A Derivational Approach to the Operational Semantics of Functional Languages

Małgorzata Biernacka

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

A Derivational Approach to the Operational Semantics of Functional Languages

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

by
Małgorzata Biernacka
November 22, 2005

Abstract

We study the connections between different forms of operational semantics for functional programming languages and we present systematic methods of inter-deriving reduction semantics, abstract machines and higher-order evaluators.

We first consider two methods based on program transformations: a syntactic correspondence relating reduction semantics and abstract machines, and a functional correspondence relating abstract machines and higher-order evaluators. We show that an extension of the syntactic correspondence provides a systematic and uniform derivation method connecting calculi of explicit substitutions and environment machines. In particular, we show that canonical environment-based abstract machines for evaluation in the lambda calculus, including the Krivine machine for call by name and the CEK machine for call by value, can be systematically derived from a calculus of explicit substitutions, given a reduction strategy such as normal order or applicative order.

Furthermore, we show that the syntactic correspondence applies to languages with context-sensitive reductions, which we illustrate by presenting a number of calculi of explicit substitutions with computational effects (e.g., control operators, input/output, stack inspection, laziness and proper tail recursion). Several of the corresponding abstract machines have been independently designed and reported as such in the literature. All the calculi are new.

As an application of the syntactic and the functional correspondences, we also provide an operational foundation for the lambda calculus with a hierarchy of delimited control operators, shift and reset.

In addition, we present a case study in program extraction from proofs, using Kreisel's modified realizability. This case study provides another example of a derivational approach to connecting semantic specifications: we formalize the reduction semantics of the simply typed lambda calculus in a first-order logic, and from the proof of weak head normalization we extract a higher-order evaluator for this language.

Acknowledgements

First, I am deeply indebted to my Ph.D. advisor Olivier Danvy, who has guided and supported me throughout my studies at BRICS. Working with him has been a great experience. I am grateful for his readiness to share his scientific expertise as well as his personal experience, and for his enthusiasm and constant encouragement.

I thank Mads Sig Ager, Darek Biernacki, Sam Lindley, Jan Midtgaard, Kevin Millikin, Henning Korsholm Rohde and Kristian Støvring for the discussions at our "weekly" meetings. Many thanks to Darek and Kristian for our enjoyable collaboration and for proof-reading parts of this dissertation. I would also like to thank Andrzej Filinski and Julia Lawall for many insightful comments and helpful suggestions about parts of this work, and Ulrich Kohlenbach for eliciting my interest in proof theory.

I am grateful to John Reynolds for sponsoring Darek and myself during our visit at Carnegie Mellon University, and to him and his wife Mary for hosting us in their home in Pittsburgh for the first two weeks.

I would like to thank Pierre-Louis Curien and Peter Sestoft for accepting to serve in my PhD committee.

Many thanks to all my friends and colleagues in Århus, in particular: Bartek, Branimir, Chris, Christopher, Irena, Jan, Johan, Kaja, Kevin, Kirill, Kristian, Maciek, Mads, Marco, Matthias, Michael, Paulo, Philipp, Piotrek, Tanya and Wojtek. Special thanks are due to Angie, Claus, Doina, Irit and Jesus for their much appreciated presence and support, and for the good conversations, and to Gabi and Saurabh for their cheerful attitude and for organizing entertainment.

I would also like to thank all the scientific and technical staff at BRICS and DAIMI, in particular: Mogens Nielsen, Lene Kjeldsteen and Uffe Engberg, and also Kaja Christiansen, Michael Glad, Hanne Friis Jensen, Ingrid Larsen, Ellen Lindstrøm and Karen Kjær Møller. Thanks to them, my studies went smoothly and in a friendly environment. I also want to thank the Danish National Research Foundation for funding my studies at BRICS.

The deepest gratitude I owe to my family: my parents and sister, and also Darek's mom, for their support, constant encouragement and all the help they have provided over the years. I especially thank my Mom for her wisdom and strength, and for supporting me in realizing my goals and dreams.

Finally, I am grateful for having Darek by my side. Without him, I wouldn't have survived these last 3 years. I thank him for his love and support, and for his inexhaustible patience and belief in me.

Małgorzata Biernacka,
Århus, November 22, 2005.

Contents

Abstract **v**

Acknowledgements **vii**

I Context **1**

1 Introduction **3**

1.1 Semantics of programming languages 3

1.2 Contributions 5

1.3 Structure of the dissertation 7

2 Operational aspects of evaluation **9**

2.1 Reduction semantics 10

2.2 Abstract machines 16

2.3 Interpreters 20

3 Connecting specifications **25**

3.1 Methods based on program transformations 26

3.1.1 A syntactic correspondence 28

3.1.2 A functional correspondence 35

3.2 Program extraction from proofs 42

3.2.1 The Curry-Howard isomorphism 43

3.2.2 Program extraction 45

3.3 Conclusion 47

II Publications **49**

4 A concrete framework for environment machines **51**

4.1 Introduction 51

4.2 One-step reduction in a calculus of closures 53

4.2.1 Curien's calculus of closures 54

4.2.2 A minimal extension to Curien's calculus of closures . . . 55

4.2.3 Specification of the normal-order reduction strategy . . . 57

4.2.4 Specification of the applicative-order reduction strategy . 58

4.3 From normal-order reduction to call-by-name environment machine 58

4.3.1 A reduction semantics for normal order 59
4.3.2 A pre-abstract machine 61
4.3.3 A staged abstract machine 62
4.3.4 An eval/apply abstract machine 63
4.3.5 A push/enter abstract machine 63
4.3.6 An optimized push/enter machine 63
4.3.7 An environment machine 64
4.3.8 Correctness 64
4.3.9 Correspondence with the λ-calculus 65
4.4 From applicative-order reduction to call-by-value environment machine 66
4.4.1 The reduction semantics for left-to-right applicative order 66
4.4.2 From evaluation function to environment machine 66
4.4.3 Correctness and correspondence with the λ-calculus . . . 67
4.5 A notion of context-sensitive reduction 68
4.5.1 Normal order: variants of Krivine's machine 68
4.5.2 Right-to-left applicative order: variants of the Zinc machine 70
4.6 A space optimization for call-by-name evaluation 71
4.7 Actual substitutions, explicit substitutions, and environments . . 72
4.7.1 A derivational taxonomy of abstract machines 72
4.7.2 Reversibility of the derivation steps 74
4.8 Conclusion 74

5 A syntactic correspondence between context-sensitive calculi and abstract machines 77
5.1 Introduction 77
5.1.1 Calculi and machines 77
5.1.2 Calculi of explicit substitution and environment-based machines 78
5.1.3 Calculi for computational effects and environment-based machines 79
5.1.4 Overview 79
5.2 Preliminaries 80
5.2.1 Our base calculus of closures: $\lambda\widehat{\rho}$ 80
5.2.2 Notion of context-sensitive reduction 81
5.3 The $\lambda\widehat{\rho}\mathcal{K}$-calculus 82
5.3.1 The language of $\lambda\widehat{\rho}\mathcal{K}$ 82
5.3.2 Notion of context-sensitive reduction 83
5.3.3 Krivine's machine 83
5.3.4 Formal correspondence 83
5.4 The $\lambda\widehat{\rho}\mu$-calculus 84
5.4.1 The language of $\lambda\widehat{\rho}\mu$ 84
5.4.2 Notion of context-sensitive reduction 85
5.4.3 An eval/apply abstract machine 85
5.4.4 Formal correspondence 85
5.5 Delimited continuations 85
5.5.1 The $\lambda\widehat{\rho}\mathcal{S}$-calculus 86

5.5.2 The $\lambda\widehat{\rho}\mathcal{F}$-calculus 89
5.5.3 Conclusion 91
5.6 A calculus of closures with input/output ($\lambda\widehat{\rho}_{\mathrm{i/o}}$) 92
5.6.1 The language of $\lambda\widehat{\rho}_{\mathrm{i/o}}$ 92
5.6.2 Notion of context-sensitive reduction 93
5.6.3 An eval/apply abstract machine 94
5.6.4 Formal correspondence 94
5.7 Stack inspection 94
5.7.1 The $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus 95
5.7.2 Properly tail-recursive stack inspection 96
5.7.3 State-based properly tail-recursive stack inspection 98
5.7.4 Conclusion 100
5.8 A calculus for proper tail-recursion 100
5.8.1 A simplified version of Clinger's abstract machine 101
5.8.2 The language of $\lambda\widehat{\rho}_{\mathrm{ptr}}$ 101
5.8.3 Notion of context-sensitive reduction 101
5.8.4 Formal correspondence 102
5.9 A lazy calculus of closures 102
5.9.1 The language of $\lambda\widehat{\rho}_{\mathrm{l}}$ 102
5.9.2 Notion of context-sensitive reduction 103
5.9.3 An eval/apply abstract machine 103
5.9.4 Formal correspondence 104
5.10 Conclusion 104

6 An operational foundation for delimited continuations in the CPS hierarchy 109
6.1 Introduction 109
6.2 From evaluator to reduction semantics for arithmetic expressions 110
6.2.1 The starting point: an evaluator in direct style 111
6.2.2 CPS transformation 111
6.2.3 Defunctionalization 112
6.2.4 Abstract machines as defunctionalized continuation-passing programs 113
6.2.5 From value-based abstract machine to term-based abstract machine 114
6.2.6 From term-based abstract machine to reduction semantics 114
6.2.7 From reduction semantics to term-based abstract machine 115
6.2.8 Summary and conclusion 115
6.3 Programming with delimited continuations 116
6.3.1 Finding prefixes by accumulating lists 116
6.3.2 Finding prefixes by accumulating list constructors 117
6.3.3 Finding prefixes in direct style 118
6.3.4 Finding prefixes in continuation-passing style 119
6.3.5 The CPS hierarchy 119
6.3.6 A note about typing 120
6.3.7 Related work 120
6.3.8 Summary and conclusion 121

6.4 From evaluator to reduction semantics for delimited continuations 122
6.4.1 An environment-based evaluator . . . 122
6.4.2 An environment-based abstract machine . . . 123
6.4.3 A substitution-based abstract machine . . . 125
6.4.4 A reduction semantics . . . 127
6.4.5 Beyond CPS . . . 129
6.4.6 Static vs. dynamic delimited continuations . . . 130
6.4.7 Summary and conclusion . . . 131
6.5 From evaluator to reduction semantics for the CPS hierarchy . . 131
6.5.1 An environment-based evaluator . . . 132
6.5.2 An environment-based abstract machine . . . 134
6.5.3 A substitution-based abstract machine . . . 136
6.5.4 A reduction semantics . . . 138
6.5.5 Beyond CPS . . . 139
6.5.6 Static vs. dynamic delimited continuations . . . 139
6.5.7 Summary and conclusion . . . 139
6.6 Programming in the CPS hierarchy . . . 139
6.6.1 Normalization by evaluation . . . 139
6.6.2 The free monoid . . . 140
6.6.3 A language of propositions . . . 141
6.6.4 A hierarchical language of units and products . . . 142
6.6.5 A note about efficiency . . . 146
6.6.6 Summary and conclusion . . . 146
6.7 Conclusion and issues . . . 147

7 Program extraction from proofs of weak head normalization 149
7.1 Introduction and related work . . . 149
7.2 Preliminaries . . . 150
7.2.1 First-order minimal logic . . . 150
7.2.2 Modified realizability . . . 151
7.3 Weak head normalization . . . 154
7.3.1 The object language . . . 155
7.3.2 Call-by-name evaluation . . . 156
7.3.3 Call-by-value evaluation . . . 160
7.3.4 Weak head normalization for closed terms of base type . . 164
7.A Implementation . . . 165
7.A.1 Call by name . . . 166
7.A.2 Call by value . . . 166

Bibliography 167

Part I

Context

Chapter 1

Introduction

A programming language is characterized by its syntax, semantics and pragmatics. The syntax defines the set of well-formed expressions allowed in the language, the semantics assigns meaning to these expressions, and the pragmatics concerns the applicability and usage of the language. Since a programming language is a notation to express computation, it is then desirable that it have a simple and unambiguous grammar, and a rigorous mathematical semantics.

Formal semantics is used as a tool for designing and analyzing programming languages, and for reasoning about programs as well as for providing standards for language implementations. It can hardly be expected that all these goals can be met by using a single formalization of the meaning of programs. Typically, a programming language is given a variety of semantic descriptions, serving different purposes and developed in response to different designer, programmer or implementor needs. What is ultimately required of all the different semantics for the same language is that they *agree* in some formal sense.

In this thesis, we focus on the operational approach to the semantics of functional programming languages. Operational specifications come in a variety of styles and forms. Since they are usually developed independently, different operational descriptions of the same language or of the same computational phenomenon tend to be disconnected from each other and often independent accounts of the same concepts are invented in different settings. This thesis provides a step towards discovering the connections between these accounts and interderiving semantic specifications mechanically.

1.1 Semantics of programming languages

Traditionally, the semantics of programming languages is formalized using the following approaches:

Operational semantics describes the meaning of programs written in a language by specifying the way programs are executed (in the context of imperative languages), or evaluated (in the context of functional languages) on a machine. This general definition indicates that the role of operational semantics is to formalize the *process* of carrying out the computation prescribed by a given program, until a result is obtained. At a reasonably high level of abstraction, this approach is particularly appealing to language users who usually think of the meaning of programs in terms of their execution—this holds true not only for imperative, hence naturally sequential, languages, but also for languages embodying declarative paradigms (functional and logic programming).

Denotational semantics describes the meaning of programs by interpreting them in a suitable mathematical model; this approach originates in the work of Scott and Strachey. A semantic valuation function maps each syntactic construct to its denotation in the model in such a way that the denotation of a compound construct is defined in terms of denotations of its subcomponents (i.e., compositionally). The high level of abstraction of denotational descriptions hides away low-level, implementation details that are unimportant for the understanding of the computational behavior of language features, and provides support during the language development phase.

Axiomatic semantics provides a way of stating and proving properties of programs within a formal proof system, as pioneered by Floyd and Hoare, but it does not explicitly assign meaning to programming constructs. The semantics is given by a set of axioms and inference rules, and a language of assertions to express relations between the state of computation before and after the execution/evaluation of a particular expression.

Since the introduction of the three approaches above, new paths emerged, spurred on both by research developments in related areas of logic and computer science, and by the practical needs arising from applications of programming languages. The former typically results in new techniques in the denotational approach (e.g., game semantics, categorical semantics); and the latter stimulates the advancement of new methods for implementation and reasoning (e.g., a number of proof assistants for formalizing and reasoning about programs). Moreover, the distinction between the three traditional approaches has blurred as new semantic frameworks have been developed which do not always fit into just one category: for example, game semantics has both operational and denotational features.

The different ways of specifying semantics turn out to be not only suited each for a different purpose, but in fact they are complementary. In most cases, it is not enough to only provide, say, an operational semantics for a language. We can implement a language whose behavior is defined operationally, e.g., via an abstract machine, but we also have to know what it is that we are computing, i.e., we need to interpret the syntactic constructs of the language in some model. For a simple example, β-reduction in the λ-calculus only becomes a meaningful symbolic manipulation if we interpret it as the application of a function (denoted by a λ-abstraction) to an argument. Furthermore, it is an obvious requirement that we prove the programs we write correct, or – at least – that we know how to do it and we can construct such proofs at any time.

So, we can only claim to have given a programming language a *complete* specification if we have all the tools necessary in each of the possible contexts it can be used in. Even better, we can explore the connections between various semantic specifications to build frameworks for interderiving different descriptions, possibly applicable in many different settings. This approach offers the following benefits:

- it promotes the understanding of the nature of computation and enables the exploration of its new aspects offered by different semantic paradigms
- it allows for adapting semantic frameworks in new settings, and enhances their reusability
- it facilitates extensions and modifications to the language and its implementations
- it helps one to discover new implementation methods and optimize existing ones

For example, denotational specifications can be used for deriving interpreters, compilers and abstract machines: a denotational definition of a language can be read as a program written in a different language (the metalanguage of denotational semantics), which in turn can be interpreted operationally. In fact, this approach has led to several compiler-generation systems and it has inspired the style of writing interpreters. In particular, the λ-notation traditionally used in denotational definitions suggests reading denotational definitions as interpreters written in a functional language. However, the correctness of such implementations and the precise correspondence between the semantics have to be established separately, and – minimally – it consists in proving the adequacy of the denotational semantics with respect to the operational one.

1.2 Contributions

In this thesis, we focus on the operational approach and we study the connections between the common operational specifications for functional languages: reduction semantics (i.e., a small-step semantics with explicit representation of reduction contexts), abstract machines, and higher-order evaluators. The goal of our work is to provide *systematic* (and, ultimately, mechanizable) derivations from one semantic description to another, and in doing so, to unify the different facets of the evaluation process into a common whole.

This general goal is realized here in two different ways, by using:

1. methods based on program-transformation techniques: a *syntactic correspondence* based on *refocusing*, relating reduction semantics and abstract machines, and a *functional correspondence* based on CPS transformation and defunctionalization, relating higher-order evaluators and abstract machines (this approach comprises most of the thesis), and
2. a method based on the Curry-Howard isomorphism between formal logics and typed λ-calculi, which allows one to extract programs from proofs

in logic (this approach is realized in a case study reported in the last chapter).

Refocusing and the functional correspondence have been introduced elsewhere. In this thesis, we apply them in new settings, and we extend the refocusing method to account for calculi with explicit substitutions and for context-sensitive reductions.

Since it is usually hard to reason about programs in terms of an abstract machine and to compare different implementations, we recognize the need to faithfully reflect the computation process realized in machines at the purely syntactic level of the underlying calculus. However, starting with Landin's SECD machine, most implementations use environments and model "delayed substitutions" rather than the actual substitutions as usually specified in the notion of β-reduction in the λ-calculus. The two diverging views can be reconciled with the introduction of intermediate calculi of closures as pioneered by Curien (these calculi are weak versions of more expressive calculi of explicit substitutions), and the refocusing method can be used to derive environment machines from these calculi with a variety of computational effects. On the other hand, the extraction of an underlying reduction semantics from an arbitrary abstract machine is not always straightforward and in general not mechanizable. The refocusing method allows us to identify the source of this difficulty, i.e., in what sense most abstract machines "optimize" the reduction sequence of the reduction semantics.

The refocusing method has been initially devised as a way of optimizing a naive implementation of the evaluation function of a reduction semantics, and only subsequently it has been noticed to produce a (substitution-based) abstract machine in effect. Here we extend the method to a new setting where the starting point is a calculus of closures and its reduction semantics, and we show a systematic derivation of environment-based abstract machines.

Together, refocusing and the functional correspondence provide a simple, uniform and versatile method of connecting distinct approaches to specifying operational semantics of functional languages. Moreover, applying them in new settings allows not only to derive new abstract machines, new evaluators, or new calculi, but it provides a connection between existing semantic specifications. In particular, it is a contribution of this work that the extended refocusing method applied to calculi of closures yields well known environment-based abstract machines: the Krivine machine, the CEK machine, the Zinc machine, etc. This fact, however, is not a coincidence; on the contrary, exhibiting the systematic derivation method has made it possible to identify different facets of the same computational phenomena: for example, it has led to realization that reduction contexts of reduction semantics and evaluation contexts of abstract machines coincide, and that they are isomorphic to defunctionalized continuations of an evaluator in continuation-passing style and of a one-step reduction function in CPS.

Finally, the correctness of the outcome of each of the derivations does not have to be proved separately for each individual case, but it follows directly from the correctness of the methods used.

1.3 Structure of the dissertation

The dissertation is divided into two parts. Part I lays down the context of our work:

- Chapter 2: We introduce the basic concepts used throughout the dissertation in connection with operational semantics of functional programming languages: small-step specifications provided by reduction semantics and abstract machines, and big-step, reduction-free semantics in the form of functional evaluators. We also discuss some of the advantages and typical uses of each of these specifications.

- Chapter 3: We expand on the previous chapter by reviewing some of the methods that allow for systematic derivations of one semantic description from another, which are put to use in Part II of the thesis. Specifically, we first show two methods based on program transformations: a syntactic correspondence based on refocusing leading from reduction semantics to a series of abstract machines, and a functional correspondence between higher-order evaluators and abstract machines. Then we show how a general method of program extraction from proofs can be used to obtain a higher-order evaluator implementing a reduction-free evaluation function from a formalization of reduction semantics in a logical system (we consider the simply typed λ-calculus as the object language).

Part II is a collection of four articles:

- Chapter 4: *A concrete framework for environment machines*, with Olivier Danvy. BRICS research report RS-05-15.

  We extend the refocusing method and present a systematic derivation method for obtaining environment-based abstract machines from one-step reduction semantics in a suitable language of closures. This work builds on Curien's introduction of a calculus of closures as an intermediate language for reasoning about implementations of the λ-calculus. Curien's calculus is minimal in that it is expressive enough to formalize the evaluation process in the λ-calculus with "delayed" substitutions as a multi-step reduction relation, but it does not allow one-step specifications. As Danvy and Nielsen's method shows, it is exactly one-step semantics that naturally corresponds to an abstract machine. Therefore, we propose an extension to Curien's language that fills the gap and we systematically derive abstract machines for call by name and call by value. We also consider context-sensitive reductions in the source language, and we identify the resulting machines as already present in the literature, but independently discovered ones: the Krivine machine, the CEK machine and the Zinc machine. We also discuss the issue of reversibility of the derivation, i.e., of extracting a reduction semantics from an abstract machine.

- Chapter 5: *A syntactic correspondence between context-sensitive calculi and abstract machines*, with Olivier Danvy. BRICS research report RS-05-22.

We further explore the syntactic correspondence between calculi of closures and environment-based abstract machines identified in Chapter 4. We consider a range of calculi with computational effects: control operators, input/output, stack inspection, laziness and proper tail recursion. In particular, we show calculi extended with a syntactic component accounting for the store component of the corresponding abstract machine. Since the store component is global in the machine, the underlying calculus does not admit the traditional reduction semantics, but a context-sensitive one, where reduction contexts take part in reductions. Similarly, context-sensitive reductions are needed for other language features that require access to some global component, e.g., control operators or stack inspection.

- Chapter 6: *An operational foundation for delimited continuations in the CPS hierarchy*, with Dariusz Biernacki and Olivier Danvy. Appears in Logical Methods in Computer Science.

  In this chapter we consider the λ-calculus extended with a hierarchy of static delimited-control operators shift_n and reset_n. Starting from a definitional evaluator, we derive the corresponding environment-based abstract machine and we extract the underlying reduction semantics. These first-order operational specifications make it possible to pinpoint the difference in behavior between static delimited continuations and dynamic ones. This difference has been further explored elsewhere [34]. Furthermore, we show examples of programming with higher levels of the CPS hierarchy.

- Chapter 7: *Program extraction from proofs of weak head normalization*, with Olivier Danvy and Kristian Støvring. Appears in the Proceedings of the 21st Conference on the Mathematical Foundations of Programming Semantics.

  We formalize the proof of weak head normalization for the simply typed λ-calculus in first-order minimal logic, and from the proof we extract a reduction-free normalization function, which can be read as an evaluator written in the simply typed λ-calculus. We carry out the development for two reduction strategies in the object language, obtaining two different evaluators: one for call by name and one for call by value. We also discuss how the method of proof (by logical relations) determines the intensional structure of the evaluator, which is an instance of normalization by evaluation using a glueing construction in the model.

Chapter 2

Operational aspects of evaluation

In functional programming languages, executing programs consists in *evaluating* expressions and applying functions to arguments. This process is usually specified by syntactic simplifications (normalization) of the program text—represented as a λ-expression—using some reduction rules and a fixed reduction strategy. The result of a computation (the *value* of an expression) is represented by a term in some *canonical form* [193, p. 186], typically a basic constant (representing an integer, a character, etc.) or a λ-abstraction (representing a function). It is common practice to use the term 'value' for a canonical form, and we will often adhere to this practice.

When it comes to formalizing the process of evaluation, various approaches are possible. Depending on their purpose, specifications may range from very high-level, abstract ones, whose primary purpose is to provide reasonable intuition for programmers about how programs are executed, to very concrete, low-level ones, facilitating more or less direct implementations on real machines.

In this thesis, we look at three ways of providing an operational understanding of evaluation in a programming language: an abstract view given by reduction semantics, a concrete view specified by abstract machines, and a more indirect view offered by interpreters as executable semantic definitions. A common slogan is that interpreters are "implementations" of denotational semantics; the "implementation" part consists precisely in imposing a specific operational strategy on the metalanguage, in order to define a precise operational behavior of the source language [155].

We consider a simple functional programming language λ_v—a restriction of Landin's ISWIM [128,129], extending λ-terms with numerals as basic constants and the successor operation:

$$t ::= \ulcorner n \urcorner \mid \mathtt{succ}\, t \mid x \mid t\, t \mid \lambda x.t$$

In the following sections, we give (three different forms of) call-by-value semantics for λ_v-programs, i.e., closed terms.

2.1 Reduction semantics

Notion of reduction

As the name suggests, the central notion in reduction semantics is a primitive *notion of reduction*, given by a binary relation on terms. The relation is often presented as a set of *contraction rules*; the term in the left-hand side of a contraction is called a *redex*, and the one in the right-hand side is called a *contractum.* The notion of reduction specifies the basic computation steps in a language.

In λ_{v}, values are numerals and (closed) λ-abstractions:[1]

$$v ::= \ulcorner n \urcorner \mid \lambda x.t.$$

The notion of computation consists of a restricted β-contraction and a δ-contraction [56]:

- An application of a λ-abstraction is contracted when the supplied argument is already reduced to a value:

  $$(\mathrm{Beta_v}) \quad (\lambda x.t)\, v \to t\{v/x\},$$

  where $t\{v/x\}$ denotes the result of the simultaneous substitution of the variable x in t with v.[2]

- An application of the successor operation is contracted when the operand is a numeral:

  $$(\mathrm{Succ}) \quad \texttt{succ}\, \ulcorner n \urcorner \to \ulcorner n+1 \urcorner.$$

A contraction prescribed by a notion of reduction can be applied to any redex inside a given term. The compatible closure of the notion of reduction defines one-step reduction (noted $\to_1$). With the use of *contexts* (i.e., as Barendregt puts it [16], terms with a hole):

$$C ::= [\,] \mid C[[\,]\, t] \mid C[t\, [\,]] \mid C[\lambda x.[\,]],$$

we can define one-step reduction concisely; if $t_1 \to t_2$, then $C[t_1] \to_1 C[t_2]$. Here $C[t]$ denotes the term obtained by plugging the hole $[\,]$ of the context C with the term t.

Reduction strategies and reduction contexts

The process of evaluation is defined through a reduction sequence induced by one-step reduction by taking its reflexive, transitive closure. In general, reduction sequences for a given term need not be unique, and they need not yield the same values; therefore, various reduction strategies are used in practice to ensure that the language is well-behaved. Furthermore, in most languages no

[1]In the call-by-value λ-calculus, variables are also included in the set of values; they are not in λ_{v}, because closed terms can never evaluate to variables.

[2]Since v is closed, there is no danger of variable capture and no variable renaming is necessary before performing substitution.

reduction is performed inside the body of a λ-abstraction—the value of such an expression is a function whose identity is determined by its extensional behavior, and not by its internal structure.

Specific reduction strategies typically restrict the one-step reduction relation, and they are conveniently specified using *reduction contexts*, as proposed by Felleisen [82,84]. A reduction context is a restricted form of a context whose grammar limits the positions in a term where reductions can occur.

For example, in λ_{v}, the grammar of reduction contexts is the following one:

$$E := [\,] \mid E[[\,]\,t] \mid E[v\,[\,]] \mid E[\mathtt{succ}\,[\,]],$$

hence permitting only leftmost-outermost redexes to be contracted.[3] Thus, the one-step reduction relation $\rightarrow_{\mathrm{v}}$ generated by the notion of reduction can be defined as follows:

$$\begin{array}{rcl} E[(\lambda x.t)\,v] & \rightarrow_{\mathrm{v}} & E[t\{v/x\}] \\ E[\mathtt{succ}\,\ulcorner n \urcorner] & \rightarrow_{\mathrm{v}} & E[\ulcorner n+1 \urcorner] \end{array}$$

Computationally, given the two ingredients—a notion of reduction and reduction contexts—a single reduction step in a given term consists in three successive steps:

1. identifying a redex and its surrounding reduction context (i.e., decomposing a term into these two components),
2. reducing the redex according to the corresponding contraction rule;
3. plugging the contractum into the hole of the reduction context (which remains unaltered during the operation).

Although in general a term can be decomposed in several ways, reduction semantics typically satisfy the *unique-decomposition property* [82] stating that for each term there is only one complete decomposition, i.e., only one way of choosing a redex to contract. A generalized formulation of this property, proposed by Danvy and Nielsen [72], introduced the notion of *potential redex*; a potential redex r is a term whose every decomposition into a context E and another term t' is such that either $r = t'$ or t' is a value.

Definition 2.1 (Unique-decomposition property). *A reduction semantics satisfies the unique-decomposition property, if for any term t, either t is a value, or it can be uniquely decomposed into a reduction context E and a potential redex r such that $E[r] = t$.*

For λ_{v}, potential redexes can be defined by the grammar:

$$r ::= v\,v \mid \mathtt{succ}\,v$$

Potential redexes contain both *actual redexes* as defined by the left-hand sides of contraction rules, and *stuck redexes*, providing a syntactic characterization of stuck terms. These typically occur when the set of programs, i.e.,

[3]Note that, due to closedness assumption, in the reduction context $E[[\,]\,t]$, t must be a closed term, and in particular it can never be a variable.

syntactically well-formed closed terms, does not exclude meaningless terms—a property that can be verified by providing a static semantics (for example, in λ_v, an application of a numeral to a value $\ulcorner n \urcorner v$ is not a sensible program fragment, and in the presence of types, such a term is not allowed since it is not well-typed).

The unique-decomposition property holds for λ_v.

Evaluation

The process of evaluating a term consists in repeatedly applying the basic computation step (one-step reduction) until it is no longer possible to apply any contraction rule, or indefinitely if the given term diverges under the chosen strategy. In the first case—when the evaluation terminates—it is either with a value, or with a stuck term; the (semantic) distinction between the two is based on their syntactic structure.

The evaluation function is a partial function mapping a program to its canonical form, if it exists.

Reasoning about programs

When it comes to reasoning about programs and verifying the correctness of various program transformations and optimizations, one is often interested in exhibiting an equational theory for the language—a calculus. For this purpose, the calculus must satisfy certain criteria of correctness and correspondence with an independently given semantic specification. This problem led Plotkin to the discovery of the λ_v-calculus corresponding to the call-by-value programming language ISWIM [148], originally specified by Landin's SECD machine. The λ_v-calculus was further extended by Felleisen et al. to account for imperative features, like control and state, also using an abstract machine [82, 85, 87]. The correctness of a calculus with respect to the programming language can be summarized by the following two criteria, identified by Plotkin [148]:

1. a standard reduction function in the calculus should produce the same result as the given evaluation function (for ISWIM—the evaluation function defined by the SECD machine); and

2. terms provably equal in the calculus should be observationally equivalent.

The first condition allows one to simulate the behavior of programs in the calculus, using an algorithmically defined deterministic reduction sequence, obtained as a restriction of the calculus.

The second condition validates the use of calculus as an equational theory for reasoning about the behavior of programs, where we would like to equate terms that are interchangeable in all program contexts. For this purpose, we use a suitably defined *observational equivalence* relation.

Formally, a program context is a general term context which, when plugged with a suitable term, yields a syntactically correct program. Furthermore, observational equivalence relies on the notion of observations, i.e., canonical forms that can be easily understood and effectively compared by the user. The choice of observable values is arbitrary, but typically they include basic constants (representing integers, strings, etc.). A functional value, on the other hand, is not observable: from the user's point of view a function is defined by its graph—it is perceived as a "black box" producing values given arguments, and its internal structure is irrelevant. Formally, the definition of observational equivalence can be stated as follows:

Definition 2.2. *Two terms t_1, t_2 are* observationally equivalent *(denoted $t_1 \simeq t_2$) if and only if, for all program contexts C, either $C[t_1]$ and $C[t_2]$ both diverge, or they both terminate and if one yields an observable value, then the other one yields the same value.*

In the present case, if we take the previously defined reduction semantics as specifying the operational behavior of the language λ_{v}, then the calculus can be obtained immediately by taking the smallest equivalence relation (denoted $=_{\mathrm{v}}$) generated by the reflexive, transitive closure of (the general) one-step reduction $\rightarrow_1$. It then follows that the restricted one-step reduction $\rightarrow_{\mathrm{v}}$ defines a standard reduction function, and hence the first condition above is vacuously satisfied.

The second criterion is also satisfied for the language λ_{v}, i.e., provability in the calculus is sound with respect to observational equivalence. However, it is not complete:

Theorem 2.1. *If $t_1 =_{\mathrm{v}} t_2$, then $t_1 \simeq t_2$. The converse does not hold.*

The proof requires the Church-Rosser and standardization theorems for the λ_{v}-calculus. We omit them here, the details can be found in Plotkin's article [148].

Proof. The convertibility relation $=_{\mathrm{v}}$ is a congruence, hence for any program context $C[t_1] =_{\mathrm{v}} C[t_2]$. Assume $C[t_1]$ evaluates to a value v; then $C[t_2] =_{\mathrm{v}} v$, and furthermore (following from the standardization theorem) $C[t_2]$ also terminates, say, with a value v'. If v is observable (i.e., a numeral), then from the Church-Rosser property it follows that $v \equiv v'$ ($\equiv$ denotes syntactic equality). □

Completeness fails because any two stuck terms are observationally equivalent (which could be indicated by a 'syntax error' message), but in general they are not provably equal in the calculus. This problem, however, could be solved by adding an 'error' constant to the language, and a set of contraction rules to handle erratic terms. A more serious problem is that of nonterminating programs: any two such programs are observationally equivalent, but not necessarily equal in the calculus (consider, e.g., $(\lambda x.x\,x)\,(\lambda x.x\,x)$ and $(\lambda x.x)\,((\lambda x.x\,x)\,(\lambda x.x\,x))$).

In general, consider any well-behaved calculus, i.e., for which the Church-Rosser and standardization theorems hold. Then a sufficient condition for soundness of convertibility with respect to observational equivalence is for convertible observable values to be equal, as stated in the following theorem. This condition is also reasonable from the user's point of view: values deemed to be "the same" should be easily identified as such.

Theorem 2.2. *Let $=$ denote convertibility in a calculus, and let the following property hold in the calculus for all observable values v:*

$$\text{For all values } v',\ \text{if } v = v',\ \text{then } v \equiv v'.$$

Then, whenever $t_1 = t_2$, then also $t_1 \simeq t_2$.

Proof. Let $t_1 = t_2$. Then for any program context C, $C[t_1] = C[t_2]$. Assume $C[t_1]$ evaluates to a value v. Then the evaluation of $C[t_2]$ also terminates, say with a value v', and $v = v'$. If v is observable, then, by assumption, we have $v \equiv v'$, and hence $t_1 \simeq t_2$. □

Non-local reductions

The above presentation of reduction semantics relies on the *locality* of reductions, i.e., at each point in the evaluation process we can identify a reduction site (redex), and the part of the program that remains intact during the contraction (reduction context). This property is particularly useful for local reasoning and local manipulations of programs. However, one can think of such notions of reduction that do not—at least not immediately—admit a local specification, but require rewriting the text of the entire program. Examples of this kind of behavior include various forms of control operators, exceptions, assignments, and other features typical especially of languages with computational effects. The impact of adding non-local reduction rules to the λ_v-calculus has been studied extensively by Felleisen and his collaborators [82,83,85,87].

A more general reduction semantics, admitting non-local reductions, is one with a generalized notion of one-step reduction of the form:

$$E[r] \to t,$$

where the right-hand side of the reduction depends on *both* the redex r and the context E, but it does not have to be of the form $E[r']$. Using this kind of rules, it is no longer possible to separate the notion of reduction (expressing the nature of the basic computation step) and one-step reduction (defined previously as the compatible closure of the notion of reduction). Since such rules are context-sensitive, they can only be applied to programs as a whole, and therefore the locality of reduction no longer holds.

As an example, consider the language $\lambda_v^{\mathcal{A}}$, obtained by extending λ_v with a control operator $\mathcal{A}$ (abort):

$$t ::= \ldots \mid \mathcal{A}\, t$$

that discards its entire reduction context:

$$(\text{Abort}) \quad E[\mathcal{A}\, t] \quad \to \quad t$$

The $\mathcal{A}$ operator has a global effect on a program, and therefore it is not possible to express it using only local rules. An alternative notion of reduction that simulates its behavior using local reductions was proposed by Felleisen [85]:

$$\begin{array}{lrcl}
(\mathcal{A}_{\mathrm{L}}) & v\,(\mathcal{A}\, t) & \to & \mathcal{A}\, t \\
(\mathcal{A}_{\mathrm{R}}) & (\mathcal{A}\, t_1)\, t_2 & \to & \mathcal{A}\, t_1 \\
(\mathcal{A}_{\mathrm{succ}}) & \mathtt{succ}\,(\mathcal{A}\, t) & \to & \mathcal{A}\, t
\end{array}$$

Intuitively, these rules allow the abort operator to consume (and discard) the surrounding reduction context piecemeal, until the empty context is reached. Consider a program that decomposes into $E[\mathcal{A}\, t]$. The rule (Abort) reduces it to t, which can be evaluated further. Using the set of rules ($\mathcal{A}_{\mathrm{L}}$)-($\mathcal{A}_{\mathrm{succ}}$) we can only reduce it to $\mathcal{A}\, t$. Hence, in order to be able to proceed with evaluation of t, we need to add a special, top-level rule, which can only be applied in the empty context:[4]

$$\mathcal{A}\, t \triangleright t.$$

Reasoning with non-local reductions

Non-local rules hinder reasoning about programs. The notion of reduction for the extended language induces an equality relation which is not a congruence, hence it does not satisfy the Church-Rosser property. Consequently, equality in the calculus is not sound with respect to observational equivalence. For example, $\mathcal{A}\, \lambda x.x$ is equal to $\lambda x.x$, but the two terms clearly do not behave the same in all program contexts; consider the context $[\,]\, \ulcorner 3 \urcorner$: $(\mathcal{A}\, \lambda x.x)\, \ulcorner 3 \urcorner$ yields $\lambda x.x$, and $(\lambda x.x)\, \ulcorner 3 \urcorner$ yields $\ulcorner 3 \urcorner$. However, for λ_{v} extended with $\mathcal{A}$ as well as with Felleisen's $\mathcal{C}$ operators, a subset of all the equations ("safe equations" [83]) provable in the calculus can be distinguished, for which soundness is preserved.

Furthermore, Felleisen observed that the calculus corresponding to his programming language with control can be simplified in two ways:

- One way is to modify the language itself, which led him to extending the λ_{v}-calculus with a *delimited* control operator $\mathcal{F}$ and a delimiter # (pronounced "prompt") [83].

- Another way is to relax Plotkin's criterion of correspondence between the calculus and a semantic evaluation function. Instead of requiring that results returned by a standard reduction function and the evaluation function coincide, a notion of *answer* is introduced in the calculus, from which the corresponding value can be extracted. This approach allowed Felleisen and Hieb to find an axiomatization of "safe equations" and simplify reasoning about programs, but at the price of introducing a complicated evaluation procedure [87].

Applications and limitations

Although fairly intuitive and easy to conduct by hand, reduction semantics is not fit for direct implementations. From the above description, it is not clear how reduction contexts should be represented in a machine, and what is the correct and most efficient way of implementing the two crucial operations: decomposition and plugging. In fact, a closer inspection of the evaluation procedure reveals that directly implementing these operations is far from efficient [72,195], and real implementations differ significantly from this naive approach.

Nonetheless, reduction semantics has its uses:

[4]Felleisen calls the special top-level contractions 'computation rules'.

1. It gives operational intuitions about how expressions are evaluated entirely in terms of the syntax of the language, without introducing additional syntactic or semantic notions, or any machine-specific data structures, like a stack or a state. Hence it may serve as the simplest abstract model of the process of evaluation.

2. It directly facilitates reasoning about programs and developing the corresponding equational theory of the programming language. For typed languages, it can be used for establishing type safety, consisting in proving that well-typed terms do not get stuck (progress), and that types are preserved under reduction (preservation) [147, p. 95] [194].

3. It constitutes a starting point for more refined and implementation-oriented specifications, obtained by more or less systematic methods.

2.2 Abstract machines

According to its name, an abstract machine is a *machine*, i.e., a device used to simulate execution of programs in a sequential manner, but at the same time it is *abstract*, since it abstracts from concrete low-level details of actual hardware used for real implementations.

A bewildering number of abstract machines have been proposed in the literature for functional languages alone, not to mention other programming paradigms. They all differ significantly in the way they are obtained, in their architecture, and in their purpose. Despite wide differences, a common definition characterizes an abstract machine as a state transition systems, i.e., a set of states—also called machine configurations—and a binary relation on this set. Two kinds of states and transitions are distinguished: an *initial state*, i.e., the starting configuration of the machine obtained by applying the *initial transition* to a program we want to execute, and a set of *final states*, from which the result of the computation can be read by applying *final transitions*.

Another definition might be given as a term rewriting system (TRS), where transitions are given by rewrite rules. The resulting TRS, however, is nonstandard: it is restricted by requiring that a rewrite rule can only be applied to the root of a term [104]. Depending on the features of the defined language and the purpose the machine is to serve, configurations contain a varying number of components precisely describing the current state of the computation, possibly using auxiliary, temporary data structures.

Following Ager et al. [3], we distinguish between *abstract* machines and *virtual* machines: the former ones operate directly on the source language, and the latter ones operate on machine code produced by a compiler.[5] In this thesis, we focus on abstract machines and their direct correspondence with reduction semantics. In the case of virtual machines, an intermediate compilation phase has to be considered.

[5]This distinction is not widely recognized in the literature: some authors use the two terms interchangeably, and some refer to implementations of abstract machines as virtual machines [77]. We find it, however, convenient.

Applications

Abstract machines bridge the gap between high-level operational semantics and low-level implementations of a programming language. The lower level of abstraction typically raises issues of efficiency and specific design choices for implementation of concrete language features, at the same time hiding other, irrelevant, aspects of real machine architecture. Hence, a machine gives a more realistic and detailed description of evaluation, at the same time retaining the single-step characterization of the evaluation process.

Most machines, even for the same language, are designed by hand, in a ad hoc manner, using as guideline human ingenuity and intuition. Consequently, such machines have to be proven correct with respect to some independently given semantics of the language. Typically, it is enough to prove that the evaluation function induced by a machine coincides with an already known evaluation function [148]. Sometimes, however, we would like to know what evaluation strategy the machine follows, and to this end, a more involved proof is needed, showing how the machine transitions simulate reduction steps [105].

Given the abundance of abstract machines [77], relatively little effort has been put to design machines in a systematic manner and connecting them with other semantic descriptions. The benefits of such a systematic method are not to be underestimated: the correctness of each new abstract machine no longer needs to be proved from scratch, but it follows from the correctness of the method itself; moreover, the connection between various language features and their efficient machine realization facilitates modular machine design and extensibility.

Machines for λ_{v}

The call-by-value λ-calculus λ_{v} is a prototypical functional programming language, and as such it has been studied extensively. As a consequence, numerous machines have been designed for this language; some of them were invented, and some were more or less systematically developed from other semantic specifications. Some prominent examples of machines for λ_{v} (and its extensions) are: Landin's SECD machine [128] (the first machine for evaluation in the λ-calculus), Felleisen and Friedman's CEK machine [85], Hannan and Miller's CLS machine [104], Leroy's ZINC machine [132], Cousineau et al.'s Categorical Abstract Machine (CAM) [50], and Cardelli's Functional Abstract Machine (FAM) [39]. The former three are abstract machines (i.e., they operate on source terms), and the latter three are virtual machines (i.e., they operate on the code generated by a compiler).

An abstract machine for λ_{v} is presented in Figure 2.1. This machine is a variant of Felleisen and Friedman's original CEK machine, restricted to our smaller language. An additional transition

$$\langle \mathcal{A}\, t,\, e,\, E\rangle_{eval} \quad \Rightarrow \quad \langle t,\, e,\, [\,]\rangle_{eval}$$

has to be added to account for the control operator $\mathcal{A}$.

The CEK machine is probably the simplest one, conceptually, for the call-by-value λ-calculus, and it can be almost directly connected to the corresponding reduction semantics (for more details, see Section 3.1.1). Its configurations

- Terms: $t ::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t_1\, t_2 \mid \mathtt{succ}\; t$
- Values (integers and closures): $v ::= \ulcorner n \urcorner \mid [x,\, t,\, e]$
- Environments: $e ::= e_{empty} \mid e[x \mapsto v]$
- Evaluation contexts: $E ::= [\,] \mid \mathtt{ARG}(t, e, E) \mid \mathtt{SUCC}\; E \mid \mathtt{FUN}(v, E)$
- Initial transition, transition rules, and final transition:

$$\begin{array}{rcl}
t & \Rightarrow & \langle t,\, e_{empty},\, E\rangle_{eval} \\[1ex]
\langle \ulcorner n \urcorner,\, e,\, E\rangle_{eval} & \Rightarrow & \langle E,\, \ulcorner n \urcorner\rangle_{apply} \\
\langle x,\, e,\, E\rangle_{eval} & \Rightarrow & \langle E,\, e\,(x)\rangle_{apply} \\
\langle \lambda x.t,\, e,\, E\rangle_{eval} & \Rightarrow & \langle E,\, [x,\, t,\, e]\rangle_{apply} \\
\langle t_1\, t_2,\, e,\, E\rangle_{eval} & \Rightarrow & \langle t_1,\, e,\, \mathtt{ARG}(t_2, e, E)\rangle_{eval} \\
\langle \mathtt{succ}\; t,\, e,\, E\rangle_{eval} & \Rightarrow & \langle t,\, e,\, \mathtt{SUCC}\; E\rangle_{eval} \\[1ex]
\langle \mathtt{ARG}(t, e, E),\, v\rangle_{apply} & \Rightarrow & \langle t,\, e,\, \mathtt{FUN}(v, E)\rangle_{eval} \\
\langle \mathtt{FUN}([x,\, t,\, e], E),\, v\rangle_{apply} & \Rightarrow & \langle t,\, e[x \mapsto v],\, E\rangle_{eval} \\[1ex]
\langle \mathtt{SUCC}\; E,\, \ulcorner n \urcorner\rangle_{apply} & \Rightarrow & \langle E,\, \ulcorner n+1 \urcorner\rangle_{apply} \\
\langle v,\, [\,]\rangle_{apply} & \Rightarrow & v
\end{array}$$

Figure 2.1: An environment-based abstract machine for the λ_v-calculus (CEK)

consist of three components: a term, an environment, and an evaluation context. The term component is the currently evaluated program fragment, the environment stores values to be substituted for free variables in the term, and the evaluation context (which uses the same constructors as reduction contexts) represents the part of the program remaining to be evaluated.

There are two kinds of configurations: *eval*-configurations consist of a term, an environment, and a stack; the *cont*-configurations are pairs consisting of a stack and a value.[6] The top transition corresponds to 'loading' the machine with a given term to evaluate; the bottom one corresponds to 'unloading' the value from the final configuration.

The term component in an *eval*-configuration is the currently evaluated part of the program, and its evaluation context is kept separate as a stack of elementary contexts (hence a slightly different, stack-like notation: elements of the stack are tagged, and $\mathtt{ARG}(t, e, E)$ corresponds to the evaluation context $E[[\,]\,(t, e)]$, $\mathtt{FUN}(v, E)$—to $E[v\,[\,]]$, and $\mathtt{SUCC}\; E$—to $E[\mathtt{succ}\,[\,]]$).

[6]The CEK machine can be presented also using only one kind of *eval*-configurations; the reason for using the current form will become apparent in Section 3.1.2.

A significant difference from the reduction semantics is the handling of substitutions. The direct realization of the abstract definition of substitution is highly inefficient, since it requires traversing a term and rewriting it. A simple and elegant solution to this problem, first introduced in Landin's SECD machine [128], is to use closures and environments instead. A closure is a representation of a functional value: it consists of a λ-abstraction together with an environment storing the values of its free variables. Thus, an environment is a mapping from variables to values.[7] A binding in the environment is created whenever a value to be substituted for a variable has been computed (the penultimate transition); and it is accessed only when actually needed in the evaluation of the body of the corresponding λ-abstraction.

Since Landin's SECD machine, the technique of using environments and closures has become standard for languages with binding constructs.

Extensions

Extending the CEK machine with primitive operations, branching constructs or recursion does not raise any conceptual problems, and can be done pretty straightforwardly. However, adding non-functional features to the language may require substantial modifications in the machine architecture. Consider, for example, extending the CEK machine with an assignment operator with its usual semantics. The environment technique is no longer enough to implement it, since assignments may affect the program globally, and environments are local to the currently evaluated subterm. Therefore, a typical approach is to extend machine configurations with a (global) *store* component [171]. The environment then maps variables to locations (i.e., addresses in the store), and the store associates closures with locations, whose assigned values are accessible for the entire program. An example machine for a minimal Scheme-like language is presented in Chapter 5.

Machines for call-by-name and call-by-need evaluation

In call-by-name evaluation, an argument to a function is not evaluated until needed, but then it might be evaluated more than once, if it is used several times in the body of the function. Although this strategy was implemented in Algol 60, later call-by-name languages adopted the *call-by-need* (or *lazy*) evaluation strategy instead, where an argument is evaluated only once, and only when needed. Due to these pragmatic considerations, relatively few machines have been proposed for ordinary call-by-name evaluation, and much more effort has been put to devising new and optimizing existing machines for call-by-need evaluation. Of the former, the most prominent and the simplest is the Krivine machine [125], which arose out of logical considerations, and which has been used mainly as a tool for theoretical studies.

[7]Both values and closures are considered also in a more general setting: closures can be arbitrary terms paired with environments, and environments can map variables to these general closures (see Chapter 4).

Of the latter, there is a number of virtual machines exploiting Wadsworth's idea of graph reduction,[8] and operating on combinators rather than source terms in order to avoid explicit variable handling [114, 182]. Examples of these machines include: the SKI-reduction machine [182], the G-machine [12, 116], its optimized version—the Spineless-Tagless G-machine [146], and the Three Instruction Machine [81]. Models of lazy evaluation via abstract machines have also been given, mainly in order to provide a direct connection to an underlying lazy calculus. For example, Sestoft derived an abstract machine from Launchbury's natural semantics [161]. In Chapter 5, we present a similar machine, due to Ager et al. [4], in the context of a calculus of closures. These machines utilize the same idea as machines dealing with assignment described above: they thread a global store for keeping values of evaluated function arguments, which can then be accessed as many times as needed in the body of the function.

2.3 Interpreters

An interpreter is a program, written in a metalanguage, that executes programs written in a, possibly different, object language.[9] More precisely, given a representation of an object-language expression as its input data, an interpreter computes the representation of the value of that expression.

As such, an interpreter provides a direct implementation of the object language, alternative to compilation into some machine language, followed by execution of the machine code. In general, interpreting programs gives greater control over the object language, though at the cost of efficiency, which in turn is the main objective of the compilation method. Therefore, interpreters are commonly used for prototyping programming languages, i.e., defining, testing and implementing new language features.

Consequently, writing an interpreter for a language provides an alternative way of defining its semantics. In this context, the object language is also called the *defined* language, and the metalanguage is called the *defining* language (this terminology is due to Reynolds [155]). More specifically, we can write a definitional interpreter, i.e., a program in the defining language, that takes as input a program in the defined language, and the answer it produces is taken as the "meaning" of the source program. Thus, an interpreter can be seen as specifying an evaluation function for the defined language. Of course, the meaning of the interpreter itself (and hence, the semantics of the defined language) depends on the semantics we give to the defining language, and writing a definitional interpreter requires a good understanding of this semantics. In fact, often the reason why we use this indirect method of assigning meaning to new languages is precisely that: we already have a simple and well-understood, well-defined and expressive enough defining language.

[8]Wadsworth observed that a program can be represented as a graph rather then a tree, thus allowing common subexpressions to be shared. Reduction is performed by graph rewriting, and, consequently, a shared expression needs to be evaluated only once, affecting all other expressions pointing to it.

[9]Interpreters for functional languages are also called evaluators, since they implement the process of evaluating expressions.

Not surprisingly, the λ-calculus (or, more precisely, a programming language based on the λ-calculus) turns out to be a good candidate for a defining language: it is concise and elegant, but at the same time expressive and precisely defined; the former features enhance clarity and intuitive understanding of interpreters, and the latter ones account for its wide applicability.

Furthermore, when the defined language possesses an independently given denotational semantics, then an interpreter can be obtained as an implementation of this semantics. Roughly, the λ-notation used for describing denotational semantics can be treated as a defining language, and the denotation of an object-language construct can be taken as its defining clause in the evaluator. However, denotational semantics does not use the λ-notation as a programming language with a fixed operational behavior, but as a description language, statically modelling relationships between its objects. Therefore, in order to obtain a well-defined interpreter from such a description, one must *impose* a specific reduction strategy on the λ-calculus. Otherwise, a rather undesirable situation might arise where the evaluator gives different semantics to the defined language, depending on its own evaluation strategy [155, 171].

Meta-circularity

An interpreter for which the defined and the defining languages are the same, is called meta-circular [155]. Such a self-interpreter, however, does not provide a complete description of the semantics of its defined language; in order to make sense of a meta-circular interpreter, the meaning of the defining language (which, in this case, might be a subset of the defined language) has to be known already. Practical programming languages often include a multitude of derived expressions ("syntactic sugar") which can be unfolded into their definitions in the core language, and one can then write a meta-circular interpreter that uses only these core features of the language. Such interpreters may serve as a pedagogical means to enhance the programmer's understanding of the defined language, or as a basis for implementations of various language extensions.[10] The first meta-circular interpreter was McCarthy's definition of Lisp [137].

Example evaluators for λ_v

Below, we present two evaluators for the language λ_v: one in direct style (in Figure 2.4) and one in continuation-passing style (in Figure 2.5). The syntax of λ_v is encoded as an ML data type `term`, presented in Figure 2.3. In addition, Figure 2.2 contains a common signature for implementation of environments used by these evaluators. In the following figures we use a structure `Env` that implements the signature `ENV`. However, the specific details of this implementation are irrelevant here, and therefore omitted.

For each of the evaluators, we choose ML as the defining language (using only its core features which boil down to a dialect of the call-by-value λ-calculus), and we represent source terms as elements of an ML data type

[10] Taking the idea of meta-circularity one step further, one can define reflective interpreters that permit the user to access the current state of computation, and extend, or even modify the interpreter. The foundational work on computational reflection was done by Smith [166].

```
signature ENV
= sig
    type 'a env
    type ide

    val init_env : 'a env
    val extend : 'a env * ide * 'a -> 'a env
    val lookup : 'a env * ide -> 'a
  end
```

Figure 2.2: An implementation of environments

```
structure Syntax
= struct
    type ide = string

    datatype term = NUM of int
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
                  | SUCC of term
end
```

Figure 2.3: An ML representation of the syntax of λ_v

`term`. Both expressible values (i.e., values of source programs) and denotable ones (i.e., entities bound to variables) are elements of the data type `value`, representing integers and functions. Both evaluators are compositional and higher order, in the sense that ML functions can be passed as arguments or returned as values of expressions.

The first evaluator is in direct style and it interprets each construct of the defined language by the corresponding ML construct.[11] In particular, for the clause handling application, in order to ensure the call-by-value order of evaluation in the defined language, we rely on the following evaluation strategy employed by ML implementations: first, the expression `eval (t0, env)` is evaluated and its result should be a function, which is then applied to the result of evaluating `eval (t1, env)`.

However, the direct-style evaluator is not directly extensible to the language $\lambda_v^{\mathcal{A}}$ and its $\mathcal{A}$ operator. In analogy to the reduction-semantic considerations of Section 2.1, we see that it is not possible to interpret $\mathcal{A}$ without explicit access to the context of computation. Therefore, we rewrite the evaluator of Figure 2.4 by adding a *continuation* parameter to each function (i.e., we write it in *continuation-passing style*, see 3.1.2.1), which provides exactly the "explicit

[11]Whenever a stuck term is reached according to the reduction semantics of Section 2.1, an ML pattern matching exception will be raised, and whenever a term diverges in the reduction semantics, so does the interpreter.

```
structure Eval
= struct
  local open Syntax in
    datatype value = INT of int
                   | FUN of value -> value

    (*  eval : term * value Env.env -> value  *)
    fun eval (NUM n, env)
        = INT n
      | eval (VAR x, env)
        = env x
      | eval (LAM (x, t), env)
        = FUN (fn v => eval (t, Env.extend (env, x, v)))
      | eval (APP (t0, t1), env)
        = let val FUN f = eval (t0, env)
          in f (eval (t1, env))
          end
      | eval (SUCC t, env)
        = let val INT n = eval (t, env)
          in INT (n + 1)
          end

    (*  evaluate : term -> value  *)
    fun evaluate t
        = eval (t, Env.init_env)
  end
end
```

Figure 2.4: A direct-style evaluator for λ_v

access to the context of computation": a continuation is a function representing the rest of the computation that remains to be completed after the current computation is finished [142, 155, 173]. The activation of the continuation is obtained by applying it to a value. The initial (or top-level) continuation is the identity function, which does nothing except returning the value obtained from previous computation. In particular, in the clause interpreting $\mathcal{A}$, the current continuation `k` is discarded, and the evaluation of the term `t` proceeds with the top-level continuation. Moreover, by using continuations, we can directly impose a specific order of evaluation for the defined language, without relying on the order of evaluation of the defining language (in the clause for application, we require that the operator is evaluated first, and when the continuation is applied, the operand will be evaluated, and only after it has been evaluated to a value, the application can take place). This observation was first made by Reynolds [155]. The CPS-transformed evaluator is presented in Figure 2.5. It is then possible to add an extra clause to interpret the $\mathcal{A}$ operator:

```
datatype term = ...
              | ABORT of term

fun eval ...
  | eval (ABORT t, env, k)
```

```
structure EvalCPS
= struct
  local open Syntax in
    datatype value = INT of int
                   | FUN of value * cont -> answer
    withtype answer = value
    and cont = value -> answer

    (*  eval : term * value Env.env * cont -> answer  *)
    fun eval (NUM n, env, k)
        = k (INT n)
      | eval (VAR x, env, k)
        = k (env x)
      | eval (LAM (x, t), env, k)
        = k (FUN (fn (v, k') =>
                    eval (t, Env.extend (env, x, v), k')))
      | eval (APP (t0, t1), env, k)
        = eval (t0, env,
                fn FUN f => eval (t1, env,
                                  fn v => f (v, k)))
      | eval (SUCC t, env, k)
        = eval (t, env, fn INT n => k (INT (n + 1)))

    (*  evaluate : term -> answer  *)
    fun evaluate t
        = eval (t, Env.init_env, fn v => v)
  end
end
```

Figure 2.5: A continuation-passing evaluator for λ_{v}

```
        = eval (t, env, fn v => v)
```

`ABORT t` is interpreted simply by evaluating `t` with the initial continuation while discarding the current one.

Examining the reduction semantics of λ_{v} and $\lambda_{\mathrm{v}}^{\mathcal{A}}$ from Section 2.1 and the continuation-based evaluator, it is not difficult to see that evaluation contexts and continuations are in close correspondence, which is further explored in Chapter 3.

Chapter 3

Connecting specifications

In this chapter, we investigate some of the interconnections between the three kinds of semantic specifications introduced in Chapter 2: reduction semantics, abstract machines and evaluators. As can be inferred from the previous chapter, reduction semantics is by far the least practical specification of the three, although it is a natural starting point for implementations, since it provides the necessary intuition about the computation process and is relatively simple to carry out by hand. On the other hand, the process of designing and optimizing intermediate-level abstract machines is typically carried out in isolation from existing high-level semantic specifications for the language. It is then a (usually tedious and unilluminating) proof of correctness that guarantees the machine induces the same evaluation function as the high-level semantics. Therefore, it certainly is advantageous to try to automatically (or at least mechanically) *transform* an existing high-level semantic description into an abstract machine so that the correctness of the outcome is guaranteed by the correctness of the transformation itself. Through such derivations, some aspects of computation left implicit at the higher level are made explicit at the lower level, which further enhances the understanding of the nature of computation and the way the language can be implemented and extended. On a global scale, this constructive approach promotes modularity in language development: many languages share certain common features, and once reasonable and efficient implementation methods have been derived for them, they can be reused for other languages.

Furthermore, reversing the derivation (if possible) abstracts from concrete implementation details, and the resulting higher-level semantics becomes more understandable for the language designer and better suited for reasoning about the evaluation process. In consequence it may allow for comparing different abstract machines with differing architectures at the level of their underlying evaluation strategy.

In the remainder of this chapter, we first report on two derivation methods—used in the second part of the thesis—connecting abstract machines with high-level specifications by means of known program transformations: the first method, a syntactic correspondence, is described in Section 3.1.1; the second method, a functional correspondence, is described in Section 3.1.2. We illustrate the derivations on the example language λ_{v}. The two methods also

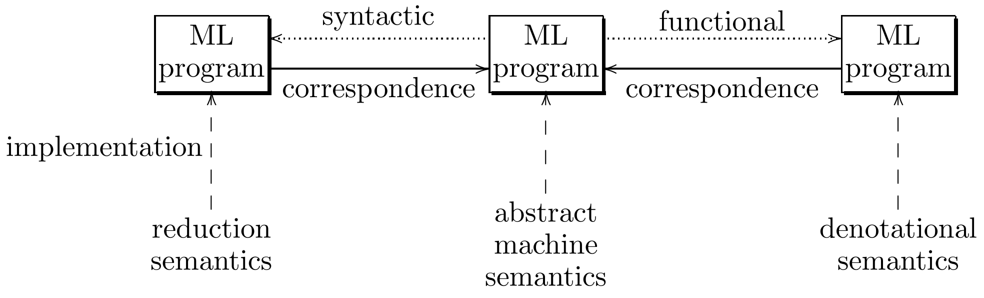

Figure 3.1: The connections between semantic specifications

provide an indirect derivation from reduction semantics and higher-order evaluators via constructing an intermediate abstract machine. A quite different approach, based on the proofs-as-programs interpretation, is described in Section 3.2. This method only works in a typed setting, whereas the program-transformation-based methods are essentially untyped.

3.1 Methods based on program transformations

The derivation methods we consider here are based on known program transformations. Therefore, before applying them, we first implement the different semantic specifications as programs in a functional language; for this purpose, we choose Standard ML [138], whose core is essentially the call-by-value λ-calculus. Figure 3.1 summarizes the situation. Functional languages are particularly well suited as implementation languages for the abstract semantic specifications used in the previous chapter, because the process of implementation boils down to almost transliterating the mathematical notation into a slightly different grammar. A formal proof of correctness for such implementations, however, can be quite involved and lies beyond the scope of this dissertation.

Example implementations

In Figures 3.2 and 3.4 we present implementations of the abstract machine and the reduction semantics for λ_v specified abstractly in Sections 2.2 and 2.1, respectively. Since in the reduction semantics there is a separate syntactic category for values (as a subset of terms), we modify the representation of the syntax of λ_v to that of Figure 3.3 (with a separate data type for values), and we use it in the subsequent derivation (throughout Figures 3.4-3.6).

Abstract machine: The abstract machine of Figure 2.1 is implemented as a first-order evaluator. The call to function `evaluate` corresponds to the initial transition, and the two mutually recursive functions `eval` and `apply` implement the two corresponding kinds of transitions of the machine. Values and evaluation contexts are represented as ML data types: `value` and `evctx`, respectively.[1]

[1]Note that values do not form a subset of terms, as in the reduction semantics.

```
structure EvaluatorAM
= struct
  local open Syntax in
    datatype value = INT of int
                   | CLO of ide * term * value Env.env

    datatype evctx = MT
                   | ARG of term * value Env.env * evctx
                   | FUN of value * evctx
                   | SUC of evctx

    (*  eval : term * value Env.env * evctx -> value  *)
    fun eval (NUM n, env, k)
        = apply (k, INT n)
      | eval (VAR x, env, k)
        = apply (k, Env.lookup (env, x))
      | eval (LAM (x, t), env, k)
        = apply (k, CLO (x, t, env))
      | eval (APP (t0, t1), env, k)
        = eval (t0, env, ARG (t1, env, k))
      | eval (SUCC t, env, k)
        = eval (t, env, SUC k)
    and
    (*  apply : evctx * value -> value  *)
        apply (ARG (t1, env, k), v)
        = eval (t1, env, FUN (v, k))
      | apply (FUN (CLO (x, t, env), k), v)
        = eval (t, Env.extend (env, x, v), k)
      | apply (SUC k, INT n)
        = apply (k, INT (n+1))
      | apply (MT, v)
        = v

    (*  evaluate : term -> value  *)
    fun evaluate t
        = eval (t, Env.init_env, MT)
  end
end
```

Figure 3.2: An ML implementation of the abstract machine for λ_v

Reduction semantics: In the previous chapter we described reduction semantics using relational notation for reductions, and assuming a decomposition of terms into reduction contexts and redexes. Due to the unique-decomposition property, decomposition is deterministic and can be implemented as an ML function `decompose`; furthermore, syntax-directed definitions of contraction and plugging naturally give rise to the functions `contract` and `plug`, respectively. We defer the implementation of `decompose` until Section 3.1.1. Call-by-value reduction contexts are represented as an ML data type `redctx`; the transitive closure of the one-step reduction is a function implemented by `iterate`, and finally, the evalua-

```
structure Syntax_with_values
= struct
    type ide = string

    datatype term = VAL of value
                  | VAR of ide
                  | APP of term * term
                  | SUCC of term
    and
        value = NUM of int
              | LAM of ide * term
    (*
    subst : term * ide * value -> term

    ... standard definition omitted
    *)
end
```

Figure 3.3: An ML representation of the terms and values of λ_v

tion function is implemented by `evaluate`. Moreover, reduction semantics treats values as syntactic entities, hence they are implemented here as a subset of terms, using the data type `value`.

The evaluation functions defined by the reduction semantics and the abstract machine from the previous chapter are undefined for stuck and open terms. In the implementations, evaluating such incorrect terms raises an ML runtime error (non-exhaustive pattern matching). This situation could easily be prevented if we explicitly handled incorrect terms in the program (which is doable, since we can characterize ill-formed terms syntactically), but for the sake of simplicity we disregard this issue here.

3.1.1 A syntactic correspondence

At an appropriate level of abstraction, an abstract machine and the corresponding reduction semantics appear to be very similar (compare, e.g., Figures 3.4 and 3.2) in that they define a small-step operational semantics with each step directed by the structure of the currently evaluated subterm, and with an explicit first-order representation of reduction/evaluation contexts. This similarity is not coincidental and can be explored to derive abstract machines from reduction semantics, as proposed by Danvy and Nielsen [72]. In this section, we briefly sketch the idea of *refocusing* and we illustrate it on the example language λ_v.

```
structure EvaluatorRS
= struct
  local open Syntax_with_values in

    datatype redctx = MT
                    | ARG of redctx * term
                    | FUN of redctx * value
                    | SUC of redctx

    fun plug (t, MT)
        = t
      | plug (t0, ARG (ec, t1))
        = plug (APP (t0, t1), ec)
      | plug (t, FUN (ec, v))
        = plug (APP (VAL v, t), ec)
      | plug (t, SUC ec)
        = plug (SUCC t, ec)

    datatype redex = BETA of value * value
                   | DELTA of value

    fun contract (BETA (LAM (x, t), v))
        = subst (t, x, v)
      | contract (DELTA (NUM n))
        = VAL (NUM (n+1))

    datatype decomposition = VL of value
                           | REDEX of redex * redctx
    (*
    decompose : term -> decomposition
    ... definition omitted
    *)

    fun iterate (VL v)
        = v
      | iterate (REDEX (r, ec))
        = iterate (decompose (plug (contract r, ec)))

    fun evaluate t
        = iterate (decompose t)
  end
end
```

Figure 3.4: An ML implementation of the reduction semantics for λ_v

The construction of a refocus function

Recall from Section 2.1 that the evaluation process in a reduction semantics consists in repeatedly applying the following three steps:

1. decomposing a term into a potential redex and a reduction context
2. contracting the redex, if an actual redex was found
3. plugging the contractum into the context

Decomposition and plugging are naturally implemented by recursive descent (linear in the size of the term to decompose) on terms and reduction contexts, respectively (cf. Figure 3.4). However, the plugging operation constructs intermediate terms that will immediately be decomposed by the following decomposition step, as depicted in the following diagram:

$$\begin{tikzcd}
{} \arrow[d, dashed] & t_0 \arrow[dl, "\texttt{decompose}"] \arrow[ddd, dotted, "\rightarrow_{\mathrm{v}}"] \\
E_1[t_1] \arrow[d, "\texttt{contract}"] & \\
E_1[t_1'] \arrow[dr, "\texttt{plug}"] \arrow[dd, dashed, "\texttt{refocus}"'] & \\
 & t_1 \arrow[dl, "\texttt{decompose}"] \arrow[ddd, dotted, "\rightarrow_{\mathrm{v}}"] \\
E_2[t_2] \arrow[d, "\texttt{contract}"] & \\
E_2[t_2'] \arrow[dr, "\texttt{plug}"] \arrow[dd, dashed, "\texttt{refocus}"'] & \\
 & t_2 \arrow[dl, "\texttt{decompose}"] \arrow[ddd, dotted, "\rightarrow_{\mathrm{v}}"] \\
E_3[t_3] \arrow[d, "\texttt{contract}"] & \\
E_3[t_3'] \arrow[dr, "\texttt{plug}"] \arrow[d, dashed] & \\
{} & t_3
\end{tikzcd}$$

This reconstruction of intermediate terms is the source of inefficiency in the direct implementation of the decompose-contract-plug loop.

The optimization identified by Danvy and Nielsen consists in replacing the composition of `decompose` and `plug` by a more efficient, yet extensionally equivalent, function `refocus` that given a current decomposition into a term `t` and a context `rc` directly computes the next decomposition into a potential redex `r` and a context `rc′`, such that `refocus (t, rc)` = `(decompose o plug)(t, rc)` = `DECOMP (r, rc′)`.

Danvy and Nielsen give an explicit algorithm for constructing the `refocus` function from scratch, provided the reduction semantics satisfies a few (reasonable) conditions:

1. unique decomposition
2. fixed order of evaluation, typically left-to-right (i.e., subterms in a syntactic construct are evaluated from left to right)
3. values and potential redexes must be generated by different sets of syntactic constructors
4. compact specification, i.e., no redundant constructs are allowed

It is then possible to define `refocus` by a pair of mutually recursive functions, both taking as arguments the current decomposition, i.e., a term and a stack of elementary contexts forming the current evaluation context. Roughly, one of them is defined by cases over the term structure: at each step—due to the unique-decomposition property—the term argument can be identified either as a value, a potential redex, or it can be further decomposed. In the latter case, the current term can be split into a new term and a (topmost) elementary context, and the function iterates on this new term and the elementary context accumulated with the previous one. In the remaining cases, either a complete decomposition is obtained, or the second function is called—one that dispatches on the reduction context to find the next subterm to evaluate. The algorithm has been proven correct and at least as efficient as the composition of `decompose` and `plug` [71].

Let us illustrate the refocusing algorithm on the example language λ_v. The ML implementation of the function `refocus` and the auxiliary function `refocus_aux` is as follows:

```
(*  refocus : term * redctx -> decomposition  *)
fun refocus (VAL v, rc)
    = refocus_aux (rc, v)
  | refocus (APP (t0, t1), rc)
    = refocus (t0, ARG (rc, t1))
  | refocus (SUCC t, rc)
    = refocus (t, SUC rc)
and
(*  refocus_aux : redctx * value -> decomposition  *)
    refocus_aux (ARG (rc, t), v)
    = refocus (t, FUN (rc, v))
  | refocus_aux (MT, v)
    = VALUE v
  | refocus_aux (FUN (rc, v), w)
    = DECOMP (BETA (v, w), rc)
  | refocus_aux (SUC rc, v)
    = DECOMP (DELTA v, rc)
```

As described above, `refocus` dispatches on the currently processed term, and `refocus_aux` dispatches on the current evaluation context when the currently evaluated subterm is identified as a value.

In the implementation of Figure 3.4, we now replace the composition of `decompose` and `plug` with `refocus`; the resulting interpreter is shown in Figure 3.5. In the definition of `evaluate`, we used the fact that `decompose t` = `(decompose o plug)(t, MT)`, and we replaced it with `refocus (t, MT)`.

This interpreter avoids reconstructing intermediate terms by using the transition functions `refocus` and `refocus_aux`, but it is not a transition system yet, because it still stops at each potential redex and returns the control to the trampoline function `iterate` (in Danvy and Nielsen's terminology, this interpreter is called a *pre-abstract machine*).

The resulting abstract machines

In a pre-abstract machine, all the components of the underlying reduction semantics are easily distinguishable: we can read the grammar of values and potential redexes from from `iterate`, the notion of reduction is given by `contract`, and decomposition is essentially defined by `refocus` (i.e., `decompose t` = `refocus (t, MT)`). Finally, reverting the transitions of `refocus` in effect defines the plug function: the evaluation contexts is deconstructed from inside out, and the intermediate term is accumulated until the context becomes empty.

All this information is not necessary if we are only interested in a transition system that computes the final result of evaluation. If we now analyze the control flow of the refocused interpreter, we can transform it into a proper abstract machine (i.e., a transition system) by a series of simplifications:

1. we inline the `contract` in the definition of `iterate`

2. we observe that each call to `iterate` always follows a call to `refocus`; we merge the two functions into one—the result is a standard eval/apply abstract machine [134]

3. finally, we can inline `refocus_aux` in `refocus`—the resulting abstract machine is a standard push/enter architecture [134]

The eval/apply machine for the example language λ_v is shown in Figure 3.6—it coincides with Felleisen and Friedman's CK machine [85]. In some cases, further intermediate simplifications are possible to eliminate redundant transitions. In Figure 3.6, in the last clause of `refocus_aux`, we unfold the transition to `refocus (VAL (NUM (n+1)), rc)` by directly calling `refocus_aux (rc, NUM (n+1))`. (Note that after this unfolding, the transformation into a push/enter machine is no longer possible, because `refocus_aux` is now recursive).

```
structure RefocusedEvaluator
= struct
  local open Syntax_with_values in
    datatype redex = BETA of value * value
                   | DELTA of value

    datatype decomposition = VALUE of value
                           | DECOMP of redex * redctx
    and redctx = MT
               | ARG of redctx * term
               | FUN of value * redctx
               | SUC of redctx

    (*
    subst : term * ide * value -> term
    ... standard definition omitted
    *)

    (*  refocus : term * redctx -> decomposition  *)
    fun refocus (VAL v, rc)
        = refocus_aux (rc, v)
      | refocus (APP (t0, t1), rc)
        = refocus (t0, ARG (rc, t1))
      | refocus (SUCC t, rc)
        = refocus (t, SUC rc)
    and
    (*  refocus_aux : redctx * value -> decomposition  *)
        refocus_aux (ARG (rc, t), v)
        = refocus (t, FUN (v, rc))
      | refocus_aux (MT, v)
        = VALUE v
      | refocus_aux (FUN (v, rc), w)
        = DECOMP (BETA (v, w), rc)
      | refocus_aux (SUC rc, v)
        = DECOMP (DELTA v, rc)

    (*  contract : redex -> term  *)
    fun contract (BETA (LAM (x, t), v))
        = subst (t, x, v)
      | contract (DELTA (NUM n))
        = VAL (NUM (n+1))

    (*  iterate : decomposition -> value  *)
    fun iterate (VALUE v)
        = v
      | iterate (DECOMP (r, rc))
        = iterate (refocus (contract r, rc))

    (*  evaluate : term -> value  *)
    fun evaluate t
        = iterate (refocus (t, MT))
  end
end
```

Figure 3.5: The refocused implementation of the reduction semantics for λ_v

```
structure CK
= struct
  local open Syntax_with_values in
    datatype redctx = MT
                    | ARG of redctx * term
                    | FUN of value * redctx
                    | SUC of redctx

    (*
    fun subst : term * ide * value -> term
    ...
    *)

    (*  refocus : term * redctx -> value  *)
    fun refocus (VAL v, rc)
        = refocus_aux (rc, v)
      | refocus (APP (t0, t1), rc)
        = refocus (t0, ARG (rc, t1))
      | refocus (SUCC t, rc)
        = refocus (t, SUC rc)
    and
    (*  refocus_aux : redctx * value -> value  *)
        refocus_aux (ARG (rc, t), v)
        = refocus (t, FUN (v, rc))
      | refocus_aux (MT, v)
        = v
      | refocus_aux (FUN (LAM (x, t), rc), v)
        = refocus (subst (t, x, v), rc)
      | refocus_aux (SUC rc, NUM n)
    (*  = refocus (VAL (NUM (n+1)), rc)  *)
        = refocus_aux (rc, NUM (n+1))

    (*  evaluate : term -> value  *)
    fun evaluate t
        = refocus (t, MT)
  end
end
```

Figure 3.6: The eval/apply machine for λ_v (the CK machine)

The final abstract machine is a transition system implementing the evaluation function of the reduction semantics, but in a more efficient way than the direct decompose-contract-plug cycle. The optimization steps performed after refocusing, however, are not reversible (at least not directly—recovering all the ingredients of the reduction semantics requires some insight and needs to be verified independently). The derivation shows that reduction contexts of a reduction semantics coincide with evaluation contexts of the derived abstract machine [62].

Environment-based vs. substitution-based abstract machines

We have shown that the method of refocusing applied to a reduction semantics yields a transition system. The method is purely syntactic, hence all the components of the resulting abstract-machine configurations must already be present in the starting point of the derivation, i.e., in the grammar of the language. In particular, when compared with the abstract machine of Figure 3.2, the transition system of Figure 3.6 differs in that it performs actual substitution in the β-contraction step instead of threading an environment (and, in consequence, the expressible values of the two machines differ as well). However, the difference in treating substitution is the only one: if we eliminate the environment from the machine of Figure 3.2 by analyzing the transitions and performing the actual substitution instead of extending the environment in the clause for beta-reduction, we obtain the machine of Figure 3.6 [28, 85]. On the other hand, if we were to systematically derive the environment-based machine using refocusing, it is clear that we would have to start off with a different language and a somehow modified reduction semantics that contains a syntactic counterpart of an environment. This idea of incorporating environments in the syntax and modelling "delayed substitutions" rather than actual substitutions has spurred a number of *calculi of explicit substitutions* [1, 105, 133]. For the purpose of deriving abstract machines, it is enough to consider simple *calculi of closures*, as first proposed by Curien [54]. In Chapters 4 and 5 we show how refocusing can be applied to calculi of closures, yielding environment machines.

Refocusing for non-local contractions

Refocusing as described above is not applicable when the starting point is a non-standard reduction semantics with non-local contractions, e.g., the language $\lambda_v^{\mathcal{A}}$ with the $\mathcal{A}$ operator. The reduction sequence cannot then be implemented as a standard decompose-contract-plug loop, but it has to be modified to account for the non-compatible notion of reduction: in particular, the contraction function must take as arguments both a redex and the current reduction context and return a contractum and a (possibly altered) context. In Chapter 5 we show how refocusing can be applied to such context-sensitive reductions.

3.1.2 A functional correspondence

The evaluator of Figure 3.2 is the first-order version of the higher-order one of Figure 2.5, and it can be mechanically derived from it using defunctional-

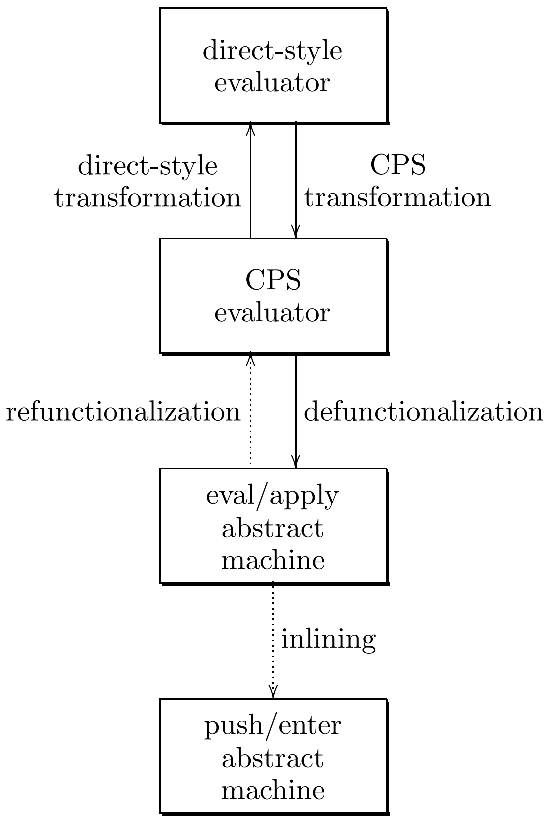

Figure 3.7: The functional correspondence between evaluators and abstract machines

ization.[2] The CPS evaluator of Figure 2.5, in turn, can be obtained from the direct-style one of Figure 2.4 by CPS transformation. A schematic view of the correspondence between all these artifacts, as identified by Ager et al. [3–5, 63], is presented in Figure 3.7. The crucial observation is that (first-order representations of) evaluation contexts arise as defunctionalized continuations of an evaluator in continuation-passing style. Let us briefly review the two transformations involved: the CPS transformation and defunctionalization.

3.1.2.1 The CPS transformation

In the previous chapter we showed how continuations can be used in a definitional interpreter to account for non-local control transfer and to ensure order-of-application independence. Supplying each function with a continuation parameter gives access to the "rest of the computation." The idea of abstracting this "rest of the computation" as a function has proven useful in other settings as well (e.g., denotational semantics and logic), but its origins can be traced back to van Wijngaarden's introduction of continuations to translate

[2]The substitution-based version of this machine—the CK machine of Figure 3.6—is also in defunctionalized form, and it can be refunctionalized into a higher-order substitution-based evaluator.

Algol 60 programs into a more restrictive language without explicit labels and jumps [183], i.e., continuations were first used for program transformations. In the context of functional languages, the continuation-passing style has been discovered multiple times, independently by several authors, e.g., by Fischer and F. L. Morris [93,142,154]; later Reynolds showed how a CPS transformation can encode a specific order of evaluation in the source language [155], and finally Plotkin formalized the call-by-value and call-by-name CPS transformations and showed the corresponding simulation properties [148].

Informally, the transformation into continuation-passing style consists in representing the "rest of the computation" as a function, sequentializing the computation, and naming the intermediate results. It is the sequentialization that disambiguates the order of evaluation by explicitly encoding a particular evaluation strategy.

Let us consider Plotkin's call-by-value CPS transformation for the source language λ_v:

$$\begin{array}{rcl}
\overline{\ulcorner n \urcorner} & = & \lambda k.k\,\ulcorner n \urcorner \\
\overline{x} & = & \lambda k.k\,x \\
\overline{\lambda x.t} & = & \lambda k.k\,(\lambda x.\overline{t}) \\
\overline{\mathtt{succ}\,t} & = & \lambda k.\overline{t}\,(\lambda v.k\,(\mathtt{succ}\,v)) \\
\overline{t_1\,t_2} & = & \lambda k.\overline{t_1}\,(\lambda v_1.\overline{t_2}\,(\lambda v_2.v_1\,v_2\,k))
\end{array}$$

This transformation introduces extra β-redexes—so-called *administrative redexes* [148]—that are not present in the source term. Since these administrative redexes significantly increase the size of resulting terms, it is necessary to reduce them away by partially evaluating the result of the transformation. Thus, Plotkin's CPS translation effectively defines a two-pass transformation, where the first step is to apply the naive translation, and the second step is to simplify the resulting term by performing all the administrative reductions. Alternatively, one can obtain the same result by using a one-pass CPS transformation such as Danvy and Filinski's that explicitly avoids the construction of the intermediate terms [68].

The evaluation of a CPS-transformed program is started with the initial continuation $k_{\mathrm{init}} = \lambda v.v$. The order-of-application independence can then be formulated as follows:

Theorem 3.1 (Plotkin's Indifference Theorem). *For any program t, evaluating $\overline{t}\,k_{\mathrm{init}}$ using the call-by-value or the call-by-name strategy yields the same result.*

The correctness of the CPS transformation is proved by showing that the result of evaluating a CPS-transformed program agrees with the result of evaluating the original program in the following sense:

Theorem 3.2 (Plotkin's Simulation Theorem). *For any program t, $\overline{t}\,k_{\mathrm{init}}$ evaluates to a value v (using either strategy) if and only if t evaluates to a value v' (using call by value) such that $\Phi(v') = v$, where*

$$\begin{array}{l}
\Phi(\ulcorner n \urcorner) = \ulcorner n \urcorner \\
\Phi(\lambda x.t) = \lambda x.\overline{t}
\end{array}$$

Plotkin also proved that both equational theories for the call-by-value and the call-by-name λ-calculus are sound with respect to their respective CPS transformations, and moreover that the call-by-name theory is complete for its CPS transformation. In order to obtain completeness for call by value, additional axioms must be added to the theory, as shown by Sabry and Felleisen [158].

In a CPS-transformed program, continuations occur as second-class objects: they are only allowed to be passed as arguments to functions or to be applied to values. The effective use of continuations as first-class objects adds expressive power to the language, and it requires the use of control operators that explicitly capture the "current continuation" and bind it to a variable that can be then used as any other variable. Not surprisingly, the semantics of control operators is usually given by an extended CPS transformation (or by a continuation semantics). For example, Plotkin's CPS transformation can be extended to account for the $\mathcal{A}$ operator of $\lambda_v^{\mathcal{A}}$ in the following way:

$$\overline{\mathcal{A}\,t} \quad = \quad \lambda k.\overline{t}\, k_{\text{init}}$$

Intuitively, the evaluation of $\mathcal{A}\,t$ with a current continuation k consists in discarding k, and proceeding with the evaluation of the term t with the initial continuation. More examples of control operators and their associated CPS transformations are discussed in Chapters 5 and 6.

As mentioned already in Section 2.3, the evaluator of Figure 2.5 is the result of the (optimized) call-by-value CPS-transformation of the evaluator of Figure 2.4, where the transformation is performed over the metalanguage, i.e., ML, and it extends Plotkin's basic transformation for the core λ-calculus with clauses accounting for ML specific features (recursion, tuples, `let`-expression, pattern matching, etc.).

Direct-style transformation

A left inverse to the call-by-value CPS transformation is Danvy's *direct-style* transformation [58]. Danvy introduced a syntactic characterization of terms in 'direct style' and of terms in CPS (using Reynolds's distinction between 'serious' and 'trivial' terms), and then defined a syntax-directed mapping from the latter set into the former one. The direct-style translation was then extended to account for first-class occurrences of continuations; the image of the extended translation is the pure λ-calculus augmented with the control operator `call/cc`, as found in Scheme [69].

3.1.2.2 Defunctionalization

The aim of defunctionalization is to transform a higher-order functional program into a first-order one. This technique was introduced by Reynolds in the context of definitional interpreters [155], and has since been used not only as a tool for transforming programs and reasoning about them, but also as a compilation technique [37, 38, 180, 192].

In a *higher-order* program, functions are passed as arguments or returned as values of other functions. In a first-order program, however, functions occurring in these positions are eliminated and are only second-class objects. More precisely, in a language with a recursive **let** expression, a first-order program can be expressed as follows:

$$\begin{array}{ll}\textbf{let} & f_1 = \lambda\vec{x}.t_1 \\ & \ldots \\ & f_n = \lambda\vec{x}.t_n \\ \textbf{in} & t \\ \textbf{end} & \end{array}$$

where the function identifiers $f_1, \ldots, f_n$ can only occur in the operator position in applications, and no anonymous functions are allowed in terms $t_1, \ldots, t_n, t$. Moreover, we assume that the source program does not contain local function definitions (which can be achieved by lambda-lifting such local definitions [117]).

Originally, Reynolds introduced defunctionalization for an untyped language. In a typed setting, defunctionalization can be defined as a type-directed translation, both on types and on (typing derivations of) expressions: a function type in the source program is converted to a first-order algebraic data type in the target program, and each λ-abstraction of the function type is then converted to a new constructor for the data type holding the values of the free variables of the original function. Applications of the converted functions are performed using an 'apply' function dispatching on the data type, where each clause corresponds to the body of the original function.

For example, consider the CPS evaluator of Figure 2.5. There are two function types: the type of functions `value * cont -> answer` and the type of continuations: `value -> answer`. In order to defunctionalize the expressible values, we identify the occurrences of this function type (tagged with the `FUN` constructor)—here there is just one, in the clause of `eval` for λ-abstraction, and its free variables are: a variable `x`, a term `t`, and an environment `env`. We thus replace the type `value * cont -> answer` of functions created with the constructor `FUN`, with a first-order *closure* of type `ide * term * value Env .env` introduced with the constructor `CLO`. The corresponding 'apply' function defines the behavior of the continuation when it is applied:

```
fun apply (CLO (x, t, env), v, k)
    = eval (t, Env.extend (env, x, v), k)
```

Here, since there is only one constructor of expressible function values, we can directly inline the corresponding 'apply' function where it is called: in the clause for the application.

The second function type gives rise to four constructors of the first-order type `cont`, corresponding to the four λ-abstractions defining continuations:

1. the initial continuation `fn v => v` gives rise to the nullary constructor `MT` of the new type `cont`

2. two continuation-constructing λ-abstractions occur in the clause for application:

    - `fn v => apply (f, v, k)` gives rise to the binary constructor `FUN of value * cont`, and for

- **`fn`** `(FUN (x, t, env'))=> eval (t1, env, ...)` we introduce the constructor `ARG` **`of`** `term * value Env.env * cont`

3. the continuation constructed in the clause for successor, **`fn`** `INT n => k (INT (n + 1))`, gives rise to the unary constructor `SUC` **`of`** `cont`

Next, all applications of continuations are replaced by a call to the corresponding 'apply' function, dispatching on the type `cont`.

The defunctionalized continuations of an interpreter implement evaluation contexts of the corresponding abstract machine (and, in consequence, the reduction contexts of the underlying reduction semantics) [3, 71]. In the present case, we used the call-by-value CPS transformation and the resulting first-order type `cont` implements the call-by-value evaluation contexts for λ_{v} defined in Section 2.2. In consequence, the defunctionalized evaluator of Figure 2.5 coincides with the implementation of the abstract machine for λ_{v} from Figure 2.1 (modulo the renaming of the defunctionalized type `cont` into `evctx`).

The defunctionalization process performed above is correct in the sense that the source and the target programs are observationally equivalent (here they are written in the same language, ML). More general and formal accounts of defunctionalization have been given by Banerjee et al. and by Nielsen for simply typed source languages [15, 144], and further extended to account for polymorphic types by Bell et al. [18] and by Pottier and Gauthier [150]. In each of these articles, a variant of defunctionalization algorithm is formalized and further proved to preserve the types and the meaning of the original program; the latter property amounts to showing that defunctionalization preserves termination, or that the source and the target programs evaluate to the same base value.

Reynolds's defunctionalization can be seen as an optimized variant of the naive transformation where no control-flow analysis is performed, and therefore all the functions in the program are defunctionalized (as in Nielsen's approach [144]). Annotating the source program with control-flow information allows for more refined translations and further optimizations, e.g., lightweight defunctionalization [15].

Closure conversion

Defunctionalization is a close relative of *closure conversion*, i.e., a program transformation converting functions into *closures*, i.e., pairs of the function code and the environment storing the values of its free variables [128]. In fact, closure conversion can be viewed as a special case of defunctionalization, where the tag associated with each abstraction is a pointer to its code, and the corresponding 'apply' function is inlined. In the abstract machine of Figure 2.1, defunctionalizing the function space in the expressible values effectively yields a closure representation of the corresponding object-level function. Thus, this particular case of defunctionalization can be seen as performing closure conversion on the object-level functions.

Both defunctionalization and closure conversion have been used as compilation techniques as well as tools for reasoning about programs [15,37,38,180,191, 192]. They have been proved correct and type-preserving [15, 18, 139, 144, 150].

Refunctionalization

Roughly, one can identify a defunctionalized higher-order function in a program, if it contains an 'apply' function dispatching on a first-order data type, and such that the data type is consumed once, and only by that function. It is then possible to *refunctionalize* the data type into a higher-order function by essentially reversing the defunctionalization process, i.e., by replacing each occurrence of a data-type constructor with a λ-abstraction whose free variables are parameters of that constructor, and whose body is defined in the corresponding clause of the 'apply' function. However, if the 'apply' function is inlined, then refunctionalization is no longer straightforward, and it requires some insight into the structure as well as the meaning of the program.

3.1.2.3 From evaluator to abstract machine

The two program transformations described above and their left inverses provide a means to systematically derive abstract machines from evaluators and vice versa, as depicted in Figure 3.7, page 36: For a direct-style, lambda-lifted evaluator, CPS-transforming it and then defunctionalizing its expressible values and its continuations yields a transition system with a first-order representation of evaluation contexts.[3] In particular, defunctionalizing the denotable and expressible values of the evaluator amounts to closure-converting the object-level functions.[4] Conversely, refunctionalizing the evaluation contexts of an eval/apply abstract machine yields a higher-order evaluation function. In general, however, it is not trivial to identify a program as being in defunctionalized form [34, 63], and in particular to single out the occurrences of the 'apply' function which may be inlined.

This derivational correspondence between higher-order evaluators and abstract machines has been initiated by Danvy, who presented an evaluator underlying Landin's SECD machine [63], and then explored by Ager et al., who have shown how several other known evaluators and abstract machines, designed independently, can in fact be derived from each other. They exhibited a number of evaluators corresponding to some of the known abstract machines, and derived new machines from known evaluators for different strategies in the λ-calculus [3, 4]. Furthermore, the derivation method can be applied to monadic-style evaluators for languages with effects, providing substantial support in designing abstract machines for such languages—often a non-trivial

[3]More generally, CPS-transforming and then defunctionalizing the continuations in an arbitrary program performing a recursive descent over a data structure, yields a program with a first-order, stack-like accumulator [71].

[4]In fact, in the original presentation of the functional correspondence, the defunctionalization of higher-order values of the direct-style evaluator was performed first, before the CPS transformation step. The order of application of these transformations does not, however, affect the resulting abstract machine. In Figure 3.7, we depict the alternative approach described above.

task. Given a generic monadic-style evaluator, one instantiates it with a specific monad and inline the monad definition, and then applies the derivation method [5].

3.2 Program extraction from proofs

In the previous section we have shown how semantic specifications can be implemented in a functional language and then interderived by means of some standard program transformations. In particular, one can obtain an evaluator from a reduction semantics by first applying refocusing and then refunctionalizing the resulting eval/apply machine into a higher-order evaluator.

In this section, we shift to a different metalanguage—that of a suitable formal logic—and we present another "derivation" of an evaluator from a reduction semantics. The method we use is an application of a general concept of program extraction from proofs, and it is based on the Curry-Howard isomorphism relating formal logics and pure typed computational calculi [56, 113]. We apply it in the setting of a many-sorted first-order minimal logic "implementing" the reduction semantics of the simply typed λ-calculus. A detailed account of two such derivations – for call by name and call by value – is presented in Chapter 7; here we only outline the general method.

The simply typed λ-calculus $\lambda^{\rightarrow}$ and its extension to dependent types are used in three different contexts in the derivation:

- as the object (defined) language encoded in the logic (the $\lambda^{\rightarrow}$-calculus)
- as the logical metalanguage, encoding proofs (a dependently typed extension of $\lambda^{\rightarrow}$)
- as the defining language of the extracted evaluator (the $\lambda^{\rightarrow}$-calculus)

The simply typed λ-calculus

The standard exposition of the $\lambda^{\rightarrow}$-calculus is given in Figure 3.8. Note that it can be extended to a simply typed version of the λ_{v}-calculus by introducing a type constant `int` and adding the following typing rules:

$$\frac{}{\vdash \ulcorner n \urcorner : \mathtt{int}} \qquad \frac{\Gamma \vdash t : \mathtt{int}}{\Gamma \vdash \mathtt{succ}\, t : \mathtt{int}}$$

(A slightly different extension with integers is considered in Section 7.3.4.)

The reduction semantics for call-by-value evaluation in $\lambda^{\rightarrow}$ is a restriction to the pure sublanguage of the semantics given in Section 2.1 for λ_{v}. Moreover, the simply typed version of λ_{v} is type sound: well-typedness is preserved under reduction and stuck expressions are untypable. Together with the fact that the calculus is strongly normalizing, type soundness ensures that all well-typed terms evaluate to values [17]. The same holds for the restriction to the $\lambda^{\rightarrow}$-calculus.

- Simple types:
  $$\varphi, \psi ::= \alpha \mid \varphi \to \psi,$$
  where α ranges over type variables
- Pseudo-terms:
  $$t ::= x \mid \lambda x^{\varphi}.t \mid t_1\, t_2,$$
  where x ranges over term variables and φ ranges over simple types
- Typing contexts:

  A typing context Γ is a finite set of pairs, of the form
  $$\{x_1 : \varphi_1, \ldots, x_n : \varphi_n\},$$
  where x_i are distinct term variables and φ_i are simple types ($1 \leq i \leq n$). A context $\Gamma \cup \{x : \varphi\}$ (where x does not occur in Γ) is abbreviated $\Gamma, x : \varphi$.
- The typability relation $\vdash$ is defined by the typing rules:

$$\frac{}{\Gamma, x : \varphi \vdash x : \varphi} \qquad \frac{\Gamma, x : \varphi \vdash t : \psi}{\Gamma \vdash \lambda x^{\varphi}.t : \varphi \to \psi} \qquad \frac{\Gamma \vdash t_1 : \varphi \to \psi \quad \Gamma \vdash t_2 : \varphi}{\Gamma \vdash t_1\, t_2 : \psi}$$

- A raw term t is a $\lambda^{\to}$-term if it is typable, i.e., if there exist Γ and φ such that $\Gamma \vdash t : \varphi$ is derivable.

Figure 3.8: The simply typed $\lambda^{\to}$-calculus à la Church

3.2.1 The Curry-Howard isomorphism

By the Curry-Howard isomorphism, the simply typed λ-calculus corresponds to minimal (intuitionistic) logic. We present the natural deduction formulation of minimal logic in Figure 3.9.

An informal interpretation of intuitionistic logic due to Brouwer, Heyting and Kolmogorow states that a proof of an implication is a *procedure* transforming a proof of the antecedent into a proof of the succedent. If we read formulas in the logic as types and write down natural deduction proofs using the λ-notation, then a procedure prescribed by such a proof is given a syntactic representation as a simply typed λ-term. Comparing Figures 3.8 and 3.9, we can then see that simple types are isomorphic to formulas and simply typed terms are isomorphic to proofs in minimal logic. Moreover, proof normalization corresponds to β-reduction in the calculus. These observations are due to Curry and a later clear presentation is due to Howard [56, 113].

This striking correspondence is not of theoretical interest only, but it can be applied in practice. Namely, when extended to higher-order logics and the corresponding more expressive type systems, it can be used for stating and

- Formulas:
$$\varphi,\ \psi ::= \alpha \mid \varphi \to \psi,$$
where α ranges over propositional variables
- Contexts: A context Γ is a finite set of formulas; $\Gamma \cup \{\varphi\}$ is abbreviated Γ, φ.
- The provability relation $\vdash$ is defined by inference rules:
$$\frac{}{\Gamma, \varphi \vdash \varphi} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi},$$
where Γ, φ abbreviates $\Gamma \cup \{\varphi\}$.
- A formula φ is provable under the assumptions Γ if there is a derivation of $\Gamma \vdash \varphi$.

Figure 3.9: Natural deduction for minimal logic

proving properties of programs, and for extracting programs from proofs [48, 123, 136, 181]. Therefore, numerous such extensions have been studied, both to identify the correspondence between existing formal systems (e.g., first-order logic corresponds to a fragment of dependent type theory and second-order logic to System F [99, 167, 187]), and to encourage cross-fertilization between logics and computational calculi by transferring ideas from one to another [20, 74, 185]. Furthermore, the correspondence is not limited to constructive systems: Griffin discovered that classical logic corresponds to the simply typed λ-calculus with a control operator $\mathcal{C}$ [100].

Proofs as programs in first-order logic

Let us now consider an extension of propositional to first-order logic (with multiple sorts). In first-order logic one can build formulas expressing properties of individual objects in a domain of interest. The objects are represented as terms over a fixed signature, and their properties are represented as predicates. A (many-sorted) signature contains a set of sorts, and a family of constant, function and predicate symbols, each of which is of a fixed arity.

Formulas can contain quantifiers, and the inference rules for first-order logic extend the rules of Figure 3.9 as shown in Figure 3.10 (here presented together with their proof terms, associated by the Curry-Howard isomorphism): By the Brouwer-Heyting-Kolmogorov interpretation of first-order logic, a proof of a universal formula $\forall x^{\iota}.\, \varphi(x)$ is a procedure transforming a term of sort ι into a proof of $\varphi\,[t/x]$, and a proof of an existential formula $\exists x^{\iota}.\, \varphi(x)$ consists of a *witness* t and a proof of the formula $\varphi\,[t/x]$. By the Curry-Howard correspondence, proof terms of first-order logic are dependently typed λ-terms: first-order

$$\frac{\Gamma \vdash d : \varphi(x)}{\Gamma \vdash \lambda x^{\iota}.d : \forall x^{\iota}.\,\varphi(x)}\ (x \text{ not free in } \Gamma) \qquad \frac{\Gamma \vdash d : \forall x^{\iota}.\,\varphi(x)}{\Gamma \vdash d\,t : \varphi\,[t/x]}$$

$$\frac{\Gamma \vdash d : \varphi\,[x/t]}{\Gamma \vdash \langle t, d\rangle : \exists x^{\iota}.\,\varphi(x)} \qquad \frac{\Gamma \vdash d : \exists x.\,\varphi \quad \Gamma, u : \varphi \vdash e : \psi}{\Gamma \vdash [d, u.e] : \psi}\ (x \text{ not free in } \Gamma, \psi)$$

Figure 3.10: An extension of minimal to first-order logic

formulas are (a subset of) dependent types, and the term constructors are as shown in the derivation rules in Figure 3.10.

3.2.2 Program extraction

Consider a formula of the form $\forall x^{\iota_1}.\,\exists y^{\iota_2}.\,\varphi(x, y)$. By the Curry-Howard isomorphism, the proof term of this formula is a procedure that for every object t_1 of sort ι_1 produces a pair consisting of an object t_2 of sort ι_2 and a proof term of the formula $\varphi(t_1, t_2)$. If we are only interested in computations at the object level, then we would like to have an algorithm computing for every object of sort ι_1 another object of sort ι_2 with the desired property (the property is represented by the relation φ), and "forget" the rest of the proof term which is irrelevant. A formalization of this informal explanation is offered, for example, by Kreisel's modified realizability interpretation [123, 181]: from a given proof of a formula in many-sorted first-order logic one can extract a simply typed λ-term with sorts as base types.

Moreover, in this interpretation a class of formulas (so-called Harrop formulas) are distinguished that do not contain any computational information, and consequently, proof terms of such formulas are omitted in the extracted program. This property can be used to state some basic relationships between objects as logical axioms which are Harrop formulas. Such axioms are then represented as free variables in proof terms, but they disappear during extraction.

Let us now informally explain how the method of program extraction can be put to use to obtain an evaluator for the simply typed λ-calculus from a proof of weak head normalization in first-order logic (a formal and detailed account is presented in Chapter 7, following Berger [21]):

1. We first represent the ingredients of the reduction semantics in the logic. There is no canonical choice of how to do this; we choose a straightforward representation, which can be summarized, roughly, as follows: The signature consists of a sort for each simple type, a function symbol for each λ-term constructor, and predicate symbols for basic relations defined by the reduction semantics (here, β-contraction and the reflexive-transitive closure of one-step reduction).

2. The reduction strategy is encoded as logical axioms with no computational content (i.e., Harrop formulas).

3. We state the existence of weak head normal forms by a formula of the form: $\forall x^T. \exists y^T. \mathbf{Ev}(x, y)$, where T is a sort of terms of type T and $\mathbf{Ev}(x, y)$ is understood as "x evaluates to y."

4. From a proof of this formula we extract a program, i.e., a simply typed λ-term $\boldsymbol{t}$ of type $T \to T$, and such that $\mathbf{Ev}(s, \boldsymbol{t}\ s)$ for each (encoding of) term s. Hence, the program we obtain is an evaluator that for each simply typed λ-term produces the result of its evaluation according to the strategy encoded in axioms.

In this informal description we glossed over a few technical details which are explained in Chapter 7, in particular: the issue of free variables, the formalization of substitution and the method of proof of the normalization theorem.

Let us now briefly consider the latter issue. The proof we show in Chapter 7 uses structural induction on both simple types and terms. We cannot do that within the logic, so we carry out the inductive argument at the meta level. Hence, we do not obtain one dependently typed term representing the proof of the formula, but rather a family of proof terms, one for each object-level term. In consequence, we do not obtain a single program, but a family of programs computing weak head normal forms.

Normalization by evaluation

The development described above leads from a reduction-based small-step operational semantics to a functional program implementing a big-step reduction-free semantics of the simply typed λ-calculus (i.e., an evaluator). This evaluator is an instance of a more general idea of *normalization by evaluation* (abbreviated NbE), first introduced by Martin-Löf as a method of proving weak normalization in the λ-calculus [135,136]. Later on, Berger and Schwichtenberg invented an NbE algorithm for normalization to long $\beta\eta$-normal forms [26]. Normalization by evaluation is based on the following idea: instead of *reducing* the source term to its normal form, we can define an alternative normalization function as a composition of:

- an interpretation function mapping an object-language term into a suitably defined model in such a way that convertible terms have the same interpretation, and

- a left inverse to the interpretation function, i.e., a so-called 'reify' function mapping an object in the model to a normal form in the object language.

If the normalization function is written in an intuitionistic metalanguage, then we can directly implement it in a programming language, i.e., write it as an expression of this language. Then *evaluating* this expression yields the normal form of the source term.

In our case, the program extracted from the proof of weak head normalization is an instance of NbE: the non-standard model is based on the so-called "glueing" construction along the lines of the one used by Coquand and Dybjer for directly constructing NbE algorithms [49,65].

3.3 Conclusion

In this chapter, we have described two different approaches to connecting operational specifications that are put to use in the remainder of the thesis for a variety of languages.

The first approach, based on the syntactic and functional correspondences, provides systematic methods of connecting reduction semantics with abstract machines, and abstract machines with higher-order evaluators. The connection between reduction semantics and higher-order evaluators can be then made via an intermediate step of constructing an abstract machine: in order to derive a higher-order evaluator, we refocus the reduction semantics into an eval/apply abstract machine, and then we refunctionalize this machine, i.e., we go from a reduction-based to reduction-free evaluation [61].

These two methods are widely applicable, since the requirements on the reduction semantics imposed by refocusing are mild, and the functional correspondence in general does not introduce any restrictions on evaluators or abstract machines. Furthermore, the two methods are extensible: we show extensions of refocusing in Chapters 4 and 5, and the functional correspondence can be applied to virtually any defined language as long as the defining language admits a CPS transformation and defunctionalization.

Furthermore, we observe that while the derivations leading from reduction semantics or evaluators to abstract machines are completely mechanical, going in the opposite direction is usually not as simple. This is due to the fact that abstract machines are geared towards efficiency of evaluation and contain only a minimal amount of information, whereas higher-level specifications tend to be more structured. Nevertheless, the derivation methods we consider allow to precisely pinpoint the "optimizations" present in abstract machines and compare them with their higher-level counterparts, which further supports modifications and extensions to language implementations.

The second approach, based on the Curry-Howard isomorphism, is naturally more restrictive, since it is applicable to typed languages. We have outlined how a general method of program extraction from proofs applied to a proof of normalization for a typed object language yields an evaluator. This development is an example of a systematic derivation of the reduction-free normalization function (implemented in the evaluator) from a reduction-based one (implemented in the axiomatization of the reduction semantics). Instead of writing such an algorithm from scratch and then proving it correct, we focus on formalizing the proof in a logical system, and then we extract the evaluator, whose correctness follows from the soundness of modified realizability and is directly provable within the system. The difficulty in applying the method (for an arbitrary object language of interest) lies in formalizing the reduction strategy within the logic and proving the normalization theorem. Very recently, Berger et al. have formalized a proof of normalization to long $\beta\eta$-normal forms in three proof assistants, pointing out the differences between them and their limitations that result in different formalizations and, in consequence, different extracted programs [23].

Part II

Publications

Chapter 4

A concrete framework for environment machines

with Olivier Danvy [29]

Abstract

We materialize the common belief that calculi with explicit substitutions provide an intermediate step between an abstract specification of substitution in the λ-calculus and its concrete implementations. To this end, we go back to Curien's original calculus of closures (an early calculus with explicit substitutions), we extend it minimally so that it can also express one-step reduction strategies, and we methodically derive a series of environment machines from the specification of two one-step reduction strategies for the λ-calculus: normal order and applicative order. The derivation extends Danvy and Nielsen's refocusing-based construction of abstract machines with two new steps: one for coalescing two successive transitions into one, and the other for unfolding a closure into a term and an environment in the resulting abstract machine. The resulting environment machines include both the idealized and the original versions of Krivine's machine, Felleisen et al.'s CEK machine, and Leroy's Zinc abstract machine.

4.1 Introduction

Krivine's machine and Felleisen et al.'s CEK machine are probably the simplest examples of abstract machines implementing an evaluation function of the λ-calculus [62, 85, 124]. Like many other abstract machines for languages with binding constructs, they are environment-based, i.e., roughly, one component of each machine configuration stores terms that are substituted for free variables during the process of evaluation. The transitions of each machine provide a precise way of handling substitution. This precision contrasts with the traditional presentations of the λ-calculus [42, page 9] [16, page 51] where the β-rule is expressed using a meta-level notion of substitution:

$$(\lambda x.t_0)\, t_1 \to t_0\{t_1/x\}$$

On the right-hand side of this rule, all the free occurrences of x in t_0 are simultaneously replaced by t_1 (which may require auxiliary α-conversions to avoid variable capture). Most implementations, however, do not realize the β-rule using actual substitutions. Instead, they keep an explicit representation of what should have been substituted and leave the term untouched. This environment technique is due to Hasenjaeger in logic [160, § 54] and to Landin in computer science [128]. In logic, it makes it possible to reason by structural induction over λ-terms (since they do not change across β-reduction), and in computer science, it makes it possible to compile λ-terms (since they do not change at run time).

To bridge the two worlds of actual substitutions and explicit representations of what should have been substituted, various calculi of 'explicit substitutions' have been proposed [1, 54, 55, 105, 133, 156, 157]. The idea behind explicit substitutions is to incorporate the notion of substitution into the syntax of the language and then specify suitable rewrite rules that realize it.

In these calculi, the syntax is extended with a 'closure' (the word is due to Landin [128]), i.e., a term together with its environment—hereby referred to as 'substitution' to follow tradition [1, 54]. Moreover, variables are often represented with de Bruijn indices [75] (i.e., lexical offsets in compiler parlance [153]) rather than explicit names; this way, substitutions can be conveniently regarded as lists and the position of a closure in such a list indicates for which variable this closure is to be substituted.

Thus, in a calculus of explicit substitutions, β-reduction is specified using one rule for extending the substitution with a closure to be substituted, such as

$$((\lambda t)[s])\ c \to t[c \cdot s],$$

where c is prepended to the list s, and another rule that replaces a variable with the corresponding closure taken from the substitution, such as

$$i[s] \to c,$$

where c is the ith element of the list s.

Calculi of explicit substitutions come in two flavors: weak calculi, typically used to express weak normalization (evaluation); and strong calculi, that are expressive enough to allow strong normalization. The greater power of strong calculi comes from a richer set of syntactic constructs forming substitutions (for instance, substitutions can be composed, and indices can be lifted).

This work: We present a completely systematic way of deriving an environment machine from a specification of one-step reduction strategy in a weak calculus of closures, by employing Danvy and Nielsen's refocusing technique [72] followed by the fusion of two steps into one and an unfolding of the data type of closures.

We first consider Curien's original calculus of closures $\lambda\rho$ [54]. Curien argues that his calculus mediates between the standard λ-calculus and its implementations via abstract machines. He illustrates his argument by constructing Krivine's machine from a multi-step normal-order reduction strategy.

We observe, however, that one-step reductions cannot be expressed in $\lambda\rho$ and therefore we propose a minimal extension to make it capable of expressing such computations (we dub it the $\lambda\widehat{\rho}$-calculus). We then present a detailed derivation of the usual idealized version of Krivine's machine [51, 54, 103] from the specification of the normal-order one-step strategy in $\lambda\widehat{\rho}$, and we outline the derivation of its applicative-order analog, the CEK machine [85]. We also outline the derivation of the original version of Krivine's machine [126] and of its applicative-order analog, which we identify as Leroy's Zinc abstract machine [132].

Prerequisites and notation: We assume a basic familiarity with the λ-calculus, explicit substitutions, and abstract machines [54]. We also follow standard usage and overload the word "closure" (as in: a term together with a substitution, a reflexive and transitive closure, a compatible closure, and also a closed term) and the word "step" (as in: a derivation step, a decomposition step, and one-step reduction).

Overview: We first present a minimal extension to Curien's original calculus of closures that is expressive enough to account for one-step reduction (Section 4.2). We then methodically derive environment machines from a series of reduction semantics à la Felleisen (Sections 4.3 to 4.6) before drawing a bigger picture (Section 4.7).

4.2 One-step reduction in a calculus of closures

In this section we first briefly review Curien's original calculus of closures $\lambda\rho$ [54], and then present an extension of $\lambda\rho$ that facilitates the specification of one-step reduction strategies. We illustrate the power of the extended calculus with the standard definitions of normal-order and applicative-order strategies.

As a reference point, we first specify the one-step and multi-step reduction relations in the standard λ-calculus.

Reduction in the λ-calculus: Let us recall the formulation of the λ-calculus with de Bruijn indices. Terms are built according to the following grammar:

$$t ::= i \mid t\,t \mid \lambda t,$$

where i ranges over natural numbers (greater than 0).

In the λ-calculus, one-step reduction is defined as the compatible closure of the notion of reduction given by the β-rule [16, Section 3.1]:

$$(\beta) \quad (\lambda t_0)\, t_1 \to t_0\{t_1/1\} \qquad\qquad (\mu) \quad \frac{t_1 \to t_1'}{t_0\, t_1 \to t_0\, t_1'}$$

$$(\nu) \quad \frac{t_0 \to t_0'}{t_0\, t_1 \to t_0'\, t_1} \qquad\qquad (\xi) \quad \frac{t \to t'}{\lambda t \to \lambda t'}$$

where in (β), $t_0\{t_1/1\}$ is a meta-level substitution operation (with suitable reindexing of variables).

Weak (nondeterministic) subsystems are obtained by discarding the ξ-rule. The usual deterministic strategies are obtained as follows:

- The normal-order strategy is obtained by a further restriction, disallowing also the right-compatibility rule (μ). In effect, one obtains the following normal-order one-step reduction strategy for the λ-calculus, producing weak head normal forms:

$$(\beta) \qquad (\lambda t_0)\, t_1 \rightarrow_{\mathrm{n}} t_0\{t_1/1\}$$

$$(\nu) \qquad \frac{t_0 \rightarrow_{\mathrm{n}} t_0'}{t_0\, t_1 \rightarrow_{\mathrm{n}} t_0'\, t_1}$$

  Alternatively, the normal-order strategy can be expressed by the following rule:

$$\frac{t_0 \rightarrow_{\mathrm{n}}^{*} \lambda t_0'}{t_0\, t_1 \rightarrow_{\mathrm{n}} t_0'\{t_1/1\}}$$

  A rule of this form specifies a multi-step reduction strategy (witness the reflexive, transitive closure used in the premise).

- The applicative-order strategy is obtained by another restriction on the β-rule:

$$(\beta_{\mathrm{v}}) \qquad (\lambda t_0)\, t_1 \rightarrow_{\mathrm{v}} t_0\{t_1/1\} \quad \text{if } t_1 \text{ is a value}$$

  Values are variables (de Bruijn indices) and λ-abstractions.

  The following restriction on the right-compatibility rule (μ) makes the reduction strategy deterministically proceed from left to right:

$$(\nu) \qquad \frac{t_0 \rightarrow_{\mathrm{v}} t_0'}{t_0\, t_1 \rightarrow_{\mathrm{v}} t_0'\, t_1}$$

$$(\mu) \qquad \frac{t_1 \rightarrow_{\mathrm{v}} t_1'}{t_0\, t_1 \rightarrow_{\mathrm{v}} t_0\, t_1'} \quad \text{if } t_0 \text{ is a value}$$

  The following restriction on the left-compatibility rule (ν) makes the reduction strategy deterministically proceed from right to left:

$$(\nu) \qquad \frac{t_0 \rightarrow_{\mathrm{v}} t_0'}{t_0\, t_1 \rightarrow_{\mathrm{v}} t_0'\, t_1} \quad \text{if } t_1 \text{ is a value}$$

$$(\mu) \qquad \frac{t_1 \rightarrow_{\mathrm{v}} t_1'}{t_0\, t_1 \rightarrow_{\mathrm{v}} t_0\, t_1'}$$

4.2.1 Curien's calculus of closures

The language of $\lambda\rho$ [54] has three syntactic categories: terms, closures and substitutions:

(Term)	$t ::= i \mid t\, t \mid \lambda t$
(Closure)	$c ::= t[s]$
(Substitution)	$s ::= \bullet \mid c \cdot s$

Terms are defined as in the λ-calculus with de Bruijn indices. A $\lambda\rho$-closure is a pair consisting of a λ-term and a $\lambda\rho$-substitution, which itself is a finite list of $\lambda\rho$-closures to be substituted for free variables in the λ-term. We abbreviate $c_1 \cdot (c_2 \cdot (c_3 \cdot \ldots (c_m \cdot \bullet) \ldots))$ as $c_1 \cdots c_m$.

The weak reduction relation $\xrightarrow{\rho}$ is specified by the following rules:

$$\text{(Eval)} \quad \frac{t_0[s] \xrightarrow{\rho}{}^* (\lambda t'_0)[s']}{(t_0\, t_1)[s] \xrightarrow{\rho} t'_0[(t_1[s]) \cdot s']}$$

$$\text{(Var)} \quad i[c_1 \cdots c_m] \xrightarrow{\rho} c_i \;\text{ if } i \leq m$$

$$\text{(Sub)} \quad \frac{c_1 \xrightarrow{\rho}{}^* c'_1 \quad \ldots \quad c_m \xrightarrow{\rho}{}^* c'_m}{t[c_1 \cdots c_m] \xrightarrow{\rho} t[c'_1 \cdots c'_m]}$$

where $\xrightarrow{\rho}{}^*$ is the reflexive, transitive closure of $\xrightarrow{\rho}$. Reductions are performed on closures, and not on individual terms. Although minimalist (it is not possible to "push" a substitution inside a λ-abstraction), this calculus is powerful enough to compute weak head normal forms. The grammar of weak head normal forms is as follows:

$$c_{\text{nf}} ::= (\lambda t)[s] \;\mid\; (i\, t_1 \ldots t_m)[s]$$

where $i[s]$ is irreducible, which happens if and only if i is greater than the length of s. If we restrict ourselves to considering only closed terms, then the only weak head normal form is a closure whose term is an abstraction.

The rules of the calculus are nondeterministic and can be restricted to define a specific deterministic reduction strategy. For instance, the normal-order strategy (denoted $\xrightarrow{\rho}_{\text{n}}$) is obtained by restricting the rules to (Eval) and (Var). This restriction specifies a multi-step reduction strategy because of the transitive closure used in the (Eval) rule.

4.2.2 A minimal extension to Curien's calculus of closures

The $\lambda\rho$-calculus is not expressive enough to specify one-step reduction because the specification of one-step reduction requires a way to "compose" intermediate results of computation—here, closures—to form a new closure that can be reduced further. In $\lambda\rho$, there is no such possibility. A simple solution is to extend the syntax of closures with a construct denoting closure application. We denote it simply by juxtaposition:

$$\begin{array}{ll} \text{(Term)} & t ::= i \mid t\, t \mid \lambda t \\ \text{(Closure)} & c ::= t[s] \mid c\, c \\ \text{(Substitution)} & s ::= \bullet \mid c \cdot s \end{array}$$

With the extended syntax we are now in position to define the one-step reduction relation as the compatible closure of the notion of (a closure-based variant of) β-reduction:

$$
\begin{array}{llll}
(\beta) & ((\lambda t)[s])\, c \xrightarrow{\widehat{\rho}} t[c \cdot s] & \text{(Var)} & i[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}} c_i \text{ if } i \leq m \\[1ex]
(\nu) & \dfrac{c_0 \xrightarrow{\widehat{\rho}} c_0'}{c_0\, c_1 \xrightarrow{\widehat{\rho}} c_0'\, c_1} & \text{(App)} & (t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\, (t_1[s]) \\[3ex]
(\mu) & \dfrac{c_1 \xrightarrow{\widehat{\rho}} c_1'}{c_0\, c_1 \xrightarrow{\widehat{\rho}} c_0\, c_1'} & \text{(Sub)} & \dfrac{c_i \xrightarrow{\widehat{\rho}} c_i'}{t[c_1 \cdots c_i \cdots c_m] \xrightarrow{\widehat{\rho}} t[c_1 \cdots c_i' \cdots c_m]} \text{ for } i \leq m
\end{array}
$$

The $\lambda\widehat{\rho}$-calculus is nondeterministic and confluent. The following proposition formalizes the simulation of $\lambda\rho$ reductions in $\lambda\widehat{\rho}$ and vice versa.

Proposition 4.1 (Simulation). *Let c_0 and c_1 be $\lambda\rho$-closures. Then the following properties hold:*

1. *If $c_0 \xrightarrow{\rho} c_1$, then $c_0 \xrightarrow{\widehat{\rho}}{}^{+} c_1$.*
2. *If $c_0 \xrightarrow{\widehat{\rho}}{}^{*} c_1$, then $c_0 \xrightarrow{\rho}{}^{*} c_1$.*

Proof. The proofs are done by induction.

Proof of *1.* We define the complexity of derivations in $\lambda\rho$ as follows: the complexity of a derivation starting with (Eval) or (Sub) is defined as the maximum of the complexities of its immediate subderivations augmented by 1, and the complexity of an instance of (Var) is 1. The complexity of a multiple-step derivation is the sum of the complexities of all its steps.

The proof proceeds by induction on the complexity n of $c_0 \xrightarrow{\rho} c_1$. If $n = 1$, then it is an instance of the rule (Var), and hence it can be immediately simulated in $\lambda\widehat{\rho}$. For the inductive step, assume that $c_0 \xrightarrow{\rho} c_1$ has complexity $n+1$. Then the first rule applied must be either (Eval) or (App), determining the structure of c_0 and c_1. We consider the two cases in turn:

Case $(t_0\, t_1)[s] \xrightarrow{\rho} t_0'[(t_1[s]) \cdot s']$. We then know that $t_0[s] \xrightarrow{\rho}{}^{*} (\lambda t_0')[s']$ has complexity n and all its steps are of the complexity less or equal to n. Then by induction hypothesis, $t_0[s] \xrightarrow{\widehat{\rho}}{}^{+} (\lambda t_0')[s']$. Hence we obtain the following reduction sequence in $\lambda\widehat{\rho}$:

$$(t_0\, t_1)[s] \xrightarrow{\widehat{\rho}} (t_0[s])\, (t_1[s]) \xrightarrow{\widehat{\rho}}{}^{+} ((\lambda t_0')[s'])\, (t_1[s]) \xrightarrow{\widehat{\rho}} t_0'[(t_1[s]) \cdot s'].$$

Case $t[c_1 \cdots c_m] \xrightarrow{\rho} t[c_1' \cdots c_m']$. By definition, each of the subderivations $c_i \xrightarrow{\rho}{}^{*} c_i'$ for $1 \leq i \leq m$ is of complexity not greater than n. Furthermore, we prove that $c_i \xrightarrow{\widehat{\rho}}{}^{+} c_i'$ for all i, by induction on the length of the reduction sequence. Hence

$$t[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}{}^{+} t[c_1' \cdots c_m] \xrightarrow{\widehat{\rho}}{}^{+} \cdots \xrightarrow{\widehat{\rho}}{}^{+} t[c_1' \cdots c_m'].$$

Note that, in particular, the rule (Var) in $\lambda\rho$ is simulated by (Var) in $\lambda\widehat{\rho}$, and the rule (Eval) in $\lambda\rho$ is simulated by (β) and (App) in $\lambda\widehat{\rho}$.

Proof of *2.* By induction on the length of the reduction sequence $c_0 \xrightarrow{\widehat{\rho}}{}^{*} c_1$. For the inductive case, assume $c_0 \xrightarrow{\widehat{\rho}}{}^{n+1} c_1$, i.e., $c_0 \xrightarrow{\widehat{\rho}} c'_0 \xrightarrow{\widehat{\rho}}{}^{n} c_1$. We distinguish two cases:

- $c'_0 = t[s]$: In this case, the first reduction step in $\lambda\widehat{\rho}$ is either according to the rule (Var), or to the rule (Sub); thus it can be simulated directly by the corresponding rule in $\lambda\rho$.
- $c'_0 = c\,c$: The only applicable reduction is (App), hence $c_0 = (t_0\,t_1)[s]$ and $c'_0 = (t_0[s])\,(t_1[s])$. Analyzing the reduction rules, we observe that in the subsequent reduction sequence the rule (β) must be applied, since c_1 is again a $\lambda\rho$-closure. Without loss of generality we can assume that

  $$c'_0 \xrightarrow{\widehat{\rho}}{}^{k_1} ((\lambda t'_0)[s'])(t_1[s]) \xrightarrow{\widehat{\rho}}{}^{k_2} ((\lambda t'_0)[s'])(t'_1[s'']) \xrightarrow{\widehat{\rho}} t'_0[(t'_1[s''])\cdot s'] \xrightarrow{\widehat{\rho}}{}^{k_3} c_1$$

  for $k_1+k_2+1+k_3 = n$. By induction hypothesis $t_0[s] \xrightarrow{\rho}{}^{*} (\lambda t'_0)[s']$ and $t_1[s] \xrightarrow{\rho}{}^{*} t'_1[s'']$, and hence $c_0 = (t_0 t_1)[s] \xrightarrow{\rho} t'_0[(t_1[s])\cdot s']$ by (Eval), and $t'_0[(t_1[s])\cdot s'] \xrightarrow{\rho}{}^{*} t'_0[(t'_1[s''])\cdot s']$ by (Sub). Finally, $t'_0[(t'_1[s''])\cdot s'] \xrightarrow{\rho}{}^{*} c_1$ by induction hypothesis.

$\square$

4.2.3 Specification of the normal-order reduction strategy

The normal-order strategy is obtained by restricting $\lambda\widehat{\rho}$ to the following rules:

$$(\beta)\quad ((\lambda t)[s])\,c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} t[c\cdot s] \qquad \text{(Var)}\quad i[c_1\cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c_i \quad \text{if } i \le m$$

$$(\nu)\quad \frac{c_0 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'_0}{c_0\,c_1 \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'_0\,c_1} \qquad \text{(App)}\quad (t_0\,t_1)[s] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[s])\,(t_1[s])$$

We consider only closed terms, and hence we omit the side condition on i in Sections 4.3 and 4.5.1.

Let $\xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*}$ (the call-by-name evaluation relation) denote the reflexive, transitive closure of $\xrightarrow{\widehat{\rho}}_{\mathrm{n}}$. The grammar of values and substitutions for call-by-name evaluation reads as follows:

(Value)	$v ::= (\lambda t)[s]$
(Substitution)	$s ::= \bullet \mid c\cdot s$

4.2.4 Specification of the applicative-order reduction strategy

Similarly, the left-to-right applicative-order strategy is obtained by restricting $\lambda\widehat{\rho}$ to the following rules:

$$(\beta)\ ((\lambda t)[s])\, c \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} t[c \cdot s] \quad \text{if } c \text{ is a value} \qquad (\mathrm{Var})\quad i[c_1 \cdots c_m] \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} c_i \quad \text{if } i \leq m$$

$$(\nu)\quad \frac{c_0 \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} c_0'}{c_0\, c_1 \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} c_0'\, c_1} \qquad (\mathrm{App})\quad (t_0\, t_1)[s] \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} (t_0[s])\,(t_1[s])$$

$$(\mu)\quad \frac{c_1 \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} c_1'}{c_0\, c_1 \overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}} c_0\, c_1'} \quad \text{if } c_0 \text{ is a value}$$

We consider only closed terms, and hence we omit the side condition on i in Sections 4.4 and 4.5.2.

Let $\overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}}{}^*$ (the call-by-value evaluation relation) denote the reflexive, transitive closure of $\overset{\widehat{\rho}}{\rightarrow}_{\mathrm{v}}$. The grammar of values and substitutions for call-by-value evaluation reads as follows:

(Value)	$v ::= (\lambda t)[s]$
(Substitution)	$s ::= \bullet \mid v \cdot s$

Under call by value, both sub-components of any application $c_0\, c_1$ (i.e., both c_0 and c_1) are evaluated. We consider left-to-right evaluation (i.e., c_0 is evaluated, and then c_1) in Section 4.4 and right-to-left evaluation in Section 4.5.2.

4.3 From normal-order reduction to call-by-name environment machine

We present a detailed and systematic derivation of an abstract machine for call-by-name evaluation in the λ-calculus, starting from the specification of the normal-order reduction strategy in the $\lambda\widehat{\rho}$-calculus. We first follow the steps outlined by Danvy and Nielsen in their work on refocusing [72]:

Section 4.3.1: We specify the normal-order reduction strategy in the form of a reduction semantics, i.e., with a one-step reduction function specified as decomposing a non-value term into a reduction context and a redex, contracting this redex, and plugging the contractum into the context. As is traditional, we also specify evaluation as the transitive closure of one-step reduction.

Section 4.3.2: We replace the combination of plugging and decomposition by a refocus function that iteratively goes from redex site to redex site in the reduction sequence. The resulting 'refocused' evaluation function is the transitive closure of the refocus function and takes the form of a 'pre-abstract machine.'

Section 4.3.3: We merge the definitions of the transitive closure and the refocus function into a 'staged abstract machine' that implements the reduction rules and the compatibility rules of the $\lambda\widehat{\rho}$-calculus with two separate functions.

Section 4.3.4: We inline the function implementing the reduction rules. The result is an eval/apply abstract machine consisting of an 'eval' transition function dispatching on closures and an 'apply' transition function dispatching on contexts.

Section 4.3.5: We inline the apply transition function. The result is a push/enter abstract machine.

We then simplify and transform the push/enter abstract machine:

Section 4.3.6: Observing that in a reduction sequence, an (App) reduction step is always followed by a decomposition step, we shortcut these two steps into one decomposition step. This simplification enables the following step.

Section 4.3.7: We unfold the data type of closures, making the simplified machine operate over two components—a term and a substitution—instead of over one—a closure. The substitution component is the traditional environment of environment machines, and the resulting machine is an environment machine. This machine coincides with the usual idealized version of Krivine's machine [51, 54, 103]. (The original version of Krivine's machine [124, 126] is a bit more complicated, and we treat it in Section 4.5.)

In Section 4.3.8, we state the correctness of the idealized version of Krivine's machine with respect to evaluation in the $\lambda\widehat{\rho}$-calculus, and in Section 4.3.9 we get back to the λ-calculus.

4.3.1 A reduction semantics for normal order

A reduction semantics [82, 84] consists of the grammar of a source language, a syntactic notion of value, a collection of reduction rules, and a reduction strategy. This reduction strategy is embodied in a grammar of reduction contexts (terms with one hole), a plug function mapping a term and a context into a new term, and a strategy for decomposing a non-value term into a redex and its reduction context. When the redexes do not overlap, the reduction rules are implemented by a contraction function that maps a redex into the corresponding contractum. When the source language satisfies a unique-decomposition property with respect to the reduction strategy, decomposing a non-value term into a redex and its reduction context is implemented by a decomposition function.

A reduction semantics for normal-order reduction in the $\lambda\widehat{\rho}$-calculus can be obtained from the specification of normal-order reduction strategy in Section 4.2.3 as follows: the syntactic notion of value and the collection of reduction rules are specified directly, the grammar of redexes is as follows:

$$(\mathsf{Redex}) \qquad r ::= ((\lambda t)[s])\,c \mid i[s] \mid (t_0\,t_1)[s],$$

and the compatibility rule (ν) induces the grammar of reduction contexts:[1]

$$(\mathsf{Context}) \qquad C ::= [\,] \mid \mathsf{ARG}(c,\, C)$$

The redexes do not overlap and the source language satisfies a unique-decomposition property. Therefore we can define the following three functions:

$$\begin{aligned} \mathsf{decompose} &: \mathsf{Closure} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\ \mathsf{contract} &: \mathsf{Redex} \rightarrow \mathsf{Closure} \\ \mathsf{plug} &: \mathsf{Closure} \times \mathsf{Context} \rightarrow \mathsf{Closure} \end{aligned}$$

Given these three functions, we can define the following one-step reduction function that tests whether a closure is a value or can be decomposed into a redex and a context, that contracts this redex, and that plugs the contractum in the context:

$$\begin{aligned} \mathsf{reduce} &: \mathsf{Closure} \rightarrow \mathsf{Closure} \\ \mathsf{reduce}\,c &= \mathit{case}\ \mathsf{decompose}\,c \\ &\qquad \mathit{of}\ v \Rightarrow v \\ &\qquad\ \mid (r,\, C) \Rightarrow \mathsf{plug}\,(c, C) \quad \text{where } c = \mathsf{contract}\,r \end{aligned}$$

The following proposition is a consequence of the unique-decomposition property.

Proposition 4.2. *For any non-value closure c and for any closure c', $c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'$ $\Leftrightarrow$ $\mathsf{reduce}\,c = c'$.*

Finally, we can define evaluation as the transitive closure of one-step reduction:

$$\begin{aligned} \mathsf{iterate} &: \mathsf{Closure} \rightarrow \mathsf{Value} \\ \mathsf{iterate}\,c &= c & \text{if } c \text{ is a value} \\ \mathsf{iterate}\,c &= \mathsf{iterate}\,(\mathsf{reduce}\,c) & \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathsf{evaluate} &: \mathsf{Term} \rightarrow \mathsf{Value} \\ \mathsf{evaluate}\,t &= \mathsf{iterate}\,(t[\bullet]) \end{aligned}$$

This evaluation function is partial because a reduction sequence might not terminate.

Proposition 4.3. *For any closed term t and any value v, $t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*} v \Leftrightarrow$ $\mathsf{evaluate}\,t = v$.*

[1]In the standard inside-out notation, this grammar can also be stated as follows:

$$C ::= [\,] \mid C[[\,]c]$$

For simplicity, we inline the definition of reduce and we use decompose to test whether a value has been reached:

$$\begin{aligned}\mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \to \mathsf{Value}\\ \mathsf{iterate}\, v &= v\\ \mathsf{iterate}\,(r,\ C) &= \mathsf{iterate}\,(\mathsf{decompose}\,(\mathsf{plug}\,(c, C))) \quad \text{where } c = \mathsf{contract}\, r\end{aligned}$$

$$\begin{aligned}\mathsf{evaluate} &: \mathsf{Term} \to \mathsf{Value}\\ \mathsf{evaluate}\, t &= \mathsf{iterate}\,(\mathsf{decompose}\,(t[\bullet]))\end{aligned}$$

4.3.2 A pre-abstract machine

The reduction sequence implemented by evaluation can be depicted as follows:

$$\begin{tikzcd}
\circ \arrow[dr, "\mathsf{decompose}"] & & & \circ \arrow[dr, "\mathsf{decompose}"] & & & \circ \arrow[dr, "\mathsf{decompose}"] & & \\
& \circ \arrow[r, "\mathsf{contract}"'] & \circ \arrow[ur, "\mathsf{plug}"] & & \circ \arrow[r, "\mathsf{contract}"'] & \circ \arrow[ur, "\mathsf{plug}"] & & \circ \arrow[r, "\mathsf{contract}"'] & \circ
\end{tikzcd}$$

In earlier work [72], Danvy and Nielsen stated formal conditions under which the composition of plug and decompose could be replaced by a more efficient function, refocus, that would directly go from redex site to redex site in the reduction sequence, and they presented an algorithm for constructing such a refocus function:

$$\begin{tikzcd}
& \circ \arrow[dr, "\mathsf{decompose}"] & & & \circ \arrow[dr, "\mathsf{decompose}"] & & & \circ \arrow[dr, "\mathsf{decompose}"] & & \\
{} \arrow[rr, dashed] & & \circ \arrow[r, "\mathsf{contract}"'] & \circ \arrow[ur, "\mathsf{plug}"] \arrow[rr, dashed, "\mathsf{refocus}"'] & & \circ \arrow[r, "\mathsf{contract}"'] & \circ \arrow[ur, "\mathsf{plug}"] \arrow[rr, dashed, "\mathsf{refocus}"'] & & \circ \arrow[r, "\mathsf{contract}"'] & \circ
\end{tikzcd}$$

The formal conditions are satisfied here: the unique-decomposition property holds and the grammar of reduction contexts is context free. The algorithm yields the following definition, which takes the form of two state-transition functions, i.e., of an abstract machine:

$$\begin{aligned}\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\ \mathsf{refocus}\,(i[s],\ C) &= (i[s],\ C)\\ \mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s])\\ \mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= ((t_0\, t_1)[s],\ C)\\ \mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))\end{aligned}$$

$$\begin{aligned}\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\ \mathsf{refocus_{aux}}\,([\,],\ v) &= v\\ \mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ v) &= (v\, c,\ C)\end{aligned}$$

In this abstract machine, the configurations are pairs of a closure and a context; the final transitions are specified by refocus$_{\mathsf{aux}}$ and by the first and third clauses of refocus; and the initial transition is specified by two clauses of the corresponding 'refocused' evaluation function, which reads as follows:

$$\begin{aligned}
\mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \to \mathsf{Value} \\
\mathsf{iterate}\, v &= v \\
\mathsf{iterate}\, (r,\ C) &= \mathsf{iterate}\, (\mathsf{refocus}\, (c,\ C)) \quad \text{where } c = \mathsf{contract}\, r
\end{aligned}$$

$$\begin{aligned}
\mathsf{evaluate} &: \mathsf{Term} \to \mathsf{Value} \\
\mathsf{evaluate}\, t &= \mathsf{iterate}\, (\mathsf{refocus}\, (t[\bullet],\ [\,]))
\end{aligned}$$

(For the initial call to iterate, we have exploited the double equality $\mathsf{decompose}\, (t[\bullet]) = \mathsf{decompose}\, (\mathsf{plug}\, (t[\bullet], [\,])) = \mathsf{refocus}\, (t[\bullet],\ [\,])$.)

Through the auxiliary function iterate, this evaluation function computes the transitive closure of refocus. Following Danvy and Nielsen, we refer to it as a 'pre-abstract machine.'

4.3.3 A staged abstract machine

To transform the pre-abstract machine into an abstract machine, we distribute the calls to iterate from the definitions of evaluate and of iterate to the definitions of refocus and $\mathsf{refocus}_{\mathsf{aux}}$:

$$\begin{aligned}
\mathsf{evaluate} &: \mathsf{Term} \to \mathsf{Value} \\
\mathsf{evaluate}\, t &= \mathsf{refocus}\, (t[\bullet],\ [\,])
\end{aligned}$$

$$\begin{aligned}
\mathsf{iterate} &: \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \to \mathsf{Value} \\
\mathsf{iterate}\, v &= v \\
\mathsf{iterate}\, (r,\ C) &= \mathsf{refocus}\, (c,\ C) \quad \text{where } c = \mathsf{contract}\, r
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}\, (i[s],\ C) &= \mathsf{iterate}\, (i[s],\ C) \\
\mathsf{refocus}\, ((\lambda t)[s],\ C) &= \mathsf{refocus}_{\mathsf{aux}}\, (C,\ (\lambda t)[s]) \\
\mathsf{refocus}\, ((t_0\, t_1)[s],\ C) &= \mathsf{iterate}\, ((t_0\, t_1)[s],\ C) \\
\mathsf{refocus}\, (c_0\, c_1,\ C) &= \mathsf{refocus}\, (c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus}_{\mathsf{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context}) \\
\mathsf{refocus}_{\mathsf{aux}}\, ([\,],\ v) &= \mathsf{iterate}\, v \\
\mathsf{refocus}_{\mathsf{aux}}\, (\mathsf{ARG}(c,\ C),\ v) &= \mathsf{iterate}\, (v\, c,\ C)
\end{aligned}$$

The resulting definitions of evaluate, iterate, refocus, and $\mathsf{refocus}_{\mathsf{aux}}$ are that of four mutually recursive transition functions that form an abstract machine. In this abstract machine, the configurations are pairs of a closure and a context, the initial transition is specified by evaluate, and the final transition in the first clause of iterate. The compatibility rules are implemented by refocus and $\mathsf{refocus}_{\mathsf{aux}}$, and the reduction rules by the call to contract in the second clause of iterate. We can make this last point even more manifest by inlining contract in the definition of iterate:

$$\begin{aligned}
\mathsf{iterate}\, v &= v \\
\mathsf{iterate}\, (i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\, (c_i,\ C) \\
\mathsf{iterate}\, ((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\, ((t_0[s])\, (t_1[s]),\ C) \\
\mathsf{iterate}\, (((\lambda t)[s])\, c,\ C) &= \mathsf{refocus}\, (t[c \cdot s],\ C)
\end{aligned}$$

By construction, the machine therefore separately implements the reduction rules and the compatibility rules; for this reason, we refer to it as a 'staged abstract machine.'

4.3.4 An eval/apply abstract machine

We now inline the calls to iterate in the staged abstract machine. The resulting machine reads as follows:

$$\begin{aligned}
\mathsf{evaluate}\, t &= \mathsf{refocus}\,(t[\bullet],\ [\,])\\[1ex]
\mathsf{refocus}\,(i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\,(c_i,\ C)\\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s])\\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\,(t_1[s]),\ C)\\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))\\[1ex]
\mathsf{refocus_{aux}}\,([\,],\ (\lambda t)[s]) &= (\lambda t)[s]\\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ (\lambda t)[s]) &= \mathsf{refocus}\,(t[c \cdot s],\ C)
\end{aligned}$$

As already observed by Danvy and Nielsen in their work on refocusing, inlining iterate yields an eval/apply abstract machine [134]: refocus (the 'eval' transition function) dispatches on closures and $\mathsf{refocus_{aux}}$ (the 'apply' function) dispatches on contexts.

4.3.5 A push/enter abstract machine

We now inline the calls to $\mathsf{refocus_{aux}}$ in the eval/apply abstract machine. The resulting machine reads as follows:

$$\begin{aligned}
\mathsf{evaluate}\, t &= \mathsf{refocus}\,(t[\bullet],\ [\,])\\[1ex]
\mathsf{refocus}\,(i[c_1 \cdots c_m],\ C) &= \mathsf{refocus}\,(c_i,\ C)\\
\mathsf{refocus}\,((\lambda t)[s],\ [\,]) &= (\lambda t)[s]\\
\mathsf{refocus}\,((\lambda t)[s],\ \mathsf{ARG}(c,\ C)) &= \mathsf{refocus}\,(t[c \cdot s],\ C)\\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\,(t_1[s]),\ C)\\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

As already observed by Ager et al. [3], inlining the apply transition function in a call-by-name eval/apply abstract machine yields a push/enter machine [134].

4.3.6 An optimized push/enter machine

The abstract machine of Section 4.3.5 only produces an application of closures through an (App) reduction step (second-to-last clause of refocus). We observe that in a reduction sequence, an (App) reduction step is always followed by a decomposition step (last clause of refocus). As an optimization, we shortcut these two consecutive steps into one decomposition step, replacing the last two clauses of refocus with the following one:

$$\mathsf{refocus}\,((t_0\, t_1)[s],\ C) = \mathsf{refocus}\,(t_0[s],\ \mathsf{ARG}(t_1[s],\ C))$$

The optimized machine never produces any application of closures and therefore works for the $\lambda\rho$-calculus as well as for the $\lambda\widehat{\rho}$-calculus with the grammar of closures restricted to that of $\lambda\rho$.

4.3.7 An environment machine

Finally, we unfold the data type of closures. If we read each syntactic category in $\lambda\rho$ as a type, then the type of closures is recursive:

$$\mathsf{Closure} \overset{\text{def}}{=} \mu X.\mathsf{Term} \times \mathsf{List}(X)$$

and furthermore

$$\mathsf{Substitution} \overset{\text{def}}{=} \mathsf{List}(\mathsf{Closure}).$$

Hence one unfolding of the type Closure yields Term $\times$ Substitution. Therefore, for any closure $t[s]$ of type Closure, its unfolding gives a pair $(t,\, s)$ of type Term $\times$ Substitution. We replace each closure in the definition of evaluate and refocus, in Section 4.3.6, by its unfolding. Flattening (Term $\times$ Substitution) $\times$ Context into Term $\times$ Substitution $\times$ Context yields the following abstract machine:

$$\begin{aligned} v &::= (\lambda t,\, s) \\ C &::= [\,] \mid \mathsf{ARG}((t,\, s),\, C) \end{aligned}$$

$$\begin{aligned} \mathsf{evaluate} &: \mathsf{Term} \to \mathsf{Value} \\ \mathsf{evaluate}\, t &= \mathsf{refocus}\,(t,\, \bullet,\, [\,]) \end{aligned}$$

$$\begin{aligned} \mathsf{refocus} &: \mathsf{Term} \times \mathsf{Substitution} \times \mathsf{Context} \to \mathsf{Value} \\ \mathsf{refocus}\,(i,\, (t_1,\, s_1)\cdots(t_m,\, s_m),\, C) &= \mathsf{refocus}\,(t_i,\, s_i,\, C) \\ \mathsf{refocus}\,(\lambda t,\, s,\, [\,]) &= (\lambda t,\, s) \\ \mathsf{refocus}\,(\lambda t,\, s,\, \mathsf{ARG}((t',\, s'),\, C)) &= \mathsf{refocus}\,(t,\, (t',\, s')\cdot s,\, C) \\ \mathsf{refocus}\,(t_0\, t_1,\, s,\, C) &= \mathsf{refocus}\,(t_0,\, s,\, \mathsf{ARG}((t_1,\, s),\, C)) \end{aligned}$$

We observe that this machine coincides with the usual idealized version of Krivine's machine [51, 54, 103], in which evaluation contexts are treated as last-in, first-out lists (i.e., stacks) of closures. In particular, the substitution component assumes the rôle of the environment.

4.3.8 Correctness

We state the correctness of the final result—the idealized version of Krivine's machine —with respect to evaluation in the $\lambda\widehat{\rho}$-calculus.

Theorem 4.1. *For any closed term t in $\lambda\widehat{\rho}$,*

$$t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*} (\lambda t')[s] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\, s).$$

Proof. The proof relies on the correctness of refocusing [72], and the (trivial) meaning preservation of each of the subsequent transformations. □

The theorem states that Krivine's machine is correct in the sense that it computes closed weak head normal forms, and it realizes the normal-order strategy in the $\lambda\widehat{\rho}$-calculus, which makes it a call-by-name machine [148]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-name evaluation in the $\lambda\widehat{\rho}$-calculus. Since the reductions according to the normal-order strategy in $\lambda\rho$ can be simulated in $\lambda\widehat{\rho}$ (see Proposition 4.1), as a byproduct we obtain the correctness of Krivine's machine also with respect to Curien's original calculus of closures:

Corollary 4.1. *For any term t in $\lambda\rho$,*

$$t[\bullet] \xrightarrow{\rho}_{\mathrm{n}}{}^{*} (\lambda t')[s] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\, s).$$

However, Krivine's machine is better known as an environment machine that computes weak head normal forms of λ-terms. We show the correspondence theorem next.

4.3.9 Correspondence with the λ-calculus

The call-by-name evaluation relation in the λ-calculus is defined as the reflexive, transitive closure of the normal-order one-step reduction strategy shown in Section 4.2.

In order to relate values in the λ-calculus with values in the language of closures, we define a function σ that forces all the delayed substitutions in a $\lambda\widehat{\rho}$-closure. The function takes a closure and a number k indicating the current depth of the processed term (with respect to the number of surrounding λ-abstractions), and returns a λ-term:

$$\begin{aligned}
\sigma(i[s], k) &= \begin{cases} i & \text{if } i \leq k \\ \sigma(c_{i-k}, k) & \text{if } k < i \leq m + k \text{ and } s = c_1 \cdots c_m \\ i - m & \text{if } i > m + k \text{ and } s = c_1 \cdots c_m \end{cases} \\
\sigma((t_0\, t_1)[s], k) &= (\sigma(t_0[s], k))\, (\sigma(t_1[s], k)) \\
\sigma((\lambda t)[s], k) &= \lambda(\sigma(t[s], k + 1)) \\
\sigma(c_0\, c_1, k) &= \sigma(c_0, k)\, \sigma(c_1, k)
\end{aligned}$$

The function σ puts us in position to state the correspondence theorem.

Theorem 4.2 (Correspondence). *For any λ-term t, $t \to_{\mathrm{n}}^{*} \lambda t'$ if and only if*

$$\mathsf{evaluate}\, t = (\lambda t'',\, s) \quad \textit{and} \quad \sigma((\lambda t'')[s], 0) = \lambda t'.$$

Proof. The left-to-right implication relies on the following property, proved by structural induction on t:

$$\text{If } t \to_{\mathrm{n}} t',\ \text{then } t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{n}}{}^{*} c \text{ in } \lambda\widehat{\rho}, \text{ and } \sigma(c, 0) = t'.$$

In order to prove the converse implication, we observe that if $c \xrightarrow{\widehat{\rho}}_{\mathrm{n}} c'$, then either $\sigma(c, 0) \to_{\mathrm{n}} \sigma(c', 0)$, if the smallest reduction step uses the rule (β), or $\sigma(c, 0) = \sigma(c', 0)$ otherwise. The proof is done by structural induction on c, using the fact that $\sigma(t[s], j + 1)\{\sigma(c, 0)/j + 1\} = \sigma(t[c \cdot s], j)$. □

Curien, Hardin and Lévy consider several weak calculi of explicit substitutions capable of simulating call-by-name evaluation [55]. They relate these calculi to the λ-calculus with de Bruijn indices in much the same way as we do above. In fact, our function σ performs exactly σ-normalization in their strong calculus $\lambda\sigma$ for the restricted grammar of closures and substitutions of the $\lambda\rho$-calculus, and the structure of the proof of Theorem 4.2 is similar to that of their Theorem 3.6 [55]. More recently, Wand has used a translation U from closures to λ-terms with names that is an analog of σ, and presented a similar simple proof of the correctness of Krivine's machine for the λ-calculus with names [189].

4.4 From applicative-order reduction to call-by-value environment machine

Starting with the applicative-order reduction strategy specified in Section 4.2.4, we follow the same procedure as in Section 4.3.

4.4.1 The reduction semantics for left-to-right applicative order

We first specify a reduction semantics for applicative-order reduction. The grammar of the source language is specified in Section 4.2.2, the syntactic notion of value and the collection of reduction rules are specified in Section 4.2.4, and the compatibility rules (ν) and (μ) induce the following grammar of reduction contexts:[2]

$$\text{(Context)} \qquad C ::= [\,] \mid \mathsf{ARG}(c,\, C) \mid \mathsf{FUN}(v,\, C)$$

The redexes do not overlap and the source language satisfies a unique-decomposition property with respect to the applicative-order reduction strategy. Therefore, as in Section 4.3.1 we can define a contraction function, a decomposition function, a plug function, a one-step reduction function, and an evaluation function.

4.4.2 From evaluation function to environment machine

We now take the same steps as in Section 4.3. The reduction semantics of Section 4.4.1 satisfies the refocusing conditions, and Danvy and Nielsen's algorithm yields the following definition:

[2]In the standard inside-out notation, this grammar can also be stated as follows:

$$C ::= [\,] \mid C[[\,]c] \mid C[v[\,]]$$

$$\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\
\mathsf{refocus}\,(i[s],\ C) &= (i[s],\ C)\\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s])\\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= ((t_0\, t_1)[s],\ C)\\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \rightarrow \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\
\mathsf{refocus_{aux}}\,([\,],\ v) &= v\\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ v) &= \mathsf{refocus}\,(c,\ \mathsf{FUN}(v,\ C))\\
\mathsf{refocus_{aux}}\,(\mathsf{FUN}(v',\ C),\ v) &= (v'\, v,\ C)
\end{aligned}$$

We successively transform the resulting pre-abstract machine into a staged abstract machine and an eval/apply abstract machine.

The eval/apply abstract machine reads as follows:

$$\mathsf{evaluate}\, t = \mathsf{refocus}\,(t[\bullet],\ [\,])$$

$$\begin{aligned}
\mathsf{refocus}\,(i[v_1 \cdots v_m],\ C) &= \mathsf{refocus_{aux}}\,(C,\ v_i)\\
\mathsf{refocus}\,((\lambda t)[s],\ C) &= \mathsf{refocus_{aux}}\,(C,\ (\lambda t)[s])\\
\mathsf{refocus}\,((t_0\, t_1)[s],\ C) &= \mathsf{refocus}\,((t_0[s])\,(t_1[s]),\ C)\\
\mathsf{refocus}\,(c_0\, c_1,\ C) &= \mathsf{refocus}\,(c_0,\ \mathsf{ARG}(c_1,\ C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus_{aux}}\,([\,],\ v) &= v\\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(c,\ C),\ (\lambda t)[s]) &= \mathsf{refocus}\,(c,\ \mathsf{FUN}((\lambda t)[s],\ C))\\
\mathsf{refocus_{aux}}\,(\mathsf{FUN}((\lambda t)[s],\ C),\ v) &= \mathsf{refocus}\,(t[v \cdot s],\ C)
\end{aligned}$$

As in Section 4.3.6, we observe that we can shortcut the third and the fourth clauses of refocus (they are the only producer and consumer of an application of closures, respectively). We can then unfold the data type of closures, as in Section 4.3.7, and obtain an eval/apply environment machine. This environment machine coincides with Felleisen et al.'s CEK machine [84].

4.4.3 Correctness and correspondence with the λ-calculus

As in Section 4.3.8, we state the correctness of the eval/apply machine with respect to the $\lambda\widehat{\rho}$-calculus.

Theorem 4.3. *For any term t in $\lambda\widehat{\rho}$,*

$$t[\bullet] \xrightarrow{\widehat{\rho}}_{\mathrm{v}}{}^{*} (\lambda t')[s'] \quad \textit{if and only if} \quad \mathsf{evaluate}\, t = (\lambda t',\ s').$$

The theorem states that the eval/apply machine is correct in the sense that it computes closed weak head normal forms, and it realizes the applicative-order strategy in the $\lambda\widehat{\rho}$-calculus, which makes it a call-by-value machine [148]. Furthermore, each of the intermediate abstract machines is also correct with respect to call-by-value evaluation in the $\lambda\widehat{\rho}$-calculus.

The CEK machine is an environment machine for call-by-value evaluation in the λ-calculus. The following theorem formalizes this correspondence, using the function σ defined in Section 4.3.9.

Theorem 4.4 (Correspondence). *For any λ-term t, $t \rightarrow_{\mathrm{v}} \lambda t'$ if and only if*

$$\mathsf{evaluate}\, t = (\lambda t'',\ s) \quad \textit{and} \quad \sigma((\lambda t'')[s], 0) = \lambda t'.$$

4.5 A notion of context-sensitive reduction

Krivine's machine, as usually presented in the literature [1, 3, 51, 52, 54, 72, 96, 103–105, 133, 157, 189], contracts one β-redex at a time. The original version [124, 126], however, grammatically distinguishes nested λ-abstractions and contracts nested β-redexes in one step. Similarly, Leroy's Zinc machine [132] optimizes curried applications.

Krivine's language of λ-terms reads as follows:

$$\text{(terms)} \quad t ::= i \mid t\,t \mid \lambda^n t$$

for $n \geq 1$, and where a nested λ-abstraction $\lambda^n t$ corresponds to $\overbrace{\lambda\lambda\ldots\lambda}^{n} t$, i.e., to n nested λ-abstractions, where t is not a λ-abstraction.

In the original version of Krivine's machine, and using the same notation as in Section 4.3, (nested) β-reduction is implemented by the following transition:

$$\begin{aligned}&\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{ARG}((t_1,\, s_1),\, \ldots \mathsf{ARG}((t_n,\, s_n),\, C)\ldots))\\ &= \mathsf{refocus}\,(t,\, (t_n,\, s_n)\cdots(t_1,\, s_1)\cdot s,\, C)\end{aligned}$$

This transition implements a nested β-reduction not just for the pair of terms forming a β-redex, or even for a tuple of terms forming a nested β-redex, but *for a nested λ-abstraction and the context of its application.* The contraction function is therefore not solely defined over the redex to contract, but over a term and its context: it is context-sensitive.

In this section, we adjust the definition of a reduction semantics with a contract function that maps a redex and its context to a contractum and its context. Nothing else changes in the definition, and therefore the refocusing method still applies. We first consider normal order, and we show how the usual idealized version of Krivine's machine arises, how the original version of Krivine's machine also arises, and how one can derive a slightly more perspicuous version of the original version. We then consider applicative order, and we show how the Zinc machine arises.

4.5.1 Normal order: variants of Krivine's machine

Let us consider the language of the $\lambda\widehat{\rho}$-calculus based on Krivine's modified grammar of terms. Together with the language comes the following grammar of reduction contexts, which is induced by the compatibility rule (ν), just as in Section 4.3.1.

$$C ::= [\,] \mid \mathsf{ARG}(c,\, C)$$

Let us adapt the rules of Section 4.2.3 for context-sensitive reduction in $\lambda\widehat{\rho}$:

$$\begin{array}{lrl}(\beta^+) & ((\lambda^n t)[s],\, \mathsf{ARG}(c_1,\, \ldots \mathsf{ARG}(c_n,\, C)\ldots)) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t[c_n\cdots c_1\cdot s],\, C)\\ (\mathrm{Var}) & (i[c_1\cdots c_m],\, C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (c_i,\, C)\\ (\mathrm{App}) & ((t_0\,t_1)[s],\, C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} ((t_0[s])\,(t_1[s]),\, C)\\ (\nu) & (c_0\,c_1,\, C) & \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (c_0,\, \mathsf{ARG}(c_1,\, C))\end{array}$$

The new (β^+) rule is the only reduction rule that actually depends on the context (in the subsequent two rules the context remains unchanged). The left-compatibility rule (ν) now explicitly constructs the reduction context.

We notice that with the context-sensitive notion of reduction we are in position to express the one-step normal-order strategy already in the $\lambda\rho$-calculus, if we combine (App) and (ν) in one rule:

$$((t_0\, t_1)[s],\ C) \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[s],\ \mathsf{ARG}(t_1[s],\ C)).$$

In the context-sensitive reduction semantics corresponding to this normal-order context-sensitive reduction strategy, $\mathsf{contract}$ has type $\mathsf{Redex} \times \mathsf{Context} \to \mathsf{Closure} \times \mathsf{Context}$ and the one-step reduction function (see Section 4.3.1) reads as follows:

```
$\mathsf{reduce}\, c =$ $case$ $\mathsf{decompose}\, c$
                 $of$ $v \Rightarrow v$
                  $\mid (r,\ C) \Rightarrow \mathsf{plug}\,(c, C')$     where $(c,\ C') = \mathsf{contract}\,(r,\ C)$
```

We then take the same steps as in Section 4.3. We consider three variants, each of which depends on the specification of n in each instance of (β^+).

4.5.1.1 The idealized version of Krivine's machine

Here, we choose n to be 1. We then take the same steps as in Section 4.3, and obtain the same machine as in Section 4.3.7, i.e., the usual idealized version of Krivine's machine.

4.5.1.2 The original version of Krivine's machine

Here, we choose n to be the "arity" of each nested λ-abstraction, i.e., the number of nested λ's surrounding a term (the body of the innermost λ-abstraction) which is not a λ-abstraction.

We then take the same steps as in Section 4.3, and obtain the same machine as Krivine [124, 126]. As pointed out by Wand [189], since the number of arguments is required to match the number of nested λ-abstractions, the machine becomes stuck if there are not enough arguments in the context. We handle this case by adapting the β^+-rule as described next.

4.5.1.3 An adjusted version of Krivine's machine

Here, we choose n to be the smallest number between the arity of each nested λ-abstraction and the number of nested applications. (So, for example, $n = 1$ for $(\lambda\lambda t,\ \mathsf{ARG}(c_1,\ [\,]))$.)

We then take the same steps as in Section 4.3, and obtain a version of Krivine's machine that directly computes weak head normal forms.

4.5.2 Right-to-left applicative order: variants of the Zinc machine

From Landin [128] to Leroy [132], implementors of call-by-value functional languages have looked fondly upon right-to-left evaluation (i.e., evaluating the actual parameter before the function part of an application) because of its fit with a stack implementation: when the function part of a (curried) application yields a functional value, its parameter(s) is (are) available on top of the stack, as in the call-by-name case. In this section, we consider right-to-left applicative order and call by value, which as in the normal order and call-by-name case, give rise to a push/enter abstract machine.

We first adapt the rules of Section 4.2.4 for context-sensitive reduction in $\lambda\widehat{\rho}$. First of all, the compatibility rules (ν) and (μ), for right-to-left applicative order, induce the following grammar of contexts:[3]

$$C ::= [\,] \mid \mathsf{ARG}(v,\, C) \mid \mathsf{FUN}(c,\, C)$$

This grammar differs from the one of Section 4.4.1 because it is for right-to-left instead of for left-to-right applicative order.

The rules therefore read as follows:

$$\begin{array}{lrl}
(\beta^+) & ((\lambda^n t)[s],\, \mathsf{ARG}(v_1, \ldots \mathsf{ARG}(v_n,\, C)\ldots)) & \overset{\widehat{\rho}}{\to}_{\mathrm{v}} (t[v_n \cdots v_1 \cdot s],\, C) \\
(\mathrm{Var}) & (i[v_1 \cdots v_m],\, C) & \overset{\widehat{\rho}}{\to}_{\mathrm{v}} (v_i,\, C) \\
(\mathrm{App}) & ((t_0\, t_1)[s],\, C) & \overset{\widehat{\rho}}{\to}_{\mathrm{v}} ((t_0[s])\, (t_1[s]),\, C) \\
(\nu') & (v,\, \mathsf{FUN}(c,\, C)) & \overset{\widehat{\rho}}{\to}_{\mathrm{v}} (c,\, \mathsf{ARG}(v,\, C)) \\
(\mu') & (c_0\, c_1,\, C) & \overset{\widehat{\rho}}{\to}_{\mathrm{v}} (c_1,\, \mathsf{FUN}(c_0,\, C))
\end{array}$$

Similarly to the call-by-name case, the only context-sensitive reduction rule is (β^+) (we specify n as in Section 4.5.1.3), and the two compatibility rules explicitly construct reduction contexts.

As in Section 4.5.1.1, we notice that with the context-sensitive notion of reduction we are in position to express the one-step applicative-order strategy in the $\lambda\rho$-calculus if we combine (App) and (μ') in one rule:

$$((t_0\, t_1)[s],\, C) \overset{\widehat{\rho}}{\to}_{\mathrm{v}} (t_1[s],\, \mathsf{FUN}(t_0[s],\, C)).$$

We now take the same steps as in Section 4.4. The reduction semantics of Section 4.4.1, adapted to the grammar of contexts and to the reduction rules above, satisfies the refocusing conditions, and Danvy and Nielsen's algorithm yields the following definition:

[3]In the standard inside-out notation, this grammar can also be stated as follows:

$$C ::= [\,] \mid C[[\,]v] \mid C[c[\,]]$$

$$\begin{aligned}
\mathsf{refocus} &: \mathsf{Closure} \times \mathsf{Context} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\
\mathsf{refocus}\,(i[s],\, C) &= (i[s],\, C)\\
\mathsf{refocus}\,((\lambda^n t)[s],\, C) &= \mathsf{refocus_{aux}}\,(C,\, (\lambda^n t)[s])\\
\mathsf{refocus}\,((t_0\, t_1)[s],\, C) &= \mathsf{refocus}\,(t_1[s],\, \mathsf{FUN}(t_0[s],\, C))
\end{aligned}$$

$$\begin{aligned}
\mathsf{refocus_{aux}} &: \mathsf{Context} \times \mathsf{Value} \to \mathsf{Value} + (\mathsf{Redex} \times \mathsf{Context})\\
\mathsf{refocus_{aux}}\,([\,],\, v) &= v\\
\mathsf{refocus_{aux}}\,(\mathsf{FUN}(c,\, C),\, v) &= \mathsf{refocus}\,(c,\, \mathsf{ARG}(v,\, C))\\
\mathsf{refocus_{aux}}\,(\mathsf{ARG}(v',\, C),\, v) &= (v,\, \mathsf{ARG}(v',\, C))
\end{aligned}$$

We then successively transform the resulting pre-abstract machine into a staged abstract machine, an eval/apply abstract machine, a push/enter abstract machine, and a push/enter abstract machine with unfolded closures.

The resulting environment machine reads as follows:

$$\mathsf{evaluate}\, t = \mathsf{refocus}\,(t,\, \bullet,\, [\,])$$

$$\begin{aligned}
\mathsf{refocus}\,(i,\, (t_1,\, s_1)\cdots(t_m,\, s_m),\, C) &= \mathsf{refocus}\,(t_i,\, s_i,\, C)\\
\mathsf{refocus}\,(\lambda^n t,\, s,\, [\,]) &= (\lambda^n t,\, s)\\
\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{FUN}((t',\, s'),\, C)) &= \mathsf{refocus}\,(t',\, s',\, \mathsf{ARG}((\lambda^n t,\, s),\, C))\\
\mathsf{refocus}\,(\lambda^n t,\, s,\, \mathsf{ARG}(v_1,\, \ldots \mathsf{ARG}(v_n,\, C)\ldots)) &= \mathsf{refocus}\,(t,\, v_n \cdot \ldots \cdot v_1 \cdot s,\, C)\\
\mathsf{refocus}\,(t_0\, t_1,\, s,\, C) &= \mathsf{refocus}\,(t_1,\, s,\, \mathsf{FUN}((t_0,\, s),\, C))
\end{aligned}$$

This machine corresponds to an instance of Leroy's Zinc machine for the pure λ-calculus [132, Chapter 3], with the proviso that it operates directly on λ-terms instead of over an instruction set (which has been said to be the difference between an abstract machine and a virtual machine [2]). Moreover, in the Zinc machine the sequence of values on the stack (denoted here by $\mathsf{ARG}(v_1,\, \ldots \mathsf{ARG}(v_n,\, C)\ldots))$ is delimited by a stack mark that separates already evaluated terms from unevaluated ones. The Zinc machine was developed independently of Krivine's machine and to the best of our knowledge the reconstruction outlined here is new.

4.6 A space optimization for call-by-name evaluation

In the compilation model of ALGOL 60, which is a call-by-name programming language, identifiers occurring as actual parameters are compiled by (1) looking up their value in the current environment, and (2) passing this value to the callee [153, Section 2.5.4.10, pages 109-110]. The rationale is that under call by name, an identifier denotes a thunk, so there is no need to create another thunk for it. This compilation rule avoids a space leak at run time and it is commonly used in implementations of lazy functional programming languages.

To circumvent the space leak, one extends Krivine's machine with the following clause for the case where the actual parameter is a variable:

$$\begin{aligned}
&\mathsf{refocus}\,(t_0\, i,\, (t_1,\, s_1)\cdots(t_m,\, s_m),\, C)\\
&= \mathsf{refocus}\,(t_0,\, (t_1,\, s_1)\cdots(t_m,\, s_m),\, \mathsf{ARG}((t_i,\, s_i),\, C))
\end{aligned}$$

We observe that circumventing the space leak has an analogue in the normal-order reduction semantics: it corresponds to adding the following reduction rule to the $\lambda\widehat{\rho}$-calculus:

$$(\mathrm{App}') \qquad (t_0\; i)[c_1 \cdots c_m] \xrightarrow{\widehat{\rho}}_{\mathrm{n}} (t_0[c_1 \cdots c_m])\; c_i$$

This addition shortens the reduction sequence of a given λ-term towards its weak head normal form.

The context-insensitive reduction semantics: Taking the same steps as in Section 4.3 mechanically leads one to an idealized version of Krivine's machine with the space optimization. Crégut has considered this space-optimized version [51] and Friedman, Ghuloum, Siek, and Winebarger have measured the impact of this optimization for a lazy version of Krivine's machine [96].

The context-sensitive reduction semantics: Taking the same steps as in Section 4.3 mechanically leads one to an adjusted version of Krivine's machine with the space optimization.

4.7 Actual substitutions, explicit substitutions, and environments

The derivations in Sections 4.3, 4.4, 4.5, and 4.6 hint at a bigger picture that we draw in Figure 4.1 and describe in Section 4.7.1. We then address the reversibility of the derivation steps in Section 4.7.2.

4.7.1 A derivational taxonomy of abstract machines

Let us analyze Figure 4.1.

The left-most column concerns the reduction and evaluation of terms with actual substitutions (λ). The second column concerns the reduction and evaluation of terms with explicit substitutions ($\lambda\widehat{\rho}$). The third column concerns the evaluation of terms with explicit substitutions ($\lambda\rho$). The right-most column concerns the evaluation of terms using an environment.

Reading down each column follows Danvy and Nielsen's work on refocusing [72]. They show how to go from an evaluation function (obtained as the reflexive-transitive closure of a one-step reduction function) to a pre-abstract machine, and then to a staged machine, an eval/apply machine, and, for call by name and right-to-left call by value, a push/enter machine.

The connections between the columns in the 4×4-matrix are new. Given a closure (i.e., a term and a substitution), carrying out the substitution in this term yields a new term. Through this operation (depicted with the short dotted arrow in the diagram), we can go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for λ-terms. To go from each of the abstract machines for $\lambda\widehat{\rho}$-closures to the corresponding abstract machine for $\lambda\rho$-closures, we shortcut the (App) reduction step with the following decomposition step. To go from each of the abstract machines for $\lambda\rho$-closures to the corresponding environment machine, we unfold the data type of closures into a

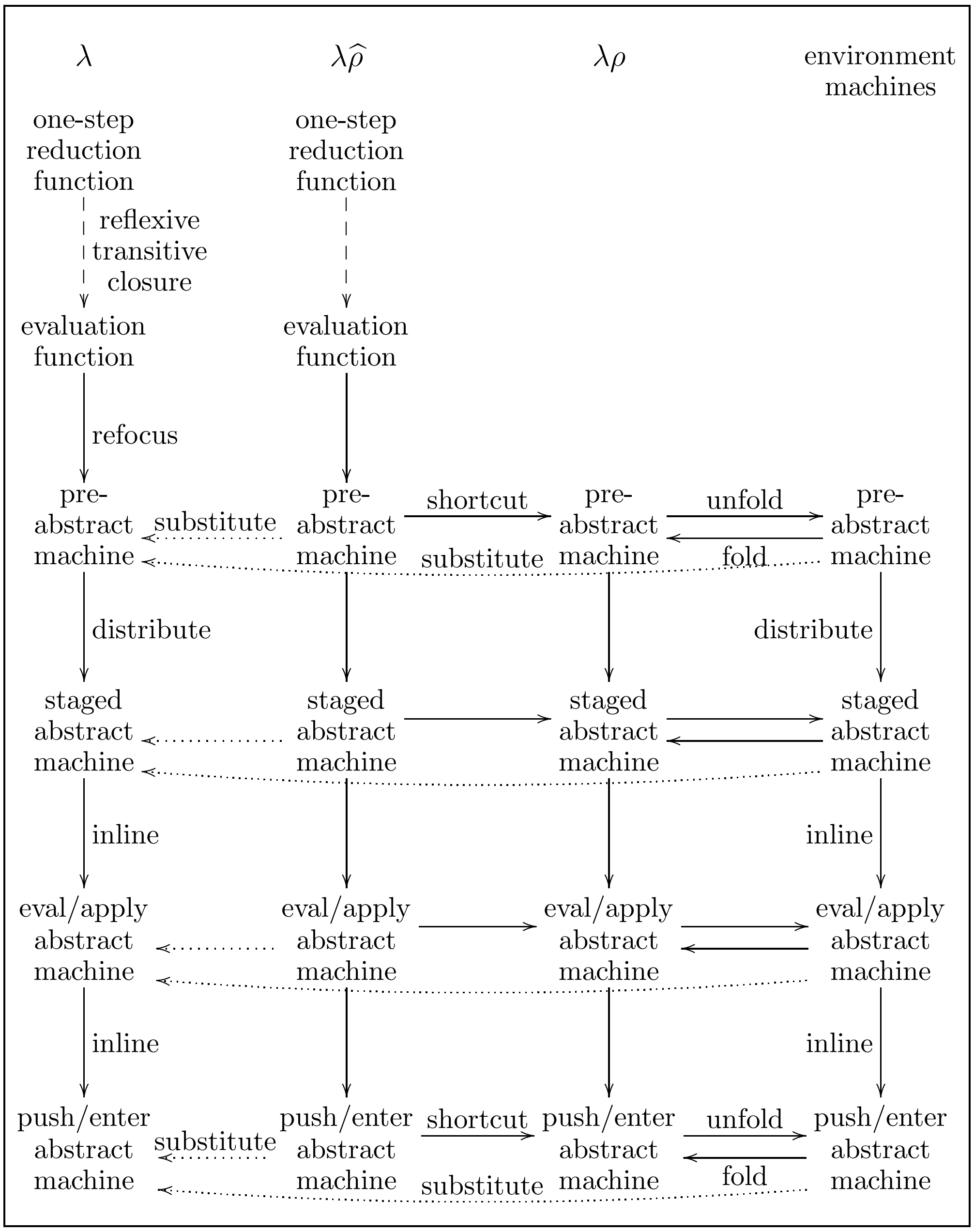

Figure 4.1: A derivational taxonomy of abstract machines

pair of term and substitution (the unfolded substitution acts as the environment of the machine). Finally, given a term and an environment, carrying out the delayed substitutions represented by the environment in the term yields a new term. Through this operation (depicted with the long dotted arrow in the diagram), we can go from each of the environment machines to the corresponding abstract machine for λ-terms.

In Section 4.3, we observed that the push/enter environment machine corresponding to normal-order reduction coincides with the usual idealized version of Krivine's machine [51,54,103]. In Section 4.4, we observed that the eval/apply environment machine corresponding to applicative-order reduction coincides with Felleisen's et al.'s CEK machine [84]. In Section 4.5, we observed that a context-sensitive reduction semantics gives rise to the original version of Krivine's machine and to the Zinc machine. In Section 4.6, we observed that the space optimization of two families of call-by-name machines corresponds to two reduction semantics.[4]

Earlier [72], Danvy and Nielsen observed that the eval/apply machine with actual substitution corresponding to applicative-order reduction coincides with Felleisen et al.'s CK machine [84]. Obtaining staged abstract machines was one of the goals of Hardin, Maranget, and Pagano's study of functional runtime systems using explicit substitutions [105]; these machines arise mechanically here. As investigated by Danvy and his students in their study of the functional correspondence between compositional evaluation functions and abstract machines [3–5,63], the eval/apply machines are in defunctionalized form [71,155] and they can be 'refunctionalized' into a continuation-passing evaluation function which itself can be written in direct style [58] and implements a big-step operational semantics. (Because of the closures, the result is again in defunctionalized form and can be refunctionalized into a compositional evaluation function implementing the valuation function of a denotational semantics.)

Each of the machines in Figure 4.1 is thus of independent value.

4.7.2 Reversibility of the derivation steps

Going from a push/enter machine to the corresponding eval/apply machine or from a push/enter machine or an eval/apply machine to a staged machine requires a degree of insight [105]. Going from a staged machine to a pre-abstract machine and from a pre-abstract machine to a reduction semantics is mechanical. We are, however, not aware of any systematic method to go from an arbitrary abstract machine to a reduction semantics [28,85].

Going from an environment machine where the environment is treated as a list to a closure-based machine is done by folding the pair (term, environment) into a closure. The resulting machine mediates between an environment-based specification and an explicit-substitution-based specification. Obtaining an explicit-substitution-based machine from this intermediate machine, however, requires a major architectural overhaul.

4.8 Conclusion

Curien originally presented a simple calculus of closures, the $\lambda\rho$-calculus, as an abstract framework for environment machines [54]. This approach gave rise to a general study of explicit substitutions [1,55,105,133,156,157] where a number

[4]In Sections 4.3 and 4.5, the derivations follow the second column all the way down to a push/enter machine, and then across the columns to the right, to the corresponding push/enter environment machine. In Section 4.4, we go down the second column to an eval/apply abstract machine, and then across the columns to the right, to the corresponding eval/apply environment machine.

of abstract machines have been obtained through a combination of skill and ingenuity.

We have presented a concrete framework for environment machines where abstract machines are methodically derived from specifications of reduction strategies, and their correctness is ensured by the correctness of the derivation method. The derivation is based on Danvy and Nielsen's refocusing method, which requires the one-step specification of a reduction strategy, i.e., a reduction semantics. For this reason, we needed to extend Curien's original $\lambda\rho$-calculus with closure application, yielding the $\lambda\widehat{\rho}$-calculus.

We have illustrated the concrete framework by deriving several known environment machines—Krivine's abstract machine both in idealized form and in original form, Felleisen et al.'s CEK machine, and Leroy's Zinc machine—from the normal-order and the applicative-order reduction strategies expressed in the $\lambda\widehat{\rho}$-calculus, both in context-insensitive and in context-sensitive form. The last step of the derivation (closure unfolding) provides a precise characterization of what has now become folklore: that explicit substitutions mediate between actual substitutions and environments.

Acknowledgments: Thanks are due to Mads Sig Ager, Dariusz Biernacki, Mayer Goldberg, Julia Lawall, Jan Midtgaard, Kevin Millikin, and Kristian Støvring for comments, and to Ulrich Kohlenbach for providing us with what appears to be the original bibliographic reference of the logic counterpart of environments [160, § 54].

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 5

A syntactic correspondence between context-sensitive calculi and abstract machines

with Olivier Danvy [30]

Abstract

We present a systematic construction of environment-based abstract machines from context-sensitive calculi of explicit substitutions, and we illustrate it with a series of calculi and machines: Krivine's machine with call/cc, the $\lambda\mu$-calculus, delimited continuations, i/o, stack inspection, proper tail-recursion, and lazy evaluation. Most of the machines already exist but have been obtained independently and are only indirectly related to the corresponding calculi. All of the calculi are new and they make it possible to directly reason about the execution of the corresponding machines. In connection with the functional correspondence between evaluation functions and abstract machines initiated by Reynolds, the present syntactic correspondence makes it possible to construct reduction-free normalization functions out of reduction-based ones, which was an open problem in the area of normalization by evaluation.

5.1 Introduction

5.1.1 Calculi and machines

Sixty-five years ago, the λ-calculus was introduced [42]. Forty-five years ago, its expressive power was observed to be relevant for computing [137,172]. Forty years ago, a first abstract machine for the λ-calculus was introduced [128]. Thirty years ago, calculi and abstract machines were formally connected [148]. Twenty years ago, a calculus format—reduction semantics—with an explicit representation of reduction contexts was introduced [82]. Today calculi and abstract machines are standard tools to study programming languages. Given a calculus, it is by now a standard activity to design a corresponding abstract machine and to prove its correctness [84].

From calculus to machine by refocusing and transition compression: Recently, Danvy and Nielsen have pointed out that the reduction strategy for a calculus actually determines the structure of the corresponding machine [72]. They present an algorithm to construct an abstract machine out of a reduction semantics satisfying the unique-decomposition property. In such a reduction semantics, a non-value term is reduced by

1. decomposing it (uniquely) into a redex and its context,
2. contracting the redex, and
3. plugging the contractum into the reduction context,

yielding a new term. An evaluation function is defined using the reflexive and transitive closure of the one-step reduction function:

$$\begin{tikzcd}
\circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & \\
& \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] & & \circ \arrow[r, "\text{contract}"'] & {}
\end{tikzcd}$$

Danvy and Nielsen have observed that the construction of the intermediate terms, in the composition of plug and decompose, could be avoided by fusing the composition into a 'refocus' function:

$$\begin{tikzcd}
& \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & & \circ \arrow[dr, "\text{decompose}"] & & \\
{} \arrow[rr, dashed] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & \circ \arrow[ur, "\text{plug}"] \arrow[rr, dashed, "\text{refocus}"'] & & \circ \arrow[r, "\text{contract}"'] & {}
\end{tikzcd}$$

The resulting 'refocused' evaluation function is defined as the reflexive and transitive closure of refocusing and contraction.

Danvy and Nielsen's algorithm yields a refocus function in the form of a state-transition function, i.e., an abstract machine. The refocused evaluation function therefore also takes the form of an abstract machine. Compressing its intermediate transitions (i.e., short-circuiting them) yields abstract machines that are often independently known: for example, for the pure λ-calculus with normal order, the resulting abstract machine is a substitution-based version of the Krivine machine (i.e., a push/enter machine); for the pure λ-calculus with applicative order, the resulting abstract machine is Felleisen et al.'s CK machine (i.e., an eval/apply machine). Refocusing has also been applied to the term language of the free monoid, yielding a reduction-free normalization function [61], and to context-based CPS transformations, improving them from quadratic time to operating in one pass [72].

5.1.2 Calculi of explicit substitution and environment-based machines

Twenty years ago, Curien observed that while most calculi use actual substitutions, most implementations use closures and environments [53]. He then developed a calculus of closures, $\lambda\rho$ [54], thereby launching the study of calculi of explicit substitutions [1, 55, 105].

From calculus to machine by refocusing, transition compression, and closure unfolding: Recently, we have applied the refocusing method to $\lambda\widehat{\rho}$, a minimal extension of $\lambda\rho$ where one can express single-step computations; we added an unfolding step to make the machine operate not on a closure, but on a term and its environment [29]. We have shown how $\lambda\widehat{\rho}$ with left-to-right applicative order directly corresponds to the CEK machine [85], how $\lambda\widehat{\rho}$ with normal order directly corresponds to the Krivine machine [51, 54], how $\lambda\widehat{\rho}$ with generalized reduction directly corresponds to the original version of Krivine's machine [124], and how $\lambda\widehat{\rho}$ with right-to-left applicative order directly corresponds to the ZINC abstract machine [132]. All of these machines are environment-based and use closures.

5.1.3 Calculi for computational effects and environment-based machines

Twenty years ago, Felleisen introduced reduction semantics – a version of small-step operational semantics with an explicit representation of reduction contexts – in order to provide calculi for control and state [82, 85]. In these calculi, reduction rules are not oblivious to their reduction context; on the contrary, they are context sensitive in that the context takes part in some reduction steps, e.g., for call/cc. Reduction semantics are in wide use today, e.g., to study the security technique of stack inspection [43, 95, 151].

From calculus to machine by refocusing, transition compression, and closure unfolding: In this article, we apply the refocusing method to context-sensitive extensions of $\lambda\widehat{\rho}$ accounting for a variety of computational effects. We present a variety of calculi of closures and the corresponding environment-based machines. What is significant here is that each machine is mechanically derived from the corresponding calculus (instead of designed and then proved correct) and also that each machine directly corresponds to this calculus (instead of indirectly via an 'unload' function at the end of each run [148] or via a compilation / decompilation scheme in the course of execution [105]).

5.1.4 Overview

We successively consider call by name: Krivine's machine with call/cc (Section 5.3) and the $\lambda\mu$-calculus (Section 5.4); call by value: static and dynamic delimited continuations (Section 5.5), i/o (Section 5.6), stack inspection (Section 5.7), and proper tail-recursion (Section 5.8); and call by need (Section 5.9). Towards this end, we first present the $\lambda\widehat{\rho}$-calculus and the notion of context-sensitive reduction (Section 5.2).

5.2 Preliminaries

5.2.1 Our base calculus of closures: $\lambda\widehat{\rho}$

Since Landin [128], most abstract machines implementing variants and extensions of the λ-calculus use closures and environments, and the substitution of terms for free variables is thus delayed until a variable is reached in the evaluation process. This implementation technique motivated the study of calculi of explicit substitutions [1, 54, 156] to mediate between the traditional abstract specifications of the λ-calculus and its traditional concrete implementations [105].

To derive an abstract machine for evaluating λ-terms, a *weak* calculus of explicit substitutions suffices. The first (and simplest) of such calculi was Curien's calculus of closures $\lambda\rho$ [54]. Although this calculus is not expressive enough to model full normalization, it is suitable for evaluating λ-terms. Its operational semantics is specified using multi-step reductions, but its syntax is too restrictive to allow single-step computations, which is what we need to apply the refocusing algorithm. For this reason, in our earlier work [29], we have proposed a minimal extension of $\lambda\rho$ with one-step reduction rules, the $\lambda\widehat{\rho}$-calculus.

The language of $\lambda\widehat{\rho}$ is as follows:

$$\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s
\end{array}$$

(For comparison, $\lambda\rho$ does not have the $c\,c$ production.)

We use de Bruijn indices for variables in a term ($i \geq 1$). A closure is a term equipped with a substitution, i.e., a list of closures to be substituted for free variables in the term. Programs are closures of the form $t[\bullet]$ where t does not contain free variables.

The notion of reduction in $\lambda\widehat{\rho}$ is given by the following rules:

$$\begin{array}{lrcll}
\text{(Var)} & i[c_1 \cdots c_j] & \rightarrow_{\widehat{\rho}} & c_i & \text{if } i \leq j \\
\text{(Beta)} & ((\lambda t)[s])\,c & \rightarrow_{\widehat{\rho}} & t[c \cdot s] & \\
\text{(Prop)} & (t_0\,t_1)[s] & \rightarrow_{\widehat{\rho}} & (t_0[s])\,(t_1[s]) &
\end{array}$$

We denote by $s(i)$ the ith element of the substitution s considered as a list. (So $[c_1 \cdots c_j](i) = c_i$ if $1 \leq i \leq j$.)

Finally, the one-step reduction relation (i.e., the compatible closure of the notion of reduction) extends the notion of reduction with the following rules:

$$\begin{array}{lcl}
\text{(L-Comp)} & \dfrac{c_0 \rightarrow_{\widehat{\rho}} c_0'}{c_0\,c_1 \rightarrow_{\widehat{\rho}} c_0'\,c_1} & \\[2ex]
\text{(R-Comp)} & \dfrac{c_1 \rightarrow_{\widehat{\rho}} c_1'}{c_0\,c_1 \rightarrow_{\widehat{\rho}} c_0\,c_1'} & \\[2ex]
\text{(Sub)} & \dfrac{c_i \rightarrow_{\widehat{\rho}} c_i'}{t[c_1 \cdots c_i \cdots c_j] \rightarrow_{\widehat{\rho}} t[c_1 \cdots c_i' \cdots c_j]} & \text{for } i \leq j
\end{array}$$

Specific, deterministic reduction strategies can be obtained by restricting the compatibility rules. In the following sections, we consider two such strategies: the normal-order strategy obtained by discarding the (R-Comp) and (Sub) rules, and the left-to-right applicative-order strategy obtained in the usual way by restricting the (Beta) and (R-Comp) rules, and discarding the (Sub) rule.

All of the calculi presented in this article are syntactic extensions of the $\lambda\widehat{\rho}$-calculus.

5.2.2 Notion of context-sensitive reduction

Traditional specifications of one-step reduction as the compatible closure of a notion of reduction provide a *local* characterization of a computation step in the form of a (potential) redex.[1] This local characterization is not fit for non-local reductions such as one involving a control operator capturing all its surrounding context in one step, or a global state. For these, one needs a notion of *context-sensitive* reduction, i.e., a binary relation defined both on redexes and on their reduction context instead of only on redexes.

A one-step reduction relation for a given notion of context-sensitive reduction is defined as follows, assuming this notion to be specified by reduction rules of the form $\langle r,\ C\rangle \rightarrow \langle c,\ C'\rangle$, where $\langle r,\ C\rangle$ denotes the decomposition of a program into a potential redex r and its context C. A program p reduces in one step to p' if decomposing p yields $\langle r,\ C\rangle$, contracting $\langle r,\ C\rangle$ yields $\langle c,\ C'\rangle$, and plugging c into C' yields p'.

Any standard notion of reduction can be trivially transformed into context-sensitive form. For example, here is the corresponding notion of reduction for the $\lambda\widehat{\rho}$-calculus:

$$\begin{array}{lrcll}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C\rangle & \rightarrow_{\widehat{\rho}} & \langle c_i,\ C\rangle & \text{if } i \leq j \\
\text{(Beta)} & \langle((\lambda t)[s])\, c,\ C\rangle & \rightarrow_{\widehat{\rho}} & \langle t[c \cdot s],\ C\rangle & \\
\text{(Prop)} & \langle(t_0\, t_1)[s],\ C\rangle & \rightarrow_{\widehat{\rho}} & \langle(t_0[s])\,(t_1[s]),\ C\rangle &
\end{array}$$

A context-sensitive reduction implicitly assumes a decomposition of the entire program, and therefore it cannot be used locally. One way to recover compatibility in the context-sensitive setting is to add explicit local control delimiters to the language (see Section 5.5 for an illustration). For a language without explicit control delimiters (as the $\lambda\widehat{\rho}$-calculus with call/cc), there is an implicit global control delimiter around the program.

For each of the calculi considered in the remainder of this article, we define a suitable notion of reduction, denoted $\rightarrow_X$, where X is a subscript identifying a particular calculus. For each of them, we then define a one-step reduction relation as the composition of: decomposing a non-value closure into a redex and a reduction context, contracting the (context-sensitive) redex, and then plugging the resulting closure into the resulting context. Finally, we define the

[1]For example, a potential redex in the call-by-name λ-calculus is the application of a value to a term. If the value is a λ-abstraction, the potential redex is an actual one and it can be β-reduced. If no reduction rule applies, the potential redex is not an actual one and the program is stuck [148].

evaluation relation (denoted $\rightarrow_X^*$) using the reflexive, transitive closure of one-step reduction, i.e., we say that c evaluates to c' if $c \rightarrow_X^* c'$ and c' is a value closure. We define the convertibility relation between closures as the smallest equivalence relation containing $\rightarrow_X^*$. If two closures c and c' are convertible, they behave similarly under evaluation (i.e., either they both evaluate to the same value, or they both diverge).

5.3 The $\lambda\widehat{\rho}\mathcal{K}$-calculus

The Krivine machine is probably the most well-known abstract machine implementing the normal-order reduction strategy in the λ-calculus [64]. In our previous work [29], we have pointed out that Krivine's original machine [124] does not coincide with the Krivine Machine As We Know It [52,54] in that it implements <u>generalized</u> instead of <u>ordinary</u> β-reduction: indeed Krivine's machine reduces the term $(\lambda\lambda t)\, t_1\, t_2$ in one step whereas the Krivine machine reduces it in two steps. Furthermore, the archival version of Krivine's machine [125] also handles call/cc (noted $\mathcal{K}$ below).

In our previous work [29], we have presented the calculus corresponding to the original version of Krivine's machine. This machine uses closures and an environment and correspondingly, the calculus is one of explicit substitutions, $\lambda\widehat{\rho}$.

Here, we present the calculus corresponding to the archival version of Krivine's machine. This machine also uses closures and an environment. In addition to generalized β-reduction, it also features $\mathcal{K}$. Correspondingly, the calculus is one of explicit substitutions, $\lambda\widehat{\rho}\mathcal{K}$. We build on top of Krivine's language of terms by specifying syntactic categories of closures and substitutions as shown below. The calculus is tied to a particular reduction strategy. Here, like Krivine, we consider the normal-order reduction strategy and therefore call by name [148].

5.3.1 The language of $\lambda\widehat{\rho}\mathcal{K}$

The abstract syntax of the language is as follows:

$$\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda^n t \mid t\, t \mid \mathcal{K}\, t \\
\text{(closures)} & c & ::= & t[s] \mid c\, c \mid \mathcal{K}\, c \mid \ulcorner C \urcorner \\
\text{(values)} & v & ::= & (\lambda^n t)[s] \mid \ulcorner C \urcorner \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\, c] \mid C[\mathcal{K}[\,]]
\end{array}$$

A nested λ-abstraction of the form $\lambda^n t$ is to be understood as a syntactic abbreviation for $\underbrace{\lambda\lambda\ldots\lambda}_{n} t$, where t is not a λ-abstraction.

In $\lambda\widehat{\rho}\mathcal{K}$, a value is either a closure with a λ-abstraction in the term part, or the representation of a reduction context captured by $\mathcal{K}$.

5.3.2 Notion of context-sensitive reduction

The notion of reduction is specified by the set of rules shown below. The rules (Var) and (Prop) are as in the $\lambda\widehat{\rho}$-calculus, and (Beta$^+$) supersedes the (Beta) rule by performing a generalized β-reduction in one step:

$$\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C\rangle & \rightarrow_{\mathcal{K}} & \langle c_i,\ C\rangle \ \text{ if } i \leq j \\
\text{(Beta}^+\text{)} & \langle(\lambda^n t)[s],\ C[[...[[\,]c_n]...]c_1]\rangle & \rightarrow_{\mathcal{K}} & \langle t[c_1 \cdots c_n \cdot s],\ C\rangle \\
\text{(Beta}_C\text{)} & \langle\ulcorner C'\urcorner,\ C[[\,]\,c]\rangle & \rightarrow_{\mathcal{K}} & \langle c,\ C'\rangle \\
\text{(Prop)} & \langle(t_0\, t_1)[s],\ C\rangle & \rightarrow_{\mathcal{K}} & \langle(t_0[s])\,(t_1[s]),\ C\rangle \\
\text{(Prop}_{\mathcal{K}}\text{)} & \langle(\mathcal{K}\, t)[s],\ C\rangle & \rightarrow_{\mathcal{K}} & \langle\mathcal{K}\,(t[s]),\ C\rangle \\
(\mathcal{K}_\lambda) & \langle\mathcal{K}\,((\lambda t)[s]),\ C\rangle & \rightarrow_{\mathcal{K}} & \langle t[\ulcorner C\urcorner \cdot s],\ C\rangle \\
(\mathcal{K}_C) & \langle\mathcal{K}\,\ulcorner C'\urcorner,\ C\rangle & \rightarrow_{\mathcal{K}} & \langle\ulcorner C'\urcorner\,\ulcorner C\urcorner,\ C\rangle
\end{array}$$

The three last rules account for call/cc: the first is an ordinary propagation rule, and the two others describe a continuation capture. In the first case, the current continuation is passed to a function, and in the second, it is passed to an already captured continuation.

5.3.3 Krivine's machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

$$\begin{array}{rcl}
\langle i,\ s,\ C\rangle & \Rightarrow_{\mathcal{K}} & \langle t',\ s',\ C\rangle \ \text{ if } s(i) = (t',\ s') \\
\langle i,\ s,\ C\rangle & \Rightarrow_{\mathcal{K}} & \langle C,\ \ulcorner C'\urcorner\rangle \ \text{ if } s(i) = \ulcorner C'\urcorner \\
\langle\lambda^n t,\ s,\ C\rangle & \Rightarrow_{\mathcal{K}} & \langle C,\ (\lambda^n t,\ s)\rangle \\
\langle t_0\, t_1,\ s,\ C\rangle & \Rightarrow_{\mathcal{K}} & \langle t_0,\ s,\ C[[\,]\,(t_1,\ s)]\rangle \\
\langle\mathcal{K}\, t,\ s,\ C\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\ s,\ C[\mathcal{K}[\,]]\rangle \\
\\
\langle[\,],\ v\rangle & \Rightarrow_{\mathcal{K}} & v \\
\langle C[[...[[\,]c_n]...]c_1],\ (\lambda^n t,\ s)\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\ c_1 \cdots c_n \cdot s,\ C\rangle \\
\langle C[[\,]\,(t,\ s)],\ \ulcorner C'\urcorner\rangle & \Rightarrow_{\mathcal{K}} & \langle t,\ s,\ C'\rangle \\
\langle C[[\,]\,\ulcorner C''\urcorner],\ \ulcorner C'\urcorner\rangle & \Rightarrow_{\mathcal{K}} & \langle C',\ \ulcorner C''\urcorner\rangle \\
\langle C[\mathcal{K}[\,]],\ v\rangle & \Rightarrow_{\mathcal{K}} & \langle C[[\,]\,\ulcorner C\urcorner],\ v\rangle
\end{array}$$

This machine coincides with the extension of Krivine's machine with $\mathcal{K}$—an extension which was designed as such [125].

5.3.4 Formal correspondence

In the remainder of the chapter, we will use the notation $\overline{v}$ for an "unfolded" value v. For example, in the present case,

$$\overline{v} ::= (\lambda t,\ s) \mid \ulcorner C\urcorner.$$

Proposition 5.1. *For any term t in the $\lambda\widehat{\rho}\mathcal{K}$-calculus,*

$$t[\bullet] \rightarrow_{\mathcal{K}}^{*} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,]\rangle \Rightarrow_{\mathcal{K}}^{*} \overline{v}.$$

The $\lambda\widehat{\rho}\mathcal{K}$-calculus therefore directly corresponds to the archival version of Krivine's machine with call/cc.

5.4 The $\lambda\widehat{\rho}\mu$-calculus

In this section we present a calculus of closures that extends Parigot's $\lambda\mu$-calculus [145] and the corresponding call-by-name abstract machine obtained by refocusing.

We want to compare our derived abstract machine with an existing one designed by de Groote [76] and therefore we adapt his syntax, which differs from Parigot's in that arbitrary terms can be abstracted by μ (not only named ones). In addition, de Groote presents a calculus of explicit substitutions built on top of the $\lambda\mu$-calculus, and uses it to prove the correctness of his machine. We show that a $\lambda\widehat{\rho}$-like calculus of closures is enough to model evaluation in the $\lambda\mu$-calculus and to derive the same abstract machine as de Groote.

The $\lambda\mu$-calculus is typed, and suitable typing rules can be given to the calculus of closures we present below. The reduction rules we show satisfy the subject reduction property, and in consequence, the machine we derive operates on typed terms. For lack of space, however, we omit all the typing considerations, focusing on the syntactic correspondence between the calculus and the machine.

5.4.1 The language of $\lambda\widehat{\rho}\mu$

We use de Bruijn indices for both the λ-bound variables and the μ-bound variables. The two kinds of variables are represented using the same set of indices, which leads one to an abstract machine with one environment [76]. Alternatively, we could use two separate sets of indices, which would then lead us to two environments in the resulting machine (one for each kind of variable).

The abstract syntax of the language is specified as follows:

$$\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \mid \mu t \mid [i]t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \\
\text{(values)} & v & ::= & (\lambda t)[s] \\
\text{(substitutions)} & s & ::= & \bullet \mid C \cdot s \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c]
\end{array}$$

We consider only closed λ-terms, and $i \geq 0$. Bound variables are indexed starting with 1, and a (free) occurrence of a variable 0 indicates a distinguished toplevel continuation (similar to `tp` in Ariola et al.'s setting [8]). A substitution is a non-empty sequence of either closures—to be substituted for λ-bound variables, or captured reduction contexts—to be used when accessing μ-bound variables.

Programs are closures of the form $t[[\,]\cdot\bullet]$, where the empty context is to be substituted for the toplevel continuation variable 0.

5.4.2 Notion of context-sensitive reduction

The notion of reduction extends that of the $\lambda\widehat{\rho}$ with two rules: (Mu), which captures the entire reduction context and stores it in the substitution, and (Rho), which reinstates a captured context when a continuation variable is applied:

$$\begin{array}{lrcl}
\text{(Beta)} & \langle(\lambda t)[s],\ C[[\,]\,c]\rangle & \to_\mu & \langle t[c\cdot s],\ C\rangle \\
\text{(Var)} & \langle i[s],\ C\rangle & \to_\mu & \langle c,\ C\rangle \quad \text{if } s(i)=c \\
\text{(Prop)} & \langle(t_0\,t_1)[s],\ C\rangle & \to_\mu & \langle(t_0[s])\,(t_1[s]),\ C\rangle \\
\text{(Mu)} & \langle(\mu t)[s],\ C\rangle & \to_\mu & \langle t[C\cdot s],\ [\,]\rangle \\
\text{(Rho)} & \langle([i]t)[s],\ [\,]\rangle & \to_\mu & \langle t[s],\ C\rangle \quad \text{if } s(i)=C
\end{array}$$

5.4.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

$$\begin{array}{rcl}
\langle\lambda t,\ s,\ C\rangle & \Rightarrow_\mu & \langle C,\ (\lambda t,\ s)\rangle \\
\langle i,\ s,\ C\rangle & \Rightarrow_\mu & \langle t',\ s',\ C\rangle \quad \text{if } s(i)=(t',\ s') \\
\langle t_0\,t_1,\ s,\ C\rangle & \Rightarrow_\mu & \langle t_0,\ s,\ C[[\,]\,(t_1,\ s)]\rangle \\
\langle\mu t,\ s,\ C\rangle & \Rightarrow_\mu & \langle t,\ C\cdot s,\ [\,]\rangle \\
\langle[i]t,\ s,\ [\,]\rangle & \Rightarrow_\mu & \langle t,\ s,\ C\rangle \quad \text{if } s(i)=C \\
\\
\langle[\,],\ v\rangle & \Rightarrow_\mu & v \\
\langle C[[\,]\,c],\ (\lambda t,\ s)\rangle & \Rightarrow_\mu & \langle t,\ c\cdot s,\ C\rangle
\end{array}$$

This machine coincides with de Groote's final abstract machine [76, p. 24], except that instead of traversing the environment as a list, we directly fetch the right substitutee for a given index i.

5.4.4 Formal correspondence

Proposition 5.2. *For any term t in the $\lambda\widehat{\rho}\mu$-calculus,*

$$t[[\,]\cdot\bullet] \to_\mu^* v \quad \textit{if and only if} \quad \langle t,\ [\,]\cdot\bullet,\ [\,]\rangle \Rightarrow_\mu^* \overline{v}.$$

The $\lambda\widehat{\rho}\mu$-calculus therefore directly corresponds to de Groote's abstract machine for the $\lambda\mu$-calculus.

5.5 Delimited continuations

Continuations have been discovered multiple times [154], but they acquired their name for describing jumps [173], using what is now known as continuation-passing style (CPS) [168]. A full-fledged control operator, J [127,177], however, existed before CPS, providing first-class continuations in direct style. Continuations therefore existed before CPS, and so one could say that it was really CPS that was discovered multiple times.

Conversely, delimited continuations, in the form of the traditional success and failure continuations [159], have been regularly used in artificial-intelligence programming [41, 109, 176] for generators and backtracking. They also occur in the study of reflective towers [166], where the notions of meta-continuation [190] and of "jumpy" vs. "pushy" continuations [70] arose. A full-fledged delimited control operator, # (pronounced "prompt"), however, was introduced independently of CPS and of reflective towers, to support operational equivalence in λ-calculi with first-class control [83, 86]. Only subsequently were control delimiters connected to success and failure continuations [67].

The goal of this section is to provide a uniform account of delimited continuations. Three data points are in presence: a calculus and an abstract machine, both invented by Felleisen [83], and CPS, as discovered by Danvy and Filinski [67].

Calculus: As we show below, an explicit-substitutions version of Felleisen's calculus of dynamic delimited continuations can be refocused into his extension of the CEK machine, which uses closures and an environment.

Abstract machine: As we have shown elsewhere [34], Felleisen's extension of the CEK machine is not in defunctionalized form (at least for the usual notion of defunctionalization [71, 155]); it needs some adjustment to be so, which leads one to a dynamic form of CPS that threads a state-like trail of delimited contexts.

CPS: Defunctionalizing Danvy and Filinski's continuation-based evaluator yields an environment-based machine [28], and we present below the corresponding calculus of static delimited continuations.

The syntactic correspondence makes it possible to directly compare (1) the calculi of dynamic and of static delimited continuations, (2) the extended CEK machine and the machine corresponding to the calculus of static delimited continuations and to the continuation-based evaluator, and (3) the evaluator corresponding to the extended CEK machine and the continuation-based evaluator. In other words, rather than having to relate heterogeneous artifacts such as a calculus with actual substitutions, an environment-based machine, and a continuation-based evaluator, we are now in position to directly compare two calculi, two abstract machines, and two continuation-based evaluators.

We address static delimited continuations in Section 5.5.1 and dynamic delimited continuations in Section 5.5.2. In both cases, we consider the left-to-right applicative-order reduction strategy and therefore left-to-right call by value.

5.5.1 The $\lambda\widehat{\rho}\mathcal{S}$-calculus

The standard λ-calculus is extended with the control operator shift (noted $\mathcal{S}$) that captures the current delimited continuation and with the control delimiter reset (noted $\langle\cdot\rangle$) that initializes the current delimited continuation.

5.5.1.1 The language of $\lambda\widehat{\rho}\mathcal{S}$

The abstract syntax of the language is as follows:

$$\begin{array}{llcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \mid \mathcal{S}\,t \mid \langle t\rangle \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \mid \mathcal{S}\,c \mid \langle c\rangle \mid \ulcorner C\urcorner \\
\text{(values)} & v & ::= & (\lambda t)[s] \mid \ulcorner C\urcorner \\
\text{(substitutions)} & s & ::= & \bullet \mid c\cdot s \\
\text{(contexts)} & C_1 & ::= & [\,] \mid C_1[v\,[\,]] \mid C_1[c\,[\,]] \mid C_1[\mathcal{S}[\,]] \\
\text{(meta-contexts)} & C_2 & ::= & \bullet \mid C_1\cdot C_2
\end{array}$$

For readability, we write $C_1 \cdot C_2$ rather than $C_2[\langle C_1[\,]\rangle]$.

The control operator $\mathcal{S}$ captures the current delimited context and replaces it with the empty context. The control delimiter $\langle\cdot\rangle$ initializes the current delimited context, saving the then-current one onto the meta-context. When a captured delimited context is resumed, the current delimited context is saved onto the meta-context. When the current delimited context completes, the previously saved one, if there is any, is resumed; otherwise, the computation terminates. This informal description paraphrases the definitional interpreter for shift and reset, which has two layers of control—a current delimited continuation (akin to a success continuation) and a meta-continuation (akin to a failure continuation), as arises naturally when one CPS-transforms a direct-style evaluator twice [67]. Elsewhere [28], we have defunctionalized this interpreter into an environment-based machine, which we present next.

5.5.1.2 The eval/apply/meta-apply abstract machine

The environment-based machine is in "eval/apply/meta-apply" form (to build on Peyton Jones's terminology [134]) because the continuation is defunctionalized into a context and the corresponding apply transition function, and the meta-continuation is defunctionalized into a meta-context (here a list of contexts) and the corresponding meta-apply transition function:

$$\begin{array}{rcl}
\langle i,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t',\, s',\, C_1,\, C_2\rangle \quad \text{if } s(i) = (t',\, s') \\
\langle \lambda t,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\, (\lambda t,\, s),\, C_2\rangle \\
\langle t_0\, t_1,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t_0,\, s,\, C_1[[\,]\,(t_1,\, s)],\, C_2\rangle \\
\langle \mathcal{S}\,t,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, C_1[\mathcal{S}[\,]],\, C_2\rangle \\
\langle \langle t\rangle,\, s,\, C_1,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, [\,],\, C_1\cdot C_2\rangle \\
\\
\langle [\,],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_2,\, v\rangle \\
\langle C_1[[\,]\,(t,\, s)],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, s,\, C_1[v\,[\,]],\, C_2\rangle \\
\langle C_1[(\lambda t,\, s)\,[\,]],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, v\cdot s,\, C_1,\, C_2\rangle \\
\langle C_1[\ulcorner C_1'\urcorner\,[\,]],\, v,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1',\, v,\, C_1\cdot C_2\rangle \\
\langle C_1[\mathcal{S}[\,]],\, (\lambda t,\, s),\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle t,\, \ulcorner C_1\urcorner\cdot s,\, [\,],\, C_2\rangle \\
\langle C_1[\mathcal{S}[\,]],\, \ulcorner C_1'\urcorner,\, C_2\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1',\, \ulcorner C_1\urcorner,\, [\,]\cdot C_2\rangle
\end{array}$$

$$\begin{array}{rcl}\langle \bullet,\, v\rangle & \Rightarrow_{\mathcal{S}} & v\\ \langle C_1 \cdot C_2,\, v\rangle & \Rightarrow_{\mathcal{S}} & \langle C_1,\, v,\, C_2\rangle\end{array}$$

We have observed that this machine is in the range of refocusing, transition compression, and closure unfolding for the following calculus $\lambda\widehat{\rho}\mathcal{S}$.

5.5.1.3 Notion of context-sensitive reduction

The $\lambda\widehat{\rho}\mathcal{S}$-calculus uses two layers of contexts: C_1 and C_2. A non-value closure is decomposed into a redex, a context C_1, and a meta-context C_2, and the notion of reduction is specified by the following rules:

$$\begin{array}{lrcl}
(\text{Var}) & \langle i[c_1 \cdots c_j],\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle c_i,\, C_1,\, C_2\rangle \quad \text{if } i \leq j\\
(\text{Beta}) & \langle ((\lambda t)[s])\, v,\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle t[v \cdot s],\, C_1,\, C_2\rangle\\
(\text{Beta}_C) & \langle \ulcorner C_1' \urcorner v\,,\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle \langle\!\langle C_1'[v] \rangle\!\rangle,\, C_1,\, C_2\rangle\\
(\text{Prop}) & \langle (t_0\, t_1)[s],\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle (t_0[s])\,(t_1[s]),\, C_1,\, C_2\rangle\\
(\text{Prop}_{\mathcal{S}}) & \langle (\mathcal{S}\, t)[s],\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle \mathcal{S}\,(t[s]),\, C_1,\, C_2\rangle\\
(\text{Prop}_{\langle\cdot\rangle}) & \langle \langle\!\langle t \rangle\!\rangle[s],\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle \langle\!\langle t[s] \rangle\!\rangle,\, C_1,\, C_2\rangle\\
(\mathcal{S}_\lambda) & \langle \mathcal{S}\,((\lambda t)[s]),\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle t[\ulcorner C_1 \urcorner \cdot s],\, [\,],\, C_2\rangle\\
(\mathcal{S}_C) & \langle \mathcal{S}\, \ulcorner C_1' \urcorner,\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle \ulcorner C_1' \urcorner\, \ulcorner C_1 \urcorner,\, [\,],\, C_2\rangle\\
(\text{Reset}) & \langle \langle\!\langle v \rangle\!\rangle,\, C_1,\, C_2\rangle & \rightarrow_{\mathcal{S}} & \langle v,\, C_1,\, C_2\rangle
\end{array}$$

Since none of the contractions depends on the meta-context, it is evident that the notion of reduction $\rightarrow_{\mathcal{S}}$ is compatible with meta-contexts, but it is not compatible with contexts, due to $\mathcal{S}_\lambda$ and $\mathcal{S}_C$. The $\langle\!\langle\cdot\rangle\!\rangle$ construct therefore delimits the parts of non-value closures in which context-sensitive reductions may occur, and partially restores the compatibility of reductions. In particular, $\langle \langle\!\langle t[s] \rangle\!\rangle,\, C_1,\, C_2\rangle$ is decomposed into $\langle t[s],\, [\,],\, C_1 \cdot C_2\rangle$ in the course of decomposition towards a context-sensitive redex.

5.5.1.4 Formal correspondence

Proposition 5.3. *For any term t in the $\lambda\widehat{\rho}\mathcal{S}$-calculus,*

$$t[\bullet] \rightarrow_{\mathcal{S}}^{*} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,],\, \bullet\rangle \Rightarrow_{\mathcal{S}}^{*} \overline{v}.$$

The $\lambda\widehat{\rho}\mathcal{S}$-calculus therefore directly corresponds to the abstract machine for shift and reset.

5.5.1.5 The CPS hierarchy

Iterating the CPS transformation on a direct-style evaluator for the λ-calculus gives rise to a family of CPS evaluators. At each iteration, one can add shift and reset to the new inner layer. The result forms a CPS hierarchy of static delimited continuations [67] which Filinski has shown to be able to represent layered monads [90]. Recently, Kameyama has proposed an axiomatization of the CPS hierarchy [118]. Elsewhere [28], we have studied its defunctionalized counterpart and the corresponding hierarchy of calculi.

5.5.2 The $\lambda\widehat{\rho}\mathcal{F}$-calculus

The standard λ-calculus is extended with the control operator $\mathcal{F}$ that captures a segment of the current context and with the control delimiter prompt (noted #) that initializes a new segment in the current context.

5.5.2.1 The language of $\lambda\widehat{\rho}\mathcal{F}$

The abstract syntax of the language is as follows:

$$\begin{array}{llcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \mid \mathcal{F}\,t \mid \#t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \mid \mathcal{F}\,c \mid \#\,c \mid \ulcorner C \urcorner \\
\text{(values)} & v & ::= & (\lambda t)[s] \mid \ulcorner C \urcorner \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \mid C[\mathcal{F}[\,]] \mid C[\#[\,]]
\end{array}$$

5.5.2.2 Notion of context-sensitive reduction

The control operator $\mathcal{F}$ captures a segment of the current context up to a mark. The control delimiter # sets a mark on the current context. When a captured segment is resumed, it is composed with the current context. For the rest, the notion of reduction is as usual:[2]

$$\begin{array}{lrcl}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C\rangle & \to_{\mathcal{F}} & \langle c_i,\ C\rangle\ \text{if}\ i \leq j \\
\text{(Beta)} & \langle((\lambda t)[s])\,v,\ C\rangle & \to_{\mathcal{F}} & \langle t[v\cdot s],\ C\rangle \\
\text{(Beta}_C) & \langle \ulcorner C' \urcorner v\ ,\ C\rangle & \to_{\mathcal{F}} & \langle C'[v],\ C\rangle \\
\text{(Prop)} & \langle(t_0\,t_1)[s],\ C\rangle & \to_{\mathcal{F}} & \langle(t_0[s])\,(t_1[s]),\ C\rangle \\
\text{(Prop}_{\mathcal{F}}) & \langle(\mathcal{F}\,t)[s],\ C\rangle & \to_{\mathcal{F}} & \langle\mathcal{F}\,(t[s]),\ C\rangle \\
\text{(Prop}_{\#}) & \langle(\#t)[s],\ C\rangle & \to_{\mathcal{F}} & \langle\#\,(t[s]),\ C\rangle \\
(\mathcal{F}_\lambda) & \langle\mathcal{F}\,((\lambda t)[s]),\ C[\#\,C']\rangle & \to_{\mathcal{F}} & \langle t[\ulcorner C' \urcorner\cdot s],\ C\rangle \\
 & & & \ \text{if}\ C'\ \text{contains no mark} \\
(\mathcal{F}_C) & \langle\mathcal{F}\,\ulcorner C'' \urcorner,\ C[\#\,C']\rangle & \to_{\mathcal{F}} & \langle\ulcorner C'' \urcorner\,\ulcorner C' \urcorner,\ C\rangle \\
 & & & \ \text{if}\ C'\ \text{contains no mark} \\
\text{(Prompt)} & \langle\#\,v,\ C\rangle & \to_{\mathcal{F}} & \langle v,\ C\rangle
\end{array}$$

Alternatively, we could specify the reduction rules using two layers of contexts, similarly to the $\lambda\widehat{\rho}\mathcal{S}$-calculus [28, 33, 34]. The difference between the two calculi would then be only in the rule (Beta$_C$):

$$\text{(Beta}_C)\quad \langle \ulcorner C_1' \urcorner v\ ,\ C_1,\ C_2\rangle \to_{\mathcal{F}} \langle C_1'[v],\ C_1,\ C_2\rangle$$

[2]The original version of $\mathcal{F}$ does not reduce its argument first, but its followers do. We do likewise here, for a more direct comparison with $\mathcal{S}$.

where there is no delimiter around $C'[v]$. As in the previous case of the $\lambda\widehat{\rho}\mathcal{S}$-calculus, such two-layered decomposition makes it evident that the contraction rules are compatible with the meta-context, since it is isolated by the use of a control delimiter.

5.5.2.3 The eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

$$\begin{array}{rcl}
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t',\, s',\, C\rangle \quad \text{if } s(i) = (t',\, s') \\
\langle \lambda t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, (\lambda t,\, s)\rangle \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle \\
\langle \mathcal{F}\, t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\mathcal{F}[\,]]\rangle \\
\langle \# t,\, s,\, C\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[\#[\,]]\rangle \\
\\
\langle C[[\,]\,(t,\, s)],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, s,\, C[v\,[\,]]\rangle \\
\langle C[(\lambda t,\, s)\,[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, v \cdot s,\, C\rangle \\
\langle C[\ulcorner C' \urcorner\,[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle C' \circ C,\, v\rangle \\
\langle C[\# C'[\mathcal{F}[\,]]],\, (\lambda t,\, s)\rangle & \Rightarrow_{\mathcal{F}} & \langle t,\, \ulcorner C' \urcorner \cdot s,\, C\rangle \\
 & & \text{where } C' \text{ contains no mark} \\
\langle C[\# C'[\mathcal{F}[\,]]],\, \ulcorner C'' \urcorner\rangle & \Rightarrow_{\mathcal{F}} & \langle C'' \circ C,\, \ulcorner C' \urcorner\rangle \\
 & & \text{where } C' \text{ contains no mark} \\
\langle C[\#[\,]],\, v\rangle & \Rightarrow_{\mathcal{F}} & \langle C,\, v\rangle
\end{array}$$

This environment-based machine coincides with Felleisen's extension of the CEK machine—an extension which was designed as such [83, Section 3].

5.5.2.4 Formal correspondence

Proposition 5.4. *For any term t in the $\lambda\widehat{\rho}\mathcal{F}$-calculus,*

$$t[\bullet] \rightarrow^*_{\mathcal{F}} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,]\rangle \Rightarrow^*_{\mathcal{F}} \overline{v}.$$

This proposition parallels Felleisen's second correspondence theorem [83, p. 186]. The $\lambda\widehat{\rho}\mathcal{F}$-calculus therefore directly corresponds to Felleisen's extension of the CEK machine.

5.5.2.5 A hierarchy of control delimiters

As described by Sitaram and Felleisen [164], one could have not one but several marks in the context and have control operators capture segments of the current context up to a particular mark. For these marks not to interfere in programming practice, they need to be organized hierarchically, forming a hierarchy of control delimiters [164, Section 7]. Alternatively, one could iterate Biernacki et al.'s dynamic CPS transformation [34] to give rise to a hierarchy of dynamic delimited continuations with a functional (CPS) counterpart. Except for the work of Gunter et al. [102] and more recently of Dybvig et al. [80], this area is little explored.

5.5.3 Conclusion

The syntactic correspondence has made it possible to exhibit the calculus corresponding to static delimited continuations as embodied in the functional idiom of success and failure continuations and more generally in the CPS hierarchy, and to show that (the explicit-substitutions version of) Felleisen's calculus of dynamic delimited continuations corresponds to his extension of the CEK machine [83]. Elsewhere, we present the abstract machine [28] and the evaluator [67] corresponding to static delimited continuations and an evaluator [34] corresponding to dynamic delimited continuations. We are now in position to compare them pointwise.

From a calculus point of view, it seems to us that one is better off with layered contexts because it is immediately obvious whether a notion of reduction is compatible with them (see Section 5.5.1.3); a context containing marks is less easy to deal with. Otherwise, the difference between static and dynamic delimited continuations is tiny (see Section 5.5.2.2), and located in the rule (Beta$_C$).

From a machine point of view, separating between the current delimited context and the other ones is also simpler, as it avoids linear searches, copies, and concatenations (in this respect, efficient implementations, e.g., with a display, in effect separate between the current delimited context and the other ones).

From the point of view of CPS, the abstract machine for dynamic delimited continuations is not in defunctionalized form whereas the abstract machine for static delimited continuations is (and corresponds to an evaluator in CPS). Conversely, defunctionalizing a CPS evaluator provides design guidelines, whereas without CPS, one is on one's own, and locally plausible choices may have unforeseen global consequences which are then taken as the norm. Two cases in point: (1) in Lisp, it was locally plausible to push both formal and actual parameters at function-call time, and to pop them at return time, but this led to dynamic scope since variable lookup then follows the dynamic link; and (2) here, it was locally plausible to concatenate a control-stack segment to the current control stack ("From this, we learn that an empty context adds no information." [88, p. 58]), but this led to dynamic delimited continuations since capturing a segment of a concatenated context then gives access to beyond the concatenation point. Granted, a degree of dynamism makes it possible to write compact programs (e.g., a breadth-first traversal without a data-queue accumulator *and* in direct style [35]), but it is very difficult to reason about them and they are not necessarily more efficient.

From the point of view of expressiveness, for example, in Lisp, one can simulate the static scope of Scheme by making each lambda-abstraction a "funarg" and in Scheme, one can simulate the dynamic scope of Lisp by threading an environment of fluid variables in a state-monad fashion. Similarly, static delimited continuations can be simulated using dynamic ones by delimiting the extent of each captured continuation [33], and dynamic delimited continuations can be simulated using static ones by threading a trail of contexts in a state-monad fashion [34, 80, 122, 162]. As to which should be used by default, the question then reduces to which behavior is the norm and which should be simulated if it is needed.

In summary, the calculi, the abstract machines, and the evaluators all differ. In one approach, continuations are composed by dynamically concatenating their representations [88] and in the other, continuations are statically composed through a meta-continuation. These differences result from distinct designs: Felleisen and his colleagues started from a calculus and wanted to go "beyond continuations" [86] and therefore beyond CPS whereas Danvy and Filinski were aiming at a CPS account of delimited control.

5.6 A calculus of closures with input/output ($\lambda\widehat{\rho}_{\mathrm{i/o}}$)

In this and the two subsequent sections we show calculi enriched with features whose implementations via abstract machines usually introduce a state, i.e., a global component of a machine configuration that can be accessed and updated at any time by a currently evaluated subclosure. We show the corresponding calculi of closures extended with a suitable syntactic entity to model state.

First, we present a simple calculus with primitives for modelling finite input and output, where closures are equipped with two additional components: an input channel (a finite list I), and an output channel (a finite list O). Since we provide explicit syntactic characterization of input and output, it is possible to give the calculus a standard reduction semantics instead of a labeled transition system with read and write actions expressed as annotations on transitions. Such a specification allows us to apply the refocusing technique and mechanically derive an abstract machine for this calculus.

5.6.1 The language of $\lambda\widehat{\rho}_{\mathrm{i/o}}$

The abstract syntax of the language is specified as follows:

$$\begin{array}{llcl}
\text{(terms)} & t & ::= & \ell \mid i \mid \lambda t \mid t\,t \mid \mathtt{in}\,t \mid \mathtt{out}\,t \mid t;t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \mid \mathtt{in}\,c \mid \mathtt{out}\,c \mid c;c \\
\text{(values)} & v & ::= & (\lambda t)[s] \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \mid \\
 & & & C[[\,];c] \mid C[\mathtt{in}\,[\,]] \mid C[\mathtt{out}\,[\,]] \\
\text{(input)} & I & ::= & \bullet \mid \ell :: I \\
\text{(output)} & O & ::= & \bullet \mid \ell :: O \\
\text{(i/o closures)} & \widetilde{c} & ::= & c[I, O] \\
\text{(i/o values)} & \widetilde{v} & ::= & v[I, O] \\
\text{(i/o contexts)} & \widetilde{C} & ::= & C[I, O]
\end{array}$$

The set of terms is extended with literals (noted ℓ), and with three new operators: `in` for reading in a literal, `out` for writing out a literal, and a binary operator $\cdot;\cdot$ to sequentialize computation. The new i/o closures are built on top of the usual closures (terms with explicit substitutions, and their compositions) equipped with two lists of literals: an input channel denoted I, and an output channel denoted O.

5.6.2 Notion of context-sensitive reduction

Input/output channels are global for the entire computation (contrary to substitutions, propagated locally to each subterm), which is obtained by ensuring that at each step only one subclosure can access and modify these channels (hence the restricted forms of closures). The reduction rules are performed only on i/o closures, making the calculus deterministic.

(Var)	$\langle i[c_1 \cdots c_j],\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle c_i,\ \widetilde{C}\rangle$
(Beta)	$\langle((\lambda t)[s])\ v,\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle t[v \cdot s],\ \widetilde{C}\rangle$
(Prop)	$\langle(t_0\ t_1)[s],\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle t_0[s]\ t_1[s],\ \widetilde{C}\rangle$
(Prop$_{seq}$)	$\langle(t_0; t_1)[s],\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle t_0[s]; t_1[s],\ \widetilde{C}\rangle$
(Prop$_{in}$)	$\langle(\mathtt{in}\, t)[s],\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle(\mathtt{in}\, t[s]),\ \widetilde{C}\rangle$
(Prop$_{out}$)	$\langle(\mathtt{out}\, t)[s],\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle(\mathtt{out}\, t[s]),\ \widetilde{C}\rangle$
(Seq)	$\langle v; c,\ \widetilde{C}\rangle$	$\to_{\mathrm{i/o}}$	$\langle c,\ \widetilde{C}\rangle$
(In)	$\langle(\mathtt{in}\,(\lambda t)[s]),\ C[\ell :: I,\ O]\rangle$	$\to_{\mathrm{i/o}}$	$\langle t[\ell[\bullet] \cdot s],\ C[I,\ O]\rangle$
(Out)	$\langle(\mathtt{out}\, \ell[s]),\ C[I,\ O]\rangle$	$\to_{\mathrm{i/o}}$	$\langle \ell[s],\ C[I,\ \ell :: O]\rangle$

The need for context-sensitive reductions arises in the last two reductions that manipulate global input and output channels, respectively. The rule (In) reads a literal from an input channel and stores it in the substitution, and the (Out) rule prints a literal to the output channel. Note that the reduction rules are compatible with contexts C, and therefore they facilitate local reasoning about programs in the following sense:

Proposition 5.5. *For all closures c_1, c_2 such that $c_1[I,\ O] =_{\mathrm{i/o}} c_2[I,\ O]$, and for every context C,*

$$(C[c_1])[I,\ O] =_{\mathrm{i/o}} (C[c_2])[I,\ O].$$

It is possible to reformulate the calculus in the usual context-insensitive form, by propagating the input/output channels down to each closure—similarly to the way a substitution is propagated. The final abstract machine obtained by refocusing is the same as the one presented below.

5.6.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following store-based machine:

$$
\begin{array}{rcl}
\langle \ell,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\ (\ell,\ s),\ I,\ O\rangle \\
\langle \lambda t,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\ (\lambda t,\ s),\ I,\ O\rangle \\
\langle i,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ s',\ C,\ I,\ O\rangle \quad \text{if } s(i) = (t,\ s') \\
\langle t_0\, t_1,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t_0,\ s,\ C[[\,]\,(t_1,\ s)],\ I,\ O\rangle \\
\langle t_0; t_1,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t_0,\ s,\ C[[\,]; (t_1,\ s)],\ I,\ O\rangle \\
\langle \mathtt{in}\, t,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ s,\ C[\mathtt{in}\,[\,]],\ I,\ O\rangle \\
\langle \mathtt{out}\, t,\ s,\ C,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ s,\ C[\mathtt{out}\,[\,]],\ I,\ O\rangle \\
\\
\langle [\,],\ v,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & (v,\ I,\ O) \\
\langle C[[\,]\,(t,\ s)],\ v,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ s,\ C[v\,[\,]],\ I,\ O\rangle \\
\langle C[(\lambda t,\ s)\,[\,]],\ v,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ v \cdot s,\ C,\ I,\ O\rangle \\
\langle C[[\,]; (t,\ s)],\ v,\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle t,\ s,\ C,\ I,\ O\rangle \\
\langle C[\mathtt{in}\,[\,]],\ (\lambda t,\ s),\ \ell :: I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\ t,\ \ell[\bullet] \cdot s,\ I,\ O\rangle \\
\langle C[\mathtt{out}\,[\,]],\ (\ell,\ s),\ I,\ O\rangle & \Rightarrow_{\mathrm{i/o}} & \langle C,\ (\ell,\ s),\ I,\ \ell :: O\rangle
\end{array}
$$

5.6.4 Formal correspondence

Proposition 5.6. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{i/o}}$-calculus,*

$$t[\bullet][I,\ \bullet] \rightarrow^{*}_{i/o} v[I',\ O] \text{ if and only if } \langle t,\ \bullet,\ [\,],\ I,\ \bullet\rangle \Rightarrow^{*}_{i/o} (\overline{v},\ I',\ O).$$

5.7 Stack inspection

This section addresses Fournet and Gordon's λ_{sec}-calculus, which formalizes security enforcement by stack inspection [95]. We first present a calculus of closures built on top of the λ_{sec}-calculus, and we construct the corresponding environment-based machine. This machine is a storeless version of the fg machine presented by Clements and Felleisen [43, Figure 1]. (We consider the issue of store-based machines in Section 5.8.) This machine is not properly tail-recursive, and so Clements and Felleisen presented another machine—the cm machine—which does implement stack inspection in a properly tail-recursive manner [43, Figure 2]. The cm machine builds on Clinger's formalization of proper tail-recursion (see Section 5.8) and it is therefore store-based; we considered its storeless version here, and we present the corresponding calculus of closures. We show that the tail-optimization of the cm machine is reflected by a non-standard plug function. Finally, we turn to the unzipped version of the cm machine [5] and we present the corresponding state-based calculus of closures.

5.7.1 The $\lambda\widehat{\rho}_{\text{sec}}$-calculus

5.7.1.1 The language of $\lambda\widehat{\rho}_{\text{sec}}$

$$\begin{array}{llcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\,t \mid \texttt{grant}\,R\,\texttt{in}\,t \mid \\
 & & & \texttt{test}\,R\,\texttt{then}\,t\,\texttt{else}\,t \mid R[t] \mid \texttt{fail} \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \mid \texttt{grant}\,R\,\texttt{in}\,c \mid \texttt{test}\,R\,\texttt{then}\,c\,\texttt{else}\,c \mid \\
 & & & R[c] \\
\text{(values)} & v & ::= & (\lambda t)[s] \mid \texttt{fail} \\
\text{(substitutions)} & s & ::= & \bullet \mid c \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \mid \\
 & & & C[\texttt{grant}\,R\,\texttt{in}\,[\,]] \mid C[R[[\,]]] \\
\text{(permissions)} & R & \subseteq & \mathcal{P}
\end{array}$$

The set of terms consists of λ-terms and four constructs for handling different levels of security specified in a set $\mathcal{P}$: $\texttt{grant}\,R\,\texttt{in}\,t$ grants the permissions R to t; $\texttt{test}\,R\,\texttt{then}\,t_0\,\texttt{else}\,t_1$ proceeds to evaluate t_0 if permissions R are available, and otherwise t_1; a frame $R[t]$ restricts the permissions of t to R; and finally, `fail` aborts the computation.

5.7.1.2 Notion of context-sensitive reduction

Given the predicate $\mathcal{OK}_{\text{sec}}(R, C)$ checking whether the permissions R are available within the context C,

$$\frac{}{\mathcal{OK}_{\text{sec}}(\emptyset, C)} \qquad \frac{}{\mathcal{OK}_{\text{sec}}(R, [\,])}$$

$$\frac{\mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[[\,]\,c])} \qquad \frac{\mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[v\,[\,]])}$$

$$\frac{R \subset R' \quad \mathcal{OK}_{\text{sec}}(R, C)}{\mathcal{OK}_{\text{sec}}(R, C[R'[[\,]]])} \qquad \frac{\mathcal{OK}_{\text{sec}}(R \setminus R', C)}{\mathcal{OK}_{\text{sec}}(R, C[\texttt{grant}\,R'\,\texttt{in}\,[\,]])}$$

the notion of reduction is given by the following set of rules:

$$\begin{array}{lrcll}
\text{(Var)} & \langle i[c_1 \cdots c_j],\ C\rangle & \rightarrow_{\text{sec}} & \langle c_i,\ C\rangle & \text{if } i \leq j \\
\text{(Beta)} & \langle ((\lambda t)[s])\,v,\ C\rangle & \rightarrow_{\text{sec}} & \langle t[v \cdot s],\ C\rangle & \\
\text{(Prop)} & \langle (t_0\,t_1)[s],\ C\rangle & \rightarrow_{\text{sec}} & \langle (t_0[s])\,(t_1[s]),\ C\rangle & \\
\text{(Prop}_G) & \langle (\texttt{grant}\,R\,\texttt{in}\,t)[s],\ C\rangle & \rightarrow_{\text{sec}} & \langle \texttt{grant}\,R\,\texttt{in}\,t[s],\ C\rangle & \\
\text{(Prop}_F) & \langle (R[t])[s],\ C\rangle & \rightarrow_{\text{sec}} & \langle R[t[s]],\ C\rangle & \\
\text{(Prop}_T) & \langle (\texttt{test}\,R\,\texttt{then}\,t_0\,\texttt{else}\,t_1)[s],\ C\rangle & \rightarrow_{\text{sec}} & \langle \texttt{test}\,R\,\texttt{then}\,t_0[s]\,\texttt{else}\,t_1[s],\ C\rangle & \\
\text{(Frame)} & \langle R[v],\ C\rangle & \rightarrow_{\text{sec}} & \langle v,\ C\rangle & \\
\text{(Grant)} & \langle \texttt{grant}\,R\,\texttt{in}\,v,\ C\rangle & \rightarrow_{\text{sec}} & \langle v,\ C\rangle & \\
\text{(Test}_1) & \langle \texttt{test}\,R\,\texttt{then}\,c_1\,\texttt{else}\,c_2,\ C\rangle & \rightarrow_{\text{sec}} & \langle c_1,\ C\rangle & \text{if } \mathcal{OK}_{\text{sec}}(R, C) \\
\text{(Test}_2) & \langle \texttt{test}\,R\,\texttt{then}\,c_1\,\texttt{else}\,c_2,\ C\rangle & \rightarrow_{\text{sec}} & \langle c_2,\ C\rangle & \text{otherwise} \\
\text{(Fail)} & \langle \texttt{fail}[s],\ C\rangle & \rightarrow_{\text{sec}} & \langle \texttt{fail},\ [\,]\rangle &
\end{array}$$

The only context-sensitive rules are (Test$_1$) and (Test$_2$), which perform a reduction step after inspecting the entire context C, and (Fail) which aborts the computation.

5.7.1.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following environment-based machine:

$$\begin{array}{rcll}
\langle i,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\, s(i)\rangle & \\
\langle \lambda t,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\, (\lambda t,\, s)\rangle & \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)]\rangle & \\
\langle \mathtt{grant}\, R\, \mathtt{in}\, t,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle t,\, s,\, C[\mathtt{grant}\, R\, \mathtt{in}\, [\,]]\rangle & \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle t_0,\, s,\, C\rangle & \text{if } \mathcal{OK}_{\mathrm{sec}}(R, C) \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle t_1,\, s,\, C\rangle & \text{otherwise} \\
\langle \mathtt{fail},\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \mathtt{fail} & \\
\langle R[t],\, s,\, C\rangle & \Rightarrow_{\mathrm{sec}} & \langle t,\, s,\, C[R[[\,]]]\rangle & \\
\\
\langle [\,],\, v\rangle & \Rightarrow_{\mathrm{sec}} & v & \\
\langle C[[\,]\,(t,\, s)],\, v\rangle & \Rightarrow_{\mathrm{sec}} & \langle t,\, s,\, C[v\,[\,]]\rangle & \\
\langle C[(\lambda t,\, s)\,[\,]],\, v\rangle & \Rightarrow_{\mathrm{sec}} & \langle t,\, v\cdot s,\, C\rangle & \\
\langle C[\mathtt{grant}\, R\, \mathtt{in}\, [\,]],\, v\rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\, v\rangle & \\
\langle C[R[[\,]]],\, v\rangle & \Rightarrow_{\mathrm{sec}} & \langle C,\, v\rangle &
\end{array}$$

This machine is a storeless version of Clements and Felleisen's fg machine [43, Figure 1].

5.7.1.4 Formal correspondence

Proposition 5.7. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus,*

$$t[\bullet] \rightarrow^*_{\mathrm{sec}} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,]\rangle \Rightarrow^*_{\mathrm{sec}} \overline{v}.$$

The $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus therefore directly corresponds to the storeless version of the fg machine.

5.7.2 Properly tail-recursive stack inspection

On the ground that the fg machine is not properly tail-recursive, Clements and Felleisen presented a new, properly tail-recursive, machine—the cm machine [43, Figure 2]—thereby debunking the folklore that stack inspection is incompatible with proper tail recursion. Below, we consider the storeless version of the cm machine and we present the underlying calculus of closures.

5.7.2.1 The storeless cm machine

The cm machine operates on a λ_{sec}-term, an environment, and an evaluation context enriched with updatable permission tables (noted m below):

$$\text{(stack frames)} \quad C \quad ::= \quad m[\,] \mid C[[\,](c, m)] \mid C[(v, m)[\,]]$$

A permission table is a partial function with a finite domain from a set of permissions $\mathcal{P}$ to the set $\{\bot = \text{not granted}, \top = \text{granted}\}$. A permission table with the empty domain is denoted ε.

Given the predicate $\mathcal{OK}^{\text{cm}}_{\text{sec}}(R, C)$,

$$\frac{}{\mathcal{OK}^{\text{cm}}_{\text{sec}}(\emptyset, C)} \qquad \frac{R \cap m^{-1}(\bot) = \emptyset}{\mathcal{OK}^{\text{cm}}_{\text{sec}}(R, m[\,])}$$

$$\frac{R \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\text{cm}}_{\text{sec}}(R \setminus m^{-1}(\top), C)}{\mathcal{OK}^{\text{cm}}_{\text{sec}}(R, C[[\,](c, m)])}$$

$$\frac{R \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\text{cm}}_{\text{sec}}(R \setminus m^{-1}(\top), C)}{\mathcal{OK}^{\text{cm}}_{\text{sec}}(R, C[(v, m)[\,]])}$$

the transitions of the storeless cm machine read as follows:

$$\begin{array}{rcl}
\langle i,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle C,\, s(i)\rangle \\
\langle \lambda t,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle C,\, (\lambda t,\, s)\rangle \\
\langle t_0\, t_1,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t_0,\, s,\, C[[\,]((t_1,\, s), \varepsilon)]\rangle \\
\langle \mathtt{grant}\, R\, \mathtt{in}\, t,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t,\, s,\, C[R \mapsto \top]\rangle \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t_0,\, s,\, C\rangle \text{ if } \mathcal{OK}^{\text{cm}}_{\text{sec}}(R, C) \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t_1,\, s,\, C\rangle \text{ otherwise} \\
\langle \mathtt{fail},\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \mathtt{fail} \\
\langle R[t],\, s,\, C\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t,\, s,\, C[\overline{R} \mapsto \bot]\rangle \\
\\
\langle m[\,],\, v\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & v \\
\langle C[[\,]((t,\, s), m)],\, v\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t,\, s,\, C[(v, \varepsilon)[\,]]\rangle \\
\langle C[((\lambda t,\, s), m)[\,]],\, v\rangle & \Rightarrow^{\text{cm}}_{\text{sec}} & \langle t,\, v \cdot s,\, C\rangle
\end{array}$$

where $\overline{R} = \mathcal{P} \setminus R$ and $C[R \mapsto v]$ is a modification of the permission table in the context C obtained by granting or restricting the permissions R, depending on v.

The following proposition states the equivalence of the fg machine and the cm machine with respect to the values they compute:

Proposition 5.8. *For any term t in the $\lambda\widehat{\rho}_{\text{sec}}$-calculus,*

$$\langle t,\, \bullet,\, [\,]\rangle \Rightarrow^{*}_{\text{sec}} v \quad \textit{if and only if} \quad \langle t,\, \bullet,\, [\,]\rangle\, (\Rightarrow^{\text{cm}}_{\text{sec}})^{*}\, v.$$

Moreover, it can be shown that each step of the fg machine is simulated by at most one step of the cm machine [43]. At the level of the calculus, this is reflected in the fact that the reduction semantics implemented by the cm machine has fewer reductions than $\lambda\widehat{\rho}_{\text{sec}}$.

5.7.2.2 The underlying calculus $\lambda\widehat{\rho}^{\mathrm{cm}}_{\mathrm{sec}}$

The calculus corresponding to the storeless cm machine is very close to the $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus. The grammar of terms, closures and substitutions is the same, but the reduction contexts (which correspond to the stack frames in the machine) contain permission tables. Consequently, the functions plug and decompose are defined in a non-standard way:

$$\begin{aligned}
\mathsf{plug}\,(c, m[\,]) &= \mathsf{build}\,(m, c)\\
\mathsf{plug}\,(c_0, C[[\,](c_1, m)]) &= \mathsf{plug}\,(\mathsf{build}\,(m, c_0\, c_1), C)\\
\mathsf{plug}\,(c, C[(v, m)[\,]]) &= \mathsf{plug}\,(\mathsf{build}\,(m, v\, c), C)
\end{aligned}$$

where the auxiliary function build conservatively constructs a closure based on the permission table of the reduction context:

$$\begin{aligned}
\mathsf{build}_G\,(m, c) &= \begin{cases} c & \text{if } m^{-1}(\top) = \emptyset\\ \mathtt{grant}\, m^{-1}(\top)\, \mathtt{in}\, c & \text{otherwise}\end{cases}\\
\mathsf{build}_F\,(m, c) &= \begin{cases} c & \text{if } m^{-1}(\bot) = \emptyset\\ \overline{m^{-1}(\bot)}[c] & \text{otherwise}\end{cases}\\
\mathsf{build}\,(m, c) &= \mathsf{build}_F\,(m, \mathsf{build}_G\,(m, c))
\end{aligned}$$

Any closure that is not already a value or a potential redex, can be further decomposed as follows:

$$\begin{aligned}
\mathsf{decompose}\,(c_0\, c_1, C) &= \mathsf{decompose}\,(c_0, C[[\,](c_1, \varepsilon)])\\
\mathsf{decompose}\,(\mathtt{grant}\, R\, \mathtt{in}\, c, C) &= \mathsf{decompose}\,(c, C[R \mapsto \top])\\
\mathsf{decompose}\,(R[c], C) &= \mathsf{decompose}\,(c, C[\overline{R} \mapsto \bot])\\
\mathsf{decompose}\,(v, C[[\,](c, m)]) &= \mathsf{decompose}\,(c, C[(v, \varepsilon)[\,]])
\end{aligned}$$

The notion of reduction includes most rules of the $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus, except for (Frame) and (Grant).

From a calculus standpoint, Clements and Felleisen therefore obtained proper tail recursion by changing the computational model (witness the change from $\mathcal{OK}_{\mathrm{sec}}$ to $\mathcal{OK}^{\mathrm{cm}}_{\mathrm{sec}}$) and by simplifying the reduction rules and modifying the compatibility rules.

5.7.3 State-based properly tail-recursive stack inspection

On the observation that the stack of the cm machine can be unzipped into the usual control stack of the CEK machine and a state-like list of permission tables, Ager et al. have presented an unzipped version of the cm machine (characterizing properly tail-recursive stack inspection as a monad in passing) [5]. We first present this machine, and then the corresponding calculus of closures.

5.7.3.1 The unzipped storeless cm machine

The unzipped cm machine operates on a λ_{sec}-term, an environment, and an ordinary evaluation context. In addition, the machine has a read-write security register m holding the current permission table and a read-only security register ms holding a list of outer permission tables. Given the predicate

$\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, ms)$,

$$\frac{}{\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(\emptyset, m, ms)} \qquad \frac{R \cap m^{-1}(\bot) = \emptyset}{\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, \bullet)}$$

$$\frac{R \cap m^{-1}(\bot) = \emptyset \quad \mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R \setminus m^{-1}(\top), m', ms)}{\mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, m' \cdot ms)}$$

the transitions of the unzipped storeless cm machine read as follows:

$$\begin{array}{rcl}
\langle i,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle C,\ s(i),\ ms\rangle \\
\langle \lambda t,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle C,\ (\lambda t,\ s),\ ms\rangle \\
\langle t_0\, t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t_0,\ s,\ \varepsilon, m \cdot ms,\ C[[\,]\,(t_1,\ s)]\rangle \\
\langle \mathtt{grant}\, R\, \mathtt{in}\, t,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t,\ s,\ m[R \mapsto \top], ms,\ C\rangle \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t_0,\ s,\ m, ms,\ C\rangle \\
 & & \text{if } \mathcal{OK}^{\mathrm{ucm}}_{\mathrm{sec}}(R, m, ms) \\
\langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1,\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t_1,\ s,\ m, ms,\ C\rangle \\
 & & \text{otherwise} \\
\langle R[t],\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t,\ s,\ m[\overline{R} \mapsto \bot], ms,\ C\rangle \\
\langle \mathtt{fail},\ s,\ m, ms,\ C\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \mathtt{fail} \\
 & & \\
\langle [\,],\ v,\ \bullet\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & v \\
\langle C[[\,]\,(t,\ s)],\ v,\ ms\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t,\ s,\ \varepsilon, ms,\ C[v\,[\,]]\rangle \\
\langle C[(\lambda t,\ s)\,[\,]],\ v,\ m \cdot ms\rangle & \Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}} & \langle t,\ v \cdot s,\ m, ms,\ C\rangle
\end{array}$$

The following proposition states the equivalence of the cm machine and the unzipped cm machine:

Proposition 5.9. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{sec}}$-calculus,*

$$\langle t,\ \bullet,\ [\,]\rangle\, (\Rightarrow^{\mathrm{cm}}_{\mathrm{sec}})^*\, v \quad \textit{if and only if} \quad \langle t,\ \bullet,\ \varepsilon, \bullet,\ [\,]\rangle\, (\Rightarrow^{\mathrm{ucm}}_{\mathrm{sec}})^*\, v.$$

Moreover, it can be shown that each step of the cm machine is simulated by one step of the unzipped cm machine.

5.7.3.2 The language of $\lambda\widehat{\rho}^{\mathrm{ucm}}_{\mathrm{sec}}$

(terms)	t	$::=$	$i \mid \lambda t \mid t\, t \mid \mathtt{grant}\, R\, \mathtt{in}\, t \mid$
			$\mathtt{test}\, R\, \mathtt{then}\, t\, \mathtt{else}\, t \mid R[t] \mid \mathtt{fail}$
(closures)	c	$::=$	$t[s]$
(values)	v	$::=$	$(\lambda t)[s] \mid \mathtt{fail}$
(substitutions)	s	$::=$	$\bullet \mid c \cdot s$
(reduction contexts)	C	$::=$	$[\,] \mid C[[\,]\, c] \mid C[v\, [\,]]$
(annotated closures)	$\widetilde{c}$	$::=$	$c[m, ms] \mid c\, \widetilde{c} \mid \widetilde{c}\, c \mid \mathtt{fail}$
(annotated values)	$\widetilde{v}$	$::=$	$v[m, ms] \mid \mathtt{fail}$

5.7.3.3 Notion of context-sensitive reduction

The notion of reduction is specified by the rules below. Compared to the rules of Section 5.7.1.2, the current permission table and the list of outer permission tables are propagated locally to each closure being evaluated. When a value is consumed, the current permission table is discarded.

$$
\begin{array}{llcl}
\text{(Prop)} & \langle (t_0\, t_1)[s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle (t_0[s][\varepsilon, m \cdot ms])\ (t_1[s]),\ C\rangle \\
\text{(Var)} & \langle i[c_1 \cdots c_j][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle c_i[m, ms],\ C\rangle\ \text{ if } i \leq j \\
\text{(Test}_1\text{)} & \langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1[s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle t_0[s][m, ms],\ C\rangle \\
 & & & \text{if } \mathcal{OK}_{\text{sec}}^{\text{ucm}}(R, m, ms) \\
\text{(Test}_2\text{)} & \langle \mathtt{test}\, R\, \mathtt{then}\, t_0\, \mathtt{else}\, t_1[s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle t_1[s][m, ms],\ C\rangle \\
 & & & \text{otherwise} \\
\text{(Fail)} & \langle \mathtt{fail}[s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle \mathtt{fail},\ [\,]\rangle \\
\text{(Switch)} & \langle (v[m, ms])\, c,\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle v\ (c[\varepsilon, ms]),\ C\rangle \\
\text{(Beta)} & \langle ((\lambda t)[s])\ (v[m, m' \cdot ms]),\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle t[v \cdot s][m', ms],\ C\rangle \\
\text{(Frame)} & \langle R[t][s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle t[s][m[\overline{R} \mapsto \bot], ms],\ C\rangle \\
\text{(Grant)} & \langle \mathtt{grant}\, R\, \mathtt{in}\, t[s][m, ms],\ C\rangle & \rightarrow_{\text{sec}}^{\text{ucm}} & \langle t[s][m[R \mapsto \top], ms],\ C\rangle
\end{array}
$$

A new reduction rule (Switch) is now necessary to go from one evaluated subclosure to a subclosure to evaluate. The (Beta) rule doubles up with discarding the permission table of the actual parameter. The (Frame) and (Grant) rules embody the state counterpart of Clements and Felleisen's design to enable proper tail recursion.

5.7.3.4 Formal correspondence

Proposition 5.10. *For any term t in the λ_{sec}-calculus,*

$$t[\bullet](\rightarrow_{\text{sec}}^{\text{ucm}})^* v \quad \textit{if and only if} \quad \langle t,\ \bullet,\ \varepsilon, \bullet,\ [\,]\rangle\ (\Rightarrow_{\text{sec}}^{\text{ucm}})^*\ \overline{v}.$$

5.7.4 Conclusion

We have presented three corresponding calculi of closures and machines for stack inspection, showing first how the storeless fg machine reflects the $\lambda\widehat{\rho}_{\text{sec}}$-calculus, second, how the $\lambda\widehat{\rho}_{\text{sec}}^{\text{cm}}$-calculus reflects the storeless cm machine, and third, how the $\lambda\widehat{\rho}_{\text{sec}}^{\text{ucm}}$-calculus reflects the unzipped storeless cm machine. In doing so, we have provided a calculus account of machine design and optimization.

5.8 A calculus for proper tail-recursion

At PLDI'98 [47], Clinger presented a properly tail-recursive semantics for Scheme in the form of a store-based abstract machine. This machine models the memory-allocation behavior of function calls in Scheme and Clinger used it to specify in which sense an implementation should not run out of memory when processing a tail-recursive program (such as a program in CPS).

We first present a similar machine for the λ-calculus with left-to-right call-by-value evaluation and assignments. This machine is in the range of refocusing, transition compression, and closure unfolding, and so we next present the corresponding store-based calculus, $\lambda\widehat{\rho}_{\text{ptr}}$.

5.8.1 A simplified version of Clinger's abstract machine

Our simplified version is an eval/apply machine with an environment and a store:

$$\begin{array}{rcl}
\langle x,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle C,\, \sigma(s(x)),\, \sigma\rangle \\
\langle \lambda x.t,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle C,\, (\lambda x.t,\, s),\, \sigma\rangle \\
\langle t_0\, t_1,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle t_0,\, s,\, C[[\,]\,(t_1,\, s)],\, \sigma\rangle \\
\langle x := t,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle t,\, s,\, C[\texttt{upd}(s(x), [\,])],\, \sigma\rangle \\
\\
\langle [\,],\, v,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & (v,\, \sigma) \\
\langle C[[\,]\,(t,\, s)],\, v,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle t,\, s,\, C[v\,[\,]],\, \sigma\rangle \\
\langle C[(\lambda x.t,\, s)\,[\,]],\, v,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle t,\, (x,\, \ell)\cdot s,\, C,\, \sigma[\ell \mapsto v]\rangle \\
 & & \text{if } \ell \text{ does not occur within } s, C, v, \sigma \\
\langle C[\texttt{upd}(\ell, [\,])],\, v,\, \sigma\rangle & \Rightarrow_{\text{ptr}} & \langle C,\, \sigma(\ell),\, \sigma[\ell \mapsto v]\rangle
\end{array}$$

A location ℓ ranges over an unspecified set of locations. A store σ is a finite mapping from locations to value closures. Denotable values are locations.

Clinger's machine also has a garbage-collection rule [47, Figure 5 and Section 3], but for simplicity we ignore it here.

5.8.2 The language of $\lambda\widehat{\rho}_{\text{ptr}}$

The abstract syntax of the language is as follows:

$$\begin{array}{llcl}
\text{(terms)} & t & ::= & x \mid \lambda x.t \mid t\,t \mid x := t \\
\text{(closures)} & c & ::= & t[s] \mid c\,c \\
\text{(values)} & v & ::= & (\lambda x.t)[s] \\
\text{(substitutions)} & s & ::= & \bullet \mid (x,\, \ell)\cdot s \\
\text{(red. contexts)} & C & ::= & [\,] \mid C[[\,]\,c] \mid C[v\,[\,]] \mid C[\texttt{upd}(\ell, [\,])] \\
\text{(store)} & \sigma & ::= & \bullet \mid \sigma[\ell \mapsto v] \\
\text{(store closures)} & \widetilde{c} & ::= & c[\sigma] \\
\text{(store values)} & \widetilde{v} & ::= & v[\sigma] \\
\text{(store contexts)} & \widetilde{C} & ::= & C[\sigma]
\end{array}$$

5.8.3 Notion of context-sensitive reduction

In the rules below, (Var) dereferences the store; (Beta) allocates a fresh location, and extends both the substitution and the store with it; (Prop) is context-insensitive and therefore essentially as in the $\lambda\widehat{\rho}$-calculus; and (Upd) updates the store.

$$\begin{array}{llll}
\text{(Var)} & \langle x[s],\ C[\sigma]\rangle & \rightarrow_{\text{ptr}} & \langle \sigma(\ell),\ C[\sigma]\rangle \quad \text{if } s(x) = \ell \\
\text{(Beta)} & \langle ((\lambda x.t)[s])\, v,\ C[\sigma]\rangle & \rightarrow_{\text{ptr}} & \langle t[(x,\ \ell)\cdot s],\ C[\sigma[\ell \mapsto v]]\rangle \\
 & & & \text{if } \ell \text{ does not occur within} s, v, C, \sigma \\
\text{(Prop)} & \langle (t_0\, t_1)[s],\ C[\sigma]\rangle & \rightarrow_{\text{ptr}} & \langle (t_0[s])\,(t_1[s]),\ C[\sigma]\rangle \\
\text{(Mark)} & \langle (x := t)[s],\ C[\sigma]\rangle & \rightarrow_{\text{ptr}} & \langle t[s],\ C[\texttt{upd}(s(x), [\,])][\sigma]\rangle \\
\text{(Upd)} & \langle \texttt{upd}(\ell, v),\ C[\sigma]\rangle & \rightarrow_{\text{ptr}} & \langle \sigma(\ell),\ C[\sigma[\ell \mapsto v]]\rangle
\end{array}$$

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the abstract machine of Section 5.8.1.

5.8.4 Formal correspondence

Proposition 5.11. *For any term t in the $\lambda\widehat{\rho}_{\text{ptr}}$-calculus,*

$$t[\bullet][\bullet] \rightarrow^*_{\text{ptr}} v[\sigma] \quad \textit{if and only if} \quad \langle t,\ \bullet,\ [\,],\ \bullet\rangle \Rightarrow^*_{\text{ptr}} (\overline{v},\ \sigma).$$

The $\lambda\widehat{\rho}_{\text{ptr}}$-calculus therefore directly corresponds to the simplified version of Clinger's properly tail-recursive machine.

In Section 5.7, we showed storeless variants of two machines for stack inspection (the fg and the cm machines). The original versions of these machines use a store in the Clinger fashion [43], and we can exhibit their underlying calculi with an explicit representation of the store, as straightforward extensions of the storeless calculi. We do not include them here for lack of space.

5.9 A lazy calculus of closures

The store-based account of proper tail-recursion from Section 5.8 suggests the following lazy calculus of closures, $\lambda\widehat{\rho}_{\text{l}}$.

5.9.1 The language of $\lambda\widehat{\rho}_{\text{l}}$

The abstract syntax of the language is as follows:

$$\begin{array}{lrcl}
\text{(terms)} & t & ::= & i \mid \lambda t \mid t\, t \\
\text{(closures)} & c & ::= & t[s] \mid c\, \ell \mid \texttt{upd}(\ell, c) \\
\text{(values)} & v & ::= & (\lambda t)[s] \\
\text{(substitutions)} & s & ::= & \bullet \mid \ell \cdot s \\
\text{(reduction contexts)} & C & ::= & [\,] \mid C[[\,]\, \ell] \mid C[\texttt{upd}(\ell, [\,])] \\
\text{(store)} & \sigma & ::= & \bullet \mid \sigma[\ell \mapsto v] \\
\text{(store closures)} & \widetilde{c} & ::= & c[\sigma] \\
\text{(store values)} & \widetilde{v} & ::= & v[\sigma] \\
\text{(store contexts)} & \widetilde{C} & ::= & C[\sigma]
\end{array}$$

5.9.2 Notion of context-sensitive reduction

The notion of reduction is specified by the five rules shown below.

$$
\begin{array}{llcll}
(\mathrm{Var}_1) & \langle i[\ell_1 \cdots \ell_j],\, C[\sigma]\rangle & \rightarrow_{\mathrm{l}} & \langle v,\, C[\sigma]\rangle & \text{if } \sigma(\ell_i) = v \\
(\mathrm{Var}_2) & \langle i[\ell_1 \cdots \ell_j],\, C[\sigma]\rangle & \rightarrow_{\mathrm{l}} & \langle \mathtt{upd}(\ell_i, c),\, C[\sigma]\rangle & \text{if } \sigma(\ell_i) = c \\
(\mathrm{Beta}) & \langle((\lambda t)[s])\,\ell,\, C[\sigma]\rangle & \rightarrow_{\mathrm{l}} & \langle t[\ell \cdot s],\, C[\sigma]\rangle & \\
(\mathrm{App}) & \langle(t_0\, t_1)[s],\, C[\sigma]\rangle & \rightarrow_{\mathrm{l}} & \langle(t_0[s])\,\ell,\, C[\sigma[\ell \mapsto t_1[s]]]\rangle & \\
 & & & \text{where } \ell \text{ does not occur in } s, C, \sigma & \\
(\mathrm{Upd}) & \langle \mathtt{upd}(\ell, v),\, C[\sigma]\rangle & \rightarrow_{\mathrm{l}} & \langle v,\, C[\sigma[\ell \mapsto v]]\rangle &
\end{array}
$$

Variables denote locations, and have two reduction rules, depending on whether the store holds a value or not at that location. In the former case—handled by (Var_1)—the result is this value, the current context, and the current store. In the latter case—handled by (Var_2)—a special closure $\mathtt{upd}(\ell, c)$ is created, indicating that c is a shared computation. When this computation completes and yields a value, the store at location ℓ should be updated with this value, which is achieved by (Upd). Since every argument to an application can potentially be shared, (App) conservatively allocates a new location in the store for such shared closures. (Beta) extends the substitution with this location.

5.9.3 An eval/apply abstract machine

Refocusing, compressing the intermediate transitions, and unfolding the data type of closures mechanically yields the following store-based machine:[3]

$$
\begin{array}{rcl}
\langle i,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle C,\, (\lambda t',\, s'),\, \sigma\rangle \\
 & & \text{where } s(i) = \ell \text{ and } \sigma(\ell) = (\lambda t')[s'] \\
\langle i,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle t',\, s',\, C[\mathtt{upd}(\ell, [\,])],\, \sigma\rangle \\
 & & \text{where } s(i) = \ell \text{ and } \sigma(\ell) = t'[s'] \\
\langle \lambda t,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle C,\, (\lambda t,\, s),\, \sigma\rangle \\
\langle t_0\, t_1,\, s,\, C,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle t_0,\, s,\, C[[\,]\,\ell],\, \sigma[\ell \mapsto (t_1,\, s)]\rangle \\
 & & \text{where } \ell \text{ does not occur in } s, C, \sigma \\
\langle [\,],\, v,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & (v,\, \sigma) \\
\langle C[[\,]\,\ell],\, (\lambda t,\, s),\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle t,\, \ell \cdot s,\, C,\, \sigma\rangle \\
\langle C[\mathtt{upd}(\ell, [\,])],\, v,\, \sigma\rangle & \Rightarrow_{\mathrm{l}} & \langle C,\, v,\, \sigma[\ell \mapsto v]\rangle
\end{array}
$$

This lazy abstract machine coincides with the one derived by Ager et al. out of a call-by-need interpreter for the λ-calculus [4], thereby connecting the present syntactic correspondence between calculi and abstract machines with the functional correspondence between evaluators and abstract machines [3, 5, 28].

[3]When a shared closure is to be evaluated, the current context is extended with what is known as an 'update marker' in the Three Instruction Machine (denoted $C[\mathtt{upd}(\ell, [\,])]$ here).

5.9.4 Formal correspondence

Proposition 5.12. *For any term t in the $\lambda\widehat{\rho}_{\mathrm{l}}$-calculus,*

$$t[\bullet][\bullet] \rightarrow_{\mathrm{l}}^{*} v[\sigma] \quad \textit{if and only if} \quad \langle t, \bullet, [\,], \bullet \rangle \Rightarrow_{\mathrm{l}}^{*} (\overline{v}, \sigma).$$

The $\lambda\widehat{\rho}_{\mathrm{l}}$-calculus therefore directly corresponds to call-by-need evaluation [184].

In $\lambda\widehat{\rho}_{\mathrm{l}}$, sharing is made possible through a global heap where actual parameters are stored. On the other hand, a number of other calculi modeling call by need extend the set of terms with a *local* let-like construct, either by statically translating the source language into an intermediate language with explicit indications of sharing (as in Launchbury's approach [130]), or by providing dynamic reduction rules to the same effect (as in Ariola et al.'s calculus [7]). A sequence of let constructs binding variables to shared computations is a local version of a global heap where shared computations are bound to locations; extra reductions are then needed to propagate all the let operators to the top level.

Another specificity is that allocation occurs early in $\lambda\widehat{\rho}_{\mathrm{l}}$, i.e., a new cell is allocated in the store every time an application is evaluated. Allocation, however, occurs late in Ariola et al.'s semantics, i.e., a new binding is created only when the operator of the application is known to be a λ-abstraction. Delaying allocation is useful in the presence of strict functions, which we do not consider here.

We can construct a local version of our calculus with either of the store propagated inside closures or of late allocation, and from there, mechanically derive the corresponding abstract machine.

5.10 Conclusion

We have presented a series of calculi and abstract machines accounting for a variety of computational effects, making it possible to directly reason about a computation in the calculus and in the corresponding abstract machine (horizontally in the diagram below) and to directly account for actual and explicit substitutions both in the world of calculi and in the world of abstract machines (vertically in the diagram below, where σ maps a closure into the corresponding λ-term and an environment-machine configuration into a configuration in the corresponding machine with actual substitutions):

$$\begin{tikzcd}
\lambda\text{-calculus} \arrow[rr, leftrightarrow, "\text{syntactic}", "\text{correspondence}"'] & & \text{machine with actual substitutions} \\
\lambda\widehat{\rho}\text{-calculus} \arrow[u, dotted, "\sigma"] \arrow[rr, leftrightarrow, "\text{syntactic}", "\text{correspondence}"'] & & \text{environment-based machine} \arrow[u, dotted, "\sigma"']
\end{tikzcd}$$

The correspondence between each calculus and each abstract machine is simple and each can be mechanically built from the other. All of the calculi are new. Many of the abstract machines are known and have been independently designed and proved correct.

The work reported here leads us to drawing the following conclusions.

Curien's calculus of closures: Once extended to account for one-step reduction, $\lambda\rho$ directly corresponds to the notions of evaluation (i.e., weak-head normalization) accounted for by environment-based machines, even in the presence of computational effects (state and control).

Refocusing: Despite its pragmatic origin—fusing a plug function and a decomposition function in a reduction-based evaluation function to improve its efficiency [72], and in combination with compressing intermediate transitions and unfolding closures, refocusing proves consistently useful to construct reduction-free evaluation functions in the form of abstract machines, even in the presence of computational effects.

Defunctionalization: Despite its practical origin—representing a higher-order function with first-order means [155], defunctionalization proves consistently useful, witness the next item and also the fact that except for the abstract machines for $\lambda\widehat{\rho}\mathcal{F}$ and the cm machine, *all* the abstract machines in this article are in defunctionalized form.

Reduction contexts and evaluation contexts: There are three objective reasons—one extensional and two intensional—why contexts are useful as well as, in some sense, unavoidable:

- reduction contexts are in one-to-one correspondence with the compatibility rules of a calculus;
- reduction contexts are the data type of the defunctionalized continuation of a one-step reduction function (as used in a reduction-based (weak-head) normalization function); and
- evaluation contexts are the data type of the defunctionalized continuation of an evaluation function (as used in a reduction-free (weak-head) normalization function).

If nothing else, each of these three reasons has practical value as a guideline for writing the grammar of reduction / evaluation contexts (which can be tricky in practice). But more significantly [62], reduction contexts and evaluation contexts *coincide*, which means that they mediate between one-step reduction and evaluation, particularly since, as initiated by Reynolds [155], defunctionalizing a continuation-passing evaluator yields an abstract machine [3–5, 28], and since as already pointed out above, a vast number of abstract machines are in defunctionalized form [32, 34, 63]:

$$\begin{tikzcd}
& \text{context} \arrow[dl, "\text{plug}"'] \arrow[dr, "\text{apply}"] & \\
\begin{array}{c}\text{one-step}\\ \text{reduction function}\end{array} & & \begin{array}{c}\text{eval/apply}\\ \text{abstract machine}\end{array} \\
& & \begin{array}{c}\text{evaluation function}\\ \text{in CPS}\end{array} \arrow[u, "\text{defunctionalization}"]
\end{tikzcd}$$

Together, the syntactic correspondence and the functional correspondence connect apparently distinct approaches to the same computational situations. We already illustrated this connection in Section 5.5 with delimited continuations; let us briefly illustrate it with the simpler example of call/cc:

> Call/cc was introduced in the Scheme programming language [45] as a Church encoding of Reynolds's escape operator [155]. A typed version of it is available in Standard ML of New Jersey [106] and Griffin has identified its logical content [100]. It is endowed with a variety of specifications: a CPS interpreter [107, 155], a denotational semantics [120], a big-step operational semantics [106], the CEK machine [85], calculi in the form of reduction semantics [84], and a number of implementation techniques [46, 112]—not to mention its call-by-name version in the archival version of Krivine's machine [125].

Question: How do we know that all the specifications in this semantic jungle define the same call/cc?

The elements of answer we contribute here are that the syntactic correspondence links calculi and abstract machines, and the functional correspondence links abstract machines and interpreters. So by construction, all the specifications that are inter-derivable are consistent.

Normalization by evaluation: Finally, refocusing provides a guideline for constructing reduction-free normalization functions out of reduction-based ones [61]. The reduction-free normalization functions take the form of eval/apply abstract machines, which usually are in defunctionalized form, which paves the way to writing normalization functions as usually encountered in the area of normalization by evaluation. We have illustrated the method with weak reduction and weak-head normalization (i.e., evaluation), but it also works for strong reduction and normalization, thus linking one-step reduction, abstract machines for strong reduction, and normalization functions.

Acknowledgments

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 6

An operational foundation for delimited continuations in the CPS hierarchy

with Dariusz Biernacki and Olivier Danvy [28]

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.

6.1 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 [9, 83, 86, 88, 102, 110, 111, 141, 152, 164, 186] which have found applications in models of control, concurrency, and type-directed partial evaluation [13, 110, 165]. Of the former, there is the work revolving around the family of control operators shift and reset [66–68, 73, 89, 90, 118, 119, 143, 186] which have found applications in non-deterministic programming, code generation, partial evaluation, normalization by evaluation, computational monads, and mobile computing [10, 11, 14, 32, 59, 60, 78, 79, 91, 97, 101, 108, 120, 122, 131, 162, 174, 175, 178].

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 [73].

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 [3] and we construct the corresponding reduction semantics [82], as pioneered by Felleisen and Friedman [85]. 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 6.2, 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 6.3, we illustrate the use of shift and reset with the classic example of finding list prefixes, using an ML-like programming language. In Section 6.4, 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 6.5, we extend this result to the CPS hierarchy. In Section 6.6, we illustrate the CPS hierarchy with a class of normalization functions for a hierarchical language of units and products.

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

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

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

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

Figure 6.1: 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 [3–5, 32, 63], which itself is based on Reynolds's seminal work on definitional interpreters [155]. 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 [85]. The last step has been studied further by Hardin, Maranget, and Pagano [105] in the context of explicit substitutions and by Biernacka, Danvy, and Nielsen [29, 30, 72].

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):

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

6.2.1 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 6.1.

6.2.2 CPS transformation

We CPS-transform the evaluator by naming intermediate results, sequentializing their computation, and introducing an extra functional parameter, the continuation [68, 148, 168]. The resulting continuation-passing evaluator is displayed in Figure 6.2.

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

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

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

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

Figure 6.2: A continuation-passing evaluator for arithmetic expressions

- Values: $\mathsf{val} \ni v ::= \ulcorner m \urcorner$
- Defunctionalized continuations: $\mathsf{cont} \ni k ::= [\,] \mid \mathsf{ADD}_2\,(e,\, k) \mid \mathsf{ADD}_1\,(v,\, k)$
- Functions $\mathsf{eval} : \mathsf{exp} \times \mathsf{cont} \to \mathsf{val}$
  $\mathsf{apply\_cont} : \mathsf{cont} \times \mathsf{val} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner,\, k) & = & \mathsf{apply\_cont}\,(k,\, \ulcorner m \urcorner) \\ \mathsf{eval}\,(e_1 + e_2,\, k) & = & \mathsf{eval}\,(e_1,\, \mathsf{ADD}_2\,(e_2,\, k)) \\ \\ \mathsf{apply\_cont}\,([\,],\, v) & = & v \\ \mathsf{apply\_cont}\,(\mathsf{ADD}_2\,(e_2,\, k),\, v_1) & = & \mathsf{eval}\,(e_2,\, \mathsf{ADD}_1\,(v_1,\, k)) \\ \mathsf{apply\_cont}\,(\mathsf{ADD}_1\,(\ulcorner m_1 \urcorner,\, k),\, \ulcorner m_2 \urcorner) & = & \mathsf{apply\_cont}\,(k,\, \ulcorner m_1 \urcorner + \ulcorner m_2 \urcorner) \end{array}$$

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

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

Figure 6.3: A defunctionalized continuation-passing evaluator for arithmetic expressions

6.2.3 Defunctionalization

The generalization of closure conversion [128] to defunctionalization is due to Reynolds [155]. 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 are represented as data types, injections as data-type constructors, and apply functions are defined by case over the corresponding data types [71].

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

$$\begin{array}{rcl}
e & \Rightarrow & \langle e,\,[\,]\rangle_{\mathit{eval}} \\[1ex]
\langle \ulcorner m \urcorner,\,C\rangle_{\mathit{eval}} & \Rightarrow & \langle C,\,\ulcorner m \urcorner\rangle_{\mathit{cont}} \\
\langle e_1 \mathbin{\texttt{+}} e_2,\,C\rangle_{\mathit{eval}} & \Rightarrow & \langle e_1,\,\mathsf{ADD}_2\,(e_2,\,C)\rangle_{\mathit{eval}} \\[1ex]
\langle \mathsf{ADD}_2\,(e_2,\,C),\,v_1\rangle_{\mathit{cont}} & \Rightarrow & \langle e_2,\,\mathsf{ADD}_1\,(v_1,\,C)\rangle_{\mathit{eval}} \\
\langle \mathsf{ADD}_1\,(\ulcorner m_1 \urcorner,\,C),\,\ulcorner m_2 \urcorner\rangle_{\mathit{cont}} & \Rightarrow & \langle C,\,\ulcorner m_1 \urcorner + \ulcorner m_2 \urcorner\rangle_{\mathit{cont}} \\[1ex]
\langle [\,],\,v\rangle_{\mathit{cont}} & \Rightarrow & v
\end{array}$$

Figure 6.4: A value-based abstract machine for evaluating arithmetic expressions

Here, we defunctionalize the continuation of the continuation-passing evaluator in Figure 6.2. 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 6.3.

6.2.4 Abstract machines as defunctionalized continuation-passing programs

Elsewhere [3,63], 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' [62, 71, 85].)

Indeed Plotkin's Indifference Theorem [148] states that continuation-passing programs are independent of their evaluation order. In Reynolds's words [155], all the subterms in applications are 'trivial'; and in Moggi's words [140], these subterms are values and not computations. Furthermore, continuation-passing programs are tail recursive [168]. Therefore, since in a continuation-passing program all calls are tail calls and all subcomputations are elementary, a defunctionalized continuation-passing program implements a transition system [149],

- 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:

$$\begin{array}{rcl}
e & \Rightarrow & \langle e,\, [\,]\rangle_{eval} \\[1ex]
\langle \ulcorner m \urcorner,\, C\rangle_{eval} & \Rightarrow & \langle C,\, \ulcorner m \urcorner\rangle_{cont} \\
\langle e_1 \texttt{+} e_2,\, C\rangle_{eval} & \Rightarrow & \langle e_1,\, \mathsf{ADD}_2\,(e_2,\, C)\rangle_{eval} \\[1ex]
\langle \mathsf{ADD}_2\,(e_2,\, C),\, v_1\rangle_{cont} & \Rightarrow & \langle e_2,\, \mathsf{ADD}_1\,(v_1,\, C)\rangle_{eval} \\
\langle \mathsf{ADD}_1\,(\ulcorner m_1 \urcorner,\, C),\, \ulcorner m_2 \urcorner\rangle_{cont} & \Rightarrow & \langle C,\, \ulcorner \ulcorner m_1 \urcorner + \ulcorner m_2 \urcorner \urcorner\rangle_{cont} \\[1ex]
\langle [\,],\, v\rangle_{cont} & \Rightarrow & v
\end{array}$$

Figure 6.5: A term-based abstract machine for processing arithmetic expressions

i.e., an abstract machine.

We thus reformat Figure 6.3 into Figure 6.4. The correctness of the abstract machine with respect to the initial evaluator follows from the correctness of CPS transformation and of defunctionalization.

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

We observe that the domain of expressible values in Figure 6.4 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 6.5; it is a syntactic theory [82].

6.2.6 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 [85] to give a reduction semantics for control operators. Let us demonstrate it.

We analyze the transitions of the abstract machine in Figure 6.5. 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:

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

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

6.2.7 From reduction semantics to term-based abstract machine

In Section 6.2.6, we have constructed the reduction semantics corresponding to the abstract machine of Figure 6.5, as pioneered by Felleisen and Friedman [84, 85]. Over the last few years [29, 30, 61, 72], 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 6.2.6 to the eval/apply abstract machine of Figure 6.5.

6.2.8 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 6.4, we apply this derivation and this construction to the first level of the CPS hierarchy, and in Section 6.5, we apply them to an arbitrary level of the CPS hierarchy. But first, let us illustrate how to program with delimited continuations.

6.3 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 6.3.1, we start with a simple solution that uses a first-order accumulator. This simple solution is in defunctionalized form. In Section 6.3.2, we present its higher-order counterpart, which uses a functional accumulator. This functional accumulator acts as a delimited continuation. In Section 6.3.3, we present its direct-style counterpart (which uses shift and reset) and in Section 6.3.4, we present its continuation-passing counterpart (which uses two layers of continuations). In Section 6.3.5, we introduce the CPS hierarchy informally. We then mention a typing issue in Section 6.3.6 and review related work in Section 6.3.7.

6.3.1 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.

```
$find\_first\_prefix\_a\,(p,\ xs)$  $\overset{\text{def}}{=}$  $letrec\ visit\,(nil,\ a)$
                                          $= nil$
                                        $|\ visit\,(x :: xs,\ a)$
                                          $= let\ a' = x :: a$
                                             $in\ if\ p\, x$
                                                 $then\ reverse\,(a',\ nil)$
                                                 $else\ visit\,(xs,\ a')$
                                     $and\ reverse\,(nil,\ xs)$
                                          $= xs$
                                        $|\ reverse\,(x :: a,\ xs)$
                                          $= reverse\,(a,\ x :: xs)$
                                   $in\ visit\,(xs,\ nil)$
```

```
$\mathit{find\_all\_prefixes\_a}\,(p,\ xs) \stackrel{\text{def}}{=}$ $\mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\ a)$
                                              $= \mathit{nil}$
                                            $|\ \mathit{visit}\,(x :: xs,\ a)$
                                              $= \mathit{let}\ a' = x :: a$
                                                 $\mathit{in}\ \mathit{if}\ p\,x$
                                                    $\mathit{then}\ (\mathit{reverse}\,(a',\ \mathit{nil}))$
                                                          $:: (\mathit{visit}\,(xs,\ a'))$
                                                    $\mathit{else}\ \mathit{visit}\,(xs,\ a')$
                                         $\mathit{and}\ \mathit{reverse}\,(\mathit{nil},\ xs)$
                                              $= xs$
                                            $|\ \mathit{reverse}\,(x :: a,\ xs)$
                                              $= \mathit{reverse}\,(a,\ x :: xs)$
                                       $\mathit{in}\ \mathit{visit}\,(xs,\ \mathit{nil})$
```

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 [71,155]: the accumulator has the data type of a defunctionalized function and *reverse* is its apply function. We present its higher-order counterpart next [115].

6.3.2 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.

```
$\mathit{find\_first\_prefix\_c}_1\,(p,\ xs) \stackrel{\text{def}}{=}$ $\mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\ k)$
                                              $= \mathit{nil}$
                                            $|\ \mathit{visit}\,(x :: xs,\ k)$
                                              $= \mathit{let}\ k' = \lambda vs.k\,(x :: vs)$
                                                 $\mathit{in}\ \mathit{if}\ p\,x$
                                                    $\mathit{then}\ k'\ \mathit{nil}$
                                                    $\mathit{else}\ \mathit{visit}\,(xs,\ k')$
                                       $\mathit{in}\ \mathit{visit}\,(xs,\ \lambda vs.vs)$
```

```
$\mathit{find\_all\_prefixes\_c}_1\,(p,\ xs) \stackrel{\text{def}}{=}$ $\mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\ k)$
                                              $= \mathit{nil}$
                                            $|\ \mathit{visit}\,(x :: xs,\ k)$
                                              $= \mathit{let}\ k' = \lambda vs.k\,(x :: vs)$
                                                 $\mathit{in}\ \mathit{if}\ p\,x$
                                                    $\mathit{then}\ (k'\ \mathit{nil}) :: (\mathit{visit}\,(xs,\ k'))$
                                                    $\mathit{else}\ \mathit{visit}\,(xs,\ k')$
                                       $\mathit{in}\ \mathit{visit}\,(xs,\ \lambda vs.vs)$
```

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 6.3.1.

The functional accumulator is a delimited continuation:

- In $\mathit{find\_first\_prefix\_c}_1$, *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 $\mathit{find\_all\_prefixes\_c}_1$, *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.

6.3.3 Finding prefixes in direct style

The two following local functions are the direct-style counterpart of the two local functions in Section 6.3.2:

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

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

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 6.3.2 [68].

6.3.4 Finding prefixes in continuation-passing style

The two following local functions are the continuation-passing counterpart of the two local functions in Section 6.3.2:

```
find_first_prefix_c_2 (p, xs) ≝ letrec visit (nil, k_1, k_2)
                                          = k_2 nil
                                        | visit (x :: xs, k_1, k_2)
                                          = let k'_1 = λ(vs, k'_2).k_1 (x :: vs, k'_2)
                                            in if p x
                                               then k'_1 (nil, k_2)
                                               else visit (xs, k'_1, k_2)
                                 in visit (xs, λ(vs, k_2).k_2 vs, λvs.vs)
```

```
find_all_prefixes_c_2 (p, xs) ≝ letrec visit (nil, k_1, k_2)
                                          = k_2 nil
                                        | visit (x :: xs, k_1, k_2)
                                          = let k'_1 = λ(vs, k'_2).k_1 (x :: vs, k'_2)
                                            in if p x
                                               then k'_1 (nil, λvs.visit (xs, k'_1,
                                                                          λvss.k_2 (vs :: vss))
                                               else visit (xs, k'_1, k_2)
                                 in visit (xs, λ(vs, k_2).k_2 vs, λvss.vss)
```

CPS-transforming the two local functions of Section 6.3.2 adds another layer of continuations and restores the syntactic characterization of all calls being tail calls and all sub-computations being elementary.

6.3.5 The CPS hierarchy

If k_2 were used non-tail recursively in a variant of the examples of Section 6.3.4, 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, 73], 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 6.4.)

6.3.6 A note about typing

The type of $\mathit{find\_all\_prefixes\_c}_1$, in Section 6.3.2, 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 $\mathit{find\_all\_prefixes\_c}_0$ provides a simple and meaningful example where Filinski's typing of shift [89] 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 [143] and in Wadler's most general monadic type system [186, 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 [9].

6.3.7 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, 163]:

- In the Lisp Pointers [57], Danvy presents three versions of this function: a typed continuation-passing version (corresponding to Section 6.3.2), one with delimited control (corresponding to Section 6.3.3), and one in assembly language.
- In his PhD thesis [163, Section 6.3], Sitaram presents two versions of this function: one with an accumulator (corresponding to Section 6.3.1) and one with delimited control (corresponding to Section 6.3.3).

In Section 6.3.2, 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.

- Terms: $\mathsf{term} \ni t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\; t \mid \langle t \rangle \mid \mathcal{S}k.t$
- Values: $\mathsf{val} \ni v ::= \ulcorner m \urcorner \mid f$
- 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: $\begin{array}{rcl} \theta_1 & = & \lambda(v,\, k_2).\, k_2\; v \\ \theta_2 & = & \lambda v.\, v \end{array}$
- Environments: $\mathsf{env} \ni e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Evaluation function: $\mathsf{eval} : \mathsf{term} \times \mathsf{env} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner,\, e,\, k_1,\, k_2) & = & k_1\,(\ulcorner m \urcorner,\, k_2) \\ \mathsf{eval}\,(x,\, e,\, k_1,\, k_2) & = & k_1\,(e(x),\, k_2) \\ \mathsf{eval}\,(\lambda x.t,\, e,\, k_1,\, k_2) & = & k_1\,(\lambda(v,\, k_1',\, k_2').\, \mathsf{eval}\,(t,\, e[x \mapsto v],\, k_1',\, k_2'),\, k_2) \\ \mathsf{eval}\,(t_0\, t_1,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t_0,\, e, \\ & & \lambda(f,\, k_2').\, \mathsf{eval}\,(t_1,\, e,\, \lambda(v,\, k_2'').\, f\;(v,\, k_1,\, k_2''),\, k_2'),\, k_2) \\ \mathsf{eval}\,(\mathit{succ}\; t,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e,\, \lambda(\ulcorner m \urcorner,\, k_2').\, k_1\,(\ulcorner m \urcorner + 1,\, k_2'),\, k_2) \\ \mathsf{eval}\,(\langle t \rangle,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e,\, \theta_1,\, \lambda v.\, k_1\,(v,\, k_2)) \\ \mathsf{eval}\,(\mathcal{S}k.t,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e[k \mapsto c],\, \theta_1,\, k_2) \\ & & \text{where } c = \lambda(v,\, k_1',\, k_2').\, k_1\,(v,\, \lambda v'.\, k_1'\,(v',\, k_2')) \end{array}$$

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

$$\mathsf{evaluate}\,(t) = \mathsf{eval}\,(t,\, e_{\mathit{empty}},\, \theta_1,\, \theta_2)$$

Figure 6.6: An environment-based evaluator for the first level of the CPS hierarchy

6.3.8 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.

6.4 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 6.2. 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:

$$t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle \mid \mathcal{S}k.t$$

Programs are closed terms.

This source language is a subset of the language used in the examples of Section 6.3. Adding the remaining constructs is a straightforward exercise and does not contribute to our point here.

6.4.1 An environment-based evaluator

Figure 6.6 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 e, 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 $\langle t \rangle$ is interpreted by evaluating t with the initial continuation and a meta-continuation on which the current continuation has been "pushed." (Indeed, and as will be shown in Section 6.4.2, defunctionalizing the meta-continuation yields the data type of a stack [71].)

The shift control operator is used to abstract (delimited) control. A shift expression $\mathcal{S}k.t$ is interpreted by capturing the current continuation, binding it to k, and evaluating t 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.

6.4.2 An environment-based abstract machine

The evaluator displayed in Figure 6.6 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 6.7.

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 [85], 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, 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 e, a context C_1, and a meta-context C_2, a reset expression $\langle t \rangle$ is processed by evaluating t with the same environment e, the empty context $\bullet$, and a meta-context where C_1 has been pushed on C_2.

Given an environment e, a context C_1, and a meta-context C_2, a shift expression $\mathcal{S}k.t$ is processed by evaluating t with an extension of e 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 t to a value v

[1]The grammar of evaluation contexts in Figure 6.7 is isomorphic to the grammar of evaluation contexts in the standard inside-out notation:

$$C_1 \quad ::= \quad [\,] \mid C_1[[\,]\,(t,e)] \mid C_1[succ\;\;[\,]] \mid C_1[v\;[\,]]$$

[2]To build on Peyton Jones's terminology [134], this abstract machine is therefore in 'eval/apply/meta-apply' form.

- Terms: $t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle\!\langle t \rangle\!\rangle \mid \mathcal{S}k.t$
- Values (integers, closures, and captured continuations):

$$v ::= \ulcorner m \urcorner \mid [x,\, t,\, e] \mid C_1$$

- Environments: $e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Evaluation contexts:

$$C_1 ::= [\,] \mid \mathtt{ARG}((t,e), C_1) \mid \mathtt{SUCC}\, C_1 \mid \mathtt{APP}(v, C_1)$$

- Meta-contexts: $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

$$\begin{array}{rcl}
t & \Rightarrow & \langle t,\, e_{\mathit{empty}},\, [\,],\, \bullet\rangle_{\mathit{eval}} \\[1ex]
\langle \ulcorner m \urcorner,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, \ulcorner m \urcorner,\, C_2\rangle_{\mathit{cont_1}} \\
\langle x,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, e\,(x),\, C_2\rangle_{\mathit{cont_1}} \\
\langle \lambda x.t,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, [x,\, t,\, e],\, C_2\rangle_{\mathit{cont_1}} \\
\langle t_0\, t_1,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle t_0,\, e,\, \mathtt{ARG}((t_1, e), C_1),\, C_2\rangle_{\mathit{eval}} \\
\langle \mathit{succ}\ t,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e,\, \mathtt{SUCC}\, C_1,\, C_2\rangle_{\mathit{eval}} \\
\langle \langle\!\langle t \rangle\!\rangle,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e,\, [\,],\, C_2 \cdot C_1\rangle_{\mathit{eval}} \\
\langle \mathcal{S}k.t,\, e,\, C_1,\, C_2\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e[k \mapsto C_1],\, [\,],\, C_2\rangle_{\mathit{eval}} \\[1ex]
\langle [\,],\, v,\, C_2\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_2,\, v\rangle_{\mathit{cont_2}} \\
\langle \mathtt{ARG}((t,e), C_1),\, v,\, C_2\rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e,\, \mathtt{APP}(v, C_1),\, C_2\rangle_{\mathit{eval}} \\
\langle \mathtt{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 \mathtt{APP}([x,\, t,\, e], C_1),\, v,\, C_2\rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e[x \mapsto v],\, C_1,\, C_2\rangle_{\mathit{eval}} \\
\langle \mathtt{APP}(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}} \\[1ex]
\langle C_2 \cdot C_1,\, v\rangle_{\mathit{cont_2}} & \Rightarrow & \langle C_1,\, v,\, C_2\rangle_{\mathit{cont_1}} \\[1ex]
\langle \bullet,\, v\rangle_{\mathit{cont_2}} & \Rightarrow & v
\end{array}$$

Figure 6.7: An environment-based abstract machine for the first level of the CPS hierarchy

whenever the environment-based machine, started with t, stops with v. The following theorem states this correctness by relating observable results:

Theorem 6.1. *For any program t and any integer value $\ulcorner m \urcorner$,* $\mathsf{evaluate}\,(t) = \ulcorner m \urcorner$ *if and only if* $\mathsf{eval}^{\mathsf{e}}\,(t) = \ulcorner m \urcorner$.

Proof. The theorem follows directly from the correctness of defunctionaliza-

- Terms and values: $t ::= v \mid x \mid t_0\, t_1 \mid succ\ t \mid \langle t \rangle \mid \mathcal{S}k.t$
  $v ::= \ulcorner m \urcorner \mid \lambda x.t \mid C_1$
- Evaluation contexts: $C_1 ::= [\,] \mid \texttt{ARG}(t, C_1) \mid \texttt{SUCC}\, C_1 \mid \texttt{APP}(v, C_1)$
- Meta-contexts: $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

$$\begin{array}{rcl}
t & \Rightarrow & \langle t,\, [\,],\, \bullet \rangle_{eval} \\[1ex]
\langle \ulcorner m \urcorner,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle C_1,\, \ulcorner m \urcorner,\, C_2 \rangle_{cont_1} \\
\langle \lambda x.t,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle C_1,\, \lambda x.t,\, C_2 \rangle_{cont_1} \\
\langle C_1',\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle C_1,\, C_1',\, C_2 \rangle_{cont_1} \\
\langle t_0\, t_1,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle t_0,\, \texttt{ARG}(t_1, C_1),\, C_2 \rangle_{eval} \\
\langle succ\ t,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle t,\, \texttt{SUCC}\, C_1,\, C_2 \rangle_{eval} \\
\langle \langle t \rangle,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle t,\, [\,],\, C_2 \cdot C_1 \rangle_{eval} \\
\langle \mathcal{S}k.t,\, C_1,\, C_2 \rangle_{eval} & \Rightarrow & \langle k\{C_1/t\},\, [\,],\, C_2 \rangle_{eval} \\[1ex]
\langle [\,],\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_2,\, v \rangle_{cont_2} \\
\langle \texttt{ARG}(t, C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle t,\, \texttt{APP}(v, C_1),\, C_2 \rangle_{eval} \\
\langle \texttt{SUCC}\, C_1,\, \ulcorner m \urcorner,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1,\, \ulcorner m+1 \urcorner,\, C_2 \rangle_{cont_1} \\
\langle \texttt{APP}(\lambda x.t, C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle x\{v/t\},\, C_1,\, C_2 \rangle_{eval} \\
\langle \texttt{APP}(C_1', C_1),\, v,\, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{cont_1} \\[1ex]
\langle C_2 \cdot C_1,\, v \rangle_{cont_2} & \Rightarrow & \langle C_1,\, v,\, C_2 \rangle_{cont_1} \\[1ex]
\langle \bullet,\, v \rangle_{cont_2} & \Rightarrow & v
\end{array}$$

Figure 6.8: A substitution-based abstract machine for the first level of the CPS hierarchy

tion [15, 144]. □

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.

6.4.3 A substitution-based abstract machine

The environment-based abstract machine of Figure 6.7, 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 6.8. 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 [85].)

We write $x\{v/t\}$ to denote the result of the usual capture-avoiding substitution of the value v for x in t.

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 6.2. *For any program t, either both $\mathsf{eval}^{\mathsf{s}}(t)$ and $\mathsf{eval}^{\mathsf{e}}(t)$ are undefined, or there exist values v, v' such that $\mathsf{eval}^{\mathsf{s}}(t) = v$, $\mathsf{eval}^{\mathsf{e}}(t) = v'$ and $\mathcal{T}(v') = v$. The function $\mathcal{T}$ relates a semantic value with its syntactic representation and is defined as follows:*[3]

$$\begin{array}{rcl}
\mathcal{T}(\ulcorner m \urcorner) & = & \ulcorner m \urcorner \\
\mathcal{T}([x,\, t,\, e]) & = & \lambda x.x_n\{\mathcal{T}(e(x_n))/x_1\{\mathcal{T}(e(x_1))/t\}\ldots\}, \\
 & & \textit{where } FV\,(\lambda x.\,t) = \{x_1, \ldots, x_n\} \\
\mathcal{T}([\,]) & = & [\,] \\
\mathcal{T}(\mathtt{ARG}((t,e), C_1)) & = & \mathtt{ARG}(x_n\{\mathcal{T}(e(x_n))/x_1\{\mathcal{T}(e(x_1))/t\}\ldots\}, \mathcal{T}(C_1)), \\
 & & \textit{where } FV\,(t) = \{x_1, \ldots, x_n\} \\
\mathcal{T}(\mathtt{APP}(v, C_1)) & = & \mathtt{APP}(\mathcal{T}(v), \mathcal{T}(C_1)) \\
\mathcal{T}(\mathtt{SUCC}\, C_1) & = & \mathtt{SUCC}\, \mathcal{T}(C_1)
\end{array}$$

Proof. We extend the translation function $\mathcal{T}$ to meta-contexts and configurations, in the expected way, e.g.,

$$\begin{array}{rcl}
\mathcal{T}(\langle t,\, e,\, C_1,\, C_2\rangle_{eval}) & = & \langle x_n\{\mathcal{T}(e(x_n))/x_1\{\mathcal{T}(e(x_1))/t\}\ldots\},\, \mathcal{T}(C_1),\, \mathcal{T}(C_2)\rangle_{eval} \\
 & & \text{where } FV\,(t) = \{x_1, \ldots, x_n\}
\end{array}$$

Then it is straightforward to show that the two abstract machines operate in lock step with respect to the translation. Hence, for any program t, both machines diverge or they both stop (after the same number of transitions) with the values v and $\mathcal{T}(v)$, respectively. $\square$

We now proceed to analyze the transitions of the machine displayed in Figure 6.8. We can think of a configuration $\langle t,\, C_1,\, C_2\rangle_{eval}$ as the following decomposition of the initial term into a meta-context C_2, a context C_1, and an intermediate term t:

$$C_2 \,\#\, C_1[t]$$

[3]$\mathcal{T}$ is a generalization of Plotkin's function Real [148].

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 $C_2 \mathbin{\texttt{\#}} C_1[t]$, 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 t, $cont_1$-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.

6.4.4 A reduction semantics

A reduction semantics provides a reduction relation on expressions by defining values, evaluation contexts, and redexes [82,84,85,195]. In the present case,

- the values are already specified in the (substitution-based) abstract machine:

$$v ::= \ulcorner m \urcorner \mid \lambda x.t \mid C_1$$

- the evaluation contexts and meta-contexts are already specified in the abstract machine, as the data-type part of defunctionalized continuations;

$$\begin{aligned} C_1 &::= [\,] \mid \texttt{ARG}(t, C_1) \mid \texttt{APP}(v, C_1) \mid \texttt{SUCC}\, C_1 \\ C_2 &::= \bullet \mid C_2 \cdot C_1 \end{aligned}$$

- we can read the redexes off the transitions of the abstract machine:

$$r ::= succ\, \ulcorner m \urcorner \mid (\lambda x.t)\, v \mid \mathcal{S}k.t \mid C_1'\, v \mid \langle v \rangle$$

Based on the distinction between decomposition and reduction, we single out the following reduction rules from the transitions of the machine:

$$\begin{array}{lrcl} (\delta) & C_2 \mathbin{\texttt{\#}} C_1[succ\, \ulcorner m \urcorner] & \rightarrow & C_2 \mathbin{\texttt{\#}} C_1[\ulcorner m+1 \urcorner] \\ (\beta_\lambda) & C_2 \mathbin{\texttt{\#}} C_1[(\lambda x.t)\, v] & \rightarrow & C_2 \mathbin{\texttt{\#}} C_1[x\{v/t\}] \\ (\mathcal{S}_\lambda) & C_2 \mathbin{\texttt{\#}} C_1[\mathcal{S}k.t] & \rightarrow & C_2 \mathbin{\texttt{\#}} [k\{C_1/t\}] \\ (\beta_{ctx}) & C_2 \mathbin{\texttt{\#}} C_1[C_1'\, v] & \rightarrow & C_2 \cdot C_1 \mathbin{\texttt{\#}} C_1'[v] \\ (\text{Reset}) & C_2 \mathbin{\texttt{\#}} C_1[\langle v \rangle] & \rightarrow & C_2 \mathbin{\texttt{\#}} C_1[v] \end{array}$$

(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 $\lambda k.t$ 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 [118, 119, 143]. 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:

$$\begin{array}{rcl}
\mathsf{decompose}\, t & = & \mathsf{decompose}'\,(t,\,[\,],\,\bullet) \\
\mathsf{decompose}'\,(t_0\,t_1,\,C_1,\,C_2) & = & \mathsf{decompose}'\,(t_0,\,\texttt{ARG}(t_1, C_1),\,C_2) \\
\mathsf{decompose}'\,(succ\;t,\,C_1,\,C_2) & = & \mathsf{decompose}'\,(t,\,\texttt{SUCC}\;C_1,\,C_2) \\
\mathsf{decompose}'\,(\langle t\rangle,\,C_1,\,C_2) & = & \mathsf{decompose}'\,(t,\,[\,],\,C_2\cdot C_1) \\
\mathsf{decompose}'\,(v,\,\texttt{ARG}(t, C_1),\,C_2) & = & \mathsf{decompose}'\,(t,\,\texttt{APP}(v, C_1),\,C_2)
\end{array}$$

In the remaining cases either a value or a redex has been found:

$$\begin{array}{rcl}
\mathsf{decompose}'\,(v,\,[\,],\,\bullet) & = & \bullet \;\texttt{\#}\; [v] \\
\mathsf{decompose}'\,(v,\,[\,],\,C_2\cdot C_1) & = & C_2 \;\texttt{\#}\; C_1[\langle v\rangle] \\
\mathsf{decompose}'\,(\mathcal{S}k.t,\,C_1,\,C_2) & = & C_2 \;\texttt{\#}\; C_1[\mathcal{S}k.t] \\
\mathsf{decompose}'\,(v,\,\texttt{APP}((\lambda x.t), C_1),\,C_2) & = & C_2 \;\texttt{\#}\; C_1[(\lambda x.t)\,v] \\
\mathsf{decompose}'\,(v,\,\texttt{APP}(C_1', C_1),\,C_2) & = & C_2 \;\texttt{\#}\; C_1[C_1'\,v] \\
\mathsf{decompose}'\,(\ulcorner m\urcorner,\,\texttt{SUCC}\;C_1,\,C_2) & = & C_2 \;\texttt{\#}\; C_1[succ\;\ulcorner m\urcorner]
\end{array}$$

An inverse of the decompose function, traditionally called plug, reconstructs a term from its decomposition:

$$\begin{array}{rcl}
\mathsf{plug}\,(\bullet \;\texttt{\#}\; [t]) & = & t \\
\mathsf{plug}\,(C_2\cdot C_1 \;\texttt{\#}\; [t]) & = & \mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[\langle t\rangle]) \\
\mathsf{plug}\,(C_2 \;\texttt{\#}\; (\texttt{ARG}(t', C_1))[t]) & = & \mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[t\,t']) \\
\mathsf{plug}\,(C_2 \;\texttt{\#}\; (\texttt{APP}(v, C_1))[t]) & = & \mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[v\,t]) \\
\mathsf{plug}\,(C_2 \;\texttt{\#}\; (\texttt{SUCC}\;C_1)[t]) & = & \mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[succ\;t])
\end{array}$$

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:

$$p ::= succ\;v \;\mid\; v_0\,v_1 \;\mid\; \mathcal{S}k.t \;\mid\; \langle v\rangle$$

Lemma 6.1 (Unique decomposition). *A program t 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 $t = \mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[p])$. In the former case* decompose $t = \bullet \;\texttt{\#}\; [v]$ *and in the latter case either* decompose $t = C_2 \;\texttt{\#}\; C_1[p]$ *if p is an actual redex, or* decompose t *is undefined.*

Proof. The first part follows by induction on the structure of t. The second part follows from the equation $\mathsf{decompose}(\mathsf{plug}\,(C_2 \;\texttt{\#}\; C_1[r])) = C_2 \;\texttt{\#}\; C_1[r]$ 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 6.3. *For any program t and any value v, $\mathsf{eval}^{\mathsf{s}}(t) = v$ if and only if $t \rightarrow^* v$, 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

$$\mathsf{decompose}'\,(t,\, C_1,\, C_2) = \mathsf{decompose}\,(\mathsf{plug}\,(C_2 \,\texttt{\#}\, C_1[t]))$$

which holds for any C_2, C_1 and t. □

We have verified that using refocusing [30,72], one can go from this reduction semantics to the abstract machine of Figure 6.8.

6.4.5 Beyond CPS

Alternatively to using the meta-context to compose delimited continuations, as in Figure 6.7, we could compose them by concatenating their representation [88]. Such a concatenation function is defined as follows:

$$\begin{array}{rcl} [\,] \star C_1' & = & C_1' \\ (\texttt{ARG}((t,e),C_1)) \star C_1' & = & \texttt{ARG}((t,e), C_1 \star C_1') \\ (\texttt{SUCC}\, C_1) \star C_1' & = & \texttt{SUCC}\, C_1 \star C_1' \\ (\texttt{APP}(v,C_1)) \star C_1' & = & \texttt{APP}(v, C_1 \star C_1') \end{array}$$

(The second clause would read $(\texttt{ARG}(t,C_1)) \star C_1' = \texttt{ARG}(t, C_1 \star C_1')$ for the contexts of Figure 6.8.)

Then, in Figures 6.7 and 6.8, we could replace the transition

$$\langle \texttt{APP}(C_1', C_1),\, v,\, C_2 \rangle_{\mathit{cont}_1} \quad \Rightarrow \quad \langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{\mathit{cont}_1}$$

by the following one:

$$\langle \texttt{APP}(C_1', C_1),\, v,\, C_2 \rangle_{\mathit{cont}_1} \quad \Rightarrow \quad \langle C_1' \star C_1,\, v,\, C_2 \rangle_{\mathit{cont}_1}$$

This replacement changes the control effect of shift to that of Felleisen et al.'s $\mathcal{F}$ operator [83]. Furthermore, the modified abstract machine is in structural correspondence with Felleisen et al.'s abstract machine for $\mathcal{F}$ and $\#$ [83,88].

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 [71] 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 6.7 and 6.8, 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 [34], 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 [80].

6.4.6 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 [35], 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:

```
foo xs  ≝  letrec visit nil                    bar xs  ≝  letrec visit nil
                    = nil                                           = nil
                  | visit (x :: xs)                               | visit (x :: xs)
                    = visit                                         = visit
                        (𝒮k.x :: (k xs))                                (ℱk.x :: (k xs))
           in ⟨visit xs⟩                                   in ⟨visit xs⟩
```

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:

$$C_1 ::= [\,] \mid \texttt{ARG}(t, C_1) \mid \texttt{SUCC}\, C_1 \mid \texttt{APP}(v, C_1) \mid \texttt{CONS}(v, C_1) \mid \ldots$$

Similarly, we need to expand the definition of concatenation as follows:

$$(\texttt{CONS}(v, C_1)) \star C_1' \quad = \quad \texttt{CONS}(v, C_1 \star C_1')$$

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 $\texttt{APP}(\mathit{visit}, [\,])$ throughout (writing *visit* in the context for simplicity):

$$\begin{array}{rcl} C_2 & \texttt{\#} & C_1[\langle \mathit{visit}\,(1 :: 2 :: \mathit{nil})\rangle] \\ C_2 \cdot C_1 & \texttt{\#} & [\mathit{visit}\,(1 :: 2 :: \mathit{nil})] \\ C_2 \cdot C_1 \cdot (\texttt{CONS}(1, [\,])) & \texttt{\#} & [\mathit{visit}\,(2 :: \mathit{nil})] \\ C_2 \cdot C_1 \cdot (\texttt{CONS}(1, [\,])) \cdot (\texttt{CONS}(2, [\,])) & \texttt{\#} & [\mathit{visit}\,\mathit{nil}] \\ C_2 \cdot C_1 \cdot (\texttt{CONS}(1, [\,])) \cdot (\texttt{CONS}(2, [\,])) & \texttt{\#} & [\mathit{nil}] \\ C_2 \cdot C_1 \cdot (\texttt{CONS}(1, [\,])) & \texttt{\#} & [2 :: \mathit{nil}] \\ C_2 \cdot C_1 & \texttt{\#} & [1 :: 2 :: \mathit{nil}] \\ C_2 & \texttt{\#} & C_1[1 :: 2 :: \mathit{nil}] \end{array}$$

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 $\texttt{APP}(\mathit{visit}, [\,])$; concatenating it to $\texttt{CONS}(1, [\,])$ yields $\texttt{CONS}(1, \texttt{APP}(\mathit{visit}, [\,]))$, which is the second captured context:

$$\begin{array}{rcl} C_2 & \texttt{\#} & C_1[\langle \mathit{visit}\,(1 :: 2 :: \mathit{nil})\rangle] \\ C_2 \cdot C_1 & \texttt{\#} & [\mathit{visit}\,(1 :: 2 :: \mathit{nil})] \\ C_2 \cdot C_1 & \texttt{\#} & (\texttt{CONS}(1, [\,]))[\mathit{visit}\,(2 :: \mathit{nil})] \\ C_2 \cdot C_1 & \texttt{\#} & (\texttt{CONS}(2, \texttt{CONS}(1, [\,])))[\mathit{visit}\,\mathit{nil}] \\ C_2 \cdot C_1 & \texttt{\#} & (\texttt{CONS}(2, \texttt{CONS}(1, [\,])))[\mathit{nil}] \\ C_2 \cdot C_1 & \texttt{\#} & (\texttt{CONS}(2, [\,]))[1 :: \mathit{nil}] \\ C_2 \cdot C_1 & \texttt{\#} & [2 :: 1 :: \mathit{nil}] \\ C_2 & \texttt{\#} & C_1[2 :: 1 :: \mathit{nil}] \end{array}$$

6.4.7 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.

6.5 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 6.4, 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.

6.5.1 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 6.9 and 6.10. If $n = 1$, it coincides with the evaluator displayed in Figure 6.6.

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 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

- Terms ($1 \leq i \leq n$):

$$\mathsf{term} \ni t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid succ\ t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t$$

- Values: $\mathsf{val} \ni v ::= \ulcorner m \urcorner \mid f$

- Answers, continuations and functions ($1 \leq i \leq n$):

$$\begin{array}{rcll} & & \mathsf{ans} & = \mathsf{val} \\ k_{n+1} & \in & \mathsf{cont}_{n+1} & = \mathsf{val} \to \mathsf{ans} \\ k_i & \in & \mathsf{cont}_i & = \mathsf{val} \times \mathsf{cont}_{i+1} \times \ldots \times \mathsf{cont}_{n+1} \to \mathsf{ans} \\ f & \in & \mathsf{fun} & = \mathsf{val} \times \mathsf{cont}_1 \times \ldots \times \mathsf{cont}_{n+1} \to \mathsf{ans} \end{array}$$

- 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: $\mathsf{env} \ni e ::= e_{empty} \mid e[x \mapsto v]$

- Evaluation function: see Figure 6.10

Figure 6.9: An environment-based evaluator for the CPS hierarchy at level n

- Evaluation function ($1 \leq i \leq n$): $\mathsf{eval}_n \;:\; \mathsf{term} \times \mathsf{env} \times \mathsf{cont}_1 \times \ldots \times \mathsf{cont}_{n+1} \rightarrow \mathsf{ans}$

$$
\begin{array}{rcl}
\mathsf{eval}_n\ (\ulcorner m \urcorner,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\ (\ulcorner m \urcorner,\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (x,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\ (e(x),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (\lambda x.t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1\ (\lambda(v,\, k'_1,\, k'_2,\, \ldots,\, k'_{n+1}).\, \mathsf{eval}_n\ (t,\, e[x \mapsto v],\, k'_1,\, k'_2,\, \ldots,\, k'_{n+1}),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (t_0\, t_1,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\ (t_0,\, e, \\
& & \quad \lambda(f,\, k'_2,\, \ldots,\, k'_{n+1}).\, \mathsf{eval}_n\ (t_1,\, e, \\
& & \qquad \lambda(v,\, k''_2,\, \ldots,\, k''_{n+1}).\, f\ (v,\, k_1,\, k''_2,\, \ldots,\, k''_{n+1}), \\
& & \qquad k'_2,\, \ldots,\, k'_{n+1}), \\
& & \quad k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (\mathit{succ}\ t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\ (t,\, e,\, \lambda(\ulcorner m \urcorner,\, k'_2,\, \ldots,\, k'_{n+1}).\, k_1\ (\ulcorner m \urcorner + 1,\, k'_2,\, \ldots,\, k'_{n+1}),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (\langle t \rangle_i,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\ (t,\, e,\, \theta_1,\, \ldots,\, \theta_i,\, \lambda(v,\, k'_{i+2},\, \ldots,\, k'_{n+1}).\, k_1\ (v,\, k_2,\, \ldots,\, k_i,\, k_{i+1},\, k'_{i+2},\, \ldots,\, k'_{n+1}),\, k_{i+2},\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n\ (\mathcal{S}_i k.t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n\ (t,\, e[k \mapsto c_i],\, \theta_1,\, \ldots,\, \theta_i,\, k_{i+1},\, \ldots,\, k_{n+1})
\end{array}
$$

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: $\mathsf{evaluate}_n \;:\; \mathsf{term} \rightarrow \mathsf{val}$

$$\mathsf{evaluate}_n\,(t) \;=\; \mathsf{eval}_n\ (t,\, e_{\mathit{empty}},\, \theta_1,\, \ldots,\, \theta_n,\, \theta_{n+1})$$

Figure 6.10: An environment-based evaluator for the CPS hierarchy at level n, ctd.

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.

6.5.2 An environment-based abstract machine

Defunctionalizing the evaluator of Figures 6.9 and 6.10 yields the environment-based abstract machine displayed in Figures 6.11 and 6.12. If $n = 1$, it coincides with the abstract machine displayed in Figure 6.7.

The abstract machine consists of $n+2$ sets of transitions: *eval* for interpreting terms and $cont_1, \ldots, cont_{n+1}$ 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 6.7 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 e and a series of successive contexts, a reset$_i$ expression $\langle t \rangle_i$ is processed by evaluating t with the same environment e, i empty contexts, and a meta$_{i+1}$-context over which all the intermediate contexts have been pushed on.

Given an environment e and a series of successive contexts, a shift expression $\mathcal{S}_i k.t$ is processed by evaluating t with an extension of e where k denotes a

- Terms ($1 \leq i \leq n$): $t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t$
- Values ($1 \leq i \leq n$): $v ::= \ulcorner m \urcorner \mid [x,\, t,\, e] \mid C_i$
- Evaluation contexts ($2 \leq i \leq n+1$):

$$\begin{aligned} C_1 &::= [\,] \mid \mathtt{ARG}((t,e), C_1) \mid \mathtt{SUCC}\, C_1 \mid \mathtt{APP}(v, C_1) \\ C_i &::= [\,] \mid C_i \cdot C_{i-1} \end{aligned}$$

- Environments: $e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Initial transition, transition rules, and final transition: see Figure 6.12

Figure 6.11: An environment-based abstract machine for the CPS hierarchy at level n

- Initial transition, transition rules, and final transition ($1 \leq i \leq n,\ 2 \leq j \leq n$):

$$\begin{array}{rcl}
t & \Rightarrow & \langle t,\, e_{\mathit{empty}},\, [\,],\, [\,],\, \ldots,\, [\,]\rangle_{\mathit{eval}} \\[6pt]
\langle \ulcorner m \urcorner,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle x,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, e\,(x),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle \lambda x.t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, [x,\, t,\, e],\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle t_0\, t_1,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t_0,\, e,\, \mathtt{ARG}((t_1, e), C_1),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathit{succ}\ t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e,\, \mathtt{SUCC}\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \langle t\rangle_i,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e,\, [\,],\, \ldots,\, [\,],\, C_{i+1}\cdot(\ldots(C_2\cdot C_1)\ldots),\, C_{i+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathcal{S}_i k.t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, e[k \mapsto C_i\cdot(\ldots(C_2\cdot C_1)\ldots)],\, [\,],\, \ldots,\, [\,],\, C_{i+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\[6pt]
\langle [\,],\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_2,\, v,\, C_3,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_2}} \\
\langle \mathtt{ARG}((t, e), C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e,\, \mathtt{APP}(v, C_1),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathtt{SUCC}\, C_1,\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_1,\, \ulcorner m \urcorner + 1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle \mathtt{APP}([x,\, t,\, e], C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, e[x \mapsto v],\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathtt{APP}(C'_i\cdot(\ldots(C'_2\cdot C'_1)\ldots), C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C'_1,\, v,\, C'_2,\, \ldots,\, C'_i,\, C_{i+1}\cdot(C_i\cdot\ldots(C_2\cdot C_1)\ldots),\, C_{i+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\[6pt]
\langle [\,],\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}} & \Rightarrow & \langle C_{j+1},\, v,\, C_{j+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_{j+1}}} \\
\langle C_j\cdot(\ldots(C_2\cdot C_1)\ldots),\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}} & \Rightarrow & \langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\[6pt]
\langle C_{n+1}\cdot(\ldots(C_2\cdot C_1)\ldots),\, v\rangle_{\mathit{cont_{n+1}}} & \Rightarrow & \langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\[6pt]
\langle [\,],\, v\rangle_{\mathit{cont_{n+1}}} & \Rightarrow & v
\end{array}$$

Figure 6.12: An environment-based abstract machine for the CPS hierarchy at level n, ctd.

- Terms and values ($1 \leq i \leq n$):

$$\begin{aligned} t &::= v \mid x \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t \\ v &::= \ulcorner m \urcorner \mid \lambda x.t \mid C_i \end{aligned}$$

- Evaluation contexts ($2 \leq i \leq n+1$):

$$\begin{aligned} C_1 &::= [\,] \mid \texttt{ARG}(t, C_1) \mid \texttt{SUCC}\ C_1 \mid \texttt{APP}(v, C_1) \\ C_i &::= [\,] \mid C_i \cdot C_{i-1} \end{aligned}$$

- Initial transition, transition rules, and final transition: see Figure 6.14

Figure 6.13: A substitution-based abstract machine for the CPS hierarchy at level n

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 t to a value v whenever the machine for level n, started with t, stops with v. The correctness of the machine with respect to the evaluator is ensured by the following theorem:

Theorem 6.4. *For any program t and any integer value $\ulcorner m \urcorner$, $\mathsf{evaluate}_n\,(t) = \ulcorner m \urcorner$ if and only if $\mathsf{eval}^{\mathsf{e}}_n\,(t) = \ulcorner m \urcorner$.*

6.5.3 A substitution-based abstract machine

In the same fashion as in Section 6.4.3, we construct the substitution-based abstract machine corresponding to the environment-based abstract machine of Section 6.5.2. The result is displayed in Figures 6.13 and 6.14. If $n = 1$, it coincides with the abstract machine displayed in Figure 6.8.

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

$$C_{n+1} \,\texttt{\#}_n\, C_n \,\texttt{\#}_{n-1}\, C_{n-1} \,\texttt{\#}_{n-2} \,\cdots\, \texttt{\#}_2\, C_2 \,\texttt{\#}_1\, C_1[t],$$

where each $\texttt{\#}_i$ 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 $\mathsf{decompose}$ 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 ($1 \leq i \leq n,\ 2 \leq j \leq n$):

$$
\begin{array}{rcl}
t & \Rightarrow & \langle t,\, [\,],\, [\,],\, \ldots,\, [\,]\rangle_{\mathit{eval}} \\[6pt]
\langle \ulcorner m \urcorner,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle \lambda x.t,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, \lambda x.t,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle C'_i,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle C_1,\, C'_i,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle t_0\, t_1,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t_0,\, \mathtt{ARG}((t_1,\, e), C_1),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathit{succ}\ t,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, \mathtt{SUCC}\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \boldsymbol{\langle} t \boldsymbol{\rangle}_i,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle t,\, [\,],\, \ldots,\, [\,],\, C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathcal{S}_i k.t,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} & \Rightarrow & \langle k\{C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)/t\},\, [\,],\, \ldots,\, [\,],\, C_{i+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\[6pt]
\langle [\,],\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_2,\, v,\, C_3,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_2}} \\
\langle \mathtt{ARG}(t, C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle t,\, \mathtt{APP}(v, C_1),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathtt{SUCC}\, C_1,\, \ulcorner m \urcorner,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle C_1,\, \ulcorner \ulcorner m \urcorner + 1 \urcorner,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\
\langle \mathtt{APP}((\lambda x.t), C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} & \Rightarrow & \langle x\{v/t\},\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}} \\
\langle \mathtt{APP}(C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots), C_1),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{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_{\mathit{cont_1}} \\[6pt]
\langle [\,],\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}} & \Rightarrow & \langle C_{j+1},\, v,\, C_{j+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_{j+1}}} \\
\langle C_j \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}} & \Rightarrow & \langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\[6pt]
\langle C_{n+1} \cdot (\ldots (C_2 \cdot C_1) \ldots),\, v\rangle_{\mathit{cont_{n+1}}} & \Rightarrow & \langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}} \\[6pt]
\langle [\,],\, v\rangle_{\mathit{cont_{n+1}}} & \Rightarrow & v
\end{array}
$$

Figure 6.14: A substitution-based abstract machine for 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 6.5. *For any program t, either both $\mathsf{eval}^{\mathsf{s}}_n(t)$ and $\mathsf{eval}^{\mathsf{e}}_n(t)$ are undefined, or there exist values v, v' such that $\mathsf{eval}^{\mathsf{s}}_n(t) = v$, $\mathsf{eval}^{\mathsf{e}}_n(t) = v'$ and $\mathcal{T}_n(v') = v$.*

The definition of $\mathcal{T}_n$ extends that of $\mathcal{T}$ from Theorem 6.2 in such a way that it is homomorphic for all the contexts C_i, with $2 \leq i \leq n$.

6.5.4 A reduction semantics

Along the same lines as in Section 6.4.4, we construct the reduction semantics for the CPS hierarchy based on the abstract machine of Figures 6.13 and 6.14. 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:

$$\begin{array}{ll}
(\delta) & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[\mathit{succ}\ \ulcorner m \urcorner] \rightarrow_n C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[\ulcorner m+1 \urcorner] \\
(\beta_\lambda) & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[(\lambda x.t)\, v] \rightarrow_n C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[x\{v/t\}] \\
(\mathcal{S}^i_\lambda) & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[\mathcal{S}_i k.t] \rightarrow_n \\
 & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_{i+1}} C_{i+1} \mathbin{\#_i} [\,] \ldots \mathbin{\#_1} [k\{C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)/t\}] \\
(\beta^i_{ctx}) & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots)\, v] \rightarrow_n \\
 & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_{i+1}} C_{i+1} \cdot (\ldots (C_2 \cdot C_1) \ldots) \mathbin{\#_i} C'_i \mathbin{\#_{i-1}} \cdots \mathbin{\#_1} C'_1[v] \\
(\mathrm{Reset}^i) & C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[\langle v \rangle_i] \rightarrow_n C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[v]
\end{array}$$

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:

$$p_n ::= \mathit{succ}\ v \mid v_0\, v_1 \mid \mathcal{S}_i k.t \mid \langle v \rangle_i \quad (1 \leq i \leq n)$$

Lemma 6.2 (Unique decomposition for level n). *A program t 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 $t = \mathsf{plug}\,(C_{n+1} \mathbin{\#_n} \cdots \mathbin{\#_1} C_1[p_n])$.*

Evaluating a term using either the derived reduction rules or the substitution-based abstract machine from Section 6.5.3 yields the same result:

Theorem 6.6. *For any program t and any value v, $\mathsf{eval}^{\mathsf{s}}_n(t) = v$ if and only if $t \rightarrow^*_n v$, where $\rightarrow^*_n$ is the reflexive, transitive closure of $\rightarrow_n$.*

As in Section 6.4.4, using refocusing, one can go from a given reduction semantics of Section 6.5.4 into a pre-abstract machine and the corresponding eval/apply abstract machine of Figures 6.13 and 6.14.

6.5.5 Beyond CPS

As in Section 6.4.5, 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.

6.5.6 Static vs. dynamic delimited continuations

As in Section 6.4.6, 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]

6.5.7 Summary and conclusion

We have generalized the results presented in Section 6.4 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.

6.6 Programming in the CPS hierarchy

To finish, we present new examples of programming in the CPS hierarchy. The examples are normalization functions. In Sections 6.6.1 and 6.6.2, we first describe normalization by evaluation and we present the simple example of the free monoid. In Section 6.6.3, we present a function mapping a proposition into its disjunctive normal form; this normalization function uses delimited continuations. In Section 6.6.4, we generalize the normalization functions of Sections 6.6.2 and 6.6.3 to a hierarchical language of units and products, and we express the corresponding normalization function in the CPS hierarchy.

6.6.1 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 [79]:

$$\begin{array}{rcl} \mathit{eval} & : & \mathit{term} \to \mathit{value} \\ \mathit{reify} & : & \mathit{value} \to \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} & : & \mathit{term} \to \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} & = & \mathit{reify} \circ \mathit{eval} \end{array}$$

Normalization by evaluation has been developed in intuitionistic type theory [49, 135], proof theory [25, 26], category theory [6], and partial evaluation [59, 60], where it has emerged as a new field of application for delimited continuations [14, 60, 79, 91, 101, 108, 175].

6.6.2 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:

$$\begin{array}{rcl} t \star (t' \star t'') & \leftrightarrow & (t \star t') \star t'' \\ t \star \varepsilon & \leftrightarrow & t \\ \varepsilon \star t & \leftrightarrow & t \end{array}$$

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:

$$\begin{array}{rcl} t \star (t' \star t'') & \leftarrow & (t \star t') \star t'' \\ \varepsilon \star t & \rightarrow & t \end{array}$$

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:

$$\begin{array}{rcl} \mathit{eval}\, x & = & \lambda t. x \star^{\mathrm{nf}} t \\ \mathit{eval}\, \varepsilon & = & \lambda t. t \\ \mathit{eval}\, (t \star t') & = & (\mathit{eval}\, t) \circ (\mathit{eval}\, t') \\ \mathit{reify}\, v & = & v\, \varepsilon^{\mathrm{nf}} \\ \mathit{normalize}\, t & = & \mathit{reify}\, (\mathit{eval}\, t) \end{array}$$

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 [27, 49, 121]. 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.

6.6.3 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:

$$\mathit{term} \ni t ::= x \mid \mathit{true} \mid t \wedge t' \mid \mathit{false} \mid t \vee t'$$

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:

$$\begin{array}{rcl} \mathit{term}^{\mathrm{nf}} & \ni & \widehat{t} ::= d \\ \mathit{term}^{\mathrm{nf}}_{\mathrm{d}} & \ni & d ::= \mathit{false}^{\mathrm{nf}} \mid c \vee^{\mathrm{nf}} d \\ \mathit{term}^{\mathrm{nf}}_{\mathrm{c}} & \ni & c ::= \mathit{true}^{\mathrm{nf}} \mid x \wedge^{\mathrm{nf}} c \end{array}$$

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

$$\begin{array}{rcl} F_1 & = & \mathit{term}^{\mathrm{nf}}_{\mathrm{c}} \rightarrow \mathit{term}^{\mathrm{nf}}_{\mathrm{c}} \\ F_2 & = & \mathit{term}^{\mathrm{nf}}_{\mathrm{d}} \rightarrow \mathit{term}^{\mathrm{nf}}_{\mathrm{d}} \end{array}$$

the non-standard domain of values is ans_1, where

$$\begin{array}{rcl} \mathit{ans}_2 & = & F_2 \\ \mathit{ans}_1 & = & (F_1 \rightarrow \mathit{ans}_2) \rightarrow \mathit{ans}_2. \end{array}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{array}{rcl} \mathit{eval}_0\, x\, k\, d & = & k\,(\lambda c.x \wedge^{\mathrm{nf}} c)\, d \\ \mathit{eval}_0\, \mathit{true}\, k\, d & = & k\,(\lambda c.c)\, d \\ \mathit{eval}_0\,(t \wedge t')\, k\, d & = & \mathit{eval}_0\, t\,(\lambda f_1.\mathit{eval}_0\, t'\,(\lambda f_1'.k\,(f_1 \circ f_1')))\, d \\ \mathit{eval}_0\, \mathit{false}\, k\, d & = & d \\ \mathit{eval}_0\,(t \vee t')\, k\, d & = & \mathit{eval}_0\, t\, k\,(\mathit{eval}_0\, t'\, k\, d) \\ \\ \mathit{reify}_0\, v & = & v\,(\lambda f_1.\lambda d.(f_1\, \mathit{true}^{\mathrm{nf}}) \vee^{\mathrm{nf}} d)\, \mathit{false}^{\mathrm{nf}} \\ \\ \mathit{normalize}\, t & = & \mathit{reify}_0\,(\mathit{eval}_0\, t) \end{array}$$

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:

$$\begin{array}{rcl}
eval_1\; x\; d &=& (\lambda c.x \wedge^{\mathrm{nf}} c,\; d)\\
eval_1\; true\; d &=& (\lambda c.c,\; d)\\
eval_1\; (t \wedge t')\; d &=& let\; (f_1,\; d) = eval_1\; t\; d\\
&& in\; let\; (f_1',\; d) = eval_1\; t'\; d\\
&& \quad in\; (f_1 \circ f_1',\; d)\\
eval_1\; false\; d &=& \mathcal{S}k.d\\
eval_1\; (t \vee t')\; d &=& \mathcal{S}k.k\,(eval_1\; t\; \langle k\,(eval_1\; t'\; d)\rangle)\\
\\
reify_1\; v &=& \langle let\; (f_1,\; d) = v\; false^{\mathrm{nf}}\\
&& \;in\; (f_1\; true^{\mathrm{nf}}) \vee^{\mathrm{nf}} d\rangle\\
normalize\; t &=& reify_1\; (eval_1\; t)
\end{array}$$

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.

6.6.4 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:

$$\begin{array}{l}
term \ni t ::= x \mid \varepsilon_i \mid t \star_i t'\\
\qquad\qquad\qquad where\; 1 \leq i \leq n.
\end{array}$$

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 6.6.2.

Boolean logic: The language corresponds to that of propositions if $n = 2$, as in Section 6.6.3: 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 [98]; for example, in case $n = 4$, the language corresponds to that of Belnap's bilattice $\mathcal{FOUR}$ [19]. 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 [140]: 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 [90], 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:

$$\begin{array}{lcl} \mathit{term}^{\mathrm{nf}} & \ni & \widehat{t} \ ::= t_n \\ \mathit{term}^{\mathrm{nf}}_n & \ni & t_n ::= \varepsilon^{\mathrm{nf}}_n \ \mid \ t_{n-1} \star^{\mathrm{nf}}_n t_n \\ & & \quad\vdots \\ \mathit{term}^{\mathrm{nf}}_1 & \ni & t_1 ::= \varepsilon^{\mathrm{nf}}_1 \ \mid \ t_0 \star^{\mathrm{nf}}_1 t_1 \\ \mathit{term}^{\mathrm{nf}}_0 & \ni & t_0 ::= x \end{array}$$

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

$$\begin{array}{lcl} F_1 & = & \mathit{term}^{\mathrm{nf}}_1 \to \mathit{term}^{\mathrm{nf}}_1 \\ F_2 & = & \mathit{term}^{\mathrm{nf}}_2 \to \mathit{term}^{\mathrm{nf}}_2 \\ F_3 & = & \mathit{term}^{\mathrm{nf}}_3 \to \mathit{term}^{\mathrm{nf}}_3 \\ F_4 & = & \mathit{term}^{\mathrm{nf}}_4 \to \mathit{term}^{\mathrm{nf}}_4 \\ F_5 & = & \mathit{term}^{\mathrm{nf}}_5 \to \mathit{term}^{\mathrm{nf}}_5 \end{array}$$

the non-standard domain of values is ans_1, where

$$\begin{array}{lcl} \mathit{ans}_5 & = & F_5 \\ \mathit{ans}_4 & = & (F_4 \to \mathit{ans}_5) \to \mathit{ans}_5 \\ \mathit{ans}_3 & = & (F_3 \to \mathit{ans}_4) \to \mathit{ans}_4 \\ \mathit{ans}_2 & = & (F_2 \to \mathit{ans}_3) \to \mathit{ans}_3 \\ \mathit{ans}_1 & = & (F_1 \to \mathit{ans}_2) \to \mathit{ans}_2. \end{array}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{array}{rcl} \mathit{eval}_0\, x\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_1\, (\lambda t_1.x \star^{\mathrm{nf}}_1 t_1)\, k_2\, k_3\, k_4\, t_5 \\ \mathit{eval}_0\, \varepsilon_1\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_1\, (\lambda t_1.t_1)\, k_2\, k_3\, k_4\, t_5 \\ \mathit{eval}_0\, (t \star_1 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, (\lambda f_1.\mathit{eval}_0\, t'\, (\lambda f'_1.k_1\, (f_1 \circ f'_1)))\, k_2\, k_3\, k_4\, t_5 \\ \mathit{eval}_0\, \varepsilon_2\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_2\, (\lambda t_2.t_2)\, k_3\, k_4\, t_5 \\ \mathit{eval}_0\, (t \star_2 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, k_1 \\ & & \quad (\lambda f_2.\mathit{eval}_0\, t'\, k_1\, (\lambda f'_2.k_2\, (f_2 \circ f'_2)))k_3 k_4 t_5 \end{array}$$

$$
\begin{array}{rcl}
\mathit{eval}_0\,\varepsilon_3\,k_1\,k_2\,k_3\,k_4\,t_5 & = & k_3\,(\lambda t_3.t_3)\,k_4\,t_5 \\
\mathit{eval}_0\,(t \star_3 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & \mathit{eval}_0\,t\,k_1\,k_2 \\
 & & \quad (\lambda f_3.\mathit{eval}_0\,t'\,k_1\,k_2\,(\lambda f_3'.k_3\,(f_3 \circ f_3')))k_4 t_5 \\
\mathit{eval}_0\,\varepsilon_4\,k_1\,k_2\,k_3\,k_4\,t_5 & = & k_4\,(\lambda t_4.t_4)\,t_5 \\
\mathit{eval}_0\,(t \star_4 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & \mathit{eval}_0\,t\,k_1\,k_2 k_3 \\
 & & \quad (\lambda f_4.\mathit{eval}_0\,t'\,k_1\,k_2\,k_3\,(\lambda f_4'.k_4\,(f_4 \circ f_4')))t_5 \\
\mathit{eval}_0\,\varepsilon_5\,k_1\,k_2\,k_3\,k_4\,t_5 & = & t_5 \\
\mathit{eval}_0\,(t \star_5 t')\,k_1\,k_2\,k_3\,k_4\,t_5 & = & \mathit{eval}_0\,t\,k_1\,k_2\,k_3\,k_4\,(\mathit{eval}_0\,t'\,k_1\,k_2\,k_3\,k_4\,t_5)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{reify}_0\,v & = & v\,(\lambda f_1.\lambda k_2.k_2\,(\lambda t_2.(f_1\,\varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2)) \\
 & & \quad (\lambda f_2.\lambda k_3.k_3\,(\lambda t_3.(f_2\,\varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3)) \\
 & & \quad (\lambda f_3.\lambda k_4.k_4\,(\lambda t_4.(f_3\,\varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4)) \\
 & & \quad (\lambda f_4.\lambda t_5.(f_4\,\varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5) \\
 & & \quad \varepsilon_5
\end{array}
$$

$$
\mathit{normalize}\,t = \mathit{reify}_0\,(\mathit{eval}_0\,t)
$$

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:

$$
\begin{array}{rcl}
\mathit{eval}_2\,x\,k_1\,k_2\,t_5 & = & k_1\,(\lambda t_1.x \star_1^{\mathrm{nf}} t_1)\,k_2\,t_5 \\
\mathit{eval}_2\,\varepsilon_1\,k_1\,k_2\,t_5 & = & k_1\,(\lambda t_1.t_1)\,k_2\,t_5 \\
\mathit{eval}_2\,(t \star_1 t')\,k_1\,k_2\,t_5 & = & \mathit{eval}_2\,t\,(\lambda f_1.\mathit{eval}_2\,t'\,(\lambda f_1'.k_1\,(f_1 \circ f_1')))\,k_2\,t_5 \\
\mathit{eval}_2\,\varepsilon_2\,k_1\,k_2\,t_5 & = & k_2\,(\lambda t_2.t_2)\,t_5 \\
\mathit{eval}_2\,(t \star_2 t')\,k_1\,k_2\,t_5 & = & \mathit{eval}_2\,t\,k_1\,(\lambda f_2.\mathit{eval}_2\,t'\,k_1\,(\lambda f_2'.k_2\,(f_2 \circ f_2')))\,t_5 \\
\mathit{eval}_2\,\varepsilon_3\,k_1\,k_2\,t_5 & = & (\lambda t_3.t_3,\, t_5) \\
\mathit{eval}_2\,(t \star_3 t')\,k_1\,k_2\,t_5 & = & \mathit{let}\,(f_3,\, t_5) = \mathit{eval}_2\,t\,k_1\,k_2\,t_5 \\
 & & \mathit{in}\ \mathit{let}\,(f_3',\, t_5) = \mathit{eval}_2\,t'\,k_1\,k_2\,t_5 \\
 & & \quad \mathit{in}\,(f_3 \circ f_3',\, t_5) \\
\mathit{eval}_2\,\varepsilon_4\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.(\lambda t_4.t_4,\, t_5) \\
\mathit{eval}_2\,(t \star_4 t')\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.\mathit{let}\,(f_4,\, t_5) = \langle k_3\,(\mathit{eval}_2\,t\,k_1\,k_2\,t_5)\rangle_1 \\
 & & \qquad \mathit{in}\ \mathit{let}\,(f_4',\, t_5) = \langle k_3\,(\mathit{eval}_2\,t'\,k_1\,k_2\,t_5)\rangle_1 \\
 & & \qquad\quad \mathit{in}\,(f_4 \circ f_4',\, t_5) \\
\mathit{eval}_2\,\varepsilon_5\,k_1\,k_2\,t_5 & = & \mathcal{S}_2 k_4.t_5 \\
\mathit{eval}_2\,(t \star_5 t')\,k_1\,k_2\,t_5 & = & \mathcal{S}_1 k_3.\mathcal{S}_2 k_4.\mathit{let}\,t_5 = \langle k_4\,\langle k_3\,(\mathit{eval}_2\,t'\,k_1\,k_2\,t_5)\rangle_1\rangle_2 \\
 & & \qquad\qquad \mathit{in}\,\langle k_4\,\langle k_3\,(\mathit{eval}_2\,t\,k_1\,k_2\,t_5)\rangle_1\rangle_2
\end{array}
$$

$$\begin{array}{rcl}
\mathit{reify}_2\, v & = & \langle \mathit{let}\ (f_4,\, t_5) = \langle \mathit{let}\ (f_3,\, t_5) \\
& & \qquad\qquad\qquad\qquad = v\,(\lambda f_1.\lambda k_2.k_2\,(\lambda t_2.(f_1\,\varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2)) \\
& & \qquad\qquad\qquad\qquad\quad\ (\lambda f_2.\lambda t_3.(f_2\,\varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3) \\
& & \qquad\qquad\qquad\qquad\quad\ \varepsilon_5 \\
& & \qquad\qquad\qquad\quad \mathit{in}\ (\lambda f_4.(f_3\,\varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4,\, t_5)\rangle_1 \\
& & \qquad \mathit{in}\ (f_4\,\varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5\rangle_2
\end{array}$$

$$\mathit{normalize}\ t \ = \ \mathit{reify}_2\,(\mathit{eval}_2\ t)$$

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{array}{rcl}
\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 \\
& & \qquad\quad \mathit{in}\ \mathit{let}\ (f_2',\, t_5) = \langle k_1\,(\mathit{eval}_4\, t'\, t_5)\rangle_1 \\
& & \qquad\qquad \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 \\
& & \qquad\qquad\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 \\
& & \qquad\qquad\qquad \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 \\
& & \qquad\qquad\qquad\quad \mathit{in}\ \mathit{let}\ (f_4',\, t_5) \\
& & \qquad\qquad\qquad\qquad = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\, t'\, t_5)\rangle_1\rangle_2\rangle_3 \\
& & \qquad\qquad\qquad\qquad \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\qquad\qquad\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\qquad\qquad\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{array}$$

$$\begin{array}{rcl}
\mathit{reify}_4\, v & = & \langle \mathit{let}\ (f_4,\, t_5) = \langle \mathit{let}\ (f_3,\, t_5) = \langle \mathit{let}\ (f_2,\, t_5) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad = \langle \mathit{let}\ (f_1,\, t_5) = v\,\varepsilon_5 \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ (\lambda f_2.(f_1\,\varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2,\, t_5)\rangle_1 \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ (\lambda f_3.(f_2\,\varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3,\, t_5)\rangle_2 \\
& & \qquad\qquad\quad \mathit{in}\ (\lambda f_4.(f_3\,\varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4,\, t_5)\rangle_3 \\
& & \mathit{in}\ (f_4\,\varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5\rangle_4
\end{array}$$

$$\mathit{normalize}\ t \ = \ \mathit{reify}_4\,(\mathit{eval}_4\ t)$$

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.

6.6.5 A note about efficiency

We have implemented all the definitions of Section 6.6.4 as well as the intermediate versions $eval_1$ and $eval_3$ in ML [73]. We have also implemented hierarchical normalization functions for other values than 5.

For high products (i.e., in Section 6.6.4, 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 6.6.4, 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 [131], 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.

6.6.6 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 $reflect_i$ 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

$$\begin{aligned} \mathit{eval}_4\,(t \star_4 t')\,t_5 \;=\; & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathit{let}\,(f_4,\,t_5) \\ & \qquad = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\rangle_2\rangle_3 \\ & \qquad \mathit{in}\ \mathit{let}\,(f_4',\,t_5) \\ & \qquad\quad = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\rangle_2\rangle_3 \\ & \qquad\quad \mathit{in}\ (f_4 \circ f_4',\,t_5) \end{aligned}$$

one could write

$$\begin{aligned} \mathit{eval}_4\,(t \star_4 t')\,t_5 \;=\; & \mathcal{L}_3\,(k_1,\,k_2,\,k_3).\mathit{let}\,(f_4',\,t_5) \\ & \qquad = \mathit{reflect}_3\,(\mathit{eval}_4\,t\,t_5,\,k_1,\,k_2,\,k_3) \\ & \qquad \mathit{in}\ \mathit{let}\,(f_4',\,t_5) \\ & \qquad\quad = \mathit{reflect}_3\,(\mathit{eval}_4\,t'\,t_5,\,k_1,\,k_2,\,k_3) \\ & \qquad\quad \mathit{in}\ (f_4 \circ f_4',\,t_5). \end{aligned}$$

Such alternative control operators can be more convenient to use, while being compatible with CPS.

6.7 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 nontrivial 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 6.4.5).

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 6.6.

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 7

Program extraction from proofs of weak head normalization

with Olivier Danvy and Kristian Støvring [31]

Abstract

We formalize two proofs of weak head normalization for the simply typed lambda-calculus in first-order minimal logic: one for normal-order reduction, and one for applicative-order reduction in the object language. Subsequently we use Kreisel's modified realizability to extract evaluation algorithms from the proofs, following Berger; the proofs are based on Tait-style reducibility predicates, and hence the extracted algorithms are instances of (weak head) normalization by evaluation, as already identified by Coquand and Dybjer.

7.1 Introduction and related work

In the early nineties, Berger and Schwichtenberg introduced normalization by evaluation in a proof-theoretic setting [26]. Berger then substantiated their normalization function by extracting it from a proof of strong normalization [21], using Kreisel's modified realizability interpretation [123]. In their own study of what also turned out to be normalization by evaluation [48, 49], Coquand and Dybjer constructed normalization functions interpreting source terms in so-called glueing models. They also outlined a process of "program extraction" with which their normalization algorithms can be obtained from simple instances of a normalization proof due to Martin-Löf, and noticed the connection with Berger's work. In this article, we use part of Berger's framework to formalize some of the relationship identified by Coquand and Dybjer between glueing models and Tait-style proofs as used by Martin-Löf. We consider two intuitionistic proofs of weak head normalization for the simply typed λ-calculus: A normal-order proof essentially due to Martin-Löf, and an applicative-order counterpart due to Hofmann [147, page 152].

Our results can be described informally as follows: Applying modified realizability to the definition of the Tait-style reducibility predicate gives the definition of a glueing model. Applying modified realizability to the proof of normalization of a particular simply typed term t gives a λ-term denoting the interpretation of t in this glueing model.

The program extraction we perform can be intuitively explained as a "program optimization" [48]: Martin-Löf's normalization proof is formalized in an intuitionistic meta-language, and such a proof can informally be regarded as a function returning the normal form, together with a proof that this result actually is a normal form [48, 79]. To go from such a normalization proof to a function returning *only* the normal form, one can then remove the redundant parts representing the axioms for convertibility, and simplify the types accordingly [48]. That is how Berger's use of the modified realizability interpretation works, in the setting of first-order logic: the axioms for convertibility can be stated as *Harrop formulas*, and subproofs which are proofs of Harrop formulas disappear during the extraction.

Coquand and Dybjer's weak normalization function for the λ-calculus can be perceived as an optimized version of the program we extract in the applicative-order case. This is not surprising, since our focus here is on formalizing the proofs and considering two different evaluation strategies in the object language rather than optimizing the extracted programs.

Our account has the following limitations:

- Like Berger, we only partially formalize the normalization proof. Since a part of the proof is performed at the meta-level, we cannot formally extract a normalization function, but only a λ-term denoting the glueing interpretation of t for every *particular* term t.

- For simplicity, we only consider normalization of closed terms. With this restriction, we do not need to formalize renaming of bound variables.

- We only treat the case of the simply typed λ-calculus with one uninterpreted base type, whereas Coquand and Dybjer consider a variety of more advanced examples.

In the remainder of this article, we first review the modified realizability interpretation (Section 7.2); we then specify the problem of weak head normalization for the λ-calculus and we extract a call-by-name λ-interpreter and then a call-by-value λ-interpreter (Section 7.3). ML implementations of the extracted normalization programs are presented in Appendix 7.A.

7.2 Preliminaries

We begin by reviewing the techniques used by Berger to extract normalization functions from proofs [21]. The key concept is Kreisel's *modified realizability* proof interpretation [123]. Our presentation is based on Berger's article [21] and Troelstra's treatise [181].

7.2.1 First-order minimal logic

We formalize the normalization proofs in a first-order logic $\mathbf{M_1}$. The language of $\mathbf{M_1}$ is that of many-sorted first-order minimal logic with conjunction, defined in a standard way. Specifically, such a language is given by:

- Sorts ι, ι_1, ι_2, ...
- Constants $\mathtt{c}^\iota$, function symbols $\mathtt{f}^{\iota_1\times\cdots\times\iota_n\to\iota}$.
- Predicate symbols $\mathbf{P}^{\iota_1\times\cdots\times\iota_n}$.

(We will see that the sorts of $\mathbf{M_1}$ are the base types of the extracted programs.) The terms and formulas of $\mathbf{M_1}$ are:

$$\begin{array}{ll} \textit{Terms} & t^\iota := x^\iota \mid \mathtt{c}^\iota \mid \mathtt{f}^{\iota_1\times\cdots\times\iota_n\to\iota}(t_1^{\iota_1},\ldots,t_n^{\iota_n}) \\ \textit{Formulas} & \varphi,\psi := \mathbf{P}^{\iota_1\times\cdots\times\iota_n}(t_1^{\iota_1},\ldots,t_n^{\iota_n}) \mid \varphi\wedge\psi \mid \varphi\to\psi \mid \forall x^\iota.\,\varphi \mid \exists x^\iota.\,\varphi \end{array}$$

A natural deduction proof system of $\mathbf{M_1}$ is shown in Figure 7.1. Instead of presenting the proof rules graphically, we directly define a proof of a formula φ to be a dependently typed λ-term d^φ. In the definition, $FV(\varphi)$ denotes the set of free variables in the formula φ, while $\mathrm{FA}(d)$ denotes the set of free assumptions in the proof d. Only the interesting defining cases of $\mathrm{FA}(d)$ are shown.

We will also use the notation $u_1:\psi_1,\ldots,u_n:\psi_n \vdash_{\mathbf{M_1}} d:\varphi$ to mean that d^φ is an $\mathbf{M_1}$-proof of φ with free assumptions contained in the set $\{u_1^{\psi_1},\ldots,u_n^{\psi_n}\}$.

7.2.2 Modified realizability

In the presentation we use one of Troelstra's variants of modified realizability [181, p. 218].

The programs extracted from proofs are terms of the simply typed λ-calculus with product and unit types, and with the sorts of $\mathbf{M_1}$ as base types:

$$\begin{array}{ll} \textit{Types} & \sigma := 1 \mid \iota_1 \mid \iota_2 \mid \ldots \mid \sigma_1\to\sigma_2 \mid \sigma_1\times\sigma_2 \\ \textit{Terms} & \mathrm{t} := x^\sigma \mid \mathrm{t}_1\,\mathrm{t}_2 \mid \lambda x^\sigma.\mathrm{t} \mid \mathrm{fst}\,\mathrm{t} \mid \mathrm{snd}\,\mathrm{t} \mid (\mathrm{t}_1,\mathrm{t}_2) \mid * \mid \mathtt{c} \mid \mathtt{f}(\mathrm{t}_1,\ldots,\mathrm{t}_n) \end{array}$$

Note that the language of λ-terms includes the constants and function symbols of $\mathbf{M_1}$. Moreover, meta-variables ranging over λ-terms are denoted with the Roman font (t), and thus differ from the notation for logical terms in $\mathbf{M_1}$ (t).

In the following, by a "program" we mean a simply typed λ-term as just defined. Only in Appendix 7.A are actual programming language implementations considered.

Definition 7.1 (Program extraction). *Given an $\mathbf{M_1}$-proof d of φ, we define*

$$
\begin{array}{ll}
(\mathit{ass.}) & u^{\varphi} \\
(\to^{+}) & (\lambda u^{\varphi}.d^{\psi})^{\varphi\to\psi} \\
(\to^{-}) & (d^{\varphi\to\psi}\, e^{\varphi})^{\psi} \\
(\wedge^{+}) & (d^{\varphi}, e^{\psi})^{\varphi\wedge\psi} \\
(\wedge^{-}_{1}) & (\mathrm{fst}\, d^{\varphi\wedge\psi})^{\varphi} \\
(\wedge^{-}_{2}) & (\mathrm{snd}\, d^{\varphi\wedge\psi})^{\psi} \\
(\forall^{+}) & (\lambda x^{\iota}.d^{\varphi})^{\forall x^{\iota}.\,\varphi} \\
 & (\text{provided } x^{\iota} \notin FV\,(\psi) \text{ for every } u^{\psi} \in \mathrm{FA}(d)) \\
(\forall^{-}) & (d^{\forall x^{\iota}.\,\varphi}\, t^{\iota})^{\varphi\,[t/x]} \\
(\exists^{+}) & \langle t, d^{\varphi\,[t/x]}\rangle^{\exists x.\,\varphi} \\
(\exists^{-}) & [e^{\exists x.\,\varphi}, u^{\varphi}.d^{\psi}]^{\psi} \\
 & (\text{provided } x \notin FV\,(\psi), \\
 & \text{and } x \notin FV\,(\chi) \text{ for every } v^{\chi} \in \mathrm{FA}(d) \setminus \{u^{\varphi}\})
\end{array}
$$

$$
\begin{array}{lll}
\text{where} & \mathrm{FA}(u^{\varphi}) & = \{u^{\varphi}\} \\
 & \mathrm{FA}((\lambda u^{\varphi}.d^{\psi})^{\varphi\to\psi}) & = \mathrm{FA}(d^{\psi}) \setminus \{u^{\varphi}\} \\
 & \mathrm{FA}([e^{\exists x.\,\varphi}, u^{\varphi}.d^{\psi}]^{\psi}) & = \mathrm{FA}(e^{\exists x.\,\varphi}) \cup (\mathrm{FA}(d^{\psi}) \setminus \{u^{\varphi}\}) \\
 & & \text{etc.}
\end{array}
$$

Figure 7.1: The proof system $\mathbf{M_1}$

a type $\tau(d)$ and a λ-term $\llbracket d\rrbracket$ of type $\tau(d)$ as follows:

$$
\begin{array}{lll@{\qquad}lll}
\tau(\mathbf{P}(t_1,\ldots,t_n)) & := & 1 & \llbracket u^{\varphi}\rrbracket & := & x_u^{\tau(\varphi)} \\
\tau(\varphi\wedge\psi) & := & \tau(\varphi)\times\tau(\psi) & \llbracket \lambda u^{\varphi}.d^{\psi}\rrbracket & := & \lambda x_u^{\tau(\varphi)}.\llbracket d\rrbracket \\
\tau(\varphi\to\psi) & := & \tau(\varphi)\to\tau(\psi) & \llbracket d^{\varphi\to\psi}\, e^{\varphi}\rrbracket & := & \llbracket d\rrbracket\,\llbracket e\rrbracket \\
\tau(\forall x^{\iota}.\,\varphi) & := & \iota\to\tau(\varphi) & \llbracket (d^{\varphi}, e^{\psi})\rrbracket & := & (\llbracket d\rrbracket, \llbracket e\rrbracket) \\
\tau(\exists x^{\iota}.\,\varphi) & := & \iota\times\tau(\varphi) & \llbracket \mathrm{fst}\, d^{\varphi\wedge\psi}\rrbracket & := & \mathrm{fst}\,\llbracket d\rrbracket \\
 & & & \llbracket \mathrm{snd}\, d^{\varphi\wedge\psi}\rrbracket & := & \mathrm{snd}\,\llbracket d\rrbracket \\
 & & & \llbracket \lambda x^{\iota}.d^{\varphi}\rrbracket & := & \lambda x^{\iota}.\llbracket d\rrbracket \\
 & & & \llbracket d^{\forall x^{\iota}.\,\varphi}\, t^{\iota}\rrbracket & := & \llbracket d\rrbracket\, t \\
 & & & \llbracket \langle t, d^{\varphi\,[t/x]}\rangle\rrbracket & := & (t, \llbracket d\rrbracket) \\
 & & & \llbracket [e^{\exists x.\,\varphi}, u^{\varphi}.d^{\psi}]\rrbracket & := & \llbracket d\rrbracket[\mathrm{fst}\,\llbracket e\rrbracket/x, \mathrm{snd}\,\llbracket e\rrbracket/x_u]
\end{array}
$$

Subsequently, we simplify the extracted terms using the isomorphisms $A\times 1\cong A$,

$1 \times B \cong B$, $A \to 1 \cong 1$, and $1 \to B \cong B$.[1] This means that the type $\tau(\varphi)$ of an extracted term will either be 1 or not contain 1 at all. The first case happens exactly when φ is a *Harrop formula*—we then informally say that φ "has no computational content."

7.2.2.1 Soundness of the extraction

We now briefly consider in what sense a λ-term extracted from a proof of φ "realizes" φ. The notion of realizability is formalized in a finite-type extension $\mathbf{M_1^-}(\lambda)$ of $\mathbf{M_1}$ [21]. The point is that every extracted term $[\![d]\!]$ is a term of $\mathbf{M_1^-}(\lambda)$.

Definition 7.2 (Modified realizability). *By induction on the* $\mathbf{M_1}$*-formula* φ *we define an* $\mathbf{M_1^-}(\lambda)$*-formula* $\mathsf{t}^{\tau(\varphi)}\ \mathsf{mr}\ \varphi$ *as follows:*

$$\begin{aligned}
\mathsf{t}^{1}\ \mathsf{mr}\ \mathbf{P}(\mathsf{t}_1,\ldots,\mathsf{t}_n) &:= \mathbf{P}(\mathsf{t}_1,\ldots,\mathsf{t}_n)\\
\mathsf{t}^{\sigma_1\times\sigma_2}\ \mathsf{mr}\ \varphi\wedge\psi &:= (\mathrm{fst}\,\mathsf{t})\ \mathsf{mr}\ \varphi\ \wedge\ (\mathrm{snd}\,\mathsf{t})\ \mathsf{mr}\ \psi\\
\mathsf{t}^{\sigma_1\to\sigma_2}\ \mathsf{mr}\ \varphi\to\psi &:= \forall y^{\sigma_1}.\,(y\ \mathsf{mr}\ \varphi \to \mathsf{t}\,y\ \mathsf{mr}\ \psi)\\
\mathsf{t}^{\iota\to\sigma}\ \mathsf{mr}\ \forall z^{\iota}.\,\varphi(z) &:= \forall z^{\iota}.\,\mathsf{t}\,z\ \mathsf{mr}\ \varphi(z)\\
\mathsf{t}^{\iota\times\sigma}\ \mathsf{mr}\ \exists z^{\iota}.\,\varphi(z) &:= (\mathrm{snd}\,\mathsf{t})\ \mathsf{mr}\ \varphi(\mathrm{fst}\,\mathsf{t})
\end{aligned}$$

Given an $\mathbf{M_1}$-proof d of φ, the goal is therefore to give an $\mathbf{M_1^-}(\lambda)$-proof of $[\![d]\!]\ \mathsf{mr}\ \varphi$. It turns out that the proof d is allowed to contain free assumptions of Harrop formulas.

Theorem 7.1 (Soundness of modified realizability). *Let* $\psi_1,\ldots,\psi_n$ *be Harrop formulas. If* $u_1 : \psi_1,\ldots,u_n : \psi_n \vdash_{\mathbf{M_1}} d : \varphi$*, then* $\psi_1,\ldots,\psi_n \vdash_{\mathbf{M_1^-}(\lambda)} [\![d]\!]\ \mathsf{mr}\ \varphi$*.*

Proof. Standard [21, 181]. □

As an example, suppose that d is a $\mathbf{M_1}$-proof of $\forall x^{\iota_1}.\,\exists y^{\iota_2}.\,\mathbf{P}(x,y)$ containing only Harrop formulas as free assumptions. Then Theorem 7.1 gives an $\mathbf{M_1^-}(\lambda)$-proof of $\forall x^{\iota_1}.\,\mathbf{P}(x,\mathrm{fst}([\![d]\!]\,x))$ from the same free assumptions. In this way, free Harrop assumptions can be thought of as "axioms" with no effect on the extracted program.

7.2.2.2 Eliminating computationally redundant variables

The extraction procedure can be refined in order to keep the resulting programs simple. We present a refinement due to Berger [21, 22] which allows computationally redundant universal variables to be eliminated from the extracted program. To this end, we add a new kind of formulas of the form $\{\forall x^{\iota}\}.\,\varphi$ with

[1]In the original version of modified realizability [123], as well as in newer variants [24], this "optimization" is built-in. We use the simpler version for presentational purposes.

the following introduction and elimination rules:

$$(\forall^+) \quad (\{\lambda x^\iota\}.d^\varphi)^{\{\forall x^\iota\}.\,\varphi}$$

(provided $x^\iota \notin FV(\psi)$ for every $u^\psi \in \mathrm{FA}(d)$) and $x \notin \mathrm{CV}(d)$)

$$(\forall^-) \quad (d^{\{\forall x^\iota\}.\,\varphi}\,\{t^\iota\})^{\varphi\,[t/x]},$$

where the set of computationally relevant variables $\mathrm{CV}(d)$ is defined as the set of all variables occurring free in a witness for an existential quantifier, or in any term instantiating a universal quantifier in d. A universally quantified variable is called redundant if it is not computationally relevant.

The type of realizers for the new formulas simply ignores the redundant variable: $\tau(\{\forall x^\iota\}.\,\varphi) := \tau(\varphi)$. The corresponding clause for modified realizability is $\mathsf{t}\,\mathsf{mr}\,\{\forall x^\iota\}.\,\varphi := \forall x^\iota.\,\mathsf{t}\,\mathsf{mr}\,\varphi$ (with $x \notin FV(\mathsf{t})$). As desired, the extracted program does not contain the redundant variable:

$$\begin{aligned} [\![\{\lambda x^\iota\}.d]\!] &:= [\![d]\!] \\ [\![d\,\{t\}]\!] &:= [\![d]\!] \end{aligned}$$

The proof of soundness of modified realizability can be extended to handle this case [21].

7.3 Weak head normalization

We now specify the problem of weak head normalization for the λ-calculus. In the presentation, we assume that all terms are well-typed, but for clarity we omit all typing annotations. We consider only closed terms.

By normalization we understand the process of reducing a term to a normal form, where the basic reduction step is β-reduction [16]:

$$(\lambda x.t)\,s \to t\,[s/x].$$

The compatible closure of β-reduction yields the one-step reduction relation.

Weak head normalization is a restricted form of normalization producing terms in weak head normal form, which—for closed terms—stops at a λ-abstraction, without normalizing its body. Therefore any λ-abstraction is in weak head normal form.

We consider two deterministic restrictions of the one-step reduction that lead to weak head normal forms: the normal-order and applicative-order reduction strategies. Since weak head normalization is closely related to *evaluation* in the λ-calculus regarded as a programming language, where computations are not performed under λ-abstractions, we also refer to the above reduction strategies as the call-by-name and call-by-value evaluation strategies, respectively [148].

Definition 7.3 (Normal-order reduction). *The normal-order reduction strategy is obtained from one-step reduction by restricting it to the following rules:*

$$(\beta) \quad (\lambda x.t_1)\,t_2 \to t_1\,[t_2/x]$$

$$(\nu) \quad \frac{t_1 \to t_1'}{t_1\,t_2 \to t_1'\,t_2}$$

Definition 7.4 (Applicative-order reduction). *The (left-to-right) applicative-order reduction strategy is obtained from one-step reduction by restricting it to the following rules:*

$$\begin{array}{lll}
(\beta_{\mathrm{v}}) & (\lambda x.t_1)\,t_2 \to t_1\,[t_2/x] & \textit{if } t_2 \textit{ is a value} \\
(\nu) & \dfrac{t_1 \to t_1'}{t_1\,t_2 \to t_1'\,t_2} & \\
(\mu_{\mathrm{v}}) & \dfrac{t_2 \to t_2'}{t_1\,t_2 \to t_1\,t_2'} & \textit{if } t_1 \textit{ is a value}
\end{array}$$

Values are λ-abstractions.

These specifications of evaluation strategies can be axiomatized directly in the logic $\mathbf{M_1}$ using only Harrop formulas, as will be shown in the following sections.

The theorem we want to prove can be stated informally as follows:

Theorem 7.2 (Weak head normalization). *The process of reducing a closed well-typed λ-term according to either of the above strategies terminates with a (weak head) normal form.*

The proof proceeds by first defining a suitable logical relation on well-typed closed terms that implies the desired property. Next we show that every well-typed term satisfies this relation. Obviously, the exact shape of the proof relies on the chosen reduction strategy (normal-order or applicative-order), and consequently the extracted program produces the result according to the corresponding strategy in the object language (call by name or call by value).

In the rest of the section we first formalize this theorem for the two evaluation strategies, and then we use modified realizability to extract the underlying programs. In the call-by-value case, our development formalizes and extends the proof of normalization for call-by-value evaluation presented in Pierce's book [147, pp. 149-152].

7.3.1 The object language

We consider an explicitly typed version of the simply typed λ-calculus with variables contained in a countable set $V = x_1^{T_1}, x_2^{T_2}, \ldots$ (infinitely many of each type). This language is now encoded in a first-order minimal logic. The variables are used to index the sorts and constants of the logic, which is given by the following:

- Sorts: For every type T and finite set of variables X, we have the sort Λ_T^X of object-level λ-terms of type T containing exactly free variables X.
- Constants: The λ-term constructors are:

$$\begin{array}{lll}
\mathtt{VAR}_x & : \Lambda_T^{\{x\}} & \text{(for each variable } x^T\text{)} \\
\mathtt{LAM}_{x,T_1,T_2,X} & : \Lambda_{T_2}^X \to \Lambda_{T_1\to T_2}^{X\setminus\{x\}} & \text{(where } x \text{ has type } T_1\text{)} \\
\mathtt{APP}_{T_1,T_2,X,Y} & : \Lambda_{T_1\to T_2}^X \to \Lambda_{T_1}^Y \to \Lambda_{T_2}^{X\cup Y} &
\end{array}$$

- Predicate symbols: the set of predicate symbols differs for call-by-name and call-by-value evaluation, and we specify each of them in Section 7.3.2 and Section 7.3.3, respectively.

Notation. For the sake of presentation, we use a number of notational abbreviations when constructing object terms, e.g., we omit type annotations from λ-term constructors—in most cases they can be inferred from the context; we use the "uncurried" versions of the term constructors; we also write $\texttt{LAM}\, x_i.\, t$ instead of $\texttt{LAM}_{x_i,T_1,T_2,X}(t)$, and $\texttt{VAR}\, x_i$ instead of $\texttt{VAR}_{x_i}$.

We abbreviate sorts of closed terms $\Lambda_T^{\emptyset}$ as Λ_T. In the formulas used in the rest of this article, we only quantify over sorts of closed terms.

We treat substitution in λ-terms at the meta level. For a variable $x_i^{T_1}$ and logical terms $s^{\Lambda_{T_1}}$ and $t^{\Lambda_{T_2}}$, we define $t\,[s/\texttt{VAR}\, x_i]$ as t with every subterm $\texttt{VAR}\, x_i$ not in scope of a $\texttt{LAM}_{x_i}$ replaced by s. As Λ_{T_1} is a sort of closed λ-terms, free object-level variables are never captured as a result of this form of substitution. For this definition of $t\,[s/\texttt{VAR}\, x_i]$ to faithfully encode substitution, we further require that all free logical variables in t range over sorts of closed object-level terms. Thus the formal definition of substitution is as follows:

Definition 7.5. *Let $x_i^{T_1}$ be a variable, and let $s^{\Lambda_{T_1}^{\emptyset}}$ and $t^{\Lambda_{T_2}^{X}}$ be logical terms such that all free logical variables in t belong to (possibly different) sorts $\Lambda_T^{\emptyset}$ of closed object-level terms. We define the term $t\,[s/\texttt{VAR}\, x_i]$ of sort $\Lambda_{T_2}^{X\setminus\{x_i\}}$ inductively:*

$$\begin{aligned}
y^{\Lambda_T^{\emptyset}}\,[s/\texttt{VAR}\, x_i] &= y \qquad \textit{(where } y \textit{ is a logical variable)}\\
\texttt{VAR}\, x_i\,[s/\texttt{VAR}\, x_i] &= s\\
\texttt{VAR}\, x_j\,[s/\texttt{VAR}\, x_i] &= \texttt{VAR}\, x_j \qquad (j \neq i)\\
\texttt{APP}(t_1, t_2)\,[s/\texttt{VAR}\, x_i] &= \texttt{APP}(t_1\,[s/\texttt{VAR}\, x_i], t_2\,[s/\texttt{VAR}\, x_i])\\
(\texttt{LAM}_{x_i,X}\, t_1)\,[s/\texttt{VAR}\, x_i] &= \texttt{LAM}_{x_i,X}\, t_1\\
(\texttt{LAM}_{x_j,X}\, t_1)\,[s/\texttt{VAR}\, x_i] &= \texttt{LAM}_{x_j,X\setminus\{x_i\}}\,(t_1\,[s/\texttt{VAR}\, x_i]) \qquad (j \neq i)
\end{aligned}$$

7.3.2 Call-by-name evaluation

First, we give an axiomatization of call-by-name evaluation in the λ-calculus. We use two primitive predicates: $\mathbf{Ev}(t, s)$, understood as "t evaluates to s," and $\mathbf{Rd}(t, s)$, understood as "t reduces to s in one step." The process of call-by-name evaluation is defined through the following axioms:

$$\begin{aligned}
&(\mathrm{A}_1) \quad \{\forall t_2\}.\, \mathbf{Rd}(\texttt{APP}(\texttt{LAM}\, \boldsymbol{x_i}.\, \boldsymbol{t}, t_2), \boldsymbol{t}\,[t_2/\texttt{VAR}\, \boldsymbol{x_i}])\\
&(\mathrm{A}_2) \quad \{\forall t_1 t_2 t_3\}.\, \mathbf{Rd}(t_1, t_2) \to \mathbf{Rd}(\texttt{APP}(t_1, t_3), \texttt{APP}(t_2, t_3))\\
&(\mathrm{A}_3) \quad \{\forall t_1 t_2 t_3\}.\, \mathbf{Rd}(t_1, t_2) \to \mathbf{Ev}(t_2, t_3) \to \mathbf{Ev}(t_1, t_3)\\
&(\mathrm{A}_4) \quad \mathbf{Ev}(\boldsymbol{t}, \boldsymbol{t}) \text{ for all terms } \boldsymbol{t} = \texttt{LAM}\, \boldsymbol{x_i}.\, \boldsymbol{s}
\end{aligned}$$

The first and the last axioms are schematic in the logical term $\boldsymbol{t}$ whose free logical variables must range over sorts of closed object-level terms. As explained above, this restriction is necessary for the meta-level definition of substitution to be correct.

The axioms formally capture the idea that (call-by-name) evaluation is the reflexive, transitive closure of (normal-order) one-step reduction as defined above. The notion of reduction is β-reduction (axiom (A_1)); it can be applied to left-most redexes (axiom (A_2)) yielding a one-step reduction relation. The evaluation stops when a λ-abstraction is reached (the family of axioms (A_4)); otherwise it is defined as the transitive closure of one-step reduction (axiom (A_3)).

In the proofs, we will use free assumption variables A_1, A_2, A_3, A_4 corresponding to the respective axioms above. Since all the axioms are Harrop formulas, these free variables will not occur in the extracted programs.

7.3.2.1 Formalizing the proof

The logical relation used in the proof is defined as follows:

$$\begin{aligned}
\mathbf{R}_b(t) \quad &:= \quad \exists v.\mathbf{Ev}(t,v)\\
\mathbf{R}_{T_1\to T_2}(t) \quad &:= \quad \exists v.\mathbf{Ev}(t,v) \wedge \forall s.\, \mathbf{R}_{T_1}(s) \to \mathbf{R}_{T_2}(\mathtt{APP}(t,s))
\end{aligned}$$

A term of an arrow type satisfying the relation $\mathbf{R}_T$ is not only required to evaluate to a value (or "halt", in Pierce's words [147, p. 150]), but it should also halt when applied to another halting term. This stronger condition allows to prove the desired theorem for both call-by-value and call-by-name evaluation strategies. If we are only interested in evaluation at base types, a weaker condition is actually enough to prove the normalization theorem for call-by-name evaluation (see Section 7.3.4), but for the call-by-value case we still need this stronger definition.

We now formalize three lemmas about the relation $\mathbf{R}_T$, using the notation p_i for the proof term corresponding to the formal proof of "Lemma 7.i". First, we immediately see that every term satisfying the relation $\mathbf{R}_T$ evaluates to a value:

Lemma 7.1. $\{\forall t\}.\, \mathbf{R}_T(t) \to \exists v.\mathbf{Ev}(t,v)$.

Proof. By induction on types at the meta level. The corresponding proof terms are:

$$\mathrm{p}_1^b = \{\lambda t\}.\lambda u^{\mathbf{R}_b(t)}.u$$
$$\mathrm{p}_1^{T_1\to T_2} = \{\lambda t\}.\lambda u^{\mathbf{R}_{T_1\to T_2}(t)}.\mathrm{fst}\, u$$

□

To prove the normalization theorem we also need the following two lemmas.

Lemma 7.2. $\{\forall rs\}.\, \mathbf{Rd}(t_1,t_2) \to \mathbf{R}_T(t_2) \to \mathbf{R}_T(t_1)$.

Proof. By induction on types at the meta level.

Case b. Assume $\mathbf{Rd}(t_1,t_2)$ and $\mathbf{R}_b(t_2)$. By Lemma 7.1, we obtain $\exists v.\mathbf{Ev}(t_2,v)$ from $\mathbf{R}_b(t_2)$. Then using axiom (A_3) we deduce $\exists v.\mathbf{Ev}(t_1,v)$.

The proof term corresponding to this case is as follows:

$$\mathrm{p}_2^b = \{\lambda rs\}.\lambda u^{\mathbf{Rd}(t_1,t_2)} v^{\mathbf{R}_b(t_2)}.[\mathrm{p}_1^b\, v, w^{\mathbf{Ev}(t_2,v')}.\langle v', A_3\,\{rsv'\}u\,w\,\rangle]$$

Case $T_1 \to T_2$. Assume $\mathbf{Rd}(t_1, t_2)$ and $\mathbf{R}_{T_1\to T_2}(t_2)$. We need to prove that $\exists v.\mathbf{Ev}(t_1, v)$ and $\forall t_3.\, \mathbf{R}_{T_1}(t_3) \to \mathbf{R}_{T_2}(\mathtt{APP}(t_1, t_3))$. The first fact is proved analogously to the base case. For the second, assume that $\mathbf{R}_{T_1}(t_3)$ holds for some t_3. By axiom (A_2) we obtain $\mathbf{Rd}(\mathtt{APP}(t_1, t_3), \mathtt{APP}(t_2, t_3))$. Next, unwinding the definition of $\mathbf{R}_{T_1\to T_2}(t_2)$ yields $\mathbf{R}_{T_2}(\mathtt{APP}(t_2, t_3))$. Hence, by induction hypothesis we conclude that $\mathbf{R}_{T_2}(\mathtt{APP}(t_1, t_3))$. Here is the corresponding proof term:

$$\mathrm{p}_2^{T_1\to T_2} = \{\lambda rs\}.\lambda u^{\mathbf{Rd}(t_1,t_2)} v^{\mathbf{R}_{T_1\to T_2}(t_2)}.(\mathrm{p}_{2,1}^{T_1\to T_2}, \mathrm{p}_{2,2}^{T_1\to T_2})$$

where

$$\mathrm{p}_{2,1}^{T_1\to T_2} = [\mathrm{p}_1^{T_1\to T_2}\, v, w^{\mathbf{Ev}(t_2,v')}.\langle v', A_3\,\{rsv'\}\,u\,w\rangle]$$

$$\mathrm{p}_{2,2}^{T_1\to T_2} = \lambda t_3^{\Lambda_{T_1}}\, z^{\mathbf{R}_{T_1}(t_3)}.\mathrm{p}_2^{T_2}\,\{\mathtt{APP}(r,t)\mathtt{APP}(s,t)\}\,(A_2\,\{rst\}\,u)\,(\mathrm{snd}\,v\,s\,z)$$

□

Lemma 7.3. *For any term* t *of type* T*, with* $FV(\mathrm{t}) = \{x_1, \ldots, x_n\}$*, the following formula is provable:*

$$\forall \vec{r}.\, (\mathbf{R}_{T_1}(r_1) \wedge \ldots \wedge \mathbf{R}_{T_n}(r_n)) \to \mathbf{R}_T(\boldsymbol{t}\,[\vec{r}/\vec{x}])$$

(where $\boldsymbol{t}^{\Lambda_T}$ *is the encoding of* t *as a logical term, and where we use the abbreviation* $\boldsymbol{t}\,[\vec{r}/\vec{x}]$ *for* $\boldsymbol{t}\,[r_1/\mathtt{VAR}\,x_1]\cdots[r_n/\mathtt{VAR}\,x_n]$*).*

Proof. By induction on the typing derivation (or, on the structure of t, parameterized by the set of free variables).

Case $\boldsymbol{t} = \mathtt{VAR}\,x_i^T$. Obvious. $\mathrm{p}_3^{\mathtt{VAR}\,x_i,\vec{x}} = \lambda\vec{r}\vec{u}.u_i$.

Case $\boldsymbol{t} = \mathtt{APP}(\boldsymbol{s_1}^{T_1\to T}, \boldsymbol{s_2}^{T_1})$. We apply the induction hypothesis to both subterms to obtain $\mathbf{R}_{T_1\to T}(\boldsymbol{s_1}\,[\vec{r}/\vec{x}])$ and $\mathbf{R}_{T_1}(\boldsymbol{s_2}\,[\vec{r}/\vec{x}])$. Unwinding the definition of $\mathbf{R}_{T_1\to T}(\boldsymbol{s_1}\,[\vec{r}/\vec{x}])$ then yields $\mathbf{R}_T(\mathtt{APP}(\boldsymbol{s_1}, \boldsymbol{s_2})\,[\vec{r}/\vec{x}])$ (using $\mathtt{APP}(\boldsymbol{s_1}\,[\vec{r}/\vec{x}], \boldsymbol{s_2}\,[\vec{r}/\vec{x}]) = \mathtt{APP}(\boldsymbol{s_1}, \boldsymbol{s_2})\,[\vec{r}/\vec{x}]$).

$$\mathrm{p}_3^{\mathtt{APP}(\boldsymbol{s_1},\boldsymbol{s_2}),\vec{x}} = \lambda\vec{r}\vec{u}.\mathrm{snd}(\mathrm{p}_3^{\boldsymbol{s_1},\vec{x}}\,\vec{r}\vec{u})\,(\boldsymbol{s_2}\,[\vec{r}/\vec{x}])\,(\mathrm{p}_3^{\boldsymbol{s_2},\vec{x}}\,\vec{r}\vec{u}).$$

Case $\boldsymbol{t} = \mathtt{LAM}\,x_{n+1}^{T_1}.\,\boldsymbol{r}^{T_2} (T = T_1 \to T_2)$. We need to show that $\exists v.\mathbf{Ev}(\boldsymbol{t}\,[\vec{r}/\vec{x}], v)$ and $\forall s.\, \mathbf{R}_{T_1}(s) \to \mathbf{R}_{T_2}(\mathtt{APP}(\boldsymbol{t}\,[\vec{r}/\vec{x}], s))$. The first fact follows from (an instance of) the axiom (A_4), since $(\mathtt{LAM}\,x_{n+1}.\,\boldsymbol{r})\,[\vec{r}/\vec{x}]$ is a λ-abstraction. For the second, assume that $\mathbf{R}_{T_1}(s)$ holds for some s. By induction hypothesis, $\mathbf{R}_{T_2}(\boldsymbol{r}\,[\vec{r}/\vec{x}]\,[s/x_{n+1}])$ holds. We now obtain $\mathbf{R}_{T_2}(\mathtt{APP}(\mathtt{LAM}\,x_{n+1}.\,\boldsymbol{r}\,[\vec{r}/\vec{x}], s))$

using axiom (A_1) and Lemma 7.2, which concludes the proof. The corresponding proof term reads as follows:

$$\begin{aligned}
&\mathrm{p}_3^{\mathtt{LAM}\,x_{n+1}.\,\boldsymbol{r},\vec{x}} = \lambda\vec{r}\vec{u}.(\mathrm{p}_{3,1},\mathrm{p}_{3,2})\\
\text{where}\quad &\mathrm{p}_{3,1} = \langle(\mathtt{LAM}\,x_{n+1}.\,\boldsymbol{r})\,[\vec{r}/\vec{x}], A_4\rangle\\
&\mathrm{p}_{3,2} = \lambda s^{\Lambda_{T_1}} v^{\mathbf{R}_{T_1}(s)}.\mathrm{p}_2^{T_2}\,\{t_1 t_2\}\,(A_1\,\{s\})\,\mathrm{p}_3^{\boldsymbol{r},\vec{x}x_{n+1}}\,(\vec{r}s)\,(\vec{u}v)\\
\text{with}\quad &t_1 = \mathtt{APP}(\mathtt{LAM}\,x_{n+1}.\,\boldsymbol{r}\,[\vec{r}/\vec{x}], s)\\
&t_2 = \boldsymbol{r}\,[\vec{r}/\vec{x}]\,[s/\mathtt{VAR}\,x_{n+1}]
\end{aligned}$$

□

The normalization theorem can now be stated formally as follows.

Theorem 7.3. *For any closed term* t *of type T with encoding $\boldsymbol{t}^{\Lambda_T}$, $\exists v.\mathbf{Ev}(\boldsymbol{t}, v)$ is provable.*

Proof. By Lemma 7.3, $\mathbf{R}_T(\boldsymbol{t})$ is provable. Hence, by Lemma 7.1, $\exists v.\mathbf{Ev}(\boldsymbol{t}, v)$ is provable.

$$\mathrm{p} = \mathrm{p}_1^T\,(\mathrm{p}_3^{\boldsymbol{t}}\,\varepsilon\,\varepsilon),$$

where ε denotes the empty tuple. □

7.3.2.2 Extracted program

Since the induction on the structure of terms in the proof of Lemma 7.3 is done at the meta level, from the proof of Theorem 7.3 we do not obtain one extracted program of type $\Lambda_T \to \Lambda_T$ realizing the formula $\forall t^{\Lambda_T}.\,\exists v^{\Lambda_T}.\,\mathbf{Ev}(t, v)$, but rather—for each term t^{Λ_T}—we extract a program ‘computing’ a term v such that $\mathbf{Ev}(t, v)$ is provable in $\mathbf{M}_{\mathbf{1}}^-(\lambda)$ [21].

We first consider the types $\tau(\mathbf{R}_T(t))$ of programs extracted from Lemma 7.3 (for specific terms t^{Λ_T}.) We see that the types $\tau(\mathbf{R}_T(t))$ are independent of t and that they can be characterized inductively:

$$\begin{aligned}
\tau(\mathbf{R}_b) &:= \Lambda_b\\
\tau(\mathbf{R}_{T_1\to T_2}) &:= \Lambda_{T_1\to T_2} \times (\Lambda_{T_1} \to \tau(\mathbf{R}_{T_1}) \to \tau(\mathbf{R}_{T_2}))
\end{aligned}$$

This inductive characterization defines the semantic domains of a glueing model similar to the ones considered by Coquand and Dybjer (relative to any particular model of $\mathbf{M}_{\mathbf{1}}^-(\lambda)$).

Let us introduce the notation eval_t for $[\![\mathrm{p}_3^t]\!]$. The terms extracted from Lemma 7.3 can then be inductively described as follows (they are parameterized by a tuple of free variables $\vec{x}$):

$$\begin{aligned}
\mathrm{eval}_{\mathtt{VAR}\,x_i,\vec{x}} &= \lambda\vec{t}\vec{u}.u_i\\
\mathrm{eval}_{\mathtt{APP}(t_1,t_2),\vec{x}} &= \lambda\vec{t}\vec{u}.\mathrm{snd}(\mathrm{eval}_{t_1,\vec{x}}\,\vec{t}\vec{u})\,(t_2\,[\vec{t}/\vec{x}])\,(\mathrm{eval}_{t_2,\vec{x}}\,\vec{t}\vec{u})\\
\mathrm{eval}_{\mathtt{LAM}\,x_{n+1}.\,t,\vec{x}} &= \lambda\vec{t}\vec{u}.(\mathtt{LAM}\,x_{n+1}.\,t\,[\vec{t}/\vec{x}], \lambda sv.[\![\mathrm{p}_2^T]\!]\,(\mathrm{eval}_{t,\vec{x}x_{n+1}}\,(\vec{t}s)(\vec{u}v)))\\
\text{with}&\\
[\![\mathrm{p}_2^b]\!] &= \lambda u.u\\
[\![\mathrm{p}_2^{T_1\to T_2}]\!] &= \lambda x.(\mathrm{fst}\,x, \lambda sv.[\![\mathrm{p}_2^{T_2}]\!]\,((\mathrm{snd}\,x)\,s\,v))
\end{aligned}$$

(Note that $[\![\mathrm{p}_2^T]\!]$ is $\beta\eta\times$-equivalent to the identity function.) For every closed term t^{Λ_T}, $\mathrm{eval}_{t,\varepsilon}$ denotes the glueing model interpretation of the object-level term denoted by t^{Λ_T}.

From Lemma 7.1 we obtain the 'reification' function mapping semantic values back to syntax (parameterized with the type of a given term):

$$\begin{aligned}
\downarrow_b &= \lambda u^{\Lambda_b}.u \\
\downarrow_{T_1\to T_2} &= \lambda u^{\Lambda_{T_1\to T_2}\times(\Lambda_{T_1}\to\tau(\mathbf{R}_{T_1})\to\tau(\mathbf{R}_{T_2}))}.\mathrm{fst}\, u
\end{aligned}$$

The complete program is the composition of the two functions and it is therefore an instance of (weak head) normalization by evaluation:

$$[\![\mathrm{p}_{tT}]\!] \;=\; \downarrow_T (\mathrm{eval}_{t,\varepsilon}\, \varepsilon\varepsilon)$$

In this presentation of the evaluation function there are two environments, represented by the vectors $\vec{t}$ and $\vec{u}$, whose elements can be substituted for the respective variables in the vector $\vec{x}$ (by construction, the length of all the vectors is the same). The program produces weak head normal forms, according to the call-by-name strategy given by the axioms, and it is correct in the sense that the formula $\mathbf{Ev}(t, [\![\mathrm{p}_{tT}]\!])$ is provable in $\mathbf{M}_{\mathbf{1}}^{-}(\lambda)$ for every closed simply typed term t of type T.

7.3.3 Call-by-value evaluation

The process of call-by-value evaluation of closed terms is defined through the following axioms:

(A$_1$) $\{\forall t_2\}.\,\mathbf{V}(t_2) \to \mathbf{Rd}(\mathtt{APP}(\mathtt{LAM}\,\boldsymbol{x_i}.\,\boldsymbol{t}, t_2), \boldsymbol{t}\,[t_2/\mathtt{VAR}\,\boldsymbol{x_i}])$

(A$_2$) $\{\forall t_1 t_2 t_3\}.\,\mathbf{Rd}(t_1,t_2) \to \mathbf{Rd}(\mathtt{APP}(t_1,t_3), \mathtt{APP}(t_2,t_3))$

(A$_2'$) $\{\forall t_1 t_2 t_3\}.\,\mathbf{V}(t_1) \to \mathbf{Rd}(t_2,t_3) \to \mathbf{Rd}(\mathtt{APP}(t_1,t_2), \mathtt{APP}(t_1,t_3))$

(A$_4$) $\mathbf{V}(\boldsymbol{t})$ for all terms $\boldsymbol{t} = \mathtt{LAM}\,\boldsymbol{x_i}.\,\boldsymbol{s}$

Similarly to the call-by-name case, these axioms directly encode the definition of the one-step call-by-value evaluation strategy. Again, the first and the last axioms are schematic in the logical term $\boldsymbol{t}$ whose free logical variables must range over sorts of closed object-level terms. To make the proof go through in the call-by-value case, however, we need a way to perform induction on the length on reduction sequences. To this end we introduce the primitive predicate $\mathbf{Rd}^*$ which is to be understood as the reflexive, transitive closure of the one-step reduction predicate $\mathbf{Rd}$. We use the following axioms for $\mathbf{Rd}^*$:

(R$_1$) $\{\forall t\}.\,\mathbf{Rd}^*(t,t)$

(R$_2$) $\{\forall stv\}.\,\mathbf{Rd}(s,t) \to \mathbf{Rd}^*(t,v) \to \mathbf{Rd}^*(s,v)$

(R$_3$) $(\{\forall t_1\}.\,\varphi(t_1,t_1)) \wedge (\{\forall t_1 t_2 t_3\}.\,\mathbf{Rd}(t_1,t_2) \wedge \varphi(t_2,t_3) \to \varphi(t_1,t_3))$
$\to \{\forall t_1 t_2\}.\,(\mathbf{Rd}^*(t_1,t_2)) \to \varphi(t_1,t_2)$
(where φ is Harrop)

The axiom (R_3) is an induction axiom. By requiring the formula φ in (R_3) to be Harrop, we ensure that every instance of the axiom is a Harrop formula.

Two further axioms are assumed:

$$\begin{array}{ll}(\mathrm{Det}) & \{\forall rst\}.\,\mathbf{Rd}(t,r) \to \mathbf{Rd}(t,s) \to r = s \\ (\mathrm{Val}) & \{\forall vt\}.\,\mathbf{V}(v) \wedge \mathbf{Rd}^*(v,t) \to t = v\end{array}$$

In order to use these two axioms, we extend the logic $\mathbf{M_1}$ with the usual axioms for equality (the soundness of modified realizability is preserved with this extension [181]).

Finally we can define the evaluation predicate as an abbreviation:

$$\mathbf{Ev}(t,v) \quad := \quad \mathbf{Rd}^*(t,v) \wedge \mathbf{V}(v).$$

7.3.3.1 Formalizing the proof

As remarked before, the logical relation used in the proof is defined as in the call-by-name case, except that now it can be refined—the universal variable becomes computationally redundant under call by value (we announce it in advance, but this observation can only be made after we actually write down the proof):

$$\begin{array}{lll}\mathbf{R}_b(t) & := & \exists v.\mathbf{Ev}(t,v) \\ \mathbf{R}_{T_1\to T_2}(t) & := & \exists v.\mathbf{Ev}(t,v) \wedge \{\forall s\}.\,\mathbf{R}_{T_1}(s) \to \mathbf{R}_{T_2}(\mathtt{APP}(t,s))\end{array}$$

The call-by-value analog of Lemma 7.1 is stated and proved in the same way:

Lemma 7.4. $\{\forall t\}.\,\mathbf{R}_T(t) \to \exists v.\,\mathbf{Ev}(t,v)$.

In order to prove the call-by-value version of Theorem 7.3 we need a few more properties of evaluation, stated in Lemmas 7.5-7.11.

Lemma 7.5. $\{\forall rst\}.\,\mathbf{Rd}^*(r,s) \to \mathbf{Rd}^*(s,t) \to \mathbf{Rd}^*(r,t)$

Proof. Using the induction axiom (R_3) with the formula

$$\varphi(r,s) = \{\forall t\}.\,\mathbf{Rd}^*(s,t) \to \mathbf{Rd}^*(r,t)$$

□

Lemma 7.6. $\{\forall rst\}.\,\mathbf{Rd}^*(r,s) \to \mathbf{Rd}^*(\mathtt{APP}(r,t),\mathtt{APP}(s,t))$

Proof. Using axiom (R_3) with the formula

$$\varphi(r,s) = \{\forall t\}.\,\mathbf{Rd}^*(\mathtt{APP}(r,t),\mathtt{APP}(s,t))$$

□

Lemma 7.7. $\{\forall rs\}.\,\mathbf{Rd}^*(r,s) \to \mathbf{R}_T(s) \to \mathbf{R}_T(r)$.

Proof. As in the call-by-name case a formal proof is constructed by induction on T. □

In the call-by-value variant of the proof, we also need to prove the formula $\{\forall rs\}.\, \mathbf{Rd}^*(r,s) \to \mathbf{R}_T(r) \to \mathbf{R}_T(s)$ for every type T. This is done in the next three lemmas.

Lemma 7.8. $\{\forall stv\}.\, \mathbf{V}(v) \to \mathbf{Rd}^*(s,v) \to \mathbf{Rd}(s,t) \to \mathbf{Rd}^*(t,v)$.

Proof. Using axiom (R$_3$) with the formula

$$\varphi(s,v) = \mathbf{Rd}^*(s,v) \wedge \{\forall t\}.\, \mathbf{Rd}(s,t) \to \mathbf{Rd}^*(t,v)$$

□

Lemma 7.9. $\{\forall stv\}.\, \mathbf{V}(v) \to \mathbf{Rd}^*(s,v) \to \mathbf{Rd}^*(s,t) \to \mathbf{Rd}^*(t,v)$.

Proof. Using axiom (R$_3$) with the formula

$$\varphi(r,s) = \{\forall v\}.\, \mathbf{V}(v) \to \mathbf{Rd}^*(r,v) \to \mathbf{Rd}^*(s,v)$$

□

Lemma 7.10. $\{\forall rs\}.\, \mathbf{Rd}^*(r,s) \to \mathbf{R}_T(r) \to \mathbf{R}_T(s)$

Proof. A formal proof is constructed by induction on T, using the two previous lemmas. □

Another lemma on reduction sequences is needed in the proof of the main lemma:

Lemma 7.11. $\{\forall stv\}.\, \mathbf{V}(v) \to \mathbf{Rd}^*(s,t) \to \mathbf{Rd}^*(\mathtt{APP}(v,s), \mathtt{APP}(v,t))$.

Proof. Using axiom (R$_3$) with the formula

$$\varphi(s,t) = \{\forall v\}.\, \mathbf{V}(v) \to \mathbf{Rd}^*(\mathtt{APP}(v,s), \mathtt{APP}(v,t))$$

□

The call-by-value analog of Lemma 7.3 is stated just as before, and its proof relies on Lemmas 7.4-7.11:

Lemma 7.12. *For any term* t *of type* T*, with* $FV(\mathrm{t}) = \{x_1, \ldots, x_n\}$*, the following formula is provable:*

$$\forall \vec{r}.\, (\mathbf{R}_{T_1}(r_1) \wedge \ldots \wedge \mathbf{R}_{T_n}(r_n)) \to \mathbf{R}_T(\boldsymbol{t}\,[\vec{r}/\vec{x}])$$

(where $\boldsymbol{t}^{\Lambda_T}$ *is the encoding of* t *as a logical term, and where we use the abbreviation* $\boldsymbol{t}\,[\vec{r}/\vec{x}]$ *for* $\boldsymbol{t}\,[r_1/\mathtt{VAR}\, x_1] \cdots [r_n/\mathtt{VAR}\, x_n]$*).*

Proof. By induction on the typing derivation (or, on the structure of t, parameterized by the set of free variables). Again we use the notation p_i for the proof term corresponding to the formal proof of "Lemma 7.i".

Case $\boldsymbol{t} = \texttt{VAR}\, x_i^T$. Obvious. $\mathrm{p}_{12}^{\texttt{VAR}\, x_i, \vec{x}} = \lambda \vec{r}\vec{u}.u_i$.

Case $\boldsymbol{t} = \texttt{APP}(\boldsymbol{s_1}^{T_1 \to T}, \boldsymbol{s_2}^{T_1})$. Just like the call-by-name case: We apply the induction hypothesis to both subterms to obtain $\mathbf{R}_{T_1 \to T}(\boldsymbol{s_1}\, [\vec{r}/\vec{x}])$ and $\mathbf{R}_{T_1}(\boldsymbol{s_2}\, [\vec{r}/\vec{x}])$.

Unwinding the definition of $\mathbf{R}_{T_1 \to T}(\boldsymbol{s_1}\, [\vec{r}/\vec{x}])$ gives $\mathbf{R}_T(\texttt{APP}(\boldsymbol{s_1}, \boldsymbol{s_2})\, [\vec{r}/\vec{x}])$.

$$\mathrm{p}_{12}^{\texttt{APP}(\boldsymbol{s_1}, \boldsymbol{s_2}), \vec{x}} = \lambda \vec{r}\vec{u}.\mathrm{snd}(\mathrm{p}_{12}^{\boldsymbol{s_1}, \vec{x}}\, \vec{r}\vec{u})\, (\boldsymbol{s_2}\, [\vec{r}/\vec{x}])\, (\mathrm{p}_{12}^{\boldsymbol{s_2}, \vec{x}}\, \vec{r}\vec{u}).$$

Case $\boldsymbol{t} = \texttt{LAM}\, x_{n+1}^{T_1}.\, \boldsymbol{r}^{T_2} (T = T_1 \to T_2)$. We need to show that $\exists v.\mathbf{Ev}(\boldsymbol{t}\, [\vec{r}/\vec{x}], v)$ and $\forall s.\, \mathbf{R}_{T_1}(s) \to \mathbf{R}_{T_2}(\texttt{APP}(\boldsymbol{t}\, [\vec{r}/\vec{x}], s))$. The first fact follows from axiom (R_1), together with (an instance of) the axiom (A_4), since $(\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r})\, [\vec{r}/\vec{x}]$ is a λ-abstraction. For the second, assume that $\mathbf{R}_{T_1}(s)$ holds for some s. Then by Lemma 7.4 there is a v such that $\mathbf{Ev}(s, v)$ holds, i.e., such that $\mathbf{Rd}^*(s, v) \wedge \mathbf{V}(v)$ holds. Then by Lemma 7.10, $\mathbf{R}_{T_1}(v)$ holds. Now, by induction hypothesis, $\mathbf{R}_{T_2}(\boldsymbol{r}\, [\vec{r}/\vec{x}]\, [v/x_{n+1}])$ holds. We then obtain $\mathbf{R}_{T_2}(\texttt{APP}(\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r}\, [\vec{r}/\vec{x}], v))$ using axiom (A_1) and Lemma 7.7. Using Lemma 7.11 and Lemma 7.7, we get $\mathbf{R}_{T_2}(\texttt{APP}(\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r}\, [\vec{r}/\vec{x}], s))$ which concludes the proof.

The corresponding proof term reads as follows (omitting some computationally irrelevant lemma instantiations for brevity):

$\mathrm{p}_{12}^{\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r}, \vec{x}} = \lambda \vec{r}\vec{u}.(\mathrm{p}_{12,1}, \mathrm{p}_{12,2})$

where

$\mathrm{p}_{12,1} = \langle (\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r})\, [\vec{r}/\vec{x}], \langle R_1\, \{(\texttt{LAM}\, x_{n+1}.\, \boldsymbol{r})\, [\vec{r}/\vec{x}]\}, A_4 \rangle\rangle$

$\mathrm{p}_{12,2} = \{\lambda s\}.\lambda e.[(\mathrm{p}_4\, \{s\}\, e)^{\exists v.\mathbf{Ev}(s,v)}, f.\mathrm{p}_7\, (\mathrm{p}_{11}\, \{s\}\, \{v\}\, A_4\, (\mathrm{fst}\, f))\, \mathrm{p}_{12,3}]$

with

$\mathrm{p}_{12,3} = \mathrm{p}_7\, (R_2\, (A_1\, (\mathrm{snd}\, f))\, R_1)\, (\mathrm{p}_{12}^{\boldsymbol{r}, \vec{x}, x_{n+1}} (\vec{r}v)\, (\vec{u}e)\, (\mathrm{p}_{10}\, \{s\}\{v\}\, (\mathrm{fst}\, f)\, e))$

□

The main theorem is also stated and proved as before.

7.3.3.2 Extracted program

Again we see that the types $\tau(\mathbf{R}_T(t))$ are independent of t. They describe the domains of a glueing model as follows:

$$\begin{aligned} \tau(\mathbf{R}_b) &:= \Lambda_b \\ \tau(\mathbf{R}_{T_1 \to T_2}) &:= \Lambda_{T_1 \to T_2} \times (\tau(\mathbf{R}_{T_1}) \to \tau(\mathbf{R}_{T_2})) \end{aligned}$$

Similarly to the previous case, the program we obtain for call by value is the composition of the term extracted from Lemma 7.4 (the same as for call by name), and the one extracted from Lemma 7.12. The term extracted from Lemma 7.12 looks as follows (using the notation eval_t for $[\![\mathrm{p}_{12}^t]\!]$):

$$\begin{aligned} \mathrm{eval}_{\texttt{VAR}\, x_i, \vec{x}} &= \lambda \vec{t}\vec{u}.u_i \\ \mathrm{eval}_{\texttt{APP}(t_1, t_2), \vec{x}} &= \lambda \vec{t}\vec{u}.\mathrm{snd}(\mathrm{eval}_{t_1, \vec{x}}\, \vec{t}\vec{u})\, (\mathrm{eval}_{t_2, \vec{x}}\, \vec{t}\vec{u}) \\ \mathrm{eval}_{\texttt{LAM}\, x_{n+1}^{T_1}.\, t^{T_2}, \vec{x}} &= \lambda \vec{t}\vec{u}.((\texttt{LAM}\, x_{n+1}.\, t)\, [\vec{t}/\vec{x}], \lambda e.\mathrm{eval}_{t, \vec{x}x_{n+1}}\, (\vec{t}(\downarrow_{T_1} e))(\vec{u}e)) \end{aligned}$$

This program also threads two environments, but the first of them (represented by the vector $\vec{t}$) contains already evaluated terms. As before, for every closed term t^{Λ_T}, $\mathrm{eval}_{t,\varepsilon}$ denotes the glueing model interpretation of the object-level term denoted by t^{Λ_T}.

Remark. In the original formulation of Lemma 7.12 in Pierce's book [147, p. 151], the terms to be substituted for free variables in a given term were required to be values. This restriction, however, is not necessary for the proof to go through, and the resulting program is exactly the same as the one obtained here.

7.3.4 Weak head normalization for closed terms of base type

We now show a variant of the proof of weak head normalization where we are only interested in evaluating terms of base type. In order to be able to observe the behavior of programs, we extend the object language with integers, formed with the zero constant `0` and the successor constant `S` in the usual way. The set of base types now includes the type ι for integers. As mentioned before, for call-by-name evaluation we can simplify the definition of the relation $\mathbf{R}_T$, which consequently leads to a simpler extracted program that we will show next.

We add the following two axioms specifying the evaluation strategy for the new terms:

$$\begin{array}{ll} (\mathrm{A}_5) & \mathbf{Ev}(\mathtt{0},\mathtt{0}) \\ (\mathrm{A}_6) & \forall tv.\, \mathbf{Ev}(t,v) \to \mathbf{Ev}(\mathtt{S}\,t, \mathtt{S}\,v) \end{array}$$

The definition of the logical relation is now less restrictive for higher types:

$$\begin{array}{lcl} \mathbf{R}_b(t) & := & \exists v.\mathbf{Ev}(t,v) \\ \mathbf{R}_{T_1\to T_2}(t) & := & \{\forall s\}.\, \mathbf{R}_{T_1}(s) \to \mathbf{R}_{T_2}(\mathtt{APP}(t,s)) \end{array}$$

Theorem 7.4. *For any closed term t of type ι, $\exists v.\mathbf{Ev}(t,v)$ holds.*

The proof is carried out almost as before, and it relies on the base-type version of Lemma 7.1, on Lemma 7.2 as before, and on the base-type counterpart of Lemma 7.3 which now reads as follows (note that the vector of terms $\vec{r}$ is now computationally redundant):

Lemma 7.13. *For any term* t *of type T, with $FV(\mathrm{t}) = \{x_1,\ldots,x_n\}$, the following formula is provable:*

$$\{\forall\vec{r}\}.\, (\mathbf{R}_{T_1}(r_1) \wedge \ldots \wedge \mathbf{R}_{T_n}(r_n)) \to \mathbf{R}_T(\boldsymbol{t}\,[\vec{r}/\vec{x}]).$$

(where $\boldsymbol{t}^{\Lambda_T}$ is the encoding of t *as a logical term, and where we use the abbreviation $\boldsymbol{t}\,[\vec{r}/\vec{x}]$ for $\boldsymbol{t}\,[r_1/\mathtt{VAR}\,x_1]\cdots[r_n/\mathtt{VAR}\,x_n]$).*

For the proof of Lemma 7.13 we need to show that $\mathbf{R}_\iota(\mathtt{0})$ holds, and that for any term t of type ι, $\mathbf{R}_\iota(t) \to \mathbf{R}_\iota(\mathtt{S}\,t)$ holds.

Remark. The proof does not go through if we use the call-by-value axiomatization instead of call by name; this is due to the fact that in the proof of the main lemma, in the case for abstraction, we must know that an arbitrary term of an arbitrary type evaluates to a value. However, with the weakened definition of the relation $\mathbf{R}_T$ we cannot prove this fact any more.

The program extracted from the proof looks as follows:

$$\begin{array}{lcl}
\mathrm{eval}_{\,\mathtt{0},\vec{x}} & = & \lambda\vec{u}.\mathtt{0} \\
\mathrm{eval}_{\,\mathtt{S}\,t,\vec{x}} & = & \lambda\vec{u}.\mathtt{S}\,(\mathrm{eval}_{\,t,\vec{x}}\,\vec{u}) \\
\mathrm{eval}_{\,\mathtt{VAR}\,x_i,\vec{x}} & = & \lambda\vec{u}.u_i \\
\mathrm{eval}_{\,\mathtt{APP}(t_1,t_2),\vec{x}} & = & \lambda\vec{u}.(\mathrm{eval}_{\,t_1,\vec{x}}\,\vec{u})\,(\mathrm{eval}_{\,t_2,\vec{x}}\,\vec{u}) \\
\mathrm{eval}_{\,\mathtt{LAM}\,x_{n+1}.\,t,\vec{x}} & = & \lambda\vec{u}.\lambda v.\mathrm{eval}_{\,t,\vec{x}x_{n+1}}\,\vec{u}v
\end{array}$$

Acknowledgments: We are grateful to Ulrich Berger for discussions on this work, and to Dariusz Biernacki, Philipp Gerhardy, and the MFPS reviewers for their comprehensive feedback. Special thanks to Andrzej Filinski for two rounds of insightful comments.

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.

7.A Implementation

This appendix contains an ML implementation of the normalization programs from Sections 7.3.2.2 and 7.3.3.2. The implementation ignores the dependencies in the definition of the object-level terms and the semantic domains:

```
type ide = string

datatype term = VAR of ide
              | APP of term * term
              | LAM of ide * term
```

The ML programs work by optimistically trying to interpret an untyped object-level term (defined in the data type `term` just above) into a semantic domain defined by a reflexive type (see the data type `R` below for call by name and call by value). However, as stressed by Filinski [79, 92], it is a non-trivial task to prove that such implementations are correct.

We use the following auxiliary functions, whose definitions are omitted here:

```
subst_all : term * (ide * term) list -> term
lookup : ide * (ide * term) list -> term
```

The function `subst_all` implements simultaneous substitution of terms for variables. The function `lookup` implements a standard association-list lookup.

7.A.1 Call by name

```
datatype R = BASE of term
           | ARROW of term * (term -> R -> R)

 fun reify (BASE t)
    = t
  | reify (ARROW (t, f))
    = t

 fun eval (VAR x, ts, us)
    = lookup (x, us)
  | eval (APP (t1, t2), ts, us)
    = let val ARROW (_, f) = eval (t1, ts, us)
      in f (subst_all (t2, ts)) (eval (t2, ts, us))
      end
  | eval (LAM (y, t1), ts, us)
    = let val t = subst_all (LAM (y, t1), ts)
          val f = fn s => fn u => eval (t1, (y, s) :: ts,
               (y, u) :: us)
      in ARROW (t, f)
      end

 fun normalize t
    = reify (eval (t, [], []))
```

7.A.2 Call by value

```
datatype R = BASE of term
           | ARROW of term * (R -> R)

 fun reify (BASE t)
    = t
  | reify (ARROW (t, f))
    = t

 fun eval (VAR x, ts, us)
    = lookup (x, us)
  | eval (APP (t1, t2), ts, us)
    = let val ARROW (_, f) = eval (t1, ts, us)
      in f (eval (t2, ts, us))
      end
  | eval (LAM (y, t1), ts, us)
    = let val t = subst_all (LAM (y, t1), ts)
          val f = fn u => eval (t1, (y, reify u) :: ts, (
              y, u) :: us)
      in ARROW (t, f)
      end


 fun normalize t
    = reify (eval (t, [], []))
```

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. 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, Mar. 2003.

[3] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, Aug. 2003.

[4] M. S. Ager, O. Danvy, and J. 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.

[5] M. S. Ager, O. Danvy, and J. 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.

[6] T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction-free normalization proof. In D. H. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, Aug. 1995. Springer-Verlag.

[7] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In P. Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, Jan. 1995. ACM Press.

[8] Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. 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.

[9] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In Fisher [94], pages 40–53.

[10] K. Asai. Online partial evaluation for shift and reset. In P. 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, Mar. 2002. ACM Press.

[11] K. Asai. Offline partial evaluation for shift and reset. In N. Heintze and P. Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2004)*, pages 3–14, Verona, Italy, Aug. 2003. ACM Press.

[12] L. Augustsson. A compiler for Lazy ML. In Steele Jr. [169], pages 218–227.

[13] V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In X. Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, Jan. 2004. ACM Press.

[14] V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In D. Batory, C. Consel, and W. 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, Oct. 2002. Springer-Verlag.

[15] A. Banerjee, N. Heintze, and J. G. Riecke. Design and correctness of program transformations based on control-flow analysis. In N. Kobayashi and B. 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, Oct. 2001. Springer-Verlag.

[16] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[17] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, chapter 2, pages 118–309. Oxford University Press, Oxford, 1992.

[18] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In Tofte [179], pages 25–37.

[19] N. D. Belnap. How a computer should think. In G. Ryle, editor, *Proceedings of the Oxford International Symposium on Contemporary Aspects of Philosophy*, pages 30–56, Oxford, England, 1976. Oriel Press.

[20] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective I. Technical report 365, Computer Laboratory, University of Cambridge, Cambridge, England, May 1995.

[21] U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, Mar. 1993. Springer-Verlag.

[22] U. Berger. Uniform Heyting arithmetic. *Annals of Pure and Applied Logic*, 133(1-3):125–148, 2005.

[23] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 2005. To appear.

[24] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114(1-3):3–25, 2002.

[25] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J. V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

[26] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In G. 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.

[27] I. Beylin and P. Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In S. Berardi and M. 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.

[28] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, Nov. 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[29] M. Biernacka and O. 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.

[30] M. Biernacka and O. 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.

[31] M. Biernacka, O. Danvy, and K. Støvring. Extracting evaluators from proofs of weak head normalization. In M. Escardó, editor, *Proceedings of the 21st Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, Birmingham, UK, May 2005. Elsevier Science Publishers. Extended version available as the technical report BRICS RS-05-12.

[32] D. Biernacki and O. Danvy. From interpreter to logic engine by defunctionalization. In M. 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, Aug. 2003. Springer-Verlag.

[33] D. Biernacki and O. Danvy. A simple proof of a folklore theorem about delimited control. Research Report BRICS RS-05-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Aug. 2005. Theoretical Pearl to appear in the Journal of Functional Programming.

[34] D. Biernacki, O. Danvy, and K. 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] D. Biernacki, O. Danvy, and C. 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.

[36] H.-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994. ACM Press.

[37] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Rapport 90/17.

[38] U. Boquist. *Code Optimization Techniques for Lazy Functional Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg University, Göteborg, Sweden, Apr. 1999.

[39] L. Cardelli. Compiling a functional language. In Steele Jr. [169], pages 208–217.

[40] R. C. Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

[41] E. Charniak, C. Riesbeck, and D. McDermott. *Artificial Intelligence Programming.* Lawrence Earlbaum Associates, 1980.

[42] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[43] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[44] W. Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[45] W. Clinger, D. P. Friedman, and M. Wand. A scheme for a higher-level semantic algebra. In J. Reynolds and M. Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[46] W. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[47] W. D. Clinger. Proper tail recursion and space efficiency. In K. D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[48] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. In *Preliminary proceedings of the 1993 TYPES Workshop*, Nijmegen, The Netherlands, May 1993.

[49] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[50] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[51] P. Crégut. An abstract machine for lambda-terms normalization. In Wand [188], pages 333–340.

[52] P. Crégut. Strongly reducing variants of the Krivine abstract machine. In Danvy [64]. To appear. Journal version of [51].

[53] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*, volume 1 of *Research Notes in Theoretical Computer Science.* Pitman, 1986.

[54] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[55] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[56] H. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[57] O. Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, Jan. 1989. ACM Press.

[58] O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[59] O. Danvy. Type-directed partial evaluation. In Steele Jr. [170], pages 242–257.

[60] O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. 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.

[61] O. Danvy. From reduction-based to reduction-free normalization. In S. Antoy and Y. 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.

[62] O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In H. Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, Jan. 2004. Invited talk.

[63] O. Danvy. A rational deconstruction of Landin's SECD machine. In C. Grelck, F. Huch, G. J. Michaelson, and P. 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, Sept. 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the technical report BRICS RS-03-33.

[64] O. Danvy, editor. *Special Issue on the Krivine Abstract Machine*, Higher-Order and Symbolic Computation. Springer, 2006. In preparation.

[65] O. Danvy and P. Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, University of Aarhus.

[66] O. Danvy and A. Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[67] O. Danvy and A. Filinski. Abstracting control. In Wand [188], pages 151–160.

[68] O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[69] O. Danvy and J. L. Lawall. Back to direct style II: First-class continuations. In Clinger [44], pages 299–310.

[70] O. Danvy and K. Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [40], pages 327–341.

[71] O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[72] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Nov. 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.

[73] O. Danvy and Z. Yang. An operational investigation of the CPS hierarchy. In S. D. Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

[74] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[75] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[76] P. de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structures in Computer Science*, 8:637–669, 1998.

[77] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[78] S. Draves. Implementing bit-addressing with specialization. In Tofte [179], pages 239–250.

[79] P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, Sept. 2000. Springer-Verlag.

[80] R. K. Dybvig, S. Peyton-Jones, and A. Sabry. A monadic framework for subcontinuations. Technical Report 615, Computer Science Department, Indiana University, Bloomington, Indiana, June 2005.

[81] J. Fairbairn and S. Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, Oregon, Sept. 1987. Springer-Verlag.

[82] M. 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, Aug. 1987.

[83] M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, Jan. 1988. ACM Press.

[84] M. Felleisen and M. Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[85] M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[86] M. Felleisen, D. P. Friedman, B. Duba, and J. Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, Feb. 1987.

[87] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[88] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [40], pages 52–62.

[89] A. Filinski. Representing monads. In Boehm [36], pages 446–457.

[90] A. Filinski. Representing layered monads. In A. Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, Jan. 1999. ACM Press.

[91] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In S. 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.

[92] A. Filinski and H. K. Rohde. A denotational account of untyped normalization by evaluation. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, number 2987 in Lecture Notes in Computer Science, pages 167–181, Barcelona, Spain, Apr. 2002. Springer-Verlag.

[93] M. J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. `<http://www.brics.dk/~hosc/vol06/03-fischer.html>` Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[94] K. Fisher, editor. *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, Snowbird, Utah, Sept. 2004. ACM Press.

[95] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[96] D. P. Friedman, A. Ghuloum, J. G. Siek, and L. Winebarger. Improving the lazy Krivine machine. In Danvy [64]. In preparation.

[97] M. Gasbichler and M. Sperber. Final shift for call/cc: direct implementation of shift and reset. In S. Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 271–282, Pittsburgh, Pennsylvania, Sept. 2002. ACM Press.

[98] M. L. Ginsberg. Multivalued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[99] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[100] T. G. Griffin. A formulae-as-types notion of control. In P. Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, Jan. 1990. ACM Press.

[101] B. Grobauer and Z. Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

[102] C. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In S. 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.

[103] C. Hankin. *Lambda Calculi, a guide for computer scientists*, volume 1 of *Graduate Texts in Computer Science.* Oxford University Press, 1994.

[104] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[105] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[106] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, Oct. 1993.

[107] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In Steele Jr. [169], pages 293–298.

[108] S. Helsen and P. Thiemann. Two flavors of offline partial evaluation. In J. Hsiang and A. Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, Dec. 1998. Springer-Verlag.

[109] C. Hewitt. Control structure as patterns of passing messages. In P. H. Winston and R. H. Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2, pages 434–465. The MIT Press, 1979.

[110] R. Hieb and R. K. Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, Mar. 1990. ACM Press.

[111] R. Hieb, R. K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1993.

[112] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In B. 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.

[113] 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.

[114] J. Hughes. Super combinators: A new implementation method for applicative languages. In D. P. Friedman and D. S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, Aug. 1982. ACM Press.

[115] J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[116] T. Johnsson. Efficient compilation of lazy evaluation. In S. L. Graham, editor, *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, pages 58–69, Montréal, Canada, June 1984. ACM Press.

[117] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, Sept. 1985. Springer-Verlag.

[118] Y. Kameyama. Axioms for delimited continuations in the CPS hierarchy. In J. Marcinkowski and A. 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, Sept. 2004. Springer.

[119] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In O. Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, Aug. 2003. ACM Press.

[120] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[121] Y. Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.

[122] O. 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, Mar. 2005.

[123] G. Kreisel. Interpretation of analysis by means of functionals of finite type. In *Constructivity in Mathematics, Proc. Colloq. Amsterdam (1957)*, Studies in Logic and the Foundation of Mathematics, pages 101–128. North-Holland, 1959.

[124] J.-L. Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`, 1985.

[125] J.-L. Krivine. A call-by-name lambda-calculus machine. In Danvy [64]. To appear. Available online at `http://www.pps.jussieu.fr/~krivine/`.

[126] J.-L. 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/`.

[127] P. 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 [177].

[128] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[129] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[130] J. Launchbury. A natural semantics for lazy evaluation. In S. L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, Jan. 1993. ACM Press.

[131] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In C. 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.

[132] X. Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, Feb. 1990.

[133] P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In Boehm [36], pages 60–69.

[134] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Fisher [94], pages 4–15.

[135] P. 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.

[136] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[137] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

[138] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[139] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In Steele Jr. [170], pages 271–283.

[140] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[141] L. Moreau and C. Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In M. Hermenegildo and J. 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, Sept. 1994. Springer-Verlag.

[142] F. L. Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.

[143] C. R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In O. Danvy and C. L. Talcott, editors, *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW 1992)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[144] L. R. Nielsen. A denotational investigation of defunctionalization. Research Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Dec. 2000.

[145] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, number 624 in Lecture Notes in Artificial Intelligence, pages 190–201, St. Petersburg, Russia, July 1992. Springer-Verlag.

[146] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[147] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[148] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[149] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Sept. 1981.

[150] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *In ACM Symposium on Principles of Programming Languages (POPL), pages 89–98, January 2004.*, 2004.

[151] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2), 2005.

[152] C. Queinnec and B. Serpette. A dynamic extent control operator for partial continuations. In R. C. Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, Jan. 1991. ACM Press.

[153] B. Randell and L. J. Russell. *ALGOL 60 Implementation.* Academic Press, London and New York, 1964.

[154] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[155] J. 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.

[156] K. H. Rose. Explicit substitution – tutorial & survey. BRICS Lecture Series LS-96-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Sept. 1996.

[157] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1996.

[158] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[159] E. 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.

[160] H. Scholz and G. Hasenjaeger. *Grundzüge der Mathematischen Logik.* Springer-Verlag, 1961.

[161] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[162] C. Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, Sept. 2004.

[163] D. Sitaram. *Models of Control and their Implications for Programming Language Design.* PhD thesis, Computer Science Department, Rice University, Houston, Texas, Apr. 1994.

[164] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, Jan. 1990.

[165] D. Sitaram and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [188], pages 161–175.

[166] B. C. Smith. Reflection and semantics in Lisp. In K. Kennedy, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, Jan. 1984. ACM Press.

[167] M. H. Sørensen and P. Urzyczyn. Lecture notes on the Curry-Howard isomorphism. Technical Report DIKU-rapport 98/14, Department of Computer Science, University of Copenhagen, 1998.

[168] G. 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.

[169] G. L. Steele Jr., editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, Aug. 1984. ACM Press.

[170] G. L. Steele Jr., editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, Jan. 1996. ACM Press.

[171] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[172] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000. Reprinted from lecture notes dated 1968.

[173] C. Strachey and C. 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.

[174] E. Sumii. An implementation of transparent migration on standard Scheme. In M. Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, Sept. 2000.

[175] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[176] G. J. Sussman and G. 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.

[177] H. Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[178] P. Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[179] M. Tofte, editor. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press.

[180] A. Tolmach and D. P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[181] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[182] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.

[183] A. 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.

[184] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[185] P. Wadler. Linear types can change the world! In *Proceedings of IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 546–566, Sea Galilee, Israel, Apr. 1990.

[186] P. Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, Jan. 1994.

[187] P. Wadler. The Girard-Reynolds isomorphism. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, pages 468–491, London, UK, 2001. Springer-Verlag.

[188] M. Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[189] M. Wand. On the correctness of the Krivine machine. In Danvy [64]. In preparation.

[190] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, May 1988.

[191] M. Wand and P. Steckler. Selective and lightweight closure conversion. In Boehm [36], pages 435–445.

[192] D. C. Wang and A. W. Appel. Type-safe garbage collectors. In H. R. Nielson, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 166–178, London, United Kingdom, Jan. 2001. ACM Press.

[193] G. Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.

[194] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[195] Y. Xiao, A. Sabry, and Z. M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.