAF 3, 2, LEAF 4), 1, LEAF 5)
```



9.5.2.1 A stack-based traversal

It is trivial to write the depth-first counterpart of Okasaki's solution: one should just replace the queue with a stack, and instead of using `last_two_and_before`, use the auxiliary function

```
first_two_and_after : int list -> int * int * int list
```

such that applying it to the list `[x1, x2, x3, ..., xn]` yields the triple `(x1, x2, [x3, ..., xn])`.

The depth-first solution reads as follows:

```
(*  depth_first_label : tree -> tree  *)
fun depth_first_label t
    = let (*  visit :  tree * int * tree list -> tree list  *)
          fun visit (LEAF _, i, ts)
              = (LEAF i) :: (continue (ts, i+1))
            | visit (NODE (t1, _, t2), i, ts)
              = let val (t1', t2', rest)
                        = first_two_and_after
                            (continue (t1 :: t2 :: ts, i+1))
                in (NODE (t1', i, t2')) :: rest
                end
          (*  continue : tree list * int -> tree list  *)
          and continue (nil, _)
              = nil
            | continue (t :: k, i)
              = visit (t, i, k)
      in hd (visit (t, 1, nil))
      end
```

In contrast to Section 9.5.1.1, where the clause for nodes was concatenating the two subtrees in the back of the list of subtrees, in a first-in, first-out fashion,

```
last_two_and_before
 (continue (k @ [t1, t2], i+1))    (* then *)
```

the clause for nodes is (essentially) concatenating the two subtrees in front of the list of subtrees, in a last-in, first-out fashion:

```
first_two_and_after
 (continue ([t1, t2] @ ts, i+1))   (* now *)
```

We can see that the algorithm uses two stacks of trees:

- the input stack, with function `visit` processing its top element, and with function `continue` processing its tail, and
- the output stack, which is pushed on in both clauses of function `visit`, and which is popped off by functions `first_two_and_after` and `hd`.

9.5.2.2 A continuation-based traversal

In the induction case of `visit`, let us unfold the call to `continue` to obtain the following clause:

```
| visit (NODE (t1, _, t2), i, ts)
  = let val (t1', t2', rest)
            = first_two_and_after
               (visit (t1, i+1, t2 :: ts))
    in (NODE (t1', i, t2')) :: rest
    end
```

The modified definition is in defunctionalized form: the data type is that of lists and `continue` is the corresponding apply function. The higher-order counterpart of this defunctionalized definition reads as follows:

```
(*  depth_first_label' : tree -> tree  *)
fun depth_first_label' t
    = let (*  visit : tree * int * (int -> tree list) -> tree list  *)
          fun visit (LEAF _, i, k)
              = (LEAF i) :: (k (i+1))
            | visit (NODE (t1, _, t2), i, k)
              = let val (t1', t2', rest)
                        = first_two_and_after
                           (visit (t1, i+1, fn i' =>
                                             visit (t2, i', k)))
                in (NODE (t1', i, t2')) :: rest
                end
      in hd (visit (t, 1, fn i => nil))
      end
```

9.5.2.3 A direct-style traversal with `shift` and `reset`

We view the function of type `int -> tree list`, in the definition just above, as a delimited continuation. This delimited continuation is initialized in the initial call to `visit`, extended in the induction case, and captured and resumed in both clauses of `visit`. In direct style, the initialization is obtained with `reset`, the extension is obtained by functional sequencing, the capture is obtained with `shift`, and the activation is obtained by function application. The result is another new example of programming with static delimited-control operators:

```
local structure SR = Shift_and_Reset
                     (type intermediate_answer = tree list)
in val shift = SR.shift
   val reset = SR.reset
end
```

```
(*  depth_first_label'' : tree -> tree  *)
fun depth_first_label'' t
    = let (*  visit :  tree * int -> tree list  *)
          fun visit (LEAF _, i)
              = shift
                 (fn k =>
                   (LEAF i) :: (k (i+1)))
            | visit (NODE (t1, _, t2), i)
              = shift
                 (fn k =>
                   let val (t1', t2', rest)
                           = first_two_and_after
                              (reset
                                (fn () => k (let val i'
                                                 = visit (t1, i+1)
                                             in visit (t2, i')
                                             end)))
                   in (NODE (t1', i, t2')) :: rest
                   end)
      in hd (reset (fn () => let val i = visit (t, 1)
                             in nil
                             end))
      end
```

CPS-transforming `visit` yields the definition of Section 9.5.2.2.

9.5.3 Summary and conclusion

Okasaki's solution relabels its input tree in breadth-first order and uses a queue. We have expressed it in direct style using `control` and `prompt`. In so doing, we have internalized the explicit data operations on the queue into implicit control operations. These control operations crucially involve delimited continuations whose extent is dynamic.

The stack-based counterpart of Okasaki's solution relabels its input tree in depth-first order. We have mechanically refunctionalized this program into another one, which is continuation-based, and we have expressed this continuation-based program in direct style using `shift` and `reset`. These control operators crucially involve delimited continuations whose extent is static.

9.6 Conclusion and issues

Over the last 15 years, it has been repeatedly claimed that `control` has more expressive power than `shift`. Even though this claim is now disproved [35, 146, 209], it is still unclear how to program with `control`-like dynamic delimited continuations. In fact, in 15 years, only toy examples have been advanced to illustrate the difference between static and dynamic delimited continuations, such as the one in Section 9.2.5.

In this article, we have filled this vacuum by using dynamic delimited continuations to program breadth-first traversal. We have accounted for the dynamic queuing mechanism inherent to breadth-first traversal with the dynamic concatenation of stack frames that is specific to `control` and that makes it go

beyond what is traditionally agreed upon as being continuation-passing style (CPS). We have presented two examples of breadth-first traversal: the breadth-first counterpart of the traditional samefringe function and Okasaki's breadth-first numbering pearl. We have recently proposed yet another example that exhibits the difference between `shift` and `control` [29, Section 4.6] [34, page 5].

One lesson we have learned here is how helpless one can feel when going beyond CPS. Unlike with `shift` and `reset`, there is no infrastructure for transforming programs that use `control` and `prompt`. We have therefore relied on CPS and on defunctionalization as guidelines, and we have built on the vision of data-structure continuations (stacks for depth-first traversals and queues for breadth-first traversals) proposed by Friedman 25 years ago [243, page 179] to infer the breadth-first traversals. We would have been hard pressed to come up with these examples only by groping for delimited continuations in direct style.[1]

Since `control`, even more dynamic delimited-control operators (some of which generate control delimiters dynamically) have been proposed [84, 111, 125, 173, 178, 191], all of which go beyond CPS but only two of which, to the best of our knowledge, come with motivating examples illustrating their specificity:

- In his PhD thesis [15], Balat uses the extra expressive power of Gunter, Rémy, and Riecke's control operators `set` and `cupto` over that of `shift` and `reset` to prototype a type-directed partial evaluator for the lambda-calculus with sums [16, 17].
- In his PhD thesis [178], Nanevski introduces two new dynamic delimited-control operators, `mark` and `recall`, and illustrates them with a function partitioning a natural number into the lists of natural numbers that add to it. He considers both depth-first and breadth-first generation strategies, and conjectures that the latter cannot be written using `shift` and `reset`. As such, his is our closest related work.

These applications are rare and so far they tend to be daunting. Dynamic delimited continuations need simpler examples, more reasoning tools, and more program transformations.

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Andrzej Filinski, Julia Lawall, Kevin Millikin, and the anonymous referees of Information Processing Letters and Science of Computer Programming for their comments.

The second author would also like to thank Andrzej Filinski, Mayer Goldberg, Julia Lawall, and Olin Shivers for participating to a one-week brain storm about continuations in August 2004, and to BRICS and DAIMI for hosting us during that week. Thanks are also due to Bernd Grobauer and Julia Lawall for sharing brain cycles about breadth-first numbering in Summer 2000.

The third author contributed to this work while at Harvard University.

[1] "You are not Superman." – Aunt May (2002)

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`), the Danish Natural Science Research Council, Grant no. 21-03-0545, and the United States National Science Foundation Grant no. BCS-0236592.

9.A An implementation of `shift` and `reset`

In his seminal article [96], Filinski has presented an ML implementation of `shift` and `reset` in terms of `callcc` and mutable state, along with its correctness proof. This implementation takes the form of a functor `Shift_and_Reset`, which maps a type of intermediate answers into a structure providing instances of `shift` and `reset` at that type:

```
signature ESCAPE
= sig
    type void
    val coerce : void -> 'a
    val escape : (('a -> void) -> 'a) -> 'a
  end

structure Escape : ESCAPE
= struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    local open SMLofNJ.Cont
    in fun escape f
           = callcc (fn k => f (fn x => throw k x))
    end
  end

signature SHIFT_AND_RESET
= sig
    type intermediate_answer
    val shift : (('a -> intermediate_answer) -> intermediate_answer)
                -> 'a
    val reset : (unit -> intermediate_answer) -> intermediate_answer
  end

functor Shift_and_Reset (type intermediate_answer) : SHIFT_AND_RESET
= struct
    open Escape

    exception MISSING_RESET

    val mk : (intermediate_answer -> void) ref
           = ref (fn _ => raise MISSING_RESET)

    fun abort x
        = coerce (!mk x)

    type intermediate_answer = intermediate_answer
```

```
    fun reset thunk
        = escape (fn k => let val m = !mk
                          in mk := (fn r => (mk := m; k r));
                             abort (thunk ())
                          end)

    fun shift function
        = escape
           (fn k => abort (function (fn v => reset
                                               (fn () => coerce (k v)))))
  end
```

9.B An implementation of control and prompt

The functor Control_and_Prompt maps a type of intermediate answers into a structure providing instances of control and prompt at that type:

```
signature CONTROL_AND_PROMPT
= sig
    type intermediate_answer
    val control : (('a -> intermediate_answer) -> intermediate_answer)
                  -> 'a
    val prompt : (unit -> intermediate_answer) -> intermediate_answer
  end

functor Control_and_Prompt (type intermediate_answer)
: CONTROL_AND_PROMPT
= struct
    datatype ('t, 'w) context'
             = CONTEXT of 't -> ('w, 'w) context' option -> 'w

    fun send v NONE
        = v
      | send v (SOME (CONTEXT mc))
        = mc v NONE

    fun compose' (CONTEXT c, NONE)
        = CONTEXT c
      | compose' (CONTEXT c, SOME mc1)
        = CONTEXT (fn v => fn mc2 => c v (SOME (compose' (mc1, mc2))))

    fun compose (CONTEXT c, NONE)
        = CONTEXT c
      | compose (CONTEXT c, SOME mc1)
        = CONTEXT (fn v => fn mc2 => c v (SOME (compose' (mc1, mc2))))

    structure SR
    = Shift_and_Reset
       (type intermediate_answer
         = (intermediate_answer, intermediate_answer) context' option
           -> intermediate_answer)
    val shift = SR.shift
    val reset = SR.reset
```

```
    type intermediate_answer = intermediate_answer

    fun prompt thunk
        = reset (fn () => send (thunk ())) NONE

    exception MISSING_PROMPT

    fun control function
        = shift
           (fn c1 =>
             fn mc1 =>
              let val k
                      = fn x =>
                         shift
                          (fn c2 =>
                            fn mc2 =>
                             let val (CONTEXT c1')
                                     = compose (CONTEXT c1, mc1)
                             in c1' x
                                    (SOME (compose (CONTEXT c2, mc2)))
                             end)
              in reset (fn () => send (function k)) NONE
              end) handle MISSING_RESET => raise MISSING_PROMPT
end
```

A delimited continuation captured by `control` may capture the context in which it is subsequently activated. To simulate this dynamic extent, the captured continuation (of type `('t, 'w) context'`) takes as arguments not just the value (of type `'t`) with which it is activated, but also the context (of type `('w, 'w) context' option`) in which it is activated. Hence the recursive definition of `datatype ('t, 'w) context'`.

Such a captured continuation can no longer be activated by mere function application; instead we define `send v c` to activate the captured continuation `c` with the value `v`. Such a captured continuation can also no longer be composed by mere function composition; instead we define `compose c mc` to concatenate the captured continuation `c` with the outer continuation (activation context) `mc`.

A direct transliteration of Shan's Scheme macros into ML results in an implementation with overly restrictive types. Due to the lack of polymorphic recursion in ML, the function `compose` would have the type:

```
('w, 'w) context' * ('w, 'w) context' option -> ('w, 'w) context'
```

and consequently, the inferred type of `control` would be:

```
((intermediate_answer -> intermediate_answer) -> intermediate_answer)
-> intermediate_answer
```

The third author has therefore cloned the function `compose` so that it has the following type:

```
('t, 'w) context' * ('w, 'w) context' option -> ('t, 'w) context'
```

Consequently, the inferred type of `control` is the same as that of `shift` in Filinski's implementation:

```
(('a -> intermediate_answer) -> intermediate_answer) -> 'a
```

Chapter 10

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

with Olivier Danvy and Kevin Millikin [35]

Abstract

We present a new abstract machine that accounts for dynamic delimited continuations. We prove the correctness of this new abstract machine with respect to a pre-existing, definitional abstract machine. Unlike this definitional abstract machine, the new abstract machine is in defunctionalized form, which makes it possible to state the corresponding higher-order evaluator. This evaluator is in continuation+state passing style and threads a trail of delimited continuations and a meta-continuation. Since this style accounts for dynamic delimited continuations, we refer to it as 'dynamic continuation-passing style.'

We show that the new machine operates more efficiently than the definitional one and that the notion of computation induced by the corresponding evaluator takes the form of a monad. We also present new examples and a new simulation of dynamic delimited continuations in terms of static ones.

10.1 Introduction

The control operator `call/cc` [46, 117, 143, 196], by now, is an accepted component in the landscape of eager functional programming, where it provides the expressive power of CPS (continuation-passing style) in direct-style programs. An integral part of its success is its surrounding array of computational artifacts: simple motivating examples as well as more complex ones, a functional encoding in the form of a continuation-passing evaluator, the corresponding continuation-passing style and CPS transformation, their first-order counterparts (e.g., the corresponding abstract machine), and the continuation monad.

The delimited-control operators `control` (alias $\mathcal{F}$) and `prompt` (alias #) [87, 93, 213] were designed to go 'beyond continuations' [90]. This vision was investigated in the early 1990's [111, 125, 173, 191, 214] and today it is receiving renewed attention: Shan and Kiselyov are studying its simulation properties [146, 209], and Dybvig, Peyton Jones, and Sabry are proposing a general framework where

multiple control delimiters can coexist [84], on the basis of Hieb, Dybvig, and Anderson's earlier work on 'subcontinuations' [125].

We observe, though, that none of these recent works on `control` and `prompt` uses the entire array of artifacts that organically surrounds `call/cc`. Our goal here is to fill this vacuum.

This work: We present a new abstract machine that accounts for dynamic delimited continuations and that is in defunctionalized form [74, 196], and we prove its equivalence with a definitional abstract machine that is not in defunctionalized form. We also present the corresponding higher-order evaluator from which one can obtain the corresponding new CPS transformer. The resulting 'dynamic continuation-passing style' (dynamic CPS) threads a list of trailing delimited continuations, i.e., it is a continuation+state-passing style. This style is equivalent to, but simpler than the one recently proposed by Shan [209], and structurally related to the one recently proposed by Dybvig, Peyton Jones, and Sabry [84]. We also show that it corresponds to a computational monad, and we present some new examples.

Overview: We first present the definitional machine for dynamic delimited continuations in Section 10.2. We then present the new machine in Section 10.3, we establish their equivalence in Section 10.4, and we compare their efficiency in Section 10.5. The new machine is in defunctionalized form and we present the corresponding higher-order evaluator in Section 10.6. This evaluator is expressed in a dynamic continuation-passing style and we present the corresponding dynamic CPS transformer in Section 10.7 and the corresponding direct-style evaluator in Section 10.8. We illustrate dynamic continuation-passing style in Section 10.9 and in Section 10.10, we show that it can be characterized with a computational monad. In Section 10.11, we present a new simulation of `control` and `prompt` based on dynamic CPS. Finally, we address related work and conclude.

Prerequisites and notation: We assume some basic familiarity with operational semantics, abstract machines, eager functional programming in (Standard) ML, defunctionalization, and continuations.

10.2 The definitional abstract machine

In our earlier work [29], we obtained an abstract machine for the static delimited-control operators `shift` and `reset` by defunctionalizing a definitional evaluator that had two layered continuations [67, 68]. In this abstract machine, the first continuation takes the form of an evaluation context and the second takes the form of a stack of evaluation contexts. By construction, this abstract machine is an extension of Felleisen et al.'s CEK machine [89], which has one evaluation context and is itself a defunctionalized evaluator with one continuation [5, 7, 64, 196].

The abstract machine for static delimited continuations implements the application of a delimited continuation (represented as a captured context) by

pushing the current context onto the stack of contexts and installing the captured context as the new current context [29]. In contrast, the abstract machine for dynamic delimited continuations implements the application of a delimited continuation (also represented as a captured context) by concatenating the captured context to the current context [93]. As a result, static and dynamic delimited continuations differ because a subsequent control operation will capture either the remainder of the reinstated context (in the static case) or the remainder of the reinstated context together with the then-current context (in the dynamic case). An abstract machine implementing dynamic delimited continuations therefore requires defining an operation to concatenate contexts.

Figure 10.1 displays the definitional abstract machine for dynamic delimited continuations, including the operation to concatenate contexts. It only differs from our earlier abstract machine for static delimited continuations [29, Figure 7 and Section 4.5] in the way captured delimited continuations are applied, by concatenating their representation with the representation of the current continuation (the shaded transition in Figure 10.1).[1]

Contexts form a monoid:

Property 10.1 *The operation $\star$ defined in Figure 10.1 satisfies the following properties:*

1. $C_1 \star \mathsf{END} = C_1 = \mathsf{END} \star C_1$,
2. $(C_1 \star C_1') \star C_1'' = C_1 \star (C_1' \star C_1'')$.

Proof. By induction on the structure of C_1. □

In the definitional machine, the constructors of contexts are not solely consumed in the *cont₁* transitions, but also by $\star$. Therefore, the definitional abstract machine is not in the range of defunctionalization [74, 196]: it does not correspond to a higher-order evaluator. In the next section, we present a new abstract machine that implements dynamic delimited continuations and is in the range of defunctionalization.

10.3 The new abstract machine

The definitional machine is not in the range of defunctionalization because of the concatenation of contexts. We therefore introduce a new component in the machine to avoid this concatenation. This new component, the *trail of contexts*, holds the then-current contexts that would have been concatenated to the captured context in the definitional machine. These then-current contexts are then reinstated in turn when the captured context completes. Together, the current context and the trail of contexts represent the current dynamic

[1] In contrast, static delimited continuations are applied as follows:

$$\langle \mathsf{FUN}\,(C_1',\ C_1),\ v,\ C_2 \rangle_{cont_1} \quad \Rightarrow_{def} \quad \langle C_1',\ v,\ C_1 :: C_2 \rangle_{cont_1}$$

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- Values (closures and captured continuations): $v ::= [x,\, e,\, \rho] \mid C_1$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\, C_1) \mid \mathsf{FUN}\,(v,\, C_1)$
- Concatenation of contexts:

$$\begin{array}{rcl} \mathsf{END} \star C_1' & \stackrel{\text{def}}{=} & C_1' \\ (\mathsf{ARG}\,((e, \rho),\, C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{ARG}\,((e, \rho),\, C_1 \star C_1') \\ (\mathsf{FUN}\,(v,\, C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{FUN}\,(v,\, C_1 \star C_1') \end{array}$$

- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow_{def}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle C_1,\, \rho(x),\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\, C_1),\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{eval}$
$\langle \mathcal{F}k.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{eval}$
$\langle \mathsf{END},\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle \mathsf{ARG}\,((e, \rho),\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\, C_1),\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,(C_1',\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle C_1' \star C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle C_1 :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow_{def}$	$\langle C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow_{def}$	v

Figure 10.1: The definitional call-by-value abstract machine for the λ-calculus extended with $\mathcal{F}$ and $\#$

context. The final component of the machine holds a stack of dynamic contexts (represented as a list: nil denotes the empty list, the infix operator :: denotes list construction, and the infix operator @ denotes list concatenation, as in ML).

Figure 10.2 displays the new abstract machine for dynamic delimited continuations. It only differs from the definitional abstract machine in the way dynamic contexts are represented (a context and a trail of contexts (represented as a list) instead of one concatenated context). In Section 10.4, we establish the equivalence of the two machines.

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- Values (closures and captured continuations): $v ::= [x,\, e,\, \rho] \mid [C_1,\, T_1]$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\, C_1) \mid \mathsf{FUN}\,(v,\, C_1)$
- Trail of contexts: $T_1 ::= \mathsf{nil} \mid C_1 :: T_1$
- Meta-contexts: $C_2 ::= \mathsf{nil} \mid (C_1, T_1) :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow_{new}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle C_1,\, \rho(x),\, T_1,\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle C_1,\, [x,\, e,\, \rho],\, T_1,\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e,\, \rho,\, \mathsf{END},\, \mathsf{nil},\, (C_1, T_1) :: C_2\rangle_{eval}$
$\langle \mathcal{F}k.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e,\, \rho\{k \mapsto [C_1,\, T_1]\},\, \mathsf{END},\, \mathsf{nil},\, C_2\rangle_{eval}$
$\langle \mathsf{END},\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle T_1,\, v,\, C_2\rangle_{trail_1}$
$\langle \mathsf{ARG}\,((e, \rho),\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([C_1',\, T_1'],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle C_1',\, v,\, T_1' \mathbin{@} (C_1 :: T_1),\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v,\, C_2\rangle_{trail_1}$	$\Rightarrow_{new}$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle C_1 :: T_1,\, v,\, C_2\rangle_{trail_1}$	$\Rightarrow_{new}$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle (C_1, T_1) :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow_{new}$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow_{new}$	v

Figure 10.2: A new call-by-value abstract machine for the λ-calculus extended with $\mathcal{F}$ and $\#$

In the new machine, the constructors of contexts are solely consumed in the $cont_1$ transitions. Therefore the new machine, unlike the definitional machine, is in the range of defunctionalization: it can be refunctionalized into a higher-order evaluator, which we present in Section 10.6.

N.B.: The trail concatenation, in Figure 10.2, could be avoided by adding a new component to the machine—a meta-trail of pairs of contexts and trails, managed last-in, first-out—and the corresponding new transitions. A captured continuation would then be a triple of context, trail, and meta-trail, and applying it would require this meta-trail to be concatenated to the current trail. In turn, this concatenation could be avoided by adding a meta-meta-trail, etc.

Because each of the meta^n-trails (for $n \geq 1$) but the last one has one point of consumption, they all are in defunctionalized form except the last one. Adding meta^n-trails amounts to trading space for time.

10.4 Equivalence of the definitional machine and of the new machine

We relate the configurations and transitions of the definitional abstract machine to those of the new abstract machine. As a diacritical convention [169], we annotate the components, configurations, and transitions of the definitional machine with a hat ($\widehat{\cdot}$). In order to relate a dynamic context of the new machine (a context and a trail of contexts) to a context of the definitional machine, we convert it into a context of the new machine:

Definition 10.1 *We define an operation $\widehat{\star}$, concatenating a new context and a trail of new contexts, by induction on its second argument:*

$$C_1 \mathbin{\widehat{\star}} \mathsf{nil} \stackrel{\text{def}}{=} C_1$$
$$C_1 \mathbin{\widehat{\star}} (C_1' :: T_1) \stackrel{\text{def}}{=} C_1 \star (C_1' \mathbin{\widehat{\star}} T_1)$$

Property 10.2 $C_1 \mathbin{\widehat{\star}} (C_1' :: T_1) = (C_1 \star C_1') \mathbin{\widehat{\star}} T_1$,

Proof. Follows from Definition 10.1 and from the associativity of $\star$ (Proposition 10.1(2)). □

Property 10.3 $(C_1 \mathbin{\widehat{\star}} T_1) \mathbin{\widehat{\star}} T_1' = C_1 \mathbin{\widehat{\star}} (T_1 \mathbin{@} T_1')$.

Proof. By induction on the structure of T_1. □

Definition 10.2 *We relate the definitional abstract machine and the new abstract machine with the following family of relations $\simeq$:*

1. *Terms: $\widehat{e} \simeq_{\text{e}} e$ iff $\widehat{e} = e$*
2. *Values:*
   (a) *$[\widehat{x}, \widehat{e}, \widehat{\rho}] \simeq_{\text{v}} [x, e, \rho]$ iff $\widehat{x} = x$, $\widehat{e} \simeq_{\text{e}} e$ and $\widehat{\rho} \simeq_{\text{env}} \rho$*
   (b) *$\widehat{C_1} \simeq_{\text{v}} [C_1, T_1]$ iff $\widehat{C_1} \simeq_{\text{c}} C_1 \mathbin{\widehat{\star}} T_1$*
3. *Environments:*
   (a) *$\widehat{\rho_{mt}} \simeq_{\text{env}} \rho_{mt}$*
   (b) *$\widehat{\rho}\{x \mapsto \widehat{v}\} \simeq_{\text{env}} \rho\{x \mapsto v\}$ iff $\widehat{v} \simeq_{\text{v}} v$ and $\widehat{\rho} \setminus \{x\} \simeq_{\text{env}} \rho \setminus \{x\}$, where $\rho \setminus \{x\}$ denotes the restriction of ρ to its domain excluding x*

4. *Contexts:*
   (a) $\widehat{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END}$
   (b) $\widehat{\mathsf{ARG}}\,((\widehat{e}, \widehat{\rho}),\ \widehat{C_1}) \simeq_{\mathrm{c}} \mathsf{ARG}\,((e, \rho),\ C_1)$ *iff* $\widehat{e} \simeq_{\mathrm{e}} e$, $\widehat{\rho} \simeq_{\mathrm{env}} \rho$, *and* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$
   (c) $\widehat{\mathsf{FUN}}\,(\widehat{v},\ \widehat{C_1}) \simeq_{\mathrm{c}} \mathsf{FUN}\,(v,\ C_1)$ *iff* $\widehat{v} \simeq_{\mathrm{v}} v$ *and* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$

5. *Meta-contexts:*
   (a) $\widehat{\mathsf{nil}} \simeq_{\mathrm{mc}} \mathsf{nil}$
   (b) $\widehat{C_1} :: \widehat{C_2} \simeq_{\mathrm{mc}} (C_1, T_1) :: C_2$ *iff* $\widehat{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$ *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$

6. *Configurations:*
   (a) $\langle \widehat{e}, \widehat{\rho}, \widehat{C_1}, \widehat{C_2} \rangle_{\widehat{eval}} \simeq \langle e, \rho, C_1, T_1, C_2 \rangle_{eval}$ *iff*
   $\widehat{e} \simeq_{\mathrm{e}} e$, $\widehat{\rho} \simeq_{\mathrm{env}} \rho$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$, *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$
   (b) $\langle \widehat{C_1}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}} \simeq \langle C_1, v, T_1, C_2 \rangle_{cont_1}$ *iff*
   $\widehat{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$, $\widehat{v} \simeq_{\mathrm{v}} v$, *and* $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$
   (c) $\langle \widehat{C_2}, \widehat{v} \rangle_{\widehat{cont_2}} \simeq \langle C_2, v \rangle_{cont_2}$ *iff*
   $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$ *and* $\widehat{v} \simeq_{\mathrm{v}} v$

By writing $\delta \Rightarrow^* \delta'$, $\delta \Rightarrow^+ \delta'$ and $\delta \Rightarrow^1 \delta'$, we mean that there is respectively zero or more, one or more, and at most one transition leading from the configuration δ to the configuration δ'.

Definition 10.3 *The partial evaluation functions* $eval_{def}$ *and* $eval_{new}$ *mapping terms to values are defined as follows:*

1. $eval_{def}\,(e) = v$ *if and only if* $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil} \rangle_{eval} \Rightarrow^+_{def} \langle \mathsf{nil}, v \rangle_{cont_2}$,

2. $eval_{new}\,(e) = v$ *if and only if* $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}, \mathsf{nil} \rangle_{eval} \Rightarrow^+_{new} \langle \mathsf{nil}, v \rangle_{cont_2}$.

We want to prove that $eval_{def}$ and $eval_{new}$ are defined on the same programs (i.e., closed terms), and that for any given program, they yield equivalent values.

Theorem 10.1 (Equivalence) *For any program* e, $eval_{def}\,(e) = \widehat{v}$ *if and only if* $eval_{new}\,(e) = v$ *and* $\widehat{v} \simeq_{\mathrm{v}} v$.

Proving Theorem 10.1 requires proving the following lemmas.

Lemma 10.1 *If* $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ *and* $\widehat{C_1'} \simeq_{\mathrm{c}} C_1'$ *then* $\widehat{C_1} \,\widehat{\star}\, \widehat{C_1'} \simeq_{\mathrm{c}} C_1 \star C_1'$.

Proof. By induction on the structure of $\widehat{C_1}$. □

The following lemma addresses the configurations of the new abstract machine that break the one-to-one correspondence with the definitional abstract machine.

Lemma 10.2 *Let* $\delta = \langle \mathsf{END}, v, T_1, C_2 \rangle_{cont_1}$.

1. *If* $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: \mathsf{nil}$, *where* $n \geq 0$, *then* $\delta \Rightarrow^{+}_{new} \langle C_2, v \rangle_{cont_2}$.
2. *If* $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: C_1 :: T_1'$, *where* $n \geq 0$ *and* $C_1 \neq \mathsf{END}$, *then* $\delta \Rightarrow^{+}_{new} \langle C_1, v, T_1', C_2 \rangle_{cont_1}$.

Proof. By induction on n. □

The following key lemma relates single transitions of the two abstract machines.

Lemma 10.3 *If* $\widehat{\delta} \simeq \delta$ *then*

1. *if* $\widehat{\delta} \Rightarrow_{def} \widehat{\delta'}$ *then there exists a configuration* δ' *such that* $\delta \Rightarrow^{+}_{new} \delta'$ *and* $\widehat{\delta'} \simeq \delta'$;
2. *if* $\delta \Rightarrow_{new} \delta'$ *then there exist configurations* $\widehat{\delta'}$ *and* δ'' *such that* $\widehat{\delta} \Rightarrow^{1}_{def} \widehat{\delta'}$, $\delta' \Rightarrow^{*}_{new} \delta''$ *and* $\widehat{\delta'} \simeq \delta''$.

Proof. By case analysis of $\widehat{\delta} \simeq \delta$. Most of the cases follow directly from the definition of the relation $\simeq$. We show the proof of one such case:

Case: $\widehat{\delta} = \langle \widehat{x}, \widehat{\rho}, \widehat{C_1}, \widehat{C_2} \rangle_{\widehat{eval}}$ and $\delta = \langle x, \rho, C_1, T_1, C_2 \rangle_{eval}$.
From the definition of the definitional abstract machine, $\widehat{\delta} \Rightarrow_{def} \widehat{\delta'}$, where $\widehat{\delta'} = \langle \widehat{C_1}, \widehat{\rho}(\widehat{x}), \widehat{C_2} \rangle_{\widehat{cont_1}}$.
From the definition of the new abstract machine, $\delta \Rightarrow_{new} \delta'$, where $\delta' = \langle C_1, \rho(x), T_1, C_2 \rangle_{cont_1}$.
By assumption, $\widehat{\rho}(\widehat{x}) \simeq_{\mathrm{v}} \rho(x)$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$ and $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widehat{\delta'} \simeq \delta'$ and both directions of Lemma 10.3 are proved in this case.

There are only three interesting cases. One of them arises when a captured continuation is applied, and the remaining two explain why the two abstract machines do not operate in lockstep:

Case: $\widehat{\delta} = \langle \widehat{\mathsf{FUN}}\,(\widehat{C_1'}, \widehat{C_1}), \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}}$ and
$\delta = \langle \mathsf{FUN}\,([C_1', T_1'], C_1), v, T_1, C_2 \rangle_{cont_1}$
From the definition of the definitional abstract machine, $\widehat{\delta} \Rightarrow_{def} \widehat{\delta'}$, where $\widehat{\delta'} = \langle \widehat{C_1'} \mathbin{\widehat{\star}} \widehat{C_1}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}}$.
From the definition of the new abstract machine, $\delta \Rightarrow_{new} \delta'$, where $\delta' = \langle C_1', v, T_1' \mathbin{@} (C_1 :: T_1), C_2 \rangle_{cont_1}$.
By assumption, $\widehat{C_1'} \simeq_{\mathrm{c}} C_1' \mathbin{\widehat{\star}} T_1'$ and $\widehat{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$.
By Lemma 10.1, we have $\widehat{C_1'} \mathbin{\widehat{\star}} \widehat{C_1} \simeq_{\mathrm{c}} (C_1' \mathbin{\widehat{\star}} T_1') \star (C_1 \mathbin{\widehat{\star}} T_1)$.
By the definition of $\widehat{\star}$, $(C_1' \mathbin{\widehat{\star}} T_1') \star (C_1 \mathbin{\widehat{\star}} T_1) = (C_1' \mathbin{\widehat{\star}} T_1') \mathbin{\widehat{\star}} (C_1 :: T_1)$.
By Proposition 10.3, $(C_1' \mathbin{\widehat{\star}} T_1') \mathbin{\widehat{\star}} (C_1 :: T_1) = C_1' \mathbin{\widehat{\star}} (T_1' \mathbin{@} (C_1 :: T_1))$.
Since $\widehat{v} \simeq_{\mathrm{v}} v$ and $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$, we infer that $\widehat{\delta'} \simeq \delta'$ and both directions of Lemma 10.3 are proved in this case.

Case: $\widehat{\delta} = \langle \widehat{\mathsf{END}}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}}$ and $\delta = \langle \mathsf{END}, v, T_1, C_2 \rangle_{cont_1}$
From the definition of the definitional abstract machine, $\widehat{\delta} \Rightarrow_{def} \widehat{\delta'}$, where $\widehat{\delta'} = \langle \widehat{C_2}, \widehat{v} \rangle_{\widehat{cont_2}}$.
By the definition of $\simeq$, $\widehat{v} \simeq_{\mathrm{v}} v$, $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$, and $\widehat{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END} \,\widehat{\star}\, T_1$. Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \,\widehat{\star}\, T_1 = \mathsf{END}$, which is possible only when $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: \mathsf{nil}$ for some $n \geq 0$.
Then by Lemma 10.2(1), $\delta \Rightarrow^{+}_{new} \delta'$, where $\delta' = \langle C_2, v \rangle_{cont_2}$ and $\widehat{\delta'} \simeq \delta'$, and both directions of the lemma are proved in this case.

Case: $\widehat{\delta} = \langle \widehat{C_1}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}}$ and $\delta = \langle \mathsf{END}, v, T_1, C_2 \rangle_{cont_1}$, where $\widehat{C_1} \neq \widehat{\mathsf{END}}$.
By the definition of $\simeq$, $\widehat{v} \simeq_{\mathrm{v}} v$, $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$, and $\widehat{C_1} \simeq_{\mathrm{c}} \mathsf{END} \,\widehat{\star}\, T_1$. Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \,\widehat{\star}\, T_1 \neq \mathsf{END}$, which is possible only when $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: C_1 :: T_1'$ for some $n \geq 0$ and $C_1 \neq \mathsf{END}$. Then by Lemma 10.2(2), $\delta \Rightarrow^{+}_{new} \delta'$, where $\delta' = \langle C_1, v, T_1', C_2 \rangle_{cont_1}$, $C_1 \neq \mathsf{END}$, and since $\mathsf{END} \,\widehat{\star}\, T_1 = C_1 \,\widehat{\star}\, T_1'$, we have $\widehat{\delta} \simeq \delta'$. Therefore, we have proved part (2) of Lemma 10.3 and reduced the proof of part (1) to one of the trivial cases (not shown in the proof), where $\widehat{\delta} \simeq \delta'$.

□

Given the relation between single-step transitions of the two abstract machines, it is straightforward to generalize it to the relation between their multi-step transitions.

Lemma 10.4 *If $\widehat{\delta} \simeq \delta$ then*

1. *if $\widehat{\delta} \Rightarrow^{+}_{def} \widehat{\delta'}$ then there exists a configuration δ' such that $\delta \Rightarrow^{+}_{new} \delta'$ and $\widehat{\delta'} \simeq \delta'$;*
2. *if $\delta \Rightarrow^{+}_{new} \delta'$ then there exist configurations $\widehat{\delta'}$ and δ'' such that $\widehat{\delta} \Rightarrow^{*}_{def} \widehat{\delta'}$, $\delta' \Rightarrow^{*}_{new} \delta''$ and $\widehat{\delta'} \simeq \delta''$.*

Proof. Both directions follow from Lemma 10.3 by induction on the number of transitions. □

We are now in position to prove the equivalence theorem.
ProofProof of Theorem 10.1. The initial configuration of the definitional abstract machine, i.e., $\langle e, \widehat{\rho_{mt}}, \widehat{\mathsf{END}}, \widehat{\mathsf{nil}} \rangle_{\widehat{eval}}$, and that of the new abstract machine, i.e., $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}, \mathsf{nil} \rangle_{eval}$, are in the relation $\simeq$. Therefore, if the definitional abstract machine reaches the final configuration $\langle \widehat{\mathsf{nil}}, \widehat{v} \rangle_{\widehat{cont_2}}$, then by Lemma 10.4(1), there is a configuration δ' such that $\delta \Rightarrow^{+}_{new} \delta'$ and $\widehat{\delta'} \simeq \delta'$. By the definition of $\simeq$, δ' must be $\langle \mathsf{nil}, v \rangle_{cont_2}$, with $\widehat{v} \simeq_{\mathrm{v}} v$. The proof of the converse direction follows similar steps.

□

10.5 Efficiency issues

The new abstract machine implements the dynamic delimited control operators $\mathcal{F}$ and # more efficiently than the definitional abstract machine. The efficiency gain comes from allowing continuations to be implemented as lists of stack segments—which is generally agreed to be the most efficient implementation for first-class continuations [47, 62, 126]—without imposing a choice of representation on the stack segments.

In particular, when the definitional abstract machine applies a captured context C_1' in a current context C_1, the new context is $C_1' \star C_1$, and constructing it requires work proportional to the length of C_1'. In contrast, when the new abstract machine applies a captured context $[C_1', T_1']$ in a current context C_1 with a current trail of contexts T_1, the new trail is $T_1' \mathbin{@} (C_1 :: T_1)$, and constructing it requires work proportional to the number of contexts (i.e., stack segments) in T_1', independently of the length of each of these contexts. In the worst case, each context in the trail has length one and the new abstract machine does the same amount of work as the definitional machine. In all other cases it does less.

The following implementation of a list copy function (written in the syntax of ML) illustrates the situation:

```
fun list_copy1 xs
    = let fun visit nil
              = control (fn k => k nil)
            | visit (x :: xs)
              = x :: (visit xs)
      in prompt (fn () => visit xs)
      end
```

The initial call to `visit` is delimited by `prompt` (alias #), and in the base case, the (delimited) continuation is captured with `control` (alias $\mathcal{F}$). This delimited continuation is represented by a context whose size is proportional to the length of the list. In the definitional abstract machine, the entire context must be traversed and copied when invoked (i.e., immediately). In the new machine, only the (empty) trail of contexts is traversed and copied. Therefore, the definitional abstract machine does work proportional to the length of the input list, whereas the new abstract machine does the same work in constant time.

A small variation on the function above causes the definitional machine to perform an amount of work which is quadratic in the length of the input list, by copying contexts whose size is proportional to the length of the list on *every* recursive call:

```
fun list_copy2 xs
    = let fun visit nil
              = control (fn k => k nil)
            | visit (x :: xs)
              = x :: (control (fn k => k (visit xs)))
      in prompt (fn () => visit xs)
      end
```

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- Answers, meta-continuations, continuations, values, and trails of continuations:

$$\begin{array}{rcl} \mathsf{Ans} & = & \mathsf{Val} \\ \theta_2, k_2 \in \mathsf{Cont}_2 & = & \mathsf{Val} \to \mathsf{Ans} \\ \theta_1, k_1 \in \mathsf{Cont}_1 & = & \mathsf{Val} \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\ v \in \mathsf{Val} & = & \mathsf{Val} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\ t_1 \in \mathsf{Trail}_1 & = & \mathsf{List}(\mathsf{Cont}_1) \end{array}$$

- Initial meta-continuation: $\theta_2 = \lambda v.\, v$
- Initial continuation: $\theta_1 = \lambda(v, t_1, k_2).\ \mathit{case}\ t_1$
  $\mathit{of}\ \mathsf{nil} \Rightarrow k_2\, v$
  $\mid k_1 :: t_1' \Rightarrow k_1\, (v, t_1', k_2)$
- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans}$

$$\begin{array}{rcl} \mathsf{eval}_{\mathsf{dcps}}\,(x, \rho, k_1, t_1, k_2) & = & k_1\,(\rho(x), t_1, k_2) \\ \mathsf{eval}_{\mathsf{dcps}}\,(\lambda x.e, \rho, k_1, t_1, k_2) & = & k_1\,(\lambda(v, k_1, t_1, k_2).\, \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho\{x \mapsto v\}, k_1, t_1, k_2), t_1, k_2) \\ \mathsf{eval}_{\mathsf{dcps}}\,(e_0\, e_1, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e_0, \rho, \lambda(v_0, t_1, k_2).\, \mathsf{eval}_{\mathsf{dcps}}\,(e_1, \rho, \lambda(v_1, t_1, k_2).\, v_0\,(v_1, k_1, t_1, k_2), t_1, k_2), t_1, k_2) \\ \mathsf{eval}_{\mathsf{dcps}}\,(\#e, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho, \theta_1, \mathsf{nil}, \lambda v.\, k_1\,(v, t_1, k_2)) \\ \mathsf{eval}_{\mathsf{dcps}}\,(\mathcal{F}k.e, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho\{k \mapsto \lambda(v, k_1', t_1', k_2).\, k_1\,(v, t_1 \mathbin{@} (k_1' :: t_1'), k_2)\}, \theta_1, \mathsf{nil}, k_2) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \to \mathsf{Val}$

$$\mathsf{evaluate}_{\mathsf{dcps}}\,(e) = \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho_{mt}, \theta_1, \mathsf{nil}, \theta_2)$$

Figure 10.3: A call-by-value evaluator for the λ-calculus extended with $\mathcal{F}$ and $\#$

In contrast to this quadratic behavior, the new abstract machine performs an amount of work that is linear in the length of the input list since it performs a constant amount of work at each application of a continuation (i.e., once per recursive call).

Implementing the composition of delimited continuations by concatenating their representations incurs an overhead proportional to the size of one of the delimited continuations, and is therefore subject to pathological situations such as the one illustrated in this section.

10.6 The evaluator corresponding to the new abstract machine

The *raison d'être* of the new abstract machine is that it is in defunctionalized form. Refunctionalizing the contexts and meta-contexts of the new abstract machine yields the higher-order evaluator of Figure 10.3. This evaluator is expressed in a continuation+state-passing style where the state consists of a trail of continuations and a meta-continuation, and defunctionalizing it gives the abstract machine of Figure 10.2. Since this continuation+state-passing style came into being to account for dynamic delimited continuations, we refer to it as a 'dynamic continuation-passing style' (dynamic CPS).

10.7 The CPS transformer corresponding to the new evaluator

The dynamic CPS transformer corresponding to the evaluator of Figure 10.3 can be immediately obtained as the associated syntax-directed encoding into the term model of the meta-language (using fresh variables):

$$
\begin{aligned}
[\![x]\!] &= \lambda(k_1, t_1, k_2).k_1\,(x, t_1, k_2)\\
[\![\lambda x.e]\!] &= \lambda(k_1, t_1, k_2).k_1\,(\lambda(x, k_1, t_1, k_2).\,[\![e]\!]\,(k_1, t_1, k_2), t_1, k_2)\\
[\![e_0\,e_1]\!] &= \lambda(k_1, t_1, k_2).[\![e_0]\!]\,(\lambda(v_0, t_1, k_2).\,[\![e_1]\!]\,(\lambda(v_1, t_1, k_2).\,v_0\,(v_1, k_1, t_1, k_2),\\
&\qquad\qquad t_1, k_2), t_1, k_2)\\
[\![\#e]\!] &= \lambda(k_1, t_1, k_2).[\![e]\!]\,(\theta_1, \mathsf{nil}, \lambda v.\,k_1\,(v, t_1, k_2))\\
[\![\mathcal{F}k.e]\!] &= \lambda(k_1, t_1, k_2).\mathit{let}\ k = \lambda(v, k_1', t_1', k_2).\,k_1\,(v, t_1 \mathbin{@} (k_1' :: t_1'), k_2)\\
&\qquad\qquad \mathit{in}\ [\![e]\!]\,(\theta_1, \mathsf{nil}, k_2)
\end{aligned}
$$

It is straightforward to write a one-pass version of the dynamic CPS transformer [68], and we have implemented it. For example, transforming `list_copy1` (in Section 10.5) yields the following program, which we write in the syntax of ML:

```
datatype 'a cont1 = CONT1 of 'a * 'a trail1 * 'a cont2 -> 'a
withtype 'a trail1 = 'a cont1 list
     and 'a cont2 = 'a -> 'a

(*  theta1 : 'a * 'a trail1 * 'a cont2 -> 'a  *)
fun theta1 (v, nil, k2)
```

```
    = k2 v
  | theta1 (v, (CONT1 k1) :: t1, k2)
    = k1 (v, t1, k2)

(*  theta2 : 'a -> 'a  *)
fun theta2 v
    = v

(*  list_copy1_dcps : 'b list -> 'b list  *)
fun list_copy1_dcps xs
    = let (*  visit : 'b list * 'b list trail1 * 'b list cont2
                    -> 'b list  *)
          fun visit (nil, k1, t1, k2)
              = let fun k (v, k1', t1', k2)
                        = k1 (v, t1 @ ((CONT1 k1') :: t1'), k2)
                in k (nil, theta1, nil, k2)
                end
            | visit (x :: xs, k1, t1, k2)
              = visit (xs, fn (r, t1', k2')
                              => k1 (x :: r, t1', k2'), t1, k2)
      in visit (xs, theta1, nil, theta2)
      end
```

or again, unfolding the inner let expression:

```
fun list_copy1_dcps_simplified xs
    = let fun visit (nil, k1, t1, k2)
              = k1 (nil, t1 @ ((CONT1 theta1) :: nil), k2)
            | visit (x :: xs, k1, t1, k2)
              = visit (xs, fn (r, t1', k2')
                              => k1 (x :: r, t1', k2'), t1, k2)
      in visit (xs, theta1, nil, theta2)
      end
```

In our experience, out-of-the-box dynamic CPS programs are rarely enlightening the way normal CPS programs tend to be (at least after some practice). However, again in our experience, a combination of simplifications (e.g., inlining `k2` and `k_init` in the example just above) and defunctionalization often clarifies the intent and the behavior of the original direct-style program. We illustrate this point in Section 10.9.

10.8 The direct-style evaluator corresponding to the new evaluator

Figure 10.4 shows a direct-style evaluator for the λ-calculus extended with $\mathcal{F}$ and # written in a meta-language enriched with $\mathcal{F}$ and #. Transforming this direct-style evaluator into dynamic continuation-passing style, using the one-pass version of the dynamic CPS transformer of Section 10.7, yields the evaluator of Figure 10.3. This experiment is an adaptation of Danvy and Filinski's experiment [67], where a direct-style evaluator for the λ-calculus extended with shift and reset written in a meta-language extended with shift and

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- Values: $v \in \mathsf{Val} = \mathsf{Val} \to \mathsf{Val}$
- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \to \mathsf{Ans}$

$$\begin{aligned}
\mathsf{eval}_{\mathsf{ds}}\,(x, \rho) &= \rho(x) \\
\mathsf{eval}_{\mathsf{ds}}\,(\lambda x.e, \rho) &= \lambda v.\, \mathsf{eval}_{\mathsf{ds}}\,(e, \rho\{x \mapsto v\}) \\
\mathsf{eval}_{\mathsf{ds}}\,(e_0\, e_1, \rho) &= \mathsf{eval}_{\mathsf{ds}}\,(e_0, \rho)\,(\mathsf{eval}_{\mathsf{ds}}\,(e_1, \rho)) \\
\mathsf{eval}_{\mathsf{ds}}\,(\#e, \rho) &= \#(\mathsf{eval}_{\mathsf{ds}}\,(e, \rho)) \\
\mathsf{eval}_{\mathsf{ds}}\,(\mathcal{F}k.e, \rho) &= \mathcal{F}v.\mathsf{eval}_{\mathsf{ds}}\,(e, \rho\{k \mapsto v\})
\end{aligned}$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \to \mathsf{Val}$

$$\mathsf{evaluate}_{\mathsf{ds}}\,(e) = \mathsf{eval}_{\mathsf{ds}}\,(e, \rho_{mt})$$

Figure 10.4: A direct-style evaluator for the λ-calculus extended with $\mathcal{F}$ and $\#$

reset was CPS-transformed into the definitional interpreter for the λ-calculus extended with shift and reset. (In the same spirit, Danvy and Lawall have transformed into direct style a continuation-passing evaluator for the λ-calculus extended with callcc, obtaining a direct-style evaluator interpreting callcc with callcc [72].)

10.9 Static and dynamic continuation-passing style

Biernacka, Biernacki, and Danvy have recently presented the following simple example to contrast the effects of `shift` and of `control` [29, Section 4.5]. We write it below in ML, using Filinski's implementation of `shift` and `reset` [96], and using the implementation of `control` and `prompt` presented in Section 10.11. In both cases, the type of the intermediate answers is `int list`:

```
(*  foo : int list -> int list  *)
fun foo xs
    = let fun visit nil
              = nil
            | visit (x :: xs)
              = visit (shift (fn k => x :: (k xs)))
      in reset (fn () => visit xs)
      end

(*  bar : int list -> int list  *)
fun bar xs
    = let fun visit nil
              = nil
```

```
            | visit (x :: xs)
              = visit (control (fn k => x :: (k xs)))
      in prompt (fn () => visit xs)
      end
```

The two functions traverse their input list recursively, and construct an output list. They only differ in that to abstract the recursive call to `visit` into a delimited continuation, `foo` uses `shift` and `reset` whereas `bar` uses `control` and `prompt`. This seemingly minor difference has a major effect since it makes `foo` behave as a *list-copying* function and `bar` as a *list-reversing* function.

To illustrate this difference of behavior, Biernacka, Biernacki, and Danvy have used contexts and meta-contexts [29, Section 4.5], and Biernacki and Danvy have used an intuitive source representation of the successive contexts [34, Section 2.3]. In this section, we use static and dynamic continuation-passing style to illustrate the difference of behavior.

10.9.1 Static continuation-passing style

Applying the canonical CPS transformation for `shift` and `reset` [67] to the definition of `foo` yields the following purely functional program:

```
fun foo_scps xs
    = let fun visit (nil, k1, k2)
              = k1 (nil, k2)
            | visit (x :: xs, k1, k2)
              = let fun k (v, k1', k2')
                        = visit (v, k1, fn v => k1' (v, k2'))
                in k (xs, fn (v, k2) => k2 (x :: v), k2)
                end
      in visit (xs, fn (v, k2) => k2 v, fn v => v)
      end
```

Inlining `k`, `k1'`, and `k1` yields the following simpler program:

```
fun foo_scps_simplified xs
    = let fun visit (nil, k2)
              = k2 nil
            | visit (x :: xs, k2)
              = visit (xs, fn v => k2 (x :: v))
      in visit (xs, fn v => v)
      end
```

Defunctionalizing `k2` yields the following first-order program:

```
fun foo_scps_defunct xs
    = let fun visit (nil, k2)
              = continue (k2, nil)
            | visit (x :: xs, k2)
              = visit (xs, x :: k2)
          and continue (nil, v)
              = v
```

```
          | continue (x :: k2, v)
            = continue (k2, x :: v)
      in visit (xs, nil)
      end
```

These equivalent views make it clearer that the program copies its input list by first reversing it using the meta-continuation as an accumulator, and then by reversing the accumulator.

10.9.2 Dynamic continuation-passing style

Applying the dynamic CPS transformation for `control` and `prompt` (Section 10.7) to the definition of `bar` yields the following purely functional program:

```
datatype 'a cont1 = CONT1 of 'a * 'a trail1 * 'a cont2 -> 'a
withtype 'a trail1 = 'a cont1 list
     and 'a cont2 = 'a -> 'a

fun theta1 (v, nil, k2)
    = k2 v
  | theta1 (v, (CONT1 k1) :: t1, k2)
    = k1 (v, t1, k2)

fun theta2 v
    = v

fun bar_dcps xs
    = let fun visit (nil, k1, t1, k2)
              = k1 (nil, t1, k2)
            | visit (x :: xs, k1, t1, k2)
              = let fun k (v, k1', t1', k2)
                        = visit (v, k1, t1 @ (k1' :: t1'), k2)
                in k (xs, CONT1 (fn (v, t1, k2)
                                    => theta1 (x :: v, t1, k2)),
                      nil, k2)
                end
      in visit (xs, theta1, nil, theta2)
      end
```

Inlining `k`, `k1'`, `k1`, and `k2`, defunctionalizing the continuation into the ML `option` type, and using an auxiliary function `continue_aux` to interpret the trail, yields the following first-order program:

```
fun bar_dcps_defunct xs
    = let fun visit (nil, t1)
              = continue (NONE, nil, t1)
            | visit (x :: xs, t1)
              = visit (xs, t1 @ ((SOME x) :: nil))
          and continue (NONE, v, t1)
              = continue_aux (t1, v)
            | continue (SOME x, v, t1)
              = continue (NONE, x :: v, t1)
```

```
            and continue_aux (nil, v)
                = v
              | continue_aux (k1 :: t1, v)
                = continue (k1, v, t1)
      in visit (xs, nil)
      end
```

Further simplifications lead one to the following program:

```
fun bar_dcps_defunct_simplified xs
    = let fun visit (nil, t1)
              = continue_aux (t1, nil)
            | visit (x :: xs, t1)
              = visit (xs, t1 @ (x :: nil))
          and continue_aux (nil, v)
              = v
            | continue_aux (k1 :: t1, v)
              = continue_aux (t1, k1 :: v)
      in visit (xs, nil)
      end
```

These equivalent views make it clearer that the program reverses its input list by first copying it to the trail through a series of concatenations, and then by reversing the trail.

10.9.3 A generalization

Let us briefly generalize the programming pattern above from lists to binary trees:

```
datatype tree = EMPTY
              | NODE of tree * int * tree
```

In the following two definitions, the type of the intermediate answers is `int list`:

- Here, the two recursive calls to `visit` are abstracted into a static delimited continuation using `shift` and `reset`:

```
fun traverse_sr t
    = let fun visit (EMPTY, a)
              = a
            | visit (NODE (t1, i, t2), a)
              = visit (t1, visit (t2, shift
                                      (fn k => i :: (k a))))
      in reset (fn () => visit (t, nil))
      end
```

- Here, the two recursive calls to `visit` are abstracted into a dynamic delimited continuation using `control` and `prompt`:

```
fun traverse_cp t
    = let fun visit (EMPTY, a)
              = a
            | visit (NODE (t1, i, t2), a)
              = visit (t1, visit (t2, control
                                      (fn k => i :: (k a))))
      in prompt (fn () => visit (t, nil))
      end
```

The static delimited continuations yield a *preorder* and *right-to-left* traversal, whereas the dynamic delimited continuation yield a *postorder* and *left-to-right* traversal. The resulting two lists are reverse of each other.

Again, CPS transformation and defunctionalization yield first-order programs whose behavior is more patent.

10.9.4 Further examples

We now turn to the lazy depth-first and breadth-first traversals recently presented by Biernacki, Danvy, and Shan [36, 37]. To support laziness, they used the following signature of generators:

```
signature GENERATOR
= sig
    type 'a computation
    datatype sequence = END
                      | NEXT of int * sequence computation

    val make_sequence : tree -> sequence
    val compute : sequence computation -> sequence
  end
```

The following generator is parameterized by a scheduler that is given four thunks to be applied in turn:

```
structure Lazy_generator : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    structure CP = Control_and_Prompt
                     (type intermediate_answer = sequence)

    fun visit EMPTY
        = ()
      | visit (NODE (t1, i, t2))
        = CP.control
            (fn k => (schedule
               (fn () => visit t1,
                fn () => CP.control (fn k' => NEXT (i, k')),
                fn () => visit t2,
                k);
```

```
            END))

  fun make_sequence t
      = CP.prompt (fn () => let val () = visit t
                            in END
                            end)

  fun compute k
      = CP.prompt (fn () => k ())
end
```

The relative scheduling of the first and third thunks determines whether the traversal of the input tree is from left to right or from right to left. The relative scheduling of the second thunk with respect to the first and the third determines whether the traversal is preorder, inorder, or postorder. The relative scheduling of the fourth thunk determines whether the traversal is depth-first, breadth-first, or a mix of both.

In each case, dynamic CPS transformation and defunctionalization yield first-order programs whose behavior is patent in that the depth-first traversal uses a stack, the breadth-first traversal uses a queue, and the mixed traversal uses a queue to hold the right (respectively the left) subtrees while visiting the left (respectively the right) ones.

10.10 A monad for dynamic continuation-passing style

The evaluator of Figure 10.3 is compositional, and has the following type:

$$\mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

Let us curry it to exhibit its notion of computation [172]:

$$\mathsf{Exp} \times \mathsf{Env} \to \mathsf{Cont}_1 \to \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

Property 10.4 *The type constructor*

$$D(\mathsf{Val}) = \mathsf{Cont}_1 \to \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

where

$$\begin{aligned}
\mathsf{Ans} &= \mathsf{Val} \\
\mathsf{Cont}_2 &= \mathsf{Val} \to \mathsf{Ans} \\
\mathsf{Cont}_1 &= \mathsf{Val} \to \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\
\mathsf{Val} &= \mathsf{Val} \to \mathsf{Cont}_1 \to \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\
\mathsf{Trail}_1 &= \mathsf{List}(\mathsf{Cont}_1)
\end{aligned}$$

together with the functions

$$\begin{aligned}
\mathsf{unit} &: \mathsf{Val} \to D(\mathsf{Val}) \\
\mathsf{unit}\,(v) &= \lambda k_1.\, \lambda(t_1, k_2).\, k_1\, v\, (t_1, k_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{bind} &: D(\mathsf{Val}) \times (\mathsf{Val} \to D(\mathsf{Val})) \to D(\mathsf{Val}) \\
\mathsf{bind}\,(c, f) &= \lambda k_1.\, \lambda(t_1, k_2).\, c\,(\lambda v.\, \lambda(t_1', k_2').\, f\, v\, k_1\, (t_1', k_2'))\, (t_1, k_2)
\end{aligned}$$

form a continuation+state monad, where the state pairs the trail of continuations and the meta-continuation. (The state could be η-reduced in the definitions of unit *and* bind*, yielding the definition of the continuation monad.)*

Proof. A simple equational verification of the three monad laws [172]. □

As in Wadler's study of monads and static delimited continuations [240], the type of bind, instead of the usual $D(\alpha) \times (\alpha \to D(\beta)) \to D(\beta)$, has $\alpha = \beta = \mathsf{Val}$, making the triple ($D$, unit, bind) more specific than a monad. As in Wadler's work, this extra specificity is coincidental here since we consider only one type, Val.

Having identified the monad for dynamic continuation-passing style, we are now in position to define `control` and `prompt` as operations in this monad:

Definition 10.4 *We define the monad operations* control, prompt *and* compute *as follows:*

$$\begin{array}{rcl}
\mathsf{prompt} & : & D(\mathsf{Val}) \to D(\mathsf{Val}) \\
\mathsf{prompt}\,(c) & = & \lambda k_1.\,\lambda(t_1, k_2).\,c\,\theta_1\,(\mathsf{nil}, \lambda v.\,k_1\,v\,(t_1, k_2)) \\
 & & \\
\mathsf{control} & : & ((\mathsf{Val} \to D(\mathsf{Val})) \to D(\mathsf{Val})) \to D(\mathsf{Val}) \\
\mathsf{control}\,(e) & = & \lambda k_1.\,\lambda(t_1, k_2).\,e\,k\,\theta_1\,(\mathsf{nil}, k_2) \\
 & & \quad \textit{where } k = \lambda v.\,\lambda k_1'.\,\lambda(t_1', k_2').\,k_1\,v\,(t_1 \mathbin{@} (k_1' :: t_1'), k_2') \\
 & & \\
\mathsf{compute} & : & D(\mathsf{Val}) \to \mathsf{Val} \\
\mathsf{compute}\,(c) & = & c\,\theta_1\,(\mathsf{nil}, \theta_2)
\end{array}$$

We can now extend the usual call-by-value monadic evaluator for the λ-calculus to $\mathcal{F}$ and # (see Figure 10.5). Inlining the abstraction layer provided by the monad yields the evaluator of Figure 10.3. Dynamic continuation-passing style therefore fits the functional correspondence between evaluators and abstract machines advocated by the first two authors [5, 7, 65]. Furthermore, and as has been observed before for other CPS transformations and for the continuation monad [120, 238], the dynamic CPS transformation itself can be factored through Moggi's monadic metalanguage and the monad above.

10.11 A new implementation of `control` and `prompt`

As pointed out in Section 10.10, if one curries the evaluator of Figure 10.3 and η-reduces the parameters t_1 and k_2 in the first three clauses interpreting the λ-calculus, one can observe that dynamic CPS conservatively extends CPS. Therefore, since the continuations k_1 live in the continuation monad, it is straightforward to express `control` and `prompt` in terms of `shift` and `reset`, essentially, by

1. transforming the evaluator of Figure 10.3 into direct style with respect to k_2 (the result is in continuation-composing style [67]), and

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- Values: $v \in \mathsf{Val} = \mathsf{Val} \to D(\mathsf{Val})$
- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \to D(\mathsf{Val})$

$$
\begin{aligned}
\mathsf{eval}_{\mathsf{mon}}\,(x, \rho) &= \mathsf{unit}\,(\rho(x)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\lambda x.e, \rho) &= \mathsf{unit}\,(\lambda v.\, \mathsf{eval}_{\mathsf{mon}}\,(e, \rho\{x \mapsto v\})) \\
\mathsf{eval}_{\mathsf{mon}}\,(e_0\, e_1, \rho) &= \mathsf{bind}\,(\mathsf{eval}_{\mathsf{mon}}\,(e_0, \rho), \lambda v_0.\, \mathsf{bind}\,(\mathsf{eval}_{\mathsf{mon}}\,(e_1, \rho), \lambda v_1.\, v_0\, v_1)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\#e, \rho) &= \mathsf{prompt}\,(\mathsf{eval}_{\mathsf{mon}}\,(e, \rho)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\mathcal{F}k.e, \rho) &= \mathsf{control}\,(\lambda v.\, \mathsf{eval}_{\mathsf{mon}}\,(e, \rho\{k \mapsto v\}))
\end{aligned}
$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \to \mathsf{Val}$

$$\mathsf{evaluate}_{\mathsf{mon}}\,(e) = \mathsf{compute}\,(\mathsf{eval}_{\mathsf{mon}}\,(e, \rho_{mt}))$$

Figure 10.5: A monadic evaluator for the λ-calculus extended with $\mathcal{F}$ and $\#$

2. transforming the resulting evaluator into direct style with respect to k_1 (as opposed to Section 10.8, where in order to obtain the direct-style evaluator written with `control` and `prompt`, we transformed the resulting evaluator into direct style with respect to both k_1 and t_1).

Building on this observation, we present below an implementation of `control` and `prompt` in Standard ML of New Jersey, based on Filinski's implementation of `shift` and `reset` [96]. The implementation takes the form of a functor mapping a type of intermediate answers to an instance of `control` and `prompt` at that type:

```
signature CONTROL_AND_PROMPT
= sig
    type intermediate_answer
    val control : (('a -> intermediate_answer) -> intermediate_answer)
                  -> 'a
    val prompt : (unit -> intermediate_answer) -> intermediate_answer
  end

functor Control_and_Prompt (type intermediate_answer)
: CONTROL_AND_PROMPT
= struct
    datatype ('a, 'b) cont1 = CONT1 of 'a -> 'b trail1 -> 'b
    withtype 'b trail1 = ('b, 'b) cont1 list
```

```
structure SR
= Shift_and_Reset
   (type intermediate_answer
         = intermediate_answer trail1 -> intermediate_answer)

type intermediate_answer = intermediate_answer

     fun theta1 v nil
         = v
       | theta1 v ((CONT1 k) :: t)
         = k v t

     fun prompt thunk
         = SR.reset (fn () => theta1 (thunk ())) nil

     exception MISSING_PROMPT

     fun control function
         = SR.shift
               (fn k1 =>
                 fn t1 =>
                  let val f = fn v =>
                                  SR.shift
                                      (fn k1' =>
                                        fn t1' =>
                                         k1 v
                                            (t1 @ (CONT1 k1' :: t1')))
                  in SR.reset (fn () => theta1 (function f)) nil
                  end) handle MISSING_RESET => raise MISSING_PROMPT
   end
```

In the definition of `prompt`, the expression delayed in `thunk` is computed with the initial continuation `theta1` and with an empty trail of continuations.

In the definition of `control`, the continuations `k1` and `k1'` are captured with `shift`, the function `f` is constructed according to its definition in the evaluator, and the application `function f` is computed with the initial continuation `theta1` and with an empty trail of continuations.

Hence, the standard continuation semantics of the definitions above coincides with the dynamic continuation semantics of `prompt` and `control` given by the evaluator. A more formal justification, however, is out of scope here.

10.12 Related work

The concept of meta-continuation and its representation as a function originates in Wand and Friedman's formalization of reflective towers [248], and its representation as a list in Danvy and Malmkjær's followup study [73]. Danvy and Filinski then realized that a meta-continuation naturally arises by "one more" CPS transformation, giving rise to success and failure continuations [67], and later Danvy and Nielsen observed that the list representation naturally arises by defunctionalization [74]. Just as repeated CPS transformations give rise to

a static CPS hierarchy [29,67,141,177], repeated dynamic CPS transformations should give rise to a dynamic CPS hierarchy—a future work.

The original approaches to delimited continuations were split between composing continuations dynamically by concatenating their representations [93] and composing them statically using continuation-passing function composition [67]. Recently, Shan [209] and Dybvig, Peyton Jones, and Sabry [84] each have proposed an account of dynamic delimited continuations using a continuation+state-passing style:

- Shan's development extends Wand et al.'s idea of an algebra of contexts [93] (the state represents the prefix of a meta-continuation and is equipped with algebraic operators *Send* and *Compose* to propagate intermediate results and compose the representation of delimited continuations). Like our dynamic continuation-passing style, Shan's continuation-passing style hinges on the requirement that the answer type of continuations be recursive. Our dynamic continuation-passing style also uses a state, namely a trail of contexts and a meta-continuation. This representation, however, only requires the usual list operations, instead of the dedicated algebraic operations provided by *Send* and *Compose*. Consequently, the abstract machine of Section 10.3 is simpler than the abstract machine corresponding to Shan's continuation-passing style. (We have constructed this abstract machine.) Shan's transformation can account for two other variations on $\mathcal{F}$. Our continuation-passing style can be adapted to account for these as well, by defunctionalizing the meta-continuation.

- Dybvig, Peyton Jones, and Sabry's continuation+state-passing style threads a state which is a list of continuations annotated with multiple control delimiters. This state is structurally similar to ours in the sense that defunctionalizing and flattening our meta-continuation and appending it to our trail of continuations yields their state without annotations. We find this coincidence of result remarkable considering the difference of motivation and methodology:

  - Dybvig, Peyton Jones, and Sabry sought "a typed monadic framework in which one can define and experiment with arbitrary [delimited] control operators" [84, Section 7], using Hieb, Dybvig, and Anderson's control operators for subcontinuations [125] as a common basis, whereas
  - we wanted an abstract machine for dynamic delimited continuations that is in the range of Reynolds's defunctionalization in order to provide a consistent spectrum of tools for programming with and reasoning about delimited continuations, both in direct style and in continuation-passing style.

Finally, as an alternative to Wand and Friedman's use of a meta-continuation [248], Bawden has used trampolining to investigate reflective towers [21].

Recently, Kiselyov has revisited trampolining to study the expressivity of static and dynamic delimited-continuation operators [146].

10.13 Conclusion and issues

In our earlier work [36, 37], we argued that dynamic delimited continuations need examples, reasoning tools, and meaning-preserving program transformations, not only new variations, new formalizations, or new implementations. The present work partly fulfills these wishes by providing, in a concerted way, an abstract machine that is in defunctionalized form, the corresponding evaluator, the corresponding continuation-passing style and CPS transformer, a monadic account of this continuation-passing style, a new simulation of a dynamic delimited-control operator in terms of a static one, and several new examples.

Compared to static delimited continuations, and despite recent implementation advances, the topic of dynamic delimited continuations still remains largely unexplored. We believe that the spectrum of compatible computational artifacts presented here—abstract machine, evaluator, computational monad, and dynamic continuation-passing style—puts one in a better position to assess them.

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Julia Lawall, Kristian Støvring, and the anonymous reviewers of MFPS XXI for their comments.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In Hudak [131], pages 31–46.

[2] Kamal S. Abdali. *A Combinatory Logic Model of Programming Languages*. PhD thesis, University of Wisconsin, 1974.

[3] Harold Abelson, Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Research Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

[5] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.

[6] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS RS-04-3.

[7] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the technical report BRICS RS-04-28.

[8] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 1991.

[9] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[11] Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium (ICALP 2003)*, number 2719 in Lecture Notes in Computer Science, pages 871–885, Eindhoven, The Netherlands, July 2003. Springer.

[12] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of continuations and prompts. In Fisher [101], pages 40–53.

[13] Kenichi Asai. Online partial evaluation for shift and reset. In Peter Thiemann, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2002)*, SIGPLAN Notices, Vol. 37, No 3, pages 19–30, Portland, Oregon, March 2002. ACM Press.

[14] Kenichi Asai. Offline partial evaluation for shift and reset. In Nevin Heintze and Peter Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2004)*, pages 3–14, Verona, Italy, August 2003. ACM Press.

[15] Vincent Balat. *Une étude des sommes fortes: isomorphismes et formes normales*. PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, December 2002.

[16] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Leroy [160], pages 64–76.

[17] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[18] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag.

[19] Chris Barker. Continuations in natural language. In Thielecke [234], pages 1–11.

[20] Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors. *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, Caminha, Portugal, September 2000. Springer-Verlag.

[21] Alan Bawden. Reification without evaluation. In Cartwright [42], pages 342–351.

[22] Nuel D. Belnap. How a computer should think. In Gilbert Ryle, editor, *Proceedings of the Oxford International Symposium on Contemporary Aspects of Philosophy*, pages 30–56, Oxford, England, 1976. Oriel Press.

[23] Josh Berdine. *Linear and Affine Typing of Continuation-Passing Style*. PhD thesis, Queen Mary, University of London, 2004.

[24] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

[25] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[26] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.

[27] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[28] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Thielecke [234], pages 25–33.

[29] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[30] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. Research Report BRICS RS-05-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[31] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Research Report BRICS RS-05-22, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2005.

[32] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[33] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[34] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 2005. To appear. Available as the technical report BRICS RS-05-10.

[35] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Research Report BRICS RS-05-16, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[36] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. *Information Processing Letters*, 96(1):7–17, 2005. Extended version available as the technical report BRICS RS-05-13.

[37] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 2006.

[38] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[39] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[40] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[41] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[42] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

[43] Robert (Corky) Cartwright, editor. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991. ACM Press.

[44] Eugene Charniak, Christopher Riesbeck, and Drew McDermott. *Artificial Intelligence Programming*. Lawrence Earlbaum Associates, 1980.

[45] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, February 1993.

[46] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[47] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[48] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[49] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[50] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [245], pages 333–340.

[51] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 2006. To appear.

[52] Tristan Crolard. A formulae-as-types interpretation of subtractive logic. *Journal of Logic and Computation*, 14(4):529–570, 2004.

[53] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[54] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser, 1993.

[55] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Wadler [241], pages 233–243.

[56] Olivier Danvy. On some functional aspects of control. In Thomas Johnsson, Simon Peyton Jones, and Kent Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 445–449. Program Methodology Group, University of Göteborg and Chalmers University of Technology, September 1988. Report 53.

[57] Olivier Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, January 1989. ACM Press.

[58] Olivier Danvy. Programming with tighter control. *Special issue of the Bigre journal: Putting Scheme to Work*, 65:10–29, July 1989.

[59] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[60] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[61] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[62] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Smolka [215], pages 88–103.

[63] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, number 124 in Electronic Notes in Theoretical Computer Science, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[64] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Thielecke [234], pages 13–23. Invited talk.

[65] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the technical report BRICS RS-03-33.

[66] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[67] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [245], pages 151–160.

[68] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[69] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, 2002. Extended version available as the technical report BRICS RS-01-29. A preliminary version was presented at the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001).

[70] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[71] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[72] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[73] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [42], pages 327–341.

[74] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[75] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[76] Olivier Danvy and Carolyn L. Talcott, editors. *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW 1992)*, Technical report STAN-CS-92-1426, Stanford University, San Francisco, California, June 1992.

[77] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[78] Arie de Bruin and Erik P. de Vink. Continuation semantics for Prolog with cut. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 351 in Lecture Notes in Computer Science, pages 178–192, Barcelona, Spain, March 1989. Springer-Verlag.

[79] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[80] Scott Draves. Implementing bit-addressing with specialization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Amsterdam, The Netherlands, June 1997. ACM Press.

[81] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In Cartwright [43], pages 163–173.

[82] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Barthe et al. [20], pages 137–192.

[83] R. Kent Dybvig. *The Scheme Programming Language, Second Edition*. Prentice Hall, 1996. Available online at `http://www.scheme.com/tspl2d/`.

[84] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. Technical Report 615, Computer Science Department, Indiana University, Bloomington, Indiana, June 2005.

[85] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[86] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

[87] Matthias Felleisen. The theory and practice of first-class prompts. In Ferrante and Mager [94], pages 180–190.

[88] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[89] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[90] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[91] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Reasoning with continuations. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 131–141, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.

[92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[93] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [42], pages 52–62.

[94] Jeanne Ferrante and Peter Mager, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.

[95] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In David H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989. Springer-Verlag.

[96] Andrzej Filinski. Representing monads. In Boehm [38], pages 446–457.

[97] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

[98] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[99] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[100] Michael J. Fischer. Lambda-calculus schemata. In Talcott [230], pages 259–288. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[101] Kathleen Fisher, editor. *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 39, No. 9, Snowbird, Utah, September 2004. ACM Press.

[102] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Wall [242], pages 237–247.

[103] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[104] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, Scotland, July 1976.

[105] Carsten Führmann. *The Structure of Call-by-Value*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 2000.

[106] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Peyton Jones [184], pages 271–282.

[107] Matthew L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[108] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Peyton Jones [184], pages 235–246.

[109] Timothy G. Griffin. A formulae-as-types notion of control. In Hudak [131], pages 47–58.

[110] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

[111] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

[112] Chris Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

[113] John Hannan. Operational semantics directed machine architecture. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.

[114] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[115] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[116] Robert Harper. Programming languages: Theory and practice. Working Draft. Available at `http://www.cs.cmu.edu/~rwh/plbook/`, Spring Semester, 2002.

[117] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[118] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1997.

[119] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[120] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [38], pages 458–471.

[121] Christopher T. Haynes. Logic continuations. *The Journal of Logic Programming*, 4:157–176, 1987.

[122] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984. ACM Press.

[123] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998. Springer-Verlag.

[124] Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, January 1976.

[125] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[126] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[127] Ralf Hinze. Deriving backtracking monad transformers. In Wadler [241], pages 186–197.

[128] Martin Hofmann. Sound and complete axiomatisations of call-by-value control operators. *Mathematical Structures in Computer Science*, 5(4):461–482, 1995.

[129] Martin Hofmann and Thomas Streicher. Continuation models are universal for lambda-mu calculus. In Glynn Winskel, editor, *Proceedings of the*

Twelfth Annual IEEE Symposium on Logic in Computer Science, pages 387–397, Warsaw, Poland, July 1997. IEEE Computer Society Press.

[130] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 470–490. Academic Press, 1980.

[131] Paul Hudak, editor. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM Press.

[132] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[133] Gregory F. Johnson. GL – a denotational testbed with continuations and partial continuations as first-class objects. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 154–176, Saint-Paul, Minnesota, June 1987. ACM Press.

[134] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In Ferrante and Mager [94], pages 158–168.

[135] Neil D. Jones. *Computability and Complexity from a Programming Perspective*. Foundations of Computing. The MIT Press, 1997.

[136] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[137] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986. ACM Press.

[138] Yukiyoshi Kameyama. A type-theoretic study on partial continuations. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 489–504, Sendai, Japan, August 2000. Springer-Verlag.

[139] Yukiyoshi Kameyama. Dynamic control operators in type theory. In *Proceedings of the Second Asian Workshop on Programming Languages and Systems (APLAS'01)*, pages 1–11, Daejon, Korea, December 2001.

[140] Yukiyoshi Kameyama. Towards logical understanding of delimited continuations. In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, Technical report 545, Computer Science Department, Indiana University, pages 27–33, London, England, January 2001.

[141] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[142] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[143] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[144] Richard B. Kieburtz, Borislav Agapiev, and James Hook. Three monads for continuations. In Danvy and Talcott [76], pages 39–48.

[145] Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.

[146] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, March 2005.

[147] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In Benjamin Pierce, editor, *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 40, No. 9, pages 192–203, Tallinn, Estonia, September 2005. ACM Press.

[148] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`, 1985.

[149] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 2006. To appear. Available online at `http://www.pps.jussieu.fr/~krivine/`.

[150] Peter Kursawe. How to invent a Prolog machine. *New Generation Computing*, 5(1):97–114, 1987.

[151] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[152] Yves Lafont, Bernhard Reus, and Thomas Streicher. Continuation semantics or expressing implication by negation. Technical report 93-21, University of Munich, 1993.

[153] Jim Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1998.

[154] Peter Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Reprinted from a technical report, UNIVAC Systems Programming Research (1965), with a foreword [232].

[155] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[156] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[157] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[158] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[159] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 227–238, Orlando, Florida, June 1994. ACM Press.

[160] Xavier Leroy, editor. *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, Venice, Italy, January 2004. ACM Press.

[161] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Fisher [101], pages 4–15.

[162] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 1991.

[163] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

[164] Antoni W. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18:220–226, 1971. (Working paper for IFIP WG2.2 dated February 1970.).

[165] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[166] John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.

[167] Chris Mellish and Steve Hardy. Integrating Prolog in the POPLOG environment. In John A. Campbell, editor, *Implementations of PROLOG*, pages 147–162. Ellis Horwood, 1984.

[168] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer-Verlag.

[169] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.

[170] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[171] Torben Æ. Mogensen. Separating binding times in language specifications. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 14–25, London, England, September 1989. ACM Press.

[172] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[173] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

[174] F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.

[175] James H. Morris Jr. A bonus from van Wijngaarden's device. *Communications of the ACM*, 15(8):773, August 1972.

[176] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

[177] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Danvy and Talcott [76], pages 49–72.

[178] Aleksandar Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 2004. Technical Report CMU-CS-04-151.

[179] Tim Nicholson and Norman Y. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, 1989.

[180] Lasse R. Nielsen. A denotational investigation of defunctionalization. Research Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.

[181] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[182] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Wadler [241], pages 131–136.

[183] C.-H. Luke Ong. A semantic view of classical proofs: Type-theoretic, categorical, and denotational characterizations (preliminary extended abstract). In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 230–241, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[184] Simon Peyton Jones, editor. *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[185] Andrew Pitts. Operational semantics and program equivalence. In Barthe et al. [20], pages 378–412.

[186] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[187] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[188] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2001. Technical Report CMU-CS-01-152.

[189] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In Leroy [160], pages 89–98.

[190] Christian Queinnec. Value transforming style. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 20–28, Bordeaux, France, September 1992. IRISA, Rennes, France.

[191] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Cartwright [43], pages 174–184.

[192] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[193] John C. Reynolds. GEDANKEN — a simple typeless language based on the principles of completeness and the reference concept. *Communications of the ACM*, 13:308–319, 1970.

[194] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974. Springer-Verlag.

[195] John C. Reynolds. The discoveries of continuations. In Talcott [230], pages 233–247.

[196] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[197] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[198] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[199] Kristoffer H. Rose. Explicit substitution – tutorial & survey. BRICS Lecture Series LS-96-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1996.

[200] Kristoffer H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1996.

[201] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms.* PhD thesis, Computer Science Department, Rice University, Houston, Texas, August 1994. Technical report 94-242.

[202] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[203] Erik Sandewall. An early use of continuations and partial evaluation for compiling rules written in FOPC. *Higher-Order and Symbolic Computation*, 12(1):105–113, 1999.

[204] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[205] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn, 1971.

[206] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11:207–260, 2001.

[207] Ravi Sethi and Adrian Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980.

[208] Chung-chieh Shan. Delimited continuations in natural language: quantification and polarity sensitivity. In Thielecke [234], pages 55–64.

[209] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[210] Chung-chieh Shan. *Linguistic Side Effects*. PhD thesis, Division of Engineering and Applied Science, Harvard University, September 2005.

[211] Dorai Sitaram. Handling control. In Wall [242], pages 147–155.

[212] Dorai Sitaram. *Models of Control and their Implications for Programming Language Design*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, April 1994.

[213] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[214] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [245], pages 161–175.

[215] Gert Smolka, editor. *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany, March 2000. Springer-Verlag.

[216] Michael Spivey and Silvija Seres. Combinators for logic programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 177–199. Palgrave Macmillan, 2003.

[217] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[218] Guy L. Steele Jr. Building interpreters by composing monads. In Boehm [38], pages 472–492.

[219] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.

[220] Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[221] Joseph Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[222] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[223] Joseph E. Stoy. The congruence of two programming language definitions. *Theoretical Computer Science*, 13(2):151–174, 1981.

[224] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000. Reprinted from lecture notes dated 1968.

[225] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[226] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[227] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, September 2000.

[228] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[229] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

[230] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, 1993.

[231] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1997. ECS-LFCS-97-376.

[232] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[233] Hayo Thielecke. On exceptions versus continuations in the presence of state. In Smolka [215], pages 397–411.

[234] Hayo Thielecke, editor. *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004.

[235] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[236] Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, 1966.

[237] Philip Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128, Nancy, France, September 1985. Springer-Verlag.

[238] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[239] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[240] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994.

[241] Philip Wadler, editor. *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, Montréal, Canada, September 2000. ACM Press.

[242] David W. Wall, editor. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, Albuquerque, New Mexico, June 1993. ACM Press.

[243] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[244] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[245] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[246] Mitchell Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, 1990.

[247] Mitchell Wand. Continuation-based multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, 1999. ⟨`http://www.brics.dk/~hosc/vol12/3-wand.html`⟩ Reprinted from the proceedings of the 1980 Lisp Conference, with a foreword.

[248] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 298–307, Cambridge, Massachusetts, August 1986. ACM Press.

[249] Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In Fisher [101], pages 54–65.

[250] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

[251] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[252] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.