The Interpretation and Inter-derivation of Small-step and Big-step Specifications

Ian Zerny

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

The Interpretation and Inter-derivation of Small-step and Big-step Specifications

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

by
Ian Zerny
June 21, 2013

Abstract

We study the interpretation and inter-derivation of *big-step* and *small-step* specifications. In particular, we consider formal specifications of programming languages, e.g., denotational semantics and operational semantics, and investigate how these specifications relate to each other. We carry out this investigation by interpreting specifications as programs in a pure functional meta-language and by constructively deriving one program from the other using program transformations. To this end, we use two derivational correspondences: The *functional correspondence* between compositional higher-order specifications and first-order transition systems, and the *syntactic correspondence* between rewriting specifications and first-order transition systems.

The main contribution of this dissertation is threefold: First, we extend these correspondences to systematically derive small-step reduction semantics and abstract machines from big-step reduction strategies. Second, we show how these correspondences can be used to relate specifications for lazy evaluation, e.g., graph reduction and call-by-need evaluation. Third, we describe an alternative interpretation of specifications as logic programs in a logical framework, and we give a logical counterpart to the functional correspondence.

Resumé

Vi studerer fortolkningen og udledelsen af *big-step* og *small-step* specifikationer. Især betragter vi formelle specifikationer af programmeringssprog, såsom denotationel semantik og operationel semantik, og vi undersøger, hvordan disse specifikationer relaterer til hinanden. Denne relation undersøger vi ved at fortolke specifikationer som programmer i et effektfri funktionel metasprog og bruger programtransformationer til konstruktivt at udlede et program fra et andet. Til dette benytter vi to udledningskorrespondancer: Den *funktionelle korrespondance* mellem kompositionelle højereordensspecifikationer og transitionssystemer, samt den *syntaktiske korrespondance* mellem omskrivningsspecifikationer og transitionssystemer.

Hovedbidraget i denne afhandling er tredelt: Først udvider vi disse korrespondancer med en systematisk udledning af en small-step reduktionssemantik og en abstrakt maskine fra en big-step reduktionsstrategi. Dernæst viser vi, hvordan disse korrespondancer kan relatere specifikationer for lazy evaluering, såsom grafreduktion og call-by-need evaluering. Slutteligt beskriver vi en alternativ fortolkning af specifikationer som logikprogrammer i et logisk system, og vi giver en logisk pendant til den funktionelle korrespondance.

Acknowledgments

As many before me, I would like to thank my supervisor Olivier Danvy, whose supervision goes well beyond the call of duty. As a teacher, Olivier exposed to me the structure, notation, and interpretation of programs. As a mentor, Olivier guided me through the academic world of writing, publishing, and teaching. As a colleague, Olivier introduced me to the wonders of research, discovery, and development. As a friend, Olivier kindled my interests in tech culture, science fiction, and many other fine details of life.

I would like to thank Jan Midtgaard. As a teaching assistant, Jan, who was then a PhD student, made a lasting impression on me that ultimately led to my own pursuit of a PhD degree. I have since been fortunate to receive Jan's support as colleague, as coauthor, as friend, and as, de facto, second advisor.

I am also grateful to Zena Ariola and Glynn Winskel for reviewing this PhD dissertation and serving in its PhD committee.

I would also like to thank Kevin Millikin for his advice and encouragements as a Google Fellowship Mentor and for hosting a fruitful internship at Google in the early fall of 2012.

I would like to thank Frank Pfenning for an inspiring stay at Carnegie Mellon University during the winter from 2011 to 2012. CMU is a truly remarkable institution which is made so by many: Thanks to Frank Pfenning for his course on logic, to Karl Crary for his course on LF, to Robert Harper for his courses on functional programming and type theory, and to Stephen Brookes for his course on semantics. Thanks to the POP students, to the tea trolls, and for the weekly D'z.

I would like to thank Kenichi Asai and his PhD students for a wonderful visit in the programming-language lab of Ochanomizu University in December 2010. Thanks to Moe Masuko, Kanae Tsushima, and Yayoi Ueda for their great hospitality and academic engagement.

I was fortunate enough to attend Henk Barendregt and Jan Willem Klop's mini-course on Lambda Calculus and Term Rewriting Systems in 2009, the Midlands Graduate School in 2009, the Central European School on Functional Programming in 2009, the Estonian Winter School in Computer Sci-

ence in 2010, and the Oregon Summer School in 2009 and in 2011. These events have shaped my PhD studies, and I want to extend grateful thanks to their organizers.

Thanks to my coauthors, Olivier Danvy, Jacob Johannsen, Jan Midtgaard, Kevin Millikin, Johan Munk, Ken Shan, Robert Simmons, Peter Thiemann, and Johnni Winther. This dissertation is but a single fruit of our collaboration. Thanks to Kent Grigo, Mathias Schwarz, and Jakob Thomsen for their feedback on this dissertation.

Thanks to the other members of the programming-languages group at Aarhus University and to Christian Christoffersen, Dennis Decker Jensen, Finn Jensen, Peter Kristensen, Carsten Nørby, Matthias Puech, Thomas Salomon, Filip Sieczkowski, and Sarah Zakarias for our discussions and also their support. Thanks to the kind and effective support of the administrative staff at Aarhus University and in particular to Michael Glad, Ellen Kjemtrup, and Ann Eg Mølhave.

My PhD studies have successively been funded by the Faculty of Science at Aarhus University and by the Google Fellowship program. I was also awarded an EliteForsk travel grant from the Ministry of Science, Innovation and Higher Education. For this support I express my sincere thanks.

Finally, I am forever grateful to Anne for her unreserved love, for her unwavering support, and for Erik, our son.

Ian Zerny,
Aarhus, June 21, 2013.

Contents

Part I

Background and overview

Chapter 1

Introduction

Many different styles of specifications have been developed for programming languages. They can coarsely be classified in two groups: *recursively* defined specifications that ascribe a final result in one *big step*, and *iteratively* defined specifications that ascribe intermediate results in successive *small steps*.

The thesis defended here is that:

> Mechanical transformations can inter-derive small-step and big-step specifications. This inter-derivation is useful to relate these specifications and to implement them.

In this dissertation, we consider the theory and practice of programming languages as our central case study. In other words, we consider semantic specifications of programming languages and implementations of these specifications. In particular, we consider the denotational and operational methodologies which are the de-facto standards to specify programming languages. We follow the tradition established by McCarthy [142], Landin [131, 132], and Reynolds [176] of using pure functional programs simultaneously as both the specification and the implementation. Our main focus is the *treatment of specifications*, not the *meta-theory of specifications*. Concretely, our developments take place in a computational framework of pure functional programs. Our method of development is to inter-derive these pure functional programs by mechanical program transformations.

The contribution of this dissertation is threefold:

1. We give a method to systematically derive small-step specifications in the form of a reduction semantics and an abstract machine from the big-step specification of a reduction strategy in the form of a compositional search function. This derivation is correct-by-construction and scales to complex cases including all of the reduction semantics in Felleisen et al.'s textbook on semantic engineering [89] and to all of the reduction semantics presented in this dissertation.

2. We give an extensive case study of these derivational techniques on semantic artifacts for lazy evaluation in the form of graph reduction and call-by-need evaluation. This study mechanically exposes the computational structure of lazy evaluation and it connects a myriad of previously disconnected semantic artifacts. Among others, we give a constructive connection of Turner's original reduction machine [209] with

Barendregt et al.'s term-graph rewriting [24]; as well as a constructive connection of Crégut's original lazy Krivine machine [51] with Ariola et al.'s call-by-need λ-calculus, and with Launchbury's natural semantics for call by need.

3. We give a logical interpretation of the inter-derivation techniques where specifications are defined by logic programs within a substructural logical framework [198]. This logical correspondence provides one possible path in formalizing a meta-theoretic framework to define and automatically inter-derive specifications.

This dissertation stands on the shoulders of many, most notably, on the shoulders of those that have jointly developed the inter-derivation techniques with Danvy. In particular, for their development of the functional correspondence, of the syntactic correspondence, and for their account of these correspondences for a wide range of applications [4, 26, 31, 119, 143, 144, 151, 153]. It is as a continuation of their work, that this dissertation extends these correspondences, uses them to account for the theory and practice of graph reduction and call-by-need evaluation, and describes a logical counterpart. This continuation is pursued in Section 3.4 and Section 4.6.

The overarching message is that these inter-derivations witness a striking unity of computation across big-step and small-step specifications and their implementations: to quote Danvy, they all truly define the same elephant, computationally speaking. The structural coincidence between contexts and continuations, in particular, plays a key rôle to connect small-step and big-step computation, as first established between reduction strategies and evaluation orders by Plotkin [170]. The semantic artifacts we derive are correct by construction but what is more, they often match what others have independently crafted by hand with ingenuity, skill, and independent soundness proofs on a case-by-case basis. The inter-derivations make it possible to concentrate this ingenuity and skill on other semantic endeavors [17].

Prerequisites and notations

We assume familiarity with basic set theory, with logical notations, and with the λ-calculus. We also assume familiarity with formal semantics, in particular denotational and operational semantics as can be gathered in, e.g., Winskel's introductory textbook [219].

We use a pure fragment[1] of core Standard ML [146] extensively throughout this dissertation. However, familiarity with another functional programming language, such as Haskell, OCaml or Scheme is sufficient to follow our developments. We use Standard ML because it has a formal semantics and has inductively defined algebraic data types. These two features make Standard ML a reasonable choice as a meta-language for specifications. Indeed, ML originated as the *M*eta *L*anguage for LCF .

We attempt to use standard notations in our developments and review some of these notations below.

Judgments A logical *judgment* is defined inductively by *rules* of the form:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

[1] By 'pure' we mean that the only computational effect of the language is divergence.

In such a rule, J is called the *conclusion* and $J_1, \cdots, J_n$ are called the *premises*. An instance of a judgment form is said to *hold* if there exists a rule with a matching conclusion for which each of the premises hold. Instances of a rule with no premises hold unconditionally.

BNF grammars A *BNF grammar* is used as a short-hand notation for defining a *formation judgment*. For example, the BNF grammar of typed λ-expressions is:

$$\Lambda \ni e ::= x \mid \lambda(x : T).\, e \mid e\, e$$

This grammar is short-hand for the formation judgment '$e \in \Lambda$' of syntactically well-formed λ-expressions. It is defined inductively by three rules:

$$\frac{}{x \in \Lambda} \qquad \frac{e \in \Lambda \quad T \in \mathsf{Type}}{\lambda(x : T).\, e \in \Lambda} \qquad \frac{e_1 \in \Lambda \quad e_2 \in \Lambda}{e_1\, e_2 \in \Lambda}$$

The first rule states that any variable, x, is a well-formed λ-expression. The second rule states that for any well-formed λ-expression, e, and well-formed type, T, the λ-abstraction, $\lambda(x : T).\, e$, is a well-formed λ-expression. Here we appeal to some suitable formation judgment for types, $T \in \mathsf{Type}$. The third rule states that for any two well-formed λ-expressions, e_1 and e_2, the application, $e_1\, e_2$, is a well-formed λ-expression. We will refer to the collection of well-formed expressions defined by such a BNF grammar as a *syntactic domain* or simply a *type*.

We let the use of a meta-variable implicitly require that the formation judgment holds, such as we have done in our use of e in the BNF grammar above. In addition, we use numeric subscripts and primes to refer to possibly distinct instances of the same formation judgment, e.g., e_1, e_2, e', e'', etc.

Types In the above, we purposefully mirror the set-theoretic notation for membership and we liberally use other such notations when appropriate. We let $\mathbb{Z}$ denote the mathematical integers and let $\mathbb{B}$ denote the two-point lattice with top element T, bottom element F and the usual binary operations $\wedge, \vee$, and $\neg$. We use $e \to e_1,\ e_2$ to denote the conditional choice of e_1 if e is T and e_2 if e is F. We denote the type of total functions by $T_1 \to T_2$; the type of partial functions by $T_1 \rightharpoonup T_2$; the type of products by $T_1 \times T_2$ with projections *fst* and *snd*; and the type of sums by $T_1 + T_2$ with injections *inl* and *inr*. We denote the unit type of products by $\mathbf{1}$ and the unit type of sums by $\mathbf{0}$.

On occasions we will make use of the type isomorphism:

$$(T_1 \times T_2) \to T_3 \;=\; T_1 \times T_2 \to T_3 \quad \cong \quad T_1 \to T_2 \to T_3 \;=\; T_1 \to (T_2 \to T_3)$$

The transformation from left to right is called *currying* and the transformation from right to left is called *uncurrying*. In addition, function types are right associative and bind more loosely than all other type constructors, as illustrated by the left-hand equality and right-hand equality in the above equation.

Contexts In general, a context is an expression with holes. In this dissertation, we consider only contexts with exactly one hole. The grammar of unrestricted one-hole contexts for the λ-expressions defined above is:

$$\mathsf{Context} \ni C ::= \Box \mid \lambda(x : T).\, C \mid C\, e \mid e\, C$$

In words, a context is either a hole (i.e., it is the empty context); a λ-abstraction with a one-hole context as its body; a λ-application with a one-hole context in the operator position; or a λ-application with a one-hole context in the operand position.

The judgment '$e = C[e']$' denotes the equality of the expression e with the *composition* of a context C and expression e' where e' replaces the hole in C. This judgment is used to denote both the *recomposition* of a context and an expression to form an expression; and the *decomposition* of an expression into a context and an expression. The latter is not, in general, unique. For example both $e_1\, e_2 = (\Box\, e_2)[e_1]$ and $e_1\, e_2 = (e_1\, \Box)[e_2]$ hold.

Abstract machines The term *'abstract machine'* has several uses in the literature ranging from simply 'an executable implementation' to 'a particular style of executable implementation'. In this dissertation, we use this term to refer to the implementation of a state-transition system, i.e., an iterative and first-order representation of atomic transitions, e.g., a finite-state automaton or a push-down automaton.

Overview

This dissertation is structured in two parts. Part I introduces background material used throughout Part II.

Chapter 2 introduces a simple imperative programming language and shows how to give it a semantics using *denotational* and *operational* specifications. The chapter is concluded with an implementation of a denotational semantics and an implementation of an operational semantics in a functional programming language.

Chapter 3 shows how to derive an *abstract machine* from the implementation of the denotational semantics using the *functional correspondence*.

Chapter 4 shows how to derive an *abstract machine* from the implementation of the operational semantics using the *syntactic correspondence*.

Part II consists of seven articles, four of which are based on previously published articles.

Chapter 5 contains the unpublished article *A prequel to reduction semantics*.

Chapter 6 contains the unpublished article *Normalization functions for Boolean propositional formulas* which is an extended version of a conference article [76].

Chapter 7 contains the journal article *Storeless call-by-need evaluation* which is an extended version of a conference article [75, 77].

Chapter 8 contains the unpublished article *A synthetic operational account of call-by-need evaluation*.

Chapter 9 contains the journal article *The inter-derivation of graph rewriting, reduction, and evaluation* which is an extended version of a conference article [222, 223].

Chapter 10 contains the journal article *Three syntactic theories for combinatory graph reduction* which is an extended version of a conference article [73, 74].

Chapter 11 contains the unpublished article *A logical correspondence between abstract machines and natural semantics*.

Chapter 2

The semantics of programming languages

This chapter introduces the semantic and implementation tools used throughout this dissertation. We exemplify them with the development of several semantics and implementations for a simple imperative programming language, $\mathscr{I}$. This programming language is a synthesis of languages from other accounts about the semantics of programming languages. Indeed, variants of $\mathscr{I}$ appear in most standard textbooks on the semantics of programming languages [148, 154, 178, 186, 201, 206, 219]. Of these, Winskel's IMP language [219, Chapter 2] and Reynolds's *simple imperative language* [178, Chapter 2] are the main sources of inspiration for $\mathscr{I}$. Our successive developments of $\mathscr{I}$ illustrate relationships between the semantics of a language and its implementation. In particular, our developments of $\mathscr{I}$ illustrate how to characterize its computational aspects. In other words, we are interested in the semantic aspects that define not just *what* is computed but also *how* it is computed. The $\mathscr{I}$ language is said to be an imperative programming language because its computational unit is that of a command (or statement). In particular, the execution of a command proceeds by affecting the ambient state of the program.

In this chapter, we consider the *denotational* and *operational* methodologies for defining the semantics of a programming language. The semantics, as specified by these methodologies, together with their implementations, in a functional programming language, form the basis on which the rest of this dissertation is carried out. Other methodologies exist for specifying the semantics of programming languages, e.g., axiomatic semantics [114], categorical semantics [149], and game semantics [2], to name a few. These alternatives and variants are not the topic of this dissertation.

Overview We start by defining the abstract syntax of $\mathscr{I}$ (Section 2.1). This syntax serves as a common definition for the subsequent semantics.

We introduce the notion of a denotational semantics (Section 2.2) with a *direct semantics* for $\mathscr{I}$ (Section 2.2.1). We consider several computational aspects of $\mathscr{I}$ and how they are specified or can be specified by the semantics. Based on these considerations, we give an alternative specification in the form of a *continuation semantics* (Section 2.2.2).

Mirroring the same structure as above, we introduce the notion of an operational semantics (Section 2.3) with a *big-step semantics* for the $\mathscr{I}$ language (Section 2.3.1). Again, we consider the same computational aspects of $\mathscr{I}$ and how they are specified or can be specified by the semantics. Based on these considerations we give two alternative semantics: the first in the form of a *small-step semantics* (Section 2.3.2); the second in the form of a *reduction semantics* (Section 2.3.3).

In conclusion, we discuss the relationship between the denotational and operational semantics as well as the relationship between a semantics and a functional representation or implementation of it (Section 2.4). To this end, we give an adequate representation of the $\mathscr{I}$ syntax in Standard ML (Section 2.4.1) together with a functional implementation of the direct-style semantics (Section 2.4.2), and a functional implementation of the reduction semantics (Section 2.4.3).

2.1 Syntax

In this section, we define the *abstract syntax* of our simple imperative programming language, $\mathscr{I}$. We do not consider aspects related to the parsing of *concrete syntax*, and simply assume that any and all textual treatment has already taken place, e.g., lexing, parsing, or preprocessing. We define each of the syntactic domains with a BNF-style grammar. A BNF grammar inductively defines several syntactic domains by a set of formation rules. Any sentence that can be inductively formed by these rules represents a well-formed sentence in our language. We do not presently assign any meaning to these sentences. Indeed, the following sections illustrate how several distinct meanings can be assigned to well-formed sentences, in effect, defining several distinct programming languages.

When defining the abstract syntax of the language we use an abstract notation for numerals and locations, i.e., we give no specific rules of formation:

$$\begin{aligned} \mathsf{Num} &\ni n \\ \mathsf{Loc} &\ni \ell \end{aligned}$$

Each numeral is a symbolic representation of a specific element in the set of integers. We assume our syntactic representation of numerals is isomorphic to $\mathbb{Z}$ with respect to the numerical operations, e.g., addition, subtraction, and multiplication. The reflection of numerals onto the integers is given by:

$$\mathscr{N} : \mathsf{Num} \to \mathbb{Z}$$

Locations represent addresses in an arbitrarily large address space. We require only that locations can be tested for equality and thus we do not consider features such as allowing arithmetic operations on locations.

We also use a syntactic representation of the truth values:

$$\mathsf{Bool} \ni b ::= \mathsf{true} \mid \mathsf{false}$$

The syntax of the language is defined by three syntactic domains: arithmetic expressions, boolean expressions, and commands. The rules of formation are given by the following BNF grammar:

$$\begin{aligned} \mathsf{AExp} &\ni A ::= n \mid \ell \mid A + A \mid A - A \mid A * A \\ \mathsf{BExp} &\ni B ::= b \mid A <= A \mid \mathsf{not}\ B \mid B\ \mathsf{and}\ B \\ \mathsf{Com} &\ni C ::= \mathsf{skip} \mid C\,;\,C \mid \ell := A \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ B\ \mathsf{do}\ C \end{aligned}$$

In words, an arithmetic expression is either a number, the dereference of a location, an addition expression over two arithmetic sub-expressions, a subtraction expression over two arithmetic sub-expressions, or a multiplication expression over two arithmetic sub-expressions. A boolean expression is either a syntactic truth value, a less-then-or-equal expression over two arithmetic sub-expressions, a negation expression over a boolean sub-expression, or a conjunction expression over two boolean sub-expressions. A command is either a skip command, a sequencing command over two sub-commands, an assignment command over a location and an arithmetic sub-expression, an if-then-else command over a boolean sub-expression and two sub-commands, or a while command over a boolean sub-expression and a sub-command.

It is worth noting that these syntactic domains are not mutually recursive. Arithmetic expressions are inductively defined in terms of numerals, locations, and arithmetic expressions. Boolean expressions are inductively defined in terms of syntactic truth values, arithmetic expressions, and boolean expressions. Commands are inductively defined in terms of all of the above as well as commands. Also, the syntax of $\mathscr{I}$ precludes type-incorrect programs, e.g., only arithmetic expressions can appear in the context of a numerical operator.

Syntactic sugar In the interest of brevity, we have tried to limit the size of the language. Following Landin [131], we can define *derived forms* as 'syntactic sugar' by translating them into *core forms* for which a semantics is given. For example, we might define numeric equality in terms of numeric ordering and conjunction, or define disjunction in terms of conjunction and negation:

$$A_1 = A_2 \quad \text{is defined as} \quad A_1 <= A_2 \text{ and } A_2 <= A_1$$
$$B_1 \text{ or } B_2 \quad \text{is defined as} \quad \text{not } ((\text{not } B_1) \text{ and } (\text{not } B_2))$$

Whether these definitions make sense depends on the semantics of the language. The first equation requires that numeric values define a partially ordered set, specifically that antisymmetry holds, and that duplicating sub-expressions cannot change the meaning of the expression. The second equation requires that the De Morgan laws hold. We will come back to illustrate how exactly such definitions can cause problems.

2.2 Denotational semantics

In this section, we define two denotational semantics for our simple imperative programming language. The first semantics defines the meaning of a program as its final result in one big step (Section 2.2.1). The second semantics defines the meaning of a program as the intermediate result of one small step followed by the successive small steps towards its final result (Section 2.2.2).

Denotational semantics was designed by Christopher Strachey and Dana Scott as a formal method for specifying and developing the semantics of programming languages [190] and it has become a standard methodology [101, 105, 178, 186, 201, 207, 219]. A denotational semantics defines the meaning of an expression by associating it with a denotation: a well-defined mathematical object that denotes this meaning. The hallmark of a denotational semantics is that this association is defined compositionally, i.e., the denotation of an expression is defined in terms of the denotations of its sub-expressions. As a corollary of compositionality, the meaning of an expression is invariant when replacing any of its

sub-expressions by an equivalent sub-expression, i.e., one with the same denotation. In other words, the meaning, or denotation, of an expression does not depend on the context of the expression. This property is sometimes referred to as *referential transparency* and it enables compositional reasoning directly upon the expressions that comprise the language.

2.2.1 A direct semantics

We start by defining a direct semantics for a simple interpretation of $\mathscr{I}$. The semantics is said to be *direct* because most of the syntactic constructs are given meaning directly in terms of their associated mathematical meanings. The qualification of a denotational semantics as being in *direct style* is often used to contrast it with other styles, e.g., *continuation style* [202] which we cover in Section 2.2.2, or *monadic style* [149] which we do not consider here.

To define the denotational semantics, we use the usual notations for λ-expressions. Meta-level abstraction is given by λ and meta-level application by juxtaposition. Other meta-level operations use their usual mathematical notations rendered in math font, e.g., $+$, $-$, and $\times$ denote meta-level addition, subtraction, and multiplication on the integers.

Before we define the denotations of expressions, we must define the meaning of the ambient state of a program. Here we use a total function from locations to the integers:

$$\Sigma = (\mathsf{Loc} \to \mathbb{Z}) \ni \sigma$$

Thus the state is defined for any location without initialization, i.e., all locations exist and hold valid content. An initial state could thus be the constant function mapping all locations to zero:

$$\Sigma \ni \mathsf{init} = \lambda(\ell : \mathsf{Loc}).\, 0$$

Given the meaning of states, we define the denotations of arithmetic expressions to be total functions from states to the integers:

$$\begin{aligned}
\mathscr{A} &: \mathsf{AExp} \to (\Sigma \to \mathbb{Z}) \\
\mathscr{A}[\![n]\!] &= \lambda(\sigma : \Sigma).\, \mathscr{N}(n) \\
\mathscr{A}[\![\ell]\!] &= \lambda(\sigma : \Sigma).\, \sigma(\ell) \\
\mathscr{A}[\![A_1 \texttt{ + } A_2]\!] &= \lambda(\sigma : \Sigma).\, (\mathscr{A}[\![A_1]\!]\sigma) + (\mathscr{A}[\![A_2]\!]\sigma) \\
\mathscr{A}[\![A_1 \texttt{ - } A_2]\!] &= \lambda(\sigma : \Sigma).\, (\mathscr{A}[\![A_1]\!]\sigma) - (\mathscr{A}[\![A_2]\!]\sigma) \\
\mathscr{A}[\![A_1 \texttt{ * } A_2]\!] &= \lambda(\sigma : \Sigma).\, (\mathscr{A}[\![A_1]\!]\sigma) \times (\mathscr{A}[\![A_2]\!]\sigma)
\end{aligned}$$

In words, the meaning of each syntactic construct is compositionally defined in terms of the mathematical meaning associated to the construct together with the meaning of its constituents: The meaning of a numeral in a given state is simply the integer it represents. The meaning of a location in a given state is the contents of that location in that state. The meaning of an addition in a given state is the mathematical meaning of addition as applied to the meaning of the sub-expressions in the same state. The meaning of a subtraction in a given state is the mathematical meaning of subtraction as applied to the meaning of the sub-expressions in the same state. The meaning of a multiplication in a given state is the mathematical meaning of multiplication as applied to the meaning of the sub-expressions in the same state.

Given the meaning of arithmetic expressions, we define the denotations of boolean expressions to be total functions from states to the truth values:

$$\begin{aligned}
\mathscr{B} &: \mathsf{BExp} \to (\Sigma \to \mathbb{B}) \\
\mathscr{B}[\![\mathsf{true}]\!] &= \lambda(\sigma : \Sigma).\, \mathrm{T} \\
\mathscr{B}[\![\mathsf{false}]\!] &= \lambda(\sigma : \Sigma).\, \mathrm{F} \\
\mathscr{B}[\![A_1 \mathsf{<=} A_2]\!] &= \lambda(\sigma : \Sigma).\, (\mathscr{A}[\![A_1]\!]\sigma) \leq (\mathscr{A}[\![A_2]\!]\sigma) \\
\mathscr{B}[\![\mathsf{not}\ B]\!] &= \lambda(\sigma : \Sigma).\, \neg(\mathscr{B}[\![B]\!]\sigma) \\
\mathscr{B}[\![B_1\ \mathsf{and}\ B_2]\!] &= \lambda(\sigma : \Sigma).\, (\mathscr{B}[\![B_1]\!]\sigma) \wedge (\mathscr{B}[\![B_2]\!]\sigma)
\end{aligned}$$

Again, as described in detail for arithmetic expressions, the meaning of each syntactic construct is defined in terms of the mathematical meaning associated to that construct together with the meaning of its constituents. This direct meaning definition is even more transparent for boolean expressions because the state is never used explicitly, it is just passed along.

Given the meaning of arithmetic expressions and boolean expressions, we define the denotations of commands to be partial functions from states to states, i.e., state transformations:

$$\begin{aligned}
\mathscr{C} &: \mathsf{Com} \to (\Sigma \rightharpoonup \Sigma) \\
\mathscr{C}[\![\mathsf{skip}]\!] &= \lambda(\sigma : \Sigma).\, \sigma \\
\mathscr{C}[\![C_1\ \mathsf{;}\ C_2]\!] &= \lambda(\sigma : \Sigma).\, \mathscr{C}[\![C_2]\!](\mathscr{C}[\![C_1]\!]\sigma) \\
\mathscr{C}[\![\ell\ \mathsf{:=}\ A]\!] &= \lambda(\sigma : \Sigma).\, \lambda(\ell' : \mathsf{Loc}).\, \ell = \ell' \to \mathscr{A}[\![A]\!]\sigma,\ \sigma(\ell') \\
\mathscr{C}[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] &= \lambda(\sigma : \Sigma).\, \mathscr{B}[\![B]\!]\sigma \to \mathscr{C}[\![C_1]\!]\sigma,\ \mathscr{C}[\![C_2]\!]\sigma \\
\mathscr{C}[\![\mathsf{while}\ B\ \mathsf{do}\ C]\!] &= \mathit{lfp}(\lambda(f : \Sigma \rightharpoonup \Sigma).\, \lambda(\sigma : \Sigma).\, \mathscr{B}[\![B]\!]\sigma \to f\,(\mathscr{C}[\![C]\!]\sigma),\ \sigma)
\end{aligned}$$

In words: The meaning of skip is the identity transformation on states, i.e., it does not affect the state. The meaning of a sequence command is the composition of the meaning of its sub-commands. The meaning of an assignment is the functional update of a state to map the assigned-to location to the meaning of the arithmetic expression. The meaning of an if command in a particular state is the conditional choice of one of its two sub-commands in the same state depending on the meaning of its predicate. The meaning of a while command is the least fixed point of the function that, if the meaning of its predicate is true, applies its argument (which by means of the fixed point represents the function itself) to the meaning of its sub-command.

We define the full evaluation of a program by applying the denotation of the program to the initial state:

$$\begin{aligned}
\mathscr{E} &: \mathsf{Com} \rightharpoonup \Sigma \\
\mathscr{E}[\![C]\!] &= \mathscr{C}[\![C]\!](\mathrm{init})
\end{aligned}$$

This definition of a direct semantics concludes our first denotational semantics for $\mathscr{I}$. Having defined a semantics for $\mathscr{I}$, we can check to see if our 'syntax sugar' definitions are justified. Consider the boolean expression to check equality of two arithmetic expressions, A_1 = A_2, which desugared to, A_1 <= A_2 and A_2 <= A_1. If we were to define its denotation directly it would be:

$$\mathscr{B}[\![A_1 \mathsf{=} A_2]\!] = \lambda(\sigma : \Sigma).\, (\mathscr{A}[\![A_1]\!]\sigma = \mathscr{A}[\![A_2]\!]\sigma)$$

We expect the denotation of its desugaring to be the same which indeed it is:

$$\begin{aligned}\mathscr{B}[\![A_1 \texttt{ <= } A_2 \texttt{ and } A_2 \texttt{ <= } A_1]\!]\sigma &= \mathscr{B}[\![A_1 \texttt{ <= } A_2]\!]\sigma \wedge \mathscr{B}[\![A_2 \texttt{ <= } A_1]\!]\sigma \\ &= (\mathscr{A}[\![A_1]\!]\sigma \leq \mathscr{A}[\![A_2]\!]\sigma) \wedge (\mathscr{A}[\![A_2]\!]\sigma \leq \mathscr{A}[\![A_1]\!]\sigma) \\ &= (\mathscr{A}[\![A_1]\!]\sigma = \mathscr{A}[\![A_2]\!]\sigma)\end{aligned}$$

Here the final step follows by antisymmetry of the integers. Because the semantics is referentially transparent, we are allowed to replace any occurrence of one by the other which justifies our definition by desugaring.

Computational aspects

The direct semantics for $\mathscr{I}$ characterizes few computational aspects of the language. It characterizes the state transformation *described* by a given program, it does not define the state transformation *prescribed* by a given program. In other words, our semantics describes *what* the meaning of a program is (i.e., the final result, should one exist) but it does not prescribe *how* that meaning is computed, i.e., the computation of a final result. For example, the multiplication in $\mathscr{I}$ is commutative because multiplication on the integers is commutative, i.e., $\mathscr{A}[\![A_1 * A_2]\!] = \mathscr{A}[\![A_2 * A_1]\!]$ for any A_1 and A_2 because $\mathscr{A}[\![A_1]\!]\sigma \times \mathscr{A}[\![A_2]\!]\sigma = \mathscr{A}[\![A_2]\!]\sigma \times \mathscr{A}[\![A_1]\!]\sigma$ for all σ. However, the direct semantics does not specify if the expression A_1 is "evaluated" before or after A_2, or really if it is to be evaluated at all, e.g., the semantics is free to disregard an operand if the denotation of the other maps any given state to zero. When defining languages with a richer set of computational effects, we are often forced to state when and how computation takes place. In the following, we discuss some common computational aspects.

Divergence Divergence, or non-termination, is a basic computational aspects in most languages. In particular, to specify any Turing-complete programming language the semantics must account for divergence. Divergence in $\mathscr{I}$ is accounted for by using partial functions as the denotations of commands. For all commands C and states σ, either $\mathscr{C}[\![C]\!]\sigma = \sigma'$ where σ' is some final state, i.e., the computation terminated in some well-defined state, or $\mathscr{C}[\![C]\!]\sigma = \bot$ where $\bot$ represents the undefined state, i.e., the computation failed to terminate. For example, $\texttt{while true do } C$ unconditionally diverges, and indeed, its denotation is undefined on all input states. To prove this we first recall that the denotation of $\texttt{while true do } C$ was defined by the least fixed point of a function, say F, and so we proceed by mathematical induction on the number of applications of this function to itself defined by:

$$\begin{aligned}F^0 &= \bot \\ F^{n+1} &= \lambda(\sigma : \Sigma).\, \mathscr{B}[\![\texttt{true}]\!]\sigma \to F^n(\mathscr{C}[\![C]\!]\sigma),\ \sigma \\ &= \lambda(\sigma : \Sigma).\, F^n(\mathscr{C}[\![C]\!]\sigma) \\ &= F^n \circ \mathscr{C}[\![C]\!]\end{aligned}$$

The base case holds trivially. The induction case is almost as simple, by the induction hypothesis we have that $F^n = \bot$, and thus $F^n \circ \mathscr{C}[\![C]\!] = \bot$.

For our particular language, if we removed the while command, we could define the denotations of commands to be total functions from states to states. Everything else would remain unchanged. In this sense, divergence is tightly controlled in our semantics of $\mathscr{I}$. For example, the possibility of divergence in commands does not affect the semantics of arithmetic expressions in any way.

Errors A more interesting operational aspect for our particular language would be the possible dereferencing of uninitialized locations. In our current semantics, this possibility is completely ruled out because the state of a program is a total function from locations to the integers. In the following, we briefly describe what needs to change and what choices arise when modeling uninitialized locations in the direct semantics. We refer to this variant of the language as $\mathscr{I}_{\text{err}}$.

We first redefine our notion of state to account for uninitialized locations. Instead of a total function mapping locations to integers, i.e., a total function of type $\mathsf{Loc} \to \mathbb{Z}$, we use a total function mapping locations to either an integer as usual or to a distinguished *error value* if the location is uninitialized, i.e., a total function of type $\mathsf{Loc} \to \mathbb{Z} + \mathbf{1}$. For short, we write $T^\bullet$ for a type T lifted to include errors, i.e., as short hand for the sum type $T + \mathbf{1}$. We write $\bullet$ for the distinguished error value, i.e., as short hand for $inr()$, and we use $\langle t \rangle_{val}$ for a non-error value, i.e., as short for $inl(t)$. We also define some auxiliary notations for working with errors. The mapping of a unary function, $f : T_1 \to T_2$, over errors is defined as:

$$\begin{aligned} &f^\bullet \,:\, {T_1}^\bullet \to {T_2}^\bullet \\ &f^\bullet = \lambda x.\mathit{case}\ x\ \mathit{of}\ \bullet \to \bullet,\ \langle x' \rangle_{val} \to \langle f\ x' \rangle_{val} \end{aligned}$$

The mapping of a binary function, $f : T_1 \times T_2 \to T_3$, over errors is defined as:

$$\begin{aligned} &f^\bullet \,:\, {T_1}^\bullet \times {T_2}^\bullet \to {T_3}^\bullet \\ &f^\bullet = \lambda\langle x_1, x_2 \rangle.\mathit{case}\ x_1\ \mathit{of} \\ &\qquad\qquad \bullet \to \bullet, \\ &\qquad \langle x_1' \rangle_{val} \to \mathit{case}\ x_2\ \mathit{of} \\ &\qquad\qquad\qquad \bullet \to \bullet, \\ &\qquad\qquad\qquad \langle x_2' \rangle_{val} \to \langle f\ \langle x_1', x_2' \rangle \rangle_{val} \end{aligned}$$

The mapping of a conditional expression over errors is defined as:

$$x_1 \overset{\bullet}{\to} x_2,\, x_3 \iff \mathit{case}\ x_1\ \mathit{of}\ \bullet \to \bullet,\ \langle x_1' \rangle_{val} \to x_1' \to x_2,\ x_3$$

Thus equipped, the ambient state of a program is defined as a total function from locations to the integers lifted over errors:

$$\Sigma_{\text{err}} = (\mathsf{Loc} \to \mathbb{Z}^\bullet) \ni \sigma$$

A reasonable initial state could then be the state that is uninitialized at every location, i.e., the constant function from locations to the error value:

$$\Sigma_{\text{err}} \ni \text{init}_{\text{err}} = \lambda(\ell : \mathsf{Loc}).\, \bullet$$

Having redefined our notion of state, we must redefine the denotations of our language. In particular, the dereference of a location can fail, and since dereferencing a location is an arithmetic expression, an arithmetic expression can fail. Thus, the type of our denotations change to include errors and we map each of the numeric operators over the error value:

$$\begin{aligned} \mathscr{A} &: \mathsf{AExp} \to (\Sigma_{\text{err}} \to \mathbb{Z}^\bullet) \\ \mathscr{A}[\![n]\!] &= \lambda(\sigma : \Sigma_{\text{err}}).\, \langle \mathscr{N}(n) \rangle_{val} \\ \mathscr{A}[\![\ell]\!] &= \lambda(\sigma : \Sigma_{\text{err}}).\, \sigma(\ell) \\ \mathscr{A}[\![A_1 + A_2]\!] &= \lambda(\sigma : \Sigma_{\text{err}}).\, (\mathscr{A}[\![A_1]\!]\sigma) +^\bullet (\mathscr{A}[\![A_2]\!]\sigma) \\ \mathscr{A}[\![A_1 \text{ - } A_2]\!] &= \lambda(\sigma : \Sigma_{\text{err}}).\, (\mathscr{A}[\![A_1]\!]\sigma) -^\bullet (\mathscr{A}[\![A_2]\!]\sigma) \\ \mathscr{A}[\![A_1 * A_2]\!] &= \lambda(\sigma : \Sigma_{\text{err}}).\, (\mathscr{A}[\![A_1]\!]\sigma) \times^\bullet (\mathscr{A}[\![A_2]\!]\sigma) \end{aligned}$$

We must redefine the denotations for boolean expressions and for commands in a similar way. However, the denotations for boolean expressions with errors expose other operational aspects of our language, in particular that of evaluation order.

Evaluation order and errors For our simple language, the inclusion of errors exposes the order of evaluation in boolean expressions.[1] *Evaluation order* refers to the order an implementation can use to evaluate the sub-expressions of an expression. For example, the evaluation order of an arithmetic expression is not determined by our semantics, even after the inclusion of errors. An implementation can choose to evaluate the sub-expressions of an addition expression from left to right, from right to left, or even in some arbitrary interleaving. The choice of concrete order cannot be observed.

The choice of evaluation order can be observed when we define the meaning of and expressions in the presence of errors. We might define its denotation by simply mapping conjunction over errors:

$$\mathscr{B}[\![B_1 \text{ and } B_2]\!] = \lambda(\sigma : \Sigma_{\text{err}}).\, (\mathscr{B}[\![B_1]\!]\sigma) \wedge^{\bullet} (\mathscr{B}[\![B_2]\!]\sigma)$$

In this case, the meaning of an and expression is the error value if either the meaning of the left-hand side or of the right-hand side is an error. This definition maps all errors in sub-expressions to an error of the composite expression. For such a semantics, a correct implementation must fully evaluate both sub-expressions in the non-error case, but could do so in any order.

We could also define the meaning of an and expression as false if the meaning of either of its sub-expressions is false, true if both are true, and the error value otherwise:

$$\begin{aligned}\mathscr{B}[\![B_1 \text{ and } B_2]\!] = \lambda(\sigma : \Sigma_{\text{err}}).\, (\mathscr{B}[\![B_1]\!]\sigma = \langle \mathrm{F} \rangle_{val}) \vee (\mathscr{B}[\![B_2]\!]\sigma = \langle \mathrm{F} \rangle_{val}) \rightarrow \\ \langle \mathrm{F} \rangle_{val}\,, \\ (\mathscr{B}[\![B_1]\!]\sigma) \wedge^{\bullet} (\mathscr{B}[\![B_2]\!]\sigma)\end{aligned}$$

This definition maps as few errors as possible to errors. For such a semantics, a correct implementation can choose to stop evaluation as soon as a sub-expression is found to be false. As before, the evaluation can happen in any order.

A more traditional choice is to define the meaning by a sequence of conditionals:

$$\mathscr{B}[\![B_1 \text{ and } B_2]\!] = \lambda(\sigma : \Sigma_{\text{err}}).\, \mathscr{B}[\![B_1]\!]\sigma \overset{\bullet}{\rightarrow} \mathscr{B}[\![B_2]\!]\sigma,\, \langle \mathrm{F} \rangle_{val}$$

This definition maps any error in B_1 to an error, but only if the meaning of B_1 is true is B_2 considered. For such a semantics, a correct implementation must consider the sub-expression one after the other (here from left to right) and stop if a value can be determined. This left-to-right order is commonly found in programming languages and often referred to as a *short-circuiting* operation. Under this semantics, a tempting but unsound compiler optimization would be to replace a boolean-and expression with a right-hand side equal to false by false, i.e., 'B_1 and false' by 'false'. However, these two expressions have distinct denotations in the short-circuiting semantics and are only equal under the condition that $\mathscr{B}[\![B_1]\!]\sigma \neq \bullet$ for all states σ.

[1] In many languages, e.g., the λ-calculus, divergence is sufficient to expose the order of evaluation. That is not the case for $\mathscr{I}$ because arithmetic expressions and boolean expressions are defined as total functions and commands are explicitly sequenced.

Another case where we must make a choice after introducing errors is for assignment commands. We can simply reuse our previous definition unchanged:

$$\mathscr{C}[\![\ell := A]\!] = \lambda(\sigma : \Sigma_{\text{err}}).\, \lambda(\ell' : \mathsf{Loc}).\, \ell = \ell' \to \mathscr{A}[\![A]\!]\sigma,\ \sigma(\ell')$$

Or we can sequence the test for errors outside the update of the state:

$$\begin{aligned}\mathscr{C}[\![\ell := A]\!] = \lambda(\sigma : \Sigma_{\text{err}}).\, \mathit{case}\ \mathscr{A}[\![A]\!]\sigma\ \mathit{of}\\ \bullet \to \bullet,\\ \langle i\rangle_{\mathit{val}} \to \lambda(\ell' : \mathsf{Loc}).\, \ell = \ell' \to \langle i\rangle_{\mathit{val}},\ \sigma(\ell')\end{aligned}$$

The former has the effect of delaying the computation of the right-hand side whereas the latter strictly maps any error in the right-hand side to an error of the assignment command itself. This difference is reminiscent of the call-by-name and call-by-value evaluation orders. In the former, we associate to the location the entire computation of the right-hand side, including errors. In the latter, we associate to the location only the non-error value, i.e., it is established at this point that a non-error value exists. This latter evaluation order is the most commonly used evaluation order in imperative programming languages.

Evaluation order and assignment expressions A common feature that entails similar considerations regarding the order of evaluation is to include assignments in the grammar of arithmetic expressions. In other words, we might move assignment from commands into arithmetic expressions. Immediately we are forced to change the denotations of arithmetic expressions to be total functions from states to an integer value in an updated state (here for the non-error semantics):

$$\mathscr{A} : \mathsf{AExp} \to (\Sigma \to \mathbb{Z} \times \Sigma)$$

The denotation of an assignment expression might then be:

$$\begin{aligned}\mathscr{A}[\![\ell := A]\!] = \lambda(\sigma : \Sigma).\, \langle i,\, \lambda(\ell' : \mathsf{Loc}).\, \ell = \ell' \to i,\ \sigma'(\ell')\rangle\\ \text{where } \langle i,\, \sigma'\rangle = \mathscr{A}[\![A]\!]\sigma\end{aligned}$$

In words, first we determine the integer value and updated state for the right-hand side. Then we pair the integer value with the functional update of the state that maps the location to the integer value. This pair is the final result of the set expression.

All of the other definitions of our semantics must now be updated to account for the possible change of state in a sub-expression. In practice, this leaves two choices for the semantics: either define any program that depends upon the order as "undefined", or explicitly thread[2] the state throughout all affected parts of the semantics.

Our semantics of $\mathscr{I}$ already explicitly threads the state for commands, and we can simply do so for arithmetic expressions and boolean expressions too. For example, the following defines a left-to-right evaluation order for `<=`:

$$\begin{aligned}\mathscr{B}[\![A_1 \texttt{<=} A_2]\!] = \lambda(\sigma : \Sigma).\, \langle i_1 \le i_2,\, \sigma''\rangle\\ \text{where } \langle i_1,\, \sigma'\rangle = \mathscr{A}[\![A_1]\!]\sigma\\ \text{and } \langle i_2,\, \sigma''\rangle = \mathscr{A}[\![A_2]\!]\sigma'\end{aligned}$$

[2] A parameter is threaded if it is sequenced through a chain of function calls, i.e., if it is passed as an argument and returned as a result which is then passed as the argument for the subsequent call, and so forth. For example, the state is threaded in the denotational definitions of commands. These definitions are sometimes said to be in 'state-passing style' to emphasize the threaded state.

This change has consequences. For example, our desugaring of equality on arithmetic expressions is not justified anymore. If we were to define the denotation directly it would be:

$$\begin{aligned}\mathscr{B}[\![A_1 = A_2]\!] = \lambda(\sigma : \Sigma).\, \langle i_1 = i_2,\, \sigma''\rangle \\ \text{where } \langle i_1,\, \sigma'\rangle = \mathscr{A}[\![A_1]\!]\sigma \\ \text{and } \langle i_2,\, \sigma''\rangle = \mathscr{A}[\![A_2]\!]\sigma'\end{aligned}$$

For this definition the denotation of $1 = (\ell := \ell + 1)$ applied to the initial state is true:

$$\mathscr{B}[\![1 = (\ell := \ell + 1)]\!]\,\text{init} = \text{T}$$

In contrast, the denotation of the desugaring applied to the initial state is false:

$$\mathscr{B}[\![1 <= (\ell := \ell + 1) \text{ and } (\ell := \ell + 1) <= 1]\!]\,\text{init} = \text{F}$$

This is because the state can now be changed while it is threaded through the sub-expressions. The desugared expression duplicates the sub-expressions thus causing the assignment to be done twice.

Control All of the computational aspects above can be related to the general concept of control flow: how the execution of a program takes place. We can often identify aspects of this control flow, e.g., by inspecting differences in behavior in the presence of errors or other computational effects. However, we can also make this control flow explicit as a construct in the semantics itself: a continuation.

2.2.2 A continuation semantics

A *continuation* is a functional abstraction of what remains to be computed by the program at a particular point. The concept of a continuation was independently developed for several purposes, among them, the ability to express unstructured jumps, i.e., goto commands [133, 177]. In this section, we define a continuation semantics and show how it can express the computational aspects discussed in Section 2.2.1.[3]

Concretely, a continuation is a function that represents the "rest of the computation", thus its input is what has been computed and its output is then the final result of the entire computation, should one exist. As defined just above, the denotation of arithmetic expressions after explicitly threading the state was a function mapping states to pairs of integers and updated states. Thus, the continuation of an arithmetic expression is a partial function that takes as input both the integer that has been computed along with the updated state and it produces the final result. We need not know the exact type of this "final result" and so we refer to it abstractly as the answer type: α. The continuation semantics defines the denotation of an arithmetic expression as the total function from continuations on arithmetic expressions to partial functions from states to answers, where

[3] Indeed, Filinski has shown that continuations define a universal monadic effect, i.e., all other monadic effects can be mapped to the continuation monad [90, 91].

a continuation is a partial function from integers and states to answers: $\mathbb{Z} \to \Sigma \rightharpoonup \alpha$.

$$
\begin{aligned}
\mathscr{A}_c : \mathsf{AExp} \to (\mathbb{Z} \to \Sigma \rightharpoonup \alpha) \to (\Sigma \rightharpoonup \alpha)& \\
\mathscr{A}_c[\![n]\!]\kappa &= \kappa(\mathscr{N}(n)) \\
\mathscr{A}_c[\![\ell]\!]\kappa &= \lambda(\sigma : \Sigma).\, \kappa(\sigma(\ell))\,\sigma \\
\mathscr{A}_c[\![A_1 \texttt{ + } A_2]\!]\kappa &= \mathscr{A}_c[\![A_1]\!](\lambda(i_1 : \mathbb{Z}).\, \mathscr{A}_c[\![A_2]\!](\lambda(i_2 : \mathbb{Z}).\, \kappa(i_1 + i_2))) \\
\mathscr{A}_c[\![A_1 \texttt{ - } A_2]\!]\kappa &= \mathscr{A}_c[\![A_1]\!](\lambda(i_1 : \mathbb{Z}).\, \mathscr{A}_c[\![A_2]\!](\lambda(i_2 : \mathbb{Z}).\, \kappa(i_1 - i_2))) \\
\mathscr{A}_c[\![A_1 \texttt{ * } A_2]\!]\kappa &= \mathscr{A}_c[\![A_1]\!](\lambda(i_1 : \mathbb{Z}).\, \mathscr{A}_c[\![A_2]\!](\lambda(i_2 : \mathbb{Z}).\, \kappa(i_1 \times i_2)))
\end{aligned}
$$

The treading of state is implicit, or in point-free style, in this description. The sequencing of continuations determines the order of evaluation and, by extension, how the state is sequenced through sub-expressions. The state can be made explicit by simply η-expanding it throughout.

The denotations of boolean expressions are defined in the same way as for arithmetic expressions. The continuations are partial functions from truth values and states to answers.

$$
\begin{aligned}
\mathscr{B}_c : \mathsf{BExp} \to (\mathbb{B} \to \Sigma \rightharpoonup \alpha) \to (\Sigma \rightharpoonup \alpha)& \\
\mathscr{B}_c[\![\mathsf{true}]\!]\kappa &= \kappa(\mathrm{T}) \\
\mathscr{B}_c[\![\mathsf{false}]\!]\kappa &= \kappa(\mathrm{F}) \\
\mathscr{B}_c[\![A_1 \texttt{ <= } A_2]\!]\kappa &= \mathscr{A}_c[\![A_1]\!](\lambda(i_1 : \mathbb{Z}).\, \mathscr{A}_c[\![A_2]\!](\lambda(i_2 : \mathbb{Z}).\, \kappa(i_1 \leq i_2))) \\
\mathscr{B}_c[\![\mathsf{not}\ B]\!]\kappa &= \mathscr{B}_c[\![B]\!](\lambda(b : \mathbb{B}).\, \kappa(\neg b)) \\
\mathscr{B}_c[\![B_1\ \mathsf{and}\ B_2]\!]\kappa &= \mathscr{B}_c[\![B_1]\!](\lambda(b_1 : \mathbb{B}).\, b_1 \to \mathscr{B}_c[\![B_2]\!]\kappa,\ \kappa(\mathrm{F}))
\end{aligned}
$$

The denotations of a command also take an extra parameter, the continuation κ, which is a partial function from states to final answers: $\Sigma \rightharpoonup \alpha$. The domain of the continuation is Σ because the intermediate result in the direct semantics, after supplying an initial state, was a state. In other words, the continuation takes such an intermediate state and produces the final result, if one exists.

$$
\begin{aligned}
\mathscr{C}_c : \mathsf{Com} \to (\Sigma \rightharpoonup \alpha) \to (\Sigma \rightharpoonup \alpha)& \\
\mathscr{C}_c[\![\mathsf{skip}]\!]\kappa &= \kappa \\
\mathscr{C}_c[\![C_1\texttt{ ; }C_2]\!]\kappa &= \mathscr{C}_c[\![C_1]\!](\mathscr{C}_c[\![C_2]\!]\kappa) \\
\mathscr{C}_c[\![\ell \texttt{ := } A]\!]\kappa &= \mathscr{A}_c[\![A]\!](\lambda(i : \mathbb{Z}).\, \lambda(\sigma : \Sigma).\, \kappa(\lambda(\ell' : \mathsf{Loc}).\, \ell = \ell' \to i,\ \sigma(\ell'))) \\
\mathscr{C}_c[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]\kappa &= \mathscr{B}_c[\![B]\!](\lambda(b : \mathbb{B}).\, b \to \mathscr{C}_c[\![C_1]\!]\kappa,\ \mathscr{C}_c[\![C_2]\!]\kappa) \\
\mathscr{C}_c[\![\mathsf{while}\ B\ \mathsf{do}\ C]\!]\kappa &= \mathit{lfp}(\lambda(f : (\Sigma \rightharpoonup \alpha) \to (\Sigma \rightharpoonup \alpha)).\, \\
&\qquad \lambda(\kappa' : \Sigma \rightharpoonup \alpha).\, \\
&\qquad\quad \mathscr{B}_c[\![B]\!](\lambda(b : \mathbb{B}).\, b \to \mathscr{C}_c[\![C]\!](f\ \kappa'),\ \kappa'))\,\kappa
\end{aligned}
$$

The denotation of a command is now a total function that produces a state transformation when given a continuation. In other words, the denotation is now a *transformation* on *state transformations*. The initial continuation for a program is the identity transformation on state transformations, i.e., the identity function. The evaluation of a program is thus the denotation of this program applied to the initial continuation and the initial state:

$$
\begin{aligned}
\mathscr{E}_c : \mathsf{Com} &\rightharpoonup \Sigma \\
\mathscr{E}_c[\![C]\!] &= \mathscr{C}_c[\![C]\!](\lambda(\sigma : \Sigma).\, \sigma)(\mathrm{init})
\end{aligned}
$$

By using the identity function we have forced the type of final answers to be states.

This continuation semantics is in *continuation-passing style*, or CPS for short. Namely, the continuation is linearly passed along, all function applications are in *tail position*, i.e., any function application is the last thing to do, and finally, instead of returning a result it is given to the continuation to produce the final result. This definition in CPS can be mechanically obtained using a mechanical and fully-correct CPS transformation. Section 3.1 covers this transformation in more detail.

Computational aspects

In contrast to the direct semantics of Section 2.2.1, the continuation semantics embodies many computational aspects of the language. Indeed, the purpose of a continuation semantics is exactly to specify these computational aspects. We review the same aspects as for the direct semantics starting with errors since divergence is still accounted for by a least fixed point, albeit the least fixed point of a more complicated function.

Errors We can define the continuation semantics for the language with errors $\mathscr{I}_{\text{err}}$ with hardly a change to the semantic definitions. Reusing the definition of states with possibly uninitialized locations, we simply restate the denotation of dereferencing a location:

$$\mathscr{A}_c[\![\ell]\!]\kappa = \lambda(\sigma : \Sigma).\, case\ \sigma(\ell)\ of\ \bullet \to \bullet,\ \langle i\rangle_{val} \to \kappa(i)\,\sigma$$

This definition simply disregards its continuation if a location is uninitialized, thereby terminating the entire computation with that error as the final answer. Thus the answer type must be the state type lifted to include errors $\alpha = \Sigma^{\bullet}$ and the initial continuation becomes the non-error injection into this answer type $\lambda(\sigma : \Sigma).\, \langle\sigma\rangle_{val}$. This continuation semantics is not in CPS because the definition above does not invoke the continuation in the case of error.

Evaluation order Interestingly, we did not need to consider errors anywhere except when dereferencing a location. In particular, we did not state how errors propagate though sub-expressions. Not even for the short-circuiting `and` expression. Indeed, the evaluation order of the continuation semantics has already been completely specified [176] and we need not reconsider it when we introduce uninitialized locations.

Control Instead of modeling unstructured jumps, we consider structured exceptions in terms of the `throw` and `try-catch` control operators. We call this extension of $\mathscr{I}$ with exceptions $\mathscr{I}_{\text{exc}}$. The syntax of commands is extended with two operators:

$$\mathsf{Com} \ni C ::= \cdots \mid \mathsf{throw} \mid \mathsf{try}\ C\ \mathsf{catch}\ C$$

A `throw` command has no sub-expressions and has the effect of aborting the computation in the current state. A `try-catch` command has two sub-commands, the first is the command to 'try', i.e., this command is executed until completion or until an exception is thrown. The second sub-command is executed if and only if an exception is thrown. We start by specifying how to throw an exception. Just as in the case of errors, throwing an exception aborts the current continuation, but instead of returning an error value we return the current state. As for errors the answer type is the sum type of either a *normal*

state (the left injection) or an *exception state* (the right injection): $\alpha = \Sigma + \Sigma$. The initial continuation produces a normal state using the left injection of the sum type:

$$\lambda(\sigma : \Sigma).\, inl(\sigma)$$

The denotations of commands are now partial functions from states to the answer type of normal states or exception states. We extend the definitions with a case for throw and for try-catch:

$$\begin{aligned}
\mathscr{C}_c : \mathsf{Com} \to (\Sigma \rightharpoonup \Sigma + \Sigma) \to (\Sigma \rightharpoonup \Sigma + \Sigma)&\\
\mathscr{C}_c[\![\mathsf{throw}]\!]\kappa &= \lambda(\sigma : \Sigma).\, inr(\sigma)\\
\mathscr{C}_c[\![\mathsf{try}\ C_1\ \mathsf{catch}\ C_2]\!]\kappa &= \lambda(\sigma : \Sigma).\, case\ \mathscr{C}_c[\![C_1]\!](\lambda(\sigma : \Sigma).\, inl(\sigma))\,\sigma\ of\\
&\qquad inl(\sigma') \to \kappa(\sigma'),\\
&\qquad inr(\sigma') \to \mathscr{C}_c[\![C_2]\!]\kappa\,\sigma'
\end{aligned}$$

The throw command produces an exception state using the right injection of the sum type. The try-catch command delimits the extent of the continuation by passing the initial continuation to the denotation of the first sub-command and case-discriminating on the result which is of answer type. If the result is a normal state then this state is passed on to the current continuation, i.e., execution proceeds as usual. If the result is an exception state then the meaning is given in terms of the second sub-command, i.e., execution continues in the second sub-command with the current continuation and the new state.

Again, this continuation semantics for $\mathscr{I}_{\mathrm{exc}}$ is not in CPS because the denotation of the throw command does not invoke its continuation and the recursive call in the denotation of a try-catch command is not in tail position. However, this continuation semantics can be transformed again into CPS yielding a continuation semantics in CPS with not one, but two continuations: a delimited continuation which is the continuation in the above semantics and a new meta-continuation [64]. For our semantics these two continuations are reminiscent of the success and failure continuations often used to describe the semantics of errors [208]. Indeed, these two continuations can be simplified to exactly a success continuation (the delimited continuation of the above semantics) and a failure continuation (the meta continuation obtained by CPS transforming the above semantics).

2.3 Operational semantics

In this section, we define three operational semantics for $\mathscr{I}$. The semantics in Section 2.3.1 is in *big-step style* and defines the meaning of a program as its final result, thereby mirroring the direct semantics of Section 2.2.1. The semantics in Section 2.3.2 is in *small-step style* and defines the meaning of a program by its intermediate results. The semantics in Section 2.3.3 is in the style of a *reduction semantics* and adds to the small-step style an explicit representation of the context, thereby mirroring the continuation semantics of Section 2.2.2.

In contrast to a denotational semantics, an operational semantics specifies rules for manipulating symbols and the meaning of an expression is the process or result of performing these manipulations. In this sense, an operational semantics defines, at face value, only *how* computation takes place but not *what* the computation is describing. The rules given by an operational semantics can reflect the program execution in a more or less abstract way. Historically, rules would specify the execution of a program on some abstract machine, possibly after first translating the source program into a machine-language

program. A seminal example of this is Landin's SECD machine for evaluating lambda expressions [131] and his subsequent definition of the Algol programming language by translation into these lambda expressions [132]. In this sense, the notion of an operational semantics has a long history predating that of denotational semantics. However, operational semantics has subsequently undergone vast changes in its use of formal methods and in its styles of presentation. A major turning point was Plotkin's design of *structural operational semantics* where the operational semantics is given by operational rules that are defined structurally on expressions of the source language. Thus, the operational rules are given directly for expressions of the language instead of indirectly by translation to another and often more complicated model. The subsequent development of standard proof techniques for relating static and dynamic aspects of a language semantics [220] further cemented the use of operational semantics as a standard tool in specifying the semantics of programming languages [89, 108, 148, 154, 166, 219].

In the following sections, we define each of the operational semantics in a purely syntactic form to clearly contrast it with the denotational semantics of the previous section. Other accounts are often more relaxed when it comes to mixing symbolic definitions with mathematical constructs.

2.3.1 A big-step structural operational semantics

Using the same syntax for $\mathscr{I}$, we define a structural operational semantics for it. As in Section 2.2.1, each syntactic construct is given meaning directly in terms of the associated mathematical meaning for that construct. However, the value or meaning of a computation is a *symbolic representation* as opposed to a *mathematical representation*. The semantics is given in a *big-step* style also known as a natural semantics [123]. This is because each judgment associates a syntactic construct to its final symbolic value in one big step.

Before we define the operational rules for expressions, we must define the symbolic representation of the ambient state of a program. Here the state is a store associating locations to numerals:

$$\mathsf{Store} \ni \sigma ::= \varepsilon \mid \sigma[\ell \mapsto n]$$

In words, the store is either empty, ε, or it contains an association of location, ℓ, with a value, n, together with some sub-store, σ. We use an auxiliary judgment to judge whether a location is associated with a value in a store defined by the following rules:

$$\frac{}{\varepsilon(\ell) = 0} \qquad \frac{}{\sigma[\ell \mapsto n](\ell) = n} \qquad \frac{\ell \neq \ell' \quad \sigma(\ell) = n}{\sigma[\ell' \mapsto n'](\ell) = n}$$

This judgment for looking up a location in a store defines a total function and can be proved by induction of the structure of the store. The first rule explicitly associates any unallocated location to the numerical representation of zero.

Given the definition of a store, we define the evaluation judgment for arithmetic expressions: $\Downarrow_A$. This judgment associates to each arithmetic expression with a given store its numeric value:

$$\Downarrow_A \subseteq \mathsf{AExp} \times \mathsf{Store} \times \mathsf{Num}$$

The following rules define the judgment using infix notation:

$$\frac{}{\langle n, \sigma\rangle \Downarrow_A n} \qquad \frac{\sigma(\ell) = n}{\langle \ell, \sigma\rangle \Downarrow_A n}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A n_1 \quad \langle A_2, \sigma\rangle \Downarrow_A n_2 \quad \mathcal{N}(n) = \mathcal{N}(n_1) + \mathcal{N}(n_2)}{\langle A_1 + A_2, \sigma\rangle \Downarrow_A n}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A n_1 \quad \langle A_2, \sigma\rangle \Downarrow_A n_2 \quad \mathcal{N}(n) = \mathcal{N}(n_1) - \mathcal{N}(n_2)}{\langle A_1 \text{ - } A_2, \sigma\rangle \Downarrow_A n}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A n_1 \quad \langle A_2, \sigma\rangle \Downarrow_A n_2 \quad \mathcal{N}(n) = \mathcal{N}(n_1) \times \mathcal{N}(n_2)}{\langle A_1 * A_2, \sigma\rangle \Downarrow_A n}$$

Each syntactic construct is given a value in terms of its associated mathematical meaning together with the value of its constituents: The value of a numeral with a given store is the numeral itself. The value of a location with a given store is the value associated with that location in the store. The value of an addition with a given store is the numeral that represents the mathematical meaning of addition as applied to the integers represented by the values of the sub-expressions. We define the value of a subtraction or multiplication similar to addition. The evaluation judgment $\Downarrow_A$ defines a total function from arithmetic expressions and stores to numerals. This can be proved by induction over the structure of arithmetic expressions.

Given the definition for arithmetic expressions, we define the evaluation judgment for boolean expressions: $\Downarrow_B$. This judgment associates to each boolean expressions with a given store its symbolic truth value:

$$\Downarrow_B \subseteq \mathsf{BExp} \times \mathsf{Store} \times \mathsf{Bool}$$

The following rules define the judgment using infix notation:

$$\frac{}{\langle b, \sigma\rangle \Downarrow_B b}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A n_1 \quad \langle A_2, \sigma\rangle \Downarrow_A n_2 \quad \mathcal{N}(n_1) \leq \mathcal{N}(n_2)}{\langle A_1 \text{ <= } A_2, \sigma\rangle \Downarrow_B \mathsf{true}}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A n_1 \quad \langle A_2, \sigma\rangle \Downarrow_A n_2 \quad \mathcal{N}(n_1) \not\leq \mathcal{N}(n_2)}{\langle A_1 \text{ <= } A_2, \sigma\rangle \Downarrow_B \mathsf{false}}$$

$$\frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{true}}{\langle \mathsf{not}\ B, \sigma\rangle \Downarrow_B \mathsf{false}} \qquad \frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{false}}{\langle \mathsf{not}\ B, \sigma\rangle \Downarrow_B \mathsf{true}}$$

$$\frac{\langle B_1, \sigma\rangle \Downarrow_B \mathsf{true} \quad \langle B_2, \sigma\rangle \Downarrow_B \mathsf{true}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \mathsf{true}} \qquad \frac{\langle B_1, \sigma\rangle \Downarrow_B \mathsf{false}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \mathsf{false}} \qquad \frac{\langle B_2, \sigma\rangle \Downarrow_B \mathsf{false}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \mathsf{false}}$$

As described in detail for the rules of arithmetic expressions, we define the value of each syntactic construct in terms of the associated mathematical construct together with the value of the sub-expressions. Again, the evaluation judgment $\Downarrow_B$ defines a total function from boolean expressions and stores to syntactic truth values. This can be proved by induction over the structure of boolean expressions.

Given the definition for arithmetic expressions and boolean expressions, we define the evaluation judgment for commands $\Downarrow_C$. This judgment associates to each command with a given store its final store value, should one exist:

$$\Downarrow_C \subseteq \mathsf{Com} \times \mathsf{Store} \times \mathsf{Store}$$

The following rules define the judgment using infix notation:

$$\frac{}{\langle \mathsf{skip}, \sigma\rangle \Downarrow_C \sigma} \qquad \frac{\langle C_1, \sigma\rangle \Downarrow_C \sigma' \quad \langle C_2, \sigma'\rangle \Downarrow_C \sigma''}{\langle C_1 ; C_2, \sigma\rangle \Downarrow_C \sigma''} \qquad \frac{\langle A, \sigma\rangle \Downarrow_A n}{\langle \ell := A, \sigma\rangle \Downarrow_C \sigma[\ell \mapsto n]}$$

$$\frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{true} \quad \langle C_1, \sigma\rangle \Downarrow_C \sigma'}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma\rangle \Downarrow_C \sigma'} \qquad \frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{false} \quad \langle C_2, \sigma\rangle \Downarrow_C \sigma'}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma\rangle \Downarrow_C \sigma'}$$

$$\frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{true} \quad \langle C, \sigma\rangle \Downarrow_C \sigma' \quad \langle \mathsf{while}\ B\ \mathsf{do}\ C, \sigma'\rangle \Downarrow_C \sigma''}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, \sigma\rangle \Downarrow_C \sigma''} \qquad \frac{\langle B, \sigma\rangle \Downarrow_B \mathsf{false}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, \sigma\rangle \Downarrow_C \sigma}$$

In words: The value of skip with a given store is the store. The value of a sequence command with a given store is the store obtained by threading the given store through the sequence of sub-commands. The value of an assignment with a given store is the extension of the store that associates the assigned-to location with the value of the arithmetic expression. The value of an if command with a given store is the value of the first or second sub-command depending on the value of the predicate. The value of a while command with a given store is the store if the value of the predicate is false. If the value of the predicate is true, then the value is the store obtained by threading the given store once through the sub-command and then through the same syntactic while command. The evaluation judgment $\Downarrow_C$ defines a partial function. This can be proved by structural induction on derivations of $\Downarrow_C$.

Finally we define program evaluation as command evaluation starting with some initial store, e.g., the empty store:

Definition 1 (program evaluation)**.** A program C evaluates to the final store σ if and only if, $\langle C, \varepsilon\rangle \Downarrow_C \sigma$ holds.

The evaluation of a program is deterministic as a corollary of $\Downarrow_C$ defining a partial function.

This definition of a big-step semantics concludes our first operational semantics for $\mathscr{I}$. The definition is similar to the direct semantics of Section 2.2.1 but differs one important way. The definition for while commands is non-compositional: it is defined recursively on the while command itself, which is not a proper sub-expression. Indeed, this semantics can only define the meaning of terminating expressions as we will discuss in the following section.

Computational aspects

Similar to the direct-style denotational semantics of Section 2.2.1, this big-step structural operational semantics characterizes few computational aspects of the language. Compared to the direct semantics, extending the big-step semantics to handle errors and account for evaluation order requires more changes: we end up duplicating rules to account for, e.g., normal evaluations and erroneous evaluations. This duplication is largely due to the lack of higher-order abstractions in the big-step semantics caused by our insistence on using a purely syntactic account.

Divergence The rules defined for the while command admit divergence and as a result they violate the principle of structural recursion. Indeed, the value definition of a while command is defined recursively in terms of the value of *the same command* where only the store has potentially changed. In other words, there is no decreasing measure in our

definition and there might not be a value associated to a while command. For example, while true do C unconditionally diverges, and indeed, our semantics cannot associate a value to this term, i.e., there exist no finite derivation for this command. Assume we have a finite derivation $\mathscr{D}$ with sub-derivations $\mathscr{E}_1$ and $\mathscr{E}_2$ that associates a value to this command:

$$\frac{\dfrac{}{\langle \mathsf{true}, \sigma\rangle \Downarrow_B \mathsf{true}} \quad \dfrac{\mathscr{E}_1}{\langle C, \sigma\rangle \Downarrow_C \sigma'} \quad \dfrac{\mathscr{E}_2}{\langle \mathsf{while\ true\ do}\ C, \sigma'\rangle \Downarrow_C \sigma''}}{\langle \mathsf{while\ true\ do}\ C, \sigma\rangle \Downarrow_C \sigma''}$$

Here the derivation $\mathscr{E}_2$ must recursively contain the entire derivation $\mathscr{D}$ where σ' has been substituted for σ. This causes the sub-derivation to be strictly larger than the entire derivation. This proves the non-existence of a finite derivation of $\mathscr{E}_2$ and thus the non-existence of a value for the command.

Errors There are several ways to account for errors in the big-step semantics. We might consider the inability to continue computation an "error". In other words, at a particular point during evaluation there exists no rule that can be applied to the current expression which is then said to be *stuck* and as a result the entire computation has *gone wrong*. Just as divergence manifests itself as an infinitely expanding derivation, errors manifest themselves as the impossibility of expanding an incomplete derivation. A problem with this approach is that, from a formal standpoint, both of these aspects amount to the non-existence of a derivation which is not immediately distinguishable. From a computational standpoint they are different. An error is the property of halting in an unexpected state whereas divergence is the property of never halting.

Alternatively we can introduce a distinguished value denoting an error, just as we did for the direct semantics of Section 2.2.1. We reuse the notation $T^\bullet$ to lift a type T to include errors and use v to denote either a value in T (the left injection) or the error value $\bullet$ (the right injection), i.e., $v \in T^\bullet = T + \mathbf{1}$ for some T. The store now maps an uninitialized location to the error value instead of zero:

$$\frac{}{\varepsilon(\ell) = \bullet} \qquad \frac{}{\sigma[\ell \mapsto n](\ell) = \langle n\rangle_{val}} \qquad \frac{\ell \neq \ell' \quad \sigma(\ell) = v}{\sigma[\ell' \mapsto n'](\ell) = v}$$

The evaluation judgment must now account for the possibility of an error value. In contrast to the direct semantics, we must explicitly state rules for both the intended return value and the error value. The evaluation judgment now associates to each arithmetic expression with a given store its symbolic truth value or an error:

$$\Downarrow_A \subseteq \mathsf{AExp} \times \mathsf{Store} \times \mathsf{Num}^\bullet$$

In the non-error case the rules are simply restated to act on the non-error value (the left injection) and errors when dereferencing are simply passed on:

$$\frac{}{\langle n, \sigma\rangle \Downarrow_A \langle n\rangle_{val}} \qquad \frac{\sigma(\ell) = v}{\langle \ell, \sigma\rangle \Downarrow_A v}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \langle n_1\rangle_{val} \quad \langle A_2, \sigma\rangle \Downarrow_A \langle n_2\rangle_{val} \quad \mathscr{N}(n) = \mathscr{N}(n_1) + \mathscr{N}(n_2)}{\langle A_1 + A_2, \sigma\rangle \Downarrow_A \langle n\rangle_{val}}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \langle n_1\rangle_{val} \quad \langle A_2, \sigma\rangle \Downarrow_A \langle n_2\rangle_{val} \quad \mathcal{N}(n) = \mathcal{N}(n_1) - \mathcal{N}(n_2)}{\langle A_1 \text{ - } A_2, \sigma\rangle \Downarrow_A \langle n\rangle_{val}}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \langle n_1\rangle_{val} \quad \langle A_2, \sigma\rangle \Downarrow_A \langle n_2\rangle_{val} \quad \mathcal{N}(n) = \mathcal{N}(n_1) \times \mathcal{N}(n_2)}{\langle A_1 * A_2, \sigma\rangle \Downarrow_A \langle n\rangle_{val}}$$

In addition we must now specify the propagation of errors. To match the direct semantics of Section 2.2.1, we map any error in a sub-expressions to an error of the composite expression:

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 + A_2, \sigma\rangle \Downarrow_A \bullet} \qquad \frac{\langle A_2, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 + A_2, \sigma\rangle \Downarrow_A \bullet}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 \text{ - } A_2, \sigma\rangle \Downarrow_A \bullet} \qquad \frac{\langle A_2, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 \text{ - } A_2, \sigma\rangle \Downarrow_A \bullet}$$

$$\frac{\langle A_1, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 * A_2, \sigma\rangle \Downarrow_A \bullet} \qquad \frac{\langle A_2, \sigma\rangle \Downarrow_A \bullet}{\langle A_1 * A_2, \sigma\rangle \Downarrow_A \bullet}$$

These rules specify a total function from arithmetic expressions and state to numerals or errors. Indeed, the same function as defined for direct semantics for $\mathscr{I}_{\text{err}}$.

Evaluation order As in the denotational case, the inclusion of error values forces us to pick a particular order when evaluating a boolean-and expression. As discussed in Section 2.2.1, there are several choices of evaluation order. We use the same short-circuiting left-to-right evaluation order:

$$\frac{\langle B_1, \sigma\rangle \Downarrow_B \bullet}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \bullet} \qquad \frac{\langle B_1, \sigma\rangle \Downarrow_B \langle \mathsf{false}\rangle_{val}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \langle \mathsf{false}\rangle_{val}}$$

$$\frac{\langle B_1, \sigma\rangle \Downarrow_B \langle \mathsf{true}\rangle_{val} \quad \langle B_2, \sigma\rangle \Downarrow_B \bullet}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \bullet}$$

$$\frac{\langle B_1, \sigma\rangle \Downarrow_B \langle \mathsf{true}\rangle_{val} \quad \langle B_2, \sigma\rangle \Downarrow_B \langle \mathsf{false}\rangle_{val}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \langle \mathsf{false}\rangle_{val}} \qquad \frac{\langle B_1, \sigma\rangle \Downarrow_B \langle \mathsf{true}\rangle_{val} \quad \langle B_2, \sigma\rangle \Downarrow_B \langle \mathsf{true}\rangle_{val}}{\langle B_1 \text{ and } B_2, \sigma\rangle \Downarrow_B \langle \mathsf{true}\rangle_{val}}$$

In words: If the left-hand side evaluates to an error, the composite expression evaluates to an error. If the left-hand side evaluates to false, the composite expression evaluates to false. If the left-hand side evaluates to true, the composite expression evaluates to the evaluation of the right-hand side, i.e., an error, false or true.

In contrast to the direct-style denotational semantics in Section 2.2.1, the structural operational semantics has already fixed the evaluation order with respect to the store because the store is defined as an association of locations with numerals. To restate the evaluation rules for assignment commands to propagate errors, we are forced to evaluate the arithmetic expression at the time of assignment as opposed to doing so at the time of dereferencing the location. We can specify the evaluation order that delays the evaluation until dereferencing the location which is reminiscent of the call-by-name evaluation order. Doing so would require changing the type of our store to associate arithmetic expressions to locations and then update the rules for assignment and dereferencing accordingly.

Control As illustrated for errors and evaluation order in the above, it is cumbersome to specify computational aspects in the big-step semantics. In some cases the specification size can double. In the following sections, we present two other semantics that provide more expressive tools to specify these computational aspects.

2.3.2 A small-step structural operational semantics

Using the same syntax for $\mathscr{I}$, we define another structural operational semantics. In the previous Section, we defined an operational semantics using a *big-step* style where each expression was associated to its *final result*. In this section, we define an operational semantics using a *small-step* style where each expression is associated with the expression resulting from a *single step*.

We reuse the syntactic definition of a store defined in Section 2.3.1 as well as the auxiliary judgment for looking up the value of a location in the store. The small-step operational semantics for $\mathscr{I}$ takes the form of several judgments that associate with each expression the expressions that it can immediately step to. The judgments can be classified in two groups: the *computational rules* that define the actual computational steps, and the *congruence rules* that define how evaluation propagates through sub-expressions. We start with the step judgment for arithmetic expressions that associates to each arithmetic expressions with a given store the arithmetic expression that is the result of a single computational step:

$$\Rightarrow_a \subseteq \mathsf{AExp} \times \mathsf{Store} \times \mathsf{AExp}$$

The computational rules are defined as follows:

$$\frac{\sigma(\ell) = n}{\langle \ell, \sigma \rangle \Rightarrow_a n}$$

$$\frac{\mathscr{N}(n) = \mathscr{N}(n_1) + \mathscr{N}(n_2)}{\langle n_1 + n_2, \sigma \rangle \Rightarrow_a n} \qquad \frac{\mathscr{N}(n) = \mathscr{N}(n_1) - \mathscr{N}(n_2)}{\langle n_1 \text{ - } n_2, \sigma \rangle \Rightarrow_a n} \qquad \frac{\mathscr{N}(n) = \mathscr{N}(n_1) \times \mathscr{N}(n_2)}{\langle n_1 * n_2, \sigma \rangle \Rightarrow_a n}$$

Each rule specifies how an arithmetic expression of a particular shape can be directly reduced, i.e., what the expression steps to. A location with a given store steps to the value associated with that location in the store. An addition expression where the two sub-expressions are numerals steps to the numeral that represents the mathematical meaning of addition as applied to the integers represented by the sub-expressions. We define the step of a subtraction or of a multiplication similar to addition.

The congruence rules are defined as follows:

$$\frac{\langle A_1, \sigma \rangle \Rightarrow_a A_1'}{\langle A_1 + A_2, \sigma \rangle \Rightarrow_a A_1' + A_2} \qquad \frac{\langle A_2, \sigma \rangle \Rightarrow_a A_2'}{\langle n_1 + A_2, \sigma \rangle \Rightarrow_a n_1 + A_2'}$$

$$\frac{\langle A_1, \sigma \rangle \Rightarrow_a A_1'}{\langle A_1 \text{ - } A_2, \sigma \rangle \Rightarrow_a A_1' \text{ - } A_2} \qquad \frac{\langle A_2, \sigma \rangle \Rightarrow_a A_2'}{\langle n_1 \text{ - } A_2, \sigma \rangle \Rightarrow_a n_1 \text{ - } A_2'}$$

$$\frac{\langle A_1, \sigma \rangle \Rightarrow_a A_1'}{\langle A_1 * A_2, \sigma \rangle \Rightarrow_a A_1' * A_2} \qquad \frac{\langle A_2, \sigma \rangle \Rightarrow_a A_2'}{\langle n_1 * A_2, \sigma \rangle \Rightarrow_a n_1 * A_2'}$$

Each congruence rule specifies how an arithmetic expression of a particular shape can be reduced in terms of reductions of its sub-expressions. In particular, these congruence

rules specify a left-to-right evaluation order for the arithmetic operators. Any addition expression, for which the first sub-expression can take a step, can itself take a step by replacing its first sub-expression by the expression that the sub-expressions steps to. Any addition expression, for which the first sub-expression is a numeral and the second sub-expression can take a step, can itself take a step by replacing its second sub-expression by the expression that the sub-expressions steps to. The congruence rules for subtraction and multiplication are defined similar to those for addition.

Given the step judgment for arithmetic expressions, we define the step judgment that associates to each boolean expression with a given store the boolean expression that is the result of a single computational step:

$$\Rightarrow_b \subseteq \mathsf{BExp} \times \mathsf{Store} \times \mathsf{BExp}$$

The computational rules are defined as follows:

$$\frac{\mathcal{N}(n_1) \leq \mathcal{N}(n_2)}{\langle n_1 \texttt{ <= } n_2, \sigma\rangle \Rightarrow_b \mathsf{true}} \qquad \frac{\mathcal{N}(n_1) \not\leq \mathcal{N}(n_2)}{\langle n_1 \texttt{ <= } n_2, \sigma\rangle \Rightarrow_b \mathsf{false}}$$

$$\frac{}{\langle \texttt{not } \mathsf{true}, \sigma\rangle \Rightarrow_b \mathsf{false}} \qquad \frac{}{\langle \texttt{not } \mathsf{false}, \sigma\rangle \Rightarrow_b \mathsf{true}}$$

$$\frac{}{\langle \mathsf{true} \texttt{ and } B_2, \sigma\rangle \Rightarrow_b B_2} \qquad \frac{}{\langle \mathsf{false} \texttt{ and } B_2, \sigma\rangle \Rightarrow_b \mathsf{false}}$$

As detailed for arithmetic expressions above, each rule specifies how a boolean expression of a particular shape can be directly reduced, i.e., what the expression steps to. The congruence rules are defined as follows:

$$\frac{\langle A_1, \sigma\rangle \Rightarrow_b A_1'}{\langle A_1 \texttt{ <= } A_2, \sigma\rangle \Rightarrow_b A_1' \texttt{ <= } A_2} \qquad \frac{\langle A_2, \sigma\rangle \Rightarrow_b A_2'}{\langle n_1 \texttt{ <= } A_2, \sigma\rangle \Rightarrow_b n_1 \texttt{ <= } A_2'}$$

$$\frac{\langle B, \sigma\rangle \Rightarrow_b B'}{\langle \texttt{not } B, \sigma\rangle \Rightarrow_b \texttt{not } B'} \qquad \frac{\langle B_1, \sigma\rangle \Rightarrow_b B_1'}{\langle B_1 \texttt{ and } B_2, \sigma\rangle \Rightarrow_b B_1' \texttt{ and } B_2}$$

Each congruence rule specifies how a boolean expression of a particular shape can be reduced in terms of reductions of its sub-expressions. Again, these congruence rules specify a left-to-right evaluation order for the arithmetic operator. For the boolean-and expression the last rule specifies a *short-circuiting* evaluation order where reduction only occurs in the left-hand sub-expression.

Finally, we define the step judgment that associates to each command with a given store the command and store that is the result of a single step:

$$\Rightarrow_c \subseteq \mathsf{Com} \times \mathsf{Store} \times \mathsf{Com} \times \mathsf{Store}$$

The computational rules are defined as follows:

$$\frac{}{\langle \texttt{skip; } C_2, \sigma\rangle \Rightarrow_c \langle C_2, \sigma\rangle} \qquad \frac{}{\langle \ell \texttt{ := } n, \sigma\rangle \Rightarrow_c \langle \texttt{skip}, \sigma[\ell \mapsto n]\rangle}$$

$$\frac{}{\langle \texttt{if true then } C_1 \texttt{ else } C_2, \sigma\rangle \Rightarrow_c \langle C_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if false then } C_1 \texttt{ else } C_2, \sigma\rangle \Rightarrow_c \langle C_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while } B \texttt{ do } C, \sigma\rangle \Rightarrow_c \langle \texttt{if } B \texttt{ then } (C \texttt{; while } B \texttt{ do } C) \texttt{ else skip}, \sigma\rangle}$$

These rules specify which commands can directly take a step. A sequence command for which the first sub-command is a skip, i.e., it is done, steps to the second sub-command leaving the store unchanged. An assignment command for which the right-hand side is a numeral steps to a skip, i.e., the assignment is done, and updates the store with a binding from the location to the numeral. An if command for which the predicate is the syntactic value true (resp. false) steps to the first (resp. second) sub-command leaving the store unchanged. A while command unconditionally expands to the sequence of its sub-command (i.e., its body) followed by the while command itself. The entire expression is then guarded by the predicate in an if command and leaves the store unchanged. The congruence rules are defined as follows:

$$\frac{\langle C_1, \sigma\rangle \Rightarrow_c \langle C_1', \sigma'\rangle}{\langle C_1 ; C_2, \sigma\rangle \Rightarrow_c \langle C_1' ; C_2, \sigma'\rangle} \qquad \frac{\langle A, \sigma\rangle \Rightarrow_a A'}{\langle \ell := A, \sigma\rangle \Rightarrow_c \langle \ell := A', \sigma\rangle}$$

$$\frac{\langle B, \sigma\rangle \Rightarrow_b B'}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma\rangle \Rightarrow_c \langle \mathsf{if}\ B'\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \sigma\rangle}$$

Each congruence rule specifies how a command of a particular shape can be reduced in terms of the reductions of its sub-expressions. In words, a sequence, for which the first sub-command can take a step, steps to the sequence where the first sub-command is replaced by the command it steps to. An assignment command, for which the right-hand side can take a step, steps to the assignment command where the right-hand side is replaced by the arithmetic expression it steps to. An if command, for which the predicate can take a step, steps to the if command where the predicate is replaced by the boolean expression it steps to.

To define the evaluation of a full program we define the reflexive and transitive evaluation judgment that expresses the iterative evaluation of commands:

$$\frac{}{\langle C, \sigma\rangle \Rightarrow_c^* \langle C, \sigma\rangle} \qquad \frac{\langle C, \sigma\rangle \Rightarrow_c \langle C', \sigma'\rangle \quad \langle C', \sigma'\rangle \Rightarrow_c^* \langle C'', \sigma''\rangle}{\langle C, \sigma\rangle \Rightarrow_c^* \langle C'', \sigma''\rangle}$$

Definition 2 (final configurations)**.** Using s and s' to denote configurations of a step judgment $\Rightarrow$, a configuration is a said to be a *final configuration* if no further step can be taken:

$$s \textit{ final} \iff \text{there exists no } s' \text{ such that } s \Rightarrow s' \text{ holds}$$

In particular, a skip command together with any store is a final configuration of the step judgment on commands, i.e., $\langle \mathsf{skip}, \sigma\rangle$ *final* holds.

The evaluation of a program is then defined as the iteration of the step judgment starting with some initial store, e.g., the empty store, to a final configuration:

Definition 3 (program evaluation)**.** A program C evaluates to the final store σ if and only if, $\langle C, \varepsilon\rangle \Rightarrow_c^* \langle \mathsf{skip}, \sigma\rangle$ holds.

The evaluation of a program is deterministic because each of the step judgments are deterministic.

Computational aspects

This small-step operational semantics expresses the individual computational steps and in so doing it expresses computational aspects of the language that the big-step semantics did not.

Divergence In the small-step semantics, divergence is characterized as a configuration, in our case an expression and a store, that cannot be evaluated to a final configuration:

Definition 4 (diverging configurations)**.** Using s, s' and s'' to denote configurations of a step judgment $\Rightarrow$, a configuration is said to diverge if for all s' such that $s \Rightarrow^* s'$ holds there exist an s'' such that $s' \Rightarrow s''$ holds, i.e., s' *final* does *not* hold.

For example, for any command C and store σ, the configuration $\langle \mathsf{while\ true\ do}\ C, \sigma\rangle$ diverges. First we show that for any configuration s, such that $\langle \mathsf{while\ true\ do}\ C, \sigma\rangle \Rightarrow_c^* s$ holds, then for some C' and σ' either

$$\begin{array}{l} s = \langle \mathsf{while\ true\ do}\ C, \sigma'\rangle, \\ s = \langle \mathsf{if\ true\ then}\ C\ ;\ \mathsf{while\ true\ do}\ C\ \mathsf{else\ skip}, \sigma'\rangle, \quad \text{or} \\ s = \langle C'\ ;\ \mathsf{while\ true\ do}\ C, \sigma'\rangle. \end{array}$$

In the first case, there exists a step which steps to an instance of the second case. In the second case, there exists a step which steps to an instance of the third case. In the third case we show that either $\langle C, \sigma\rangle$ diverges, in which case so does the above and we are done, or $\langle C', \sigma'\rangle \Rightarrow_c^* \langle \mathsf{skip}, \sigma''\rangle$ holds. In the latter case, a non-empty sequence of steps exists to $\langle \mathsf{while\ true\ do}\ C, \sigma''\rangle$ which is itself an instance of the first case.

Errors In the small-step operational semantics we define errors as stuck configurations. A *stuck configuration* is a configuration that cannot take a step and which is not an expected final configuration. The final configurations are those containing value expressions where values are a suitable subset of expressions. For arithmetic expressions, values are the numerals. For boolean expressions, values are the representations of the truth values. For commands, the skip command is the only value form.

Definition 5 (stuck configurations)**.** An arithmetic expression with a store is a stuck configuration if it is a final configuration and the arithmetic expression is not a numeral. A boolean expression with a store is a stuck configuration if it is a final configuration and the boolean expression is not true or false. A command with a store is a stuck configuration if it is a final configuration and the command is not skip.

$$\begin{array}{l} \langle A, \sigma\rangle\ \mathit{stuck} \iff \langle A, \sigma\rangle\ \mathit{final}\ \text{ and }\ A \notin \mathsf{Num} \\ \langle B, \sigma\rangle\ \mathit{stuck} \iff \langle B, \sigma\rangle\ \mathit{final}\ \text{ and }\ B \notin \mathsf{Bool} \\ \langle C, \sigma\rangle\ \mathit{stuck} \iff \langle C, \sigma\rangle\ \mathit{final}\ \text{ and }\ C \neq \mathsf{skip} \end{array}$$

To introduce errors we simply remove one of the rules in the store-lookup judgment. Instead of associating zero to an unallocated location we simply remove the rule altogether. Nothing else is changed. In this formulation errors are represented by a property of the semantics, i.e., by the meta-level language. This in contrast to the other semantics we have seen so far that all represent errors by an explicit semantic object.

In addition to specifying errors, this small-step operational semantics has already made explicit the order of evaluation. Defining the semantics in sufficiently small steps forces the specification of evaluation order. However, the semantics does not easily allow us to specify arbitrary control effects which is the topic of the following section.

2.3.3 A reduction semantics

Using the same syntax for $\mathscr{I}$, we define another operational semantics for the $\mathscr{I}$ language. This operational semantics is given in the form of a *reduction semantics* and exposes additional aspects of $\mathscr{I}$'s computational structure. The notion of a reduction semantics was invented by Felleisen to facilitate syntactic definitions of control and state [85]. Section 2.2.2 illustrated how the continuation semantics made explicit the context of the computation as a *mathematical* construct called the *continuation*. To similar effect, the reduction semantics makes explicit the context of the computation as a *syntactic* construct called the *reduction context* or sometimes the *evaluation context*.

A reduction semantics is typically specified in three parts: 1. the grammar of expressions, 2. the grammar of reduction contexts, and 3. the contraction rules. The expressions of our language are the same as defined in Section 2.1. The reduction context define the places where reduction can occur. These contexts correspond to the places in an expression where the congruence rules defined in Section 2.3.2 allow steps in sub-expressions to take place. The contraction rules define which expressions can be contracted, within some context, to form a reduced expression. These contraction rules correspond to the computational rules defined in Section 2.3.2.

We start by defining the contraction rules for each of the syntactic domains. These contraction rules are transliterated directly from the computational rules of the small-step structural operational semantics in Section 2.3.2. The four computational rules for arithmetic expressions become four contraction rules for arithmetic expressions:

$$\begin{array}{rll}
\langle \ell, \sigma\rangle \rightarrow_a n & \text{where } \sigma(\ell) = n \\
\langle n_1 \text{ + } n_2, \sigma\rangle \rightarrow_a n & \text{where } \mathscr{N}(n) = \mathscr{N}(n_1) + \mathscr{N}(n_2) \\
\langle n_1 \text{ - } n_2, \sigma\rangle \rightarrow_a n & \text{where } \mathscr{N}(n) = \mathscr{N}(n_1) - \mathscr{N}(n_2) \\
\langle n_1 * n_2, \sigma\rangle \rightarrow_a n & \text{where } \mathscr{N}(n) = \mathscr{N}(n_1) \times \mathscr{N}(n_2)
\end{array}$$

These rules define a contraction judgment using a slightly different notation: the judgment holds if and only if the premises hold, where the premises are specified in a conditional clause to the right of the rule. An expression that matches the left-hand side of a rule is called a *redex* (short hand for *red*ucible *ex*pression) and the result of contracting a redex to match the right-hand side of a rule is called the *contractum*.

The six computational rules for boolean expressions become six contraction rules for boolean expressions:

$$\begin{array}{rll}
\langle n_1 \text{ <= } n_2, \sigma\rangle \rightarrow_b \text{true} & \text{where } \mathscr{N}(n_1) \leq \mathscr{N}(n_2) \\
\langle n_1 \text{ <= } n_2, \sigma\rangle \rightarrow_b \text{false} & \text{where } \mathscr{N}(n_1) \not\leq \mathscr{N}(n_2) \\
\langle \text{not true}, \sigma\rangle \rightarrow_b \text{false} \\
\langle \text{not false}, \sigma\rangle \rightarrow_b \text{true} \\
\langle \text{true and } B_2, \sigma\rangle \rightarrow_b B_2 \\
\langle \text{false and } B_2, \sigma\rangle \rightarrow_b \text{false}
\end{array}$$

The five computational rules for commands become five contraction rules for commands:

$$\begin{array}{rl}
\langle \text{skip; } C_2, \sigma\rangle \rightarrow_c & \langle C_2, \sigma\rangle \\
\langle \ell \text{ := } n, \sigma\rangle \rightarrow_c & \langle \text{skip}, \sigma[\ell \mapsto n]\rangle \\
\langle \text{if true then } C_1 \text{ else } C_2, \sigma\rangle \rightarrow_c & \langle C_1, \sigma\rangle \\
\langle \text{if false then } C_1 \text{ else } C_2, \sigma\rangle \rightarrow_c & \langle C_2, \sigma\rangle \\
\langle \text{while } B \text{ do } C, \sigma\rangle \rightarrow_c & \langle \text{if } B \text{ then } (C \text{; while } B \text{ do } C) \text{ else skip}, \sigma\rangle
\end{array}$$

We define the grammar of reduction contexts for each of the syntactic domains in correspondence with the congruence rules of the small-step structural operational semantics in Section 2.3.2. The six congruence rules for arithmetic expressions give rise to six non-terminals in the grammar of arithmetic reduction contexts together with a single terminal denoting the empty context:

$$\mathsf{ACtx} \ni E_a ::= \Box \mid E_a + A_2 \mid n_1 + E_a \mid E_a - A_2 \mid n_1 - E_a \mid E_a * A_2 \mid n_1 * E_a$$

The four congruence rules for boolean expressions give rise to four non-terminals in the grammar of boolean reduction contexts together with a single terminal denoting the empty context:

$$\mathsf{BCtx} \ni E_b ::= \Box \mid E_a <= A_2 \mid n_1 <= E_a \mid \mathsf{not}\ E_b \mid E_b\ \mathsf{and}\ B_2$$

The three congruence rules for commands give rise to three non-terminals in the grammar of command reduction contexts together with a single terminal denoting the empty context:

$$\mathsf{CCtx} \ni E_c ::= \Box \mid E_c\,;\ C_2 \mid \ell := E_a \mid \mathsf{if}\ E_b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$$

With these definitions we define one-step reduction as the closure of the contraction rules over the reduction contexts. There are three reduction rules, one for each type of contraction that can take place: contraction of an arithmetic expression, contraction of a boolean expression, and contraction of a command:

Definition 6 (one-step reduction)**.** The one-step-reduction judgment is inductively defined by the three rules:

$$\frac{\langle A, \sigma\rangle \to_a A'}{\langle E_c[A], \sigma\rangle \mapsto_c \langle E_c[A'], \sigma\rangle} \qquad \frac{\langle B, \sigma\rangle \to_a B'}{\langle E_c[B], \sigma\rangle \mapsto_c \langle E_c[B'], \sigma\rangle} \qquad \frac{\langle C, \sigma\rangle \to_c \langle C', \sigma'\rangle}{\langle E_c[C], \sigma\rangle \mapsto_c \langle E_c[C'], \sigma'\rangle}$$

In words, if a command C decomposes into a reduction context E_c and a reducible arithmetic expression, the command reduces to the recomposition of the reduction context with the contractum of the reducible arithmetic expression leaving the store unchanged. Likewise for a reducible boolean expression and a reducible command, only for commands the store can be changed.

Evaluation is the reflexive transitive closure of one-step reduction:

Definition 7 (reduction-based evaluation)**.**

$$\frac{}{\langle C, \sigma\rangle \mapsto_c^* \langle C, \sigma\rangle} \qquad \frac{\langle C, \sigma\rangle \mapsto_c \langle C', \sigma'\rangle \quad \langle C', \sigma'\rangle \mapsto_c^* \langle C'', \sigma''\rangle}{\langle C, \sigma\rangle \mapsto_c^* \langle C'', \sigma''\rangle}$$

Evaluation of a program is then defined as the iteration of one-step reduction with some initial store, e.g., the empty store, to a final value:

Definition 8 (program evaluation)**.** A program C evaluates to the final store σ, if and only if $\langle C, \varepsilon\rangle \mapsto_c^* \langle \mathsf{skip}, \sigma\rangle$ holds.

The evaluation of a program is deterministic if and only if decomposition, contraction and recomposition are deterministic. Recomposition is deterministic by definition and in our case contraction and decomposition are also deterministic. In general, unique or deterministic decomposition can be hard to prove [221]. In Section 4.1, we give a method to derive the grammar of reduction contexts and the decomposition function mechanically starting from a compositional specification and where unique decomposition follows as a corollary.

Computational aspects

The reduction semantics can almost be seen as simply a stylistic alternative to a small-step structural operational semantics. Indeed, the semantics as stated specifies the same computational aspects using the same mechanisms as in Section 2.3.2. The key difference is the explicit specification of contexts as a semantic object in its own right as opposed to a set of congruence rules. This difference allows use of the context of a computation to specify control operators, e.g., `goto`, `call/cc`, and co-routines.

Control To illustrate the specification of control operators, we consider the extended language $\mathscr{I}_{\text{exc}}$ with the same try-catch exceptions as in Section 2.2.2. The syntax of commands is extended with two operators:

$$\mathsf{Com} \ni C ::= \cdots \mid \mathsf{throw} \mid \mathsf{try}\ C\ \mathsf{catch}\ C$$

The contexts are likewise extended to allow reduction in the body of a try-catch expression:

$$\mathsf{CCtx} \ni E_c ::= \cdots \mid \mathsf{try}\ E_c\ \mathsf{catch}\ C$$

In addition we define a subtype of command contexts to denote the context in which an exception can be thrown. This can happen in a command sequence or in the body of a try command:

$$\mathsf{ExcCtx} \ni F ::= \Box \mid F\,;\ C_2$$

This is a proper subtype such that $\mathsf{ExcCtx} \subset \mathsf{CCtx}$.

The contraction rules for these control operators are simply added to the existing rules defining the contraction judgment for commands:

$$\begin{aligned} \langle \mathsf{try\ skip\ catch}\ C, \sigma\rangle &\to_c \langle \mathsf{skip}, \sigma\rangle \\ \langle \mathsf{try}\ F[\mathsf{throw}]\ \mathsf{catch}\ C, \sigma\rangle &\to_c \langle C, \sigma\rangle \end{aligned}$$

In words: A try-catch where the body is reduced to a skip command reduces to a skip command, i.e., the exception-handling command is disregarded. A try-catch where the body is reduced to a throw command, in the restricted context allowing exceptions, reduces to the exception-handling command. In both cases the store remains unchanged.

An alternative but equivalent specification of exceptions could define throw as a command value in addition to the skip value. The reduction rules would then propagate the throw value through commands. These propagation rules would correspond exactly to the exception contexts defined above. Indeed, this style of specification would be used to add exceptions to the small-step semantics in Section 2.3.2. However, with an explicit grammar of contexts, we can state operators that *capture the context* and allow to later reinstantiate it, e.g., for co-routines or other advanced control operators.

2.4 Representations and implementations

In this section we give two implementations of $\mathscr{I}$: the first implements the direct semantics of Section 2.2.1, and the second implements the reduction semantics of Section 2.3.3. The implementation language is a pure functional subset of core Standard ML [146]. The implementations are used in the following two chapters to illustrate the functional correspondence (Chapter 3) and the syntactic correspondence (Chapter 4). We will largely take

these implementations in a pure functional languages to be our specifications, i.e., they are definitional interpreters in the sense of Reynolds [176]. Indeed, the specification of a programming language by way of a functional implementation has a long-standing tradition going back to McCarthy's original specification of LISP in LISP [142], and Landin's specification of Algol as a translation to lambda expressions for execution on the SECD machine [131, 132]. The discussion of what is the meaning of a specification for a programming language is just as long-standing going back to Scott's critique of the untyped λ-calculus [189]. This discussion has fostered a large amount of research in developing theories for semantic specifications and methods to relate them. It has also fostered research on semantics-based generation of implementations as can be gathered in Schmidt's textbook [186, Chapter 10]. Before defining the implementations of $\mathscr{I}$, we review a few central concepts for relating semantic specifications, representations and implementations.

Adequacy Following LF terminology [109, 110], an *adequate* representation is an encoding of the semantic apparatus that is isomorphic to the informal specification.[4] Thus, any element of the informal semantics must be represented in the encoding by a bijection. Furthermore, this bijection must be structure preserving, i.e., it must preserve contextual properties of the informal semantics such as substitution.

The term 'adequacy' is sometimes used to refer to the *soundness* of a semantics with respect to another semantics or to an implementation. We discuss soundness in the subsequent paragraphs.

Observational equivalence Two expressions are said to be *observationally equivalent* if we can freely replace any occurrence of one by the other in any program without an observable difference. More precisely, e_1 and e_2 are observationally equivalent if and only if, for all contexts C:

$$\mathscr{O}(C[e_1]) = \mathscr{O}(C[e_2])$$

Where C is a one-hole context such that $C[e_1]$ and $C[e_2]$ are well-formed and closed expressions. The observation function, $\mathscr{O}$, is some function from expressions to *observable outcomes*. The definition of this function, and thus what constitutes an observable outcome, is relative to our intended use for the semantics. An observable outcome could be as simple as the final result produced by an implementation, or it could be as complicated as the trace of input-output behavior.

Soundness A semantics is *sound* if the expressions it equates are observationally equivalent. More precisely, a semantics defined by the semantic function, $[\![\cdot]\!]$, is sound if and only if, $[\![e_1]\!] = [\![e_2]\!]$ implies that e_1 and e_2 are observationally equivalent.

Conversely, an implementation is *correct* if all semantically equivalent expressions are observationally equivalent where the observable outcomes are defined in terms of the implementation.

[4] By an informal specification, we mean a specification as defined "on paper" as opposed to a mechanized specification in a fully formalized system, such as LF.

```
eqtype int
val + : int * int -> int
val - : int * int -> int
val * : int * int -> int
```

```
eqtype num
val num  : int -> num
val refl : num -> int
```

```
eqtype loc
val loc : int -> loc
```

Figure 2.1: Abstract signatures for integers, numerals and locations

```
datatype aexp
  = NUM of num
  | REF of loc
  | ADD of aexp * aexp
  | SUB of aexp * aexp
  | MUL of aexp * aexp
```

```
datatype bexp
  = TRUE
  | FALSE
  | LEQ of aexp * aexp
  | NOT of bexp
  | AND of bexp * bexp
```

```
datatype com
  = SKIP
  | SEQ   of com * com
  | SET   of loc * aexp
  | IF    of bexp * com * com
  | WHILE of bexp * com
```

Figure 2.2: A functional representation of the abstract syntax

Full abstraction A semantics is *fully abstract* if it is in complete agreement with observational equivalence. More precisely, a semantics defined by the semantic function, $[\![\cdot]\!]$, is fully abstract if and only if, the semantics is sound, and if e_1 and e_2 are observationally equivalent, then $[\![e_1]\!] = [\![e_2]\!]$.

Let us define the observable outcomes of a command in $\mathscr{I}$ to be either that it diverges or that it converges with some value at a particular locations in the store, as defined by any one of the operational evaluation definitions in Section 2.3. Then both the direct semantics and the continuation semantics are fully abstract and thus all of the semantics of Chapter 2, both denotational and operational, define the same language.[5] However, for more realistic languages that include elements at higher types, defining a fully abstract denotational semantics has proved difficult, as initially pointed out by Plotkin [170].

2.4.1 A functional representation of the abstract syntax

In this section, we give a functional representation of the syntax of $\mathscr{I}$ as defined in Section 2.1. We leave the implementation of numerals and locations opaque and construct them by injections from the integers. Similarly, we leave the implementation of the integers opaque and assume the actual implementation adequate. The Standard ML signatures for integers, numerals and locations are displayed in Figure 2.1. All remaining definitions in Standard ML assume that a valid implementation of each is lexically available.

The functional representation of well-formed expressions is shown in Figure 2.2. We represent each syntactic domain by an inductively defined data type, e.g., AExp becomes the data type aexp. Each production within the syntactic domain is represented by a constructor of that data type, e.g., + which takes two subexpressions in AExp becomes the constructor ADD which takes two values of type aexp. This encoding is an adequate representation of the syntax of $\mathscr{I}$. The algebraic data types of Standard ML define inductive types[6] these are easily shown to be in one-to-one correspondence with the BNF grammar of $\mathscr{I}$. There are no binding constructs in the language and thus no need for substitution, i.e., it is sufficient to witness this bijection to show adequacy.

[5] Reynolds discusses full abstraction for a similar imperative language [178, Chapter 2.8].

[6] This is in contrast to many other languages, such as OCaml and Haskell, where algebraic data types do not define inductive types.

```
(* aexp -> state -> int *)
fun aeval (NUM n)        = (fn s => refl n)
  | aeval (REF l)        = (fn s => s l)
  | aeval (ADD (a1, a2)) = (fn s => aeval a1 s + aeval a2 s)
  | aeval (SUB (a1, a2)) = (fn s => aeval a1 s - aeval a2 s)
  | aeval (MUL (a1, a2)) = (fn s => aeval a1 s * aeval a2 s)

(* bexp -> state -> bool *)
fun beval (TRUE)         = (fn s => true)
  | beval (FALSE)        = (fn s => false)
  | beval (LEQ (a1, a2)) = (fn s => aeval a1 s <= aeval a2 s)
  | beval (NOT b)        = (fn s => not (beval b s))
  | beval (AND (b1, b2)) = (fn s => beval b1 s andalso beval b2 s)

(* com -> state -> state *)
fun ceval (SKIP)           = (fn s => s)
  | ceval (SEQ (c1, c2))   = (fn s => ceval c2 (ceval c1 s))
  | ceval (SET (l, a))     = (fn s => fn l' => if l = l' then aeval a s else s l')
  | ceval (IF (b, c1, c2)) = (fn s => if beval b s then ceval c1 s else ceval c2 s)
  | ceval (WHILE (b, c))   = let fun f s = if beval b s then f (ceval c s) else s
                             in f
                             end

(* com -> state *)
fun eval c = ceval c (fn l => 0)
```

Figure 2.3: A functional implementation of the direct-style semantics

2.4.2 A functional implementation of the direct semantics

In this section, we give a functional implementation of the direct semantics of $\mathscr{I}$ as defined in Section 2.2.1. The implementation uses the signature of integers in Figure 2.1 to represent $\mathbb{Z}$. The built-in Standard ML type `bool` represents $\mathbb{B}$. The two constructors `true` and `false` represent T and F respectively, `not` represents negation, **`andalso`** represents conjunction (which happens to be short-circuiting), and the conditional expression **`if`** _ **`then`** _ **`else`** _ represents the mathematical conditional expression. The total function space for state is represented by the function type: **`type`** `state = loc -> int`. This function space is total in the encoding. Just as in the semantics, we define evaluation of a program with respect to an initial state defined by a total function.

We transliterate the definition in Section 2.2.1 to ML as shown in Figure 2.3. Each syntactic function ($\mathscr{A}[\![\cdot]\!]$, $\mathscr{B}[\![\cdot]\!]$, and $\mathscr{C}[\![\cdot]\!]$) is transliterated to an ML function in curried form (`aeval`, `beval`, and `ceval`). To represent the least fixed point we use local recursion which is, in Standard ML, defined by the least fixed point. The implementation is correct. Expressions of undefined meaning fail to terminate and expressions of well-defined meaning terminate with a value that represents this meaning. The implementation is not easily shown to be adequate. In particular, the function space of ML is partial and as such there are representable expressions in the encoding of state that cannot be mapped back to elements in the total function space of the informal semantics. As it happens, we can show that our encoding never constructs such undefined expressions, e.g., any function in the encoding of state is a total function. Proving adequacy is possible but is complicated by these issues and outside the scope of these dissertation. However, in Chapter 11, we illustrate how to prove adequacy of encodings in a logical framework and how to apply derivational techniques similar to those presented in the remainder of this thesis.

```
datatype aval = VNUM of num
datatype ared
  = PR_REF of loc
  | PR_ADD of aval * aval
  | PR_SUB of aval * aval
  | PR_MUL of aval * aval
```

```
datatype bval = VTRUE
              | VFALSE
datatype bred
  = PR_LEQ of aval * aval
  | PR_NOT of bval
  | PR_AND of bval * bexp
```

```
datatype cval = VSKIP
datatype cred
  = PR_SEQ   of cval * com
  | PR_SET   of loc  * aval
  | PR_IF    of bval * com * com
  | PR_WHILE of bexp * com
```

Figure 2.4: A functional representation of values and redexes

```
datatype store = EMPTY
               | ASSOC of loc * aval * store

(*  loc * store -> aval option  *)
fun lookup (l, EMPTY)             = VNUM 0
  | lookup (l, ASSOC (l', v, st)) = if l = l' then v else lookup (l, st)
```

Figure 2.5: A functional representation of the store

```
(*  ared * store -> aexp  *)
fun contract_ared (PR_REF l,                  st) = aexp_of_val (lookup (l, st))
  | contract_ared (PR_ADD (VNUM n1, VNUM n2), st) = NUM (n1 + n2)
  | contract_ared (PR_SUB (VNUM n1, VNUM n2), st) = NUM (n1 - n2)
  | contract_ared (PR_MUL (VNUM n1, VNUM n2), st) = NUM (n1 * n2)

(*  bred -> bexp  *)
fun contract_bred (PR_LEQ (VNUM n1, VNUM n2)) = if n1 <= n2 then TRUE else FALSE
  | contract_bred (PR_NOT VTRUE)              = FALSE
  | contract_bred (PR_NOT VFALSE)             = TRUE
  | contract_bred (PR_AND (VTRUE, b2))        = b2
  | contract_bred (PR_AND (VFALSE, b2))       = FALSE

(*  cred * store -> com * store  *)
fun contract_cred (PR_SEQ (VSKIP, c2),     st) = (c2, st)
  | contract_cred (PR_SET (l, v),          st) = (SKIP, ASSOC (l, v, st))
  | contract_cred (PR_IF (VTRUE, c1, c2),  st) = (c1, st)
  | contract_cred (PR_IF (VFALSE, c1, c2), st) = (c2, st)
  | contract_cred (PR_WHILE (b, c),        st) = (IF (b, SEQ (c, WHILE (b, c)), SKIP), st)
```

Figure 2.6: A functional implementation of the contraction rules

2.4.3 A functional implementation of the reduction semantics

In this section, we give a functional implementation of the reduction semantics of $\mathscr{I}$ as defined in Section 2.3.3. The implementation uses syntactic representations for subtypes of values and represents each of these subtypes by a distinct data type. By doing so, the type system proves that if a value is constructed it is indeed a value of the respective syntactic domain. We use `fun aexp_of_val (VNUM n) = NUM n` to embed numeric values in numeric expressions. The value of an arithmetic expression is a numeral, the value of a boolean expression is a representation of a truth value, the value of a command is the unit of a command, i.e., the skip command. In addition to the subtypes of values, we define the subtypes of redexes as a data type (Figure 2.4). The store is implemented with a data type and an auxiliary lookup function which is a total function that looks up the value at a location (Figure 2.5). Contraction is implemented by one function for each of the contraction judgments defined over the structure of redexes (Figure 2.6).

```
datatype ccont
  = C0                          (* []       *)
  | C1 of ccont * com           (* E[_; c2] *)

datatype bcont
  = B1 of bcont                 (* E[not _]               *)
  | B2 of bcont * bexp          (* E[_ and b2]            *)
  | B3 of ccont * com * com     (* E[if _ then c1 else c2] *)

datatype acont
  = A1 of acont * aexp          (* E[_ + a2]  *)
  | A2 of acont * aexp          (* E[_ - a2]  *)
  | A3 of acont * aexp          (* E[_ * a2]  *)
  | A4 of aval * acont          (* E[v1 + _]  *)
  | A5 of aval * acont          (* E[v1 - _]  *)
  | A6 of aval * acont          (* E[v1 * _]  *)
  | A7 of bcont * aexp          (* E[_ <= a2] *)
  | A8 of aval * bcont          (* E[v1 <= _] *)
  | A9 of loc * ccont           (* E[l := _]  *)
```

Figure 2.7: A functional representations of contexts

```
(*  ccont * com -> com  *)
fun recompose_ccont (C0,         c ) = c
  | recompose_ccont (C1 (k, c2), c1) = recompose_ccont (k, SEQ (c1, c2))

(*  bcont * bexp -> com  *)
fun recompose_bcont (B1 k,           b ) = recompose_bcont (k, NOT b)
  | recompose_bcont (B2 (k, b2),     b1) = recompose_bcont (k, AND (b1, b2))
  | recompose_bcont (B3 (k, c1, c2), b ) = recompose_ccont (k, IF (b, c1, c2))

(*  acont * aexp -> com  *)
fun recompose_acont (A1 (k, a2), a1) = recompose_acont (k, ADD (a1, a2))
  | recompose_acont (A2 (k, a2), a1) = recompose_acont (k, SUB (a1, a2))
  | recompose_acont (A3 (k, a2), a1) = recompose_acont (k, MUL (a1, a2))
  | recompose_acont (A4 (v1, k), a2) = recompose_acont (k, ADD (aexp_of_val v1, a2))
  | recompose_acont (A5 (v1, k), a2) = recompose_acont (k, SUB (aexp_of_val v1, a2))
  | recompose_acont (A6 (v1, k), a2) = recompose_acont (k, MUL (aexp_of_val v1, a2))
  | recompose_acont (A7 (k, a2), a1) = recompose_bcont (k, LEQ (a1, a2))
  | recompose_acont (A8 (v1, k), a2) = recompose_bcont (k, LEQ (aexp_of_val v1, a2))
  | recompose_acont (A9 (l, k),  a ) = recompose_ccont (k, SET (l, a))
```

Figure 2.8: Recomposition of an expression and its context into a command

The representation of contexts is implemented *inside-out* (Figure 2.7). By inside-out, we mean that the outermost data constructor represents the inner-most expression context. Recomposing, or plugging, an expression into a context to construct an expression thus proceeds by iteratively constructing a larger expression until the context is empty, i.e., it is the left fold over the data type of contexts (Figure 2.8). Also, in our encoding we only ever consider the operation of recomposing into a full command, i.e., we always reconstruct the entire program.

Decomposition is implemented by two functions for each of the syntactic domains. For arithmetic expressions, `decompose_aexp` is defined over the structure of arithmetic expres-

```
datatype val_or_dec
  = VAL  of cval                (* C = skip                  *)
  | ADEC of ared * acont        (* C = E[a] and a is a redex *)
  | BDEC of bred * bcont        (* C = E[b] and b is a redex *)
  | CDEC of cred * ccont        (* C = E[c] and c is a redex *)


(*  acont * aval -> val_or_dec  *)
fun decompose_acont (A1 (k, a2), v1) = decompose_aexp (a2, A4 (v1, k))
  | decompose_acont (A2 (k, a2), v1) = decompose_aexp (a2, A5 (v1, k))
  | decompose_acont (A3 (k, a2), v1) = decompose_aexp (a2, A6 (v1, k))
  | decompose_acont (A4 (v1, k), v2) = ADEC (PR_ADD (v1, v2), k)
  | decompose_acont (A5 (v1, k), v2) = ADEC (PR_SUB (v1, v2), k)
  | decompose_acont (A6 (v1, k), v2) = ADEC (PR_MUL (v1, v2), k)
  | decompose_acont (A7 (k, a2), v1) = decompose_aexp (a2, A8 (v1, k))
  | decompose_acont (A8 (v1, k), v2) = BDEC (PR_LEQ (v1, v2), k)
  | decompose_acont (A9 (l, k),  v ) = CDEC (PR_SET (l, v), k)

(*  aexp * acont -> val_or_dec  *)
and decompose_aexp (NUM n,        k) = decompose_acont (k, VNUM n)
  | decompose_aexp (REF l,        k) = ADEC (PR_REF l, k)
  | decompose_aexp (ADD (a1, a2), k) = decompose_aexp (a1, A1 (k, a2))
  | decompose_aexp (SUB (a1, a2), k) = decompose_aexp (a1, A2 (k, a2))
  | decompose_aexp (MUL (a1, a2), k) = decompose_aexp (a1, A3 (k, a2))
```

Figure 2.9: Decomposition of a non-value arithmetic expression into the next redex and its evaluation context

```
(*  bcont * bval -> val_or_dec  *)
fun decompose_bcont (B1 k,           v ) = BDEC (PR_NOT v, k)
  | decompose_bcont (B2 (k, b2),     v1) = BDEC (PR_AND (v1, b2), k)
  | decompose_bcont (B3 (k, c1, c2), v ) = CDEC (PR_IF (v, c1, c2), k)

(*  bexp * bcont -> val_or_dec  *)
fun decompose_bexp (TRUE,         k) = decompose_bcont (k, VTRUE)
  | decompose_bexp (FALSE,        k) = decompose_bcont (k, VFALSE)
  | decompose_bexp (LEQ (a1, a2), k) = decompose_aexp (a1, A7 (k, a2))
  | decompose_bexp (NOT b,        k) = decompose_bexp (b, B1 k)
  | decompose_bexp (AND (b1, b2), k) = decompose_bexp (b1, B2 (k, b2))
```

Figure 2.10: Decomposition of a non-value boolean expression into the next redex and its evaluation context

sions and `decompose_acont` is defined over the structure of arithmetic expression contexts (Figure 2.9). Given an arithmetic expression and the context of the expression, these functions find the next redex and its reduction context. Likewise, the decomposition of boolean expressions is defined by `decompose_bexp` and `decompose_bcont` (Figure 2.10), and the decomposition of commands is defined by `decompose_com` and `decompose_ccont` (Figure 2.11).

One-step reduction is implemented as the composition of decomposition, contraction and recomposition (Figure 2.12). And reduction-based evaluation as the iteration of one-step reduction (Figure 2.13).

```
(*  ccont * cval -> val_or_dec  *)
fun decompose_ccont (C0,         v ) = VAL v
  | decompose_ccont (C1 (k, c2), v1) = CDEC (PR_SEQ (v1, c2), k)

(*  com * ccont -> val_or_dec  *)
fun decompose_com (SKIP,           k) = decompose_ccont (k, VSKIP)
  | decompose_com (SEQ (c1, c2),   k) = decompose_com (c1, C1 (k, c2))
  | decompose_com (SET (l, a),     k) = decompose_aexp (a, A9 (l, k))
  | decompose_com (IF (b, c1, c2), k) = decompose_bexp (b, B3 (k, c1, c2))
  | decompose_com (WHILE (b, c),   k) = CDEC (PR_WHILE (b, c), k)

(*  com -> val_or_dec  *)
fun decompose t = decompose_com (t, C0)
```

Figure 2.11: Decomposition of a non-value command into the next redex and its evaluation context

```
(*  com * state -> com * state  *)
fun reduce (c, st)
    = (case decompose c
        of VAL v       => (com_of_val v, st)
         | ADEC (r, E) => (recompose_acont (E, contract_ared (r, st)), st)
         | BDEC (r, E) => (recompose_bcont (E, contract_bred r), st)
         | CDEC (r, E) => let val (c', st') = contract_cred (r, st)
                          in (recompose_ccont (E, c'), st')
                          end)
```

Figure 2.12: A functional implementation of one-step reduction

```
(*  val_or_dec * store -> store  *)
fun iterate (VAL VSKIP, st)
    = st
  | iterate (ADEC (r, E), st)
    = iterate (decompose (recompose_acont (E, contract_ared (r, st))), st)
  | iterate (BDEC (r, E), st)
    = iterate (decompose (recompose_bcont (E, contract_bred r)), st)
  | iterate (CDEC (r, E), st)
    = let val (c', st') = contract_cred (r, st)
      in iterate (decompose (recompose_ccont (E, c')), st')
      end

(*  com -> store  *)
fun eval c = iterate (decompose c, EMPTY)
```

Figure 2.13: A functional implementation of reduction-based evaluation

Each component in this implementation can be shown adequate by structural induction. The proofs are straightforward because all of the data representations are first order, i.e., there are no local forms of binding, and the judgments of the reduction semantics are deterministic and total save the final evaluation judgment.

Chapter 3

A functional correspondence: from higher-order interpreter to abstract machine

The functional correspondence makes it possible to inter-derive a higher-order and compositional interpreter and the functional implementation of an abstract machine. This chapter describes each step of the functional correspondence as it applies to the definitional interpreters for $\mathscr{I}$ introduced in Chapter 2. Our starting point is the functional implementation of the direct semantics defined in Section 2.4.2. The first step is the continuation-passing style transformation that given an implementation of the semantics in direct style will produce an implementation of the semantics in continuation-passing style. The second step is defunctionalization that given an implementation of the semantics in continuation-passing style, i.e., an implementation using higher-order functions, will produce a first-order implementation in the style of an abstract machine, i.e., in the form of a state-transition system. Each of these programs are obtained by mechanical and fully-correct program transformations and represent different encodings of the *same* computation.

3.1 The CPS transformation: from direct-style to continuation-style

The list of co-discoverers of continuations, CPS transformations, and control operators is long and the list of their uses is even longer. We refer the interested reader to the historical notes by Reynolds [177] and Landin [133] and to Danvy's doctorate [60].

Danvy summarizes the transformation into continuation-passing style in three steps [55]: 1. name all intermediate results, 2. sequentialize these results, and 3. pass an extra argument to each call that will map these named and sequentialized results to the final result.

Consider the evaluation function for arithmetic expressions, here using ML's notation for curried arguments:

```
(*  aexp -> state -> int  *)
fun aeval (NUM n)        s = refl n
  | aeval (REF l)        s = s l
  | aeval (ADD (a1, a2)) s = aeval a1 s + aeval a2 s
  | aeval (SUB (a1, a2)) s = aeval a1 s - aeval a2 s
  | aeval (MUL (a1, a2)) s = aeval a1 s * aeval a2 s
```

Let us CPS transform this function in three steps:

1. We name intermediate results with let-expressions:

```
(*  aexp -> state -> int  *)
fun aeval_1 (NUM n) s
    = let val v = refl n in v end
  | aeval_1 (REF l) s
    = let val v = s l in v end
  | aeval_1 (ADD (a1, a2)) s
    = let val v3 = (let val v1 = aeval_1 a1 s in v1 end) +
                   (let val v2 = aeval_1 a2 s in v2 end)
      in v3 end
  | aeval_1 (SUB (a1, a2)) s
    = let val v3 = (let val v1 = aeval_1 a1 s in v1 end) -
                   (let val v2 = aeval_1 a2 s in v2 end)
      in v3 end
  | aeval_1 (MUL (a1, a2)) s
    = let val v3 = (let val v1 = aeval_1 a1 s in v1 end) *
                   (let val v2 = aeval_1 a2 s in v2 end)
      in v3 end
```

   The numeral case has one intermediate result named v which is the integer reflection of the numeral. The dereferencing case has one intermediate result named v which is the result of looking up the location in the store. This result is the final result. The arithmetic operators have three intermediate results. The first intermediate result named v1 is the result of the recursive call on the first subexpression. The second intermediate result named v2 is the result of the recursive on the second subexpression. The third intermediate result named v3 is the result of the arithmetic operator applied to the previous two results. The third intermediate result is the final result.

2. We sequentialize these let-expressions:

```
(*  aexp -> state -> int  *)
fun aeval_2 (NUM n) s
    = let val v = refl n in v end
  | aeval_2 (REF l) s
    = let val v = s l in v end
  | aeval_2 (ADD (a1, a2)) s
    = let val v1 = aeval_2 a1 s
      in let val v2 = aeval_2 a2 s
         in let val v3 = v1 + v2
            in v3 end end end
  | aeval_2 (SUB (a1, a2)) s
    = let val v1 = aeval_2 a1 s
      in let val v2 = aeval_2 a2 s
         in let val v3 = v1 - v2
            in v3 end end end
  | aeval_2 (MUL (a1, a2)) s
    = let val v1 = aeval_2 a1 s
      in let val v2 = aeval_2 a2 s
```

```
        in let val v3 = v1 * v2
           in v3 end end end
```

For this program, there are two possible choices when sequentializing: either `v1` comes before `v2` or `v2` comes before `v1`. There is no choice for `v3` since the computation producing the named result `v3` depends on the named results `v1` and `v2`. This step results in a computation in 'monadic normal form' [93, 111, 149].

3. Finally, we introduce continuations as an extra functional argument to each function. These continuation functions map the named result to the final value:

```
type 'a state_cps = loc -> (int -> 'a) -> 'a

fun refl_cps v k = k (refl v)
fun add_cps v1 v2 k = k (v1 + v2)
fun sub_cps v1 v2 k = k (v1 - v2)
fun mul_cps v1 v2 k = k (v1 * v2)

(*  aexp -> 'a state_cps -> (int -> 'a) -> 'a  *)
fun aeval_3 (NUM n) s k
    = refl_cps n (fn v => k v)
  | aeval_3 (REF l) s k
    = s l (fn v => k v)
  | aeval_3 (ADD (a1, a2)) s k
    = aeval_3 a1 s (fn v1 =>
        aeval_3 a2 s (fn v2 =>
          add_cps v1 v2 (fn v3 =>
            k v3)))
  | aeval_3 (SUB (a1, a2)) s k
    = aeval_3 a1 s (fn v1 =>
        aeval_3 a2 s (fn v2 =>
          sub_cps v1 v2 (fn v3 =>
            k v3)))
  | aeval_3 (MUL (a1, a2)) s k
    = aeval_3 a1 s (fn v1 =>
        aeval_3 a2 s (fn v2 =>
          mul_cps v1 v2 (fn v3 =>
            k v3)))
```

The result of the CPS transformation is an evaluation function in continuation-passing style: all functions take a continuation and all calls are in tail position, i.e., each call is locally the last thing to do.

When CPS transforming a function, we will often perform two additional transformations in passing:

1. The continuation `k` arising from the CPS transformation is a value and thus cannot diverge, i.e., it is a trivial expression in the terminology of Reynolds [176]. Therefore, the expression `fn v => k v` is functionally equivalent to its η-reduction `k`:

```
(*  aexp -> state_cps -> (int -> 'a) -> 'a  *)
fun aeval_4 (NUM n) s k
    = refl_cps n k
  | aeval_4 (REF l) s k
    = s l k
  | aeval_4 (ADD (a1, a2)) s k
    = aeval_4 a1 s (fn v1 =>
        aeval_4 a2 s (fn v2 =>
          add_cps v1 v2 k))
```

```
    | aeval_4 (SUB (a1, a2)) s k
      = aeval_4 a1 s (fn v1 =>
          aeval_4 a2 s (fn v2 =>
            sub_cps v1 v2 k))
    | aeval_4 (MUL (a1, a2)) s k
      = aeval_4 a1 s (fn v1 =>
          aeval_4 a2 s (fn v2 =>
            mul_cps v1 v2 k))
```

2. Total and effect-free functions can be left in *direct-style* without affecting the correctness of the transformation. Typically, we will leave 'primitive' operations, e.g., addition, subtraction and multiplication and in this case our encoding of state, in direct-style:

```
(* aexp -> state -> (int -> 'a) -> 'a *)
fun aeval_5 (NUM n) s k
    = k (refl n)
  | aeval_5 (REF l) s k
    = k (s l)
  | aeval_5 (ADD (a1, a2)) s k
    = aeval_5 a1 s (fn v1 =>
        aeval_5 a2 s (fn v2 =>
          k (v1 + v2)))
  | aeval_5 (SUB (a1, a2)) s k
    = aeval_5 a1 s (fn v1 =>
        aeval_5 a2 s (fn v2 =>
          k (v1 - v2)))
  | aeval_5 (MUL (a1, a2)) s k
    = aeval_5 a1 s (fn v1 =>
        aeval_5 a2 s (fn v2 =>
          k (v1 * v2)))
```

This CPS transformation is but one among many. Each choice of sequentialization gives rise to a CPS transformation. Here we choose to eagerly evaluate subexpressions from left to right. This happens to be the evaluation order of Standard ML, and is thus a fully correct program transformation on any program in Standard ML. Also, we have used the same eager and left-to-right evaluation order consistently throughout Chapter 2. The λ-calculus has two canonical evaluation orders: call by value and call by name. Given a direct semantics of the λ-calculus these evaluation orders can be realized respectively the call-by-value and the call-by-name CPS transformations [92, 169].

Having fixed the evaluation order by means of a CPS transformation, we change the encoding of state to be in state-passing style, in accordance with Section 2.2.1. The passing of state inherits the order from the CPS transformation and is fully η-reduced, i.e., we simply swap the order of the formal parameters, let the continuation take both the integer result and the threaded state, and finally η-reduce the state throughout:

```
(* aexp -> (int -> state -> 'a) -> state -> 'a *)
fun aeval_6 (NUM n) k
    = k (refl n)
  | aeval_6 (REF l) k
    = (fn s => k (s l) s)
  | aeval_6 (ADD (a1, a2)) k
    = aeval_6 a1 (fn v1 =>
        aeval_6 a2 (fn v2 =>
          k (v1 + v2)))
  | aeval_6 (SUB (a1, a2)) k
```

```
      = aeval_6 a1 (fn v1 =>
          aeval_6 a2 (fn v2 =>
            k (v1 - v2)))
    | aeval_6 (MUL (a1, a2)) k
      = aeval_6 a1 (fn v1 =>
          aeval_6 a2 (fn v2 =>
            k (v1 * v2)))
```

Completing this transformation on the evaluation functions for boolean expressions and commands yields the definitional interpreter in CPS displayed in Figure 3.1. In summary, the evaluation function for arithmetic expressions in state-passing style takes an arithmetic expression and a state (in curried form) and produces an integer and the threaded state. Thus the continuation is a function from results, here integers and threaded states, to some answer type, here denoted by the type variable `'a`. Likewise, the continuation for boolean expressions is a function from boolean values and threaded states to answers. The continuation for commands is a function from states, which is the only result produced by the direct-style denotation of commands, to answers.

This definitional interpreter in CPS is a correct implementation of the direct semantics in Section 2.2.1 because we have mechanically derived the implementation directly from the implementation of the direct style semantics. Furthermore, this definitional interpreter in CPS is actually an implementation of the continuation semantics and can be transliterated directly from definitions in Section 2.2.2.

```
type 'a acont = int -> state -> 'a
type 'a bcont = bool -> state -> 'a
type 'a ccont = state -> 'a

(*  aexp -> 'a acont -> state -> 'a  *)
fun aeval (NUM n)        k = k (refl n)
  | aeval (REF l)        k = (fn s => k (s l) s)
  | aeval (ADD (a1, a2)) k = aeval a1 (fn v1 => aeval a2 (fn v2 => k (v1 + v2)))
  | aeval (SUB (a1, a2)) k = aeval a1 (fn v1 => aeval a2 (fn v2 => k (v1 - v2)))
  | aeval (MUL (a1, a2)) k = aeval a1 (fn v1 => aeval a2 (fn v2 => k (v1 * v2)))

(*  bexp -> 'a bcont -> state -> 'a  *)
fun beval (TRUE)         k = k true
  | beval (FALSE)        k = k false
  | beval (LEQ (a1, a2)) k = aeval a1 (fn v1 => aeval a2 (fn v2 => k (v1 <= v2)))
  | beval (NOT b)        k = beval b  (fn v => k (not v))
  | beval (AND (b1, b2)) k = beval b1 (fn v => if v then beval b2 k else k false)

(*  com -> 'a ccont -> state -> 'a  *)
fun ceval (SKIP)           k = k
  | ceval (SEQ (c1, c2))   k = ceval c1 (ceval c2 k)
  | ceval (SET (l, a))     k = aeval a (fn v => fn s => k (fn l' => if l = l' then v else s l'))
  | ceval (IF (b, c1, c2)) k = beval b (fn v => if v then ceval c1 k else ceval c2 k)
  | ceval (WHILE (b, c))   k = let fun f k = beval b (fn v => if v then ceval c (f k) else k)
                               in f k end

(*  com -> state  *)
fun eval c = ceval c (fn s => s) (fn l => 0)
```

Figure 3.1: A definitional interpreter for $\mathscr{I}$ in continuation-passing style

3.2 Defunctionalization: from higher order to first order

The defunctionalization transformation was introduced by Reynolds [176] as a method for replacing the use of higher-order functions by a first-order representation. Defunctionalization is a whole-program transformation, and works by replacing each instantiation of a higher-order function by the allocation of a first-order representation of its environment and each application of a higher-order function by the invocation of an 'apply' function to the first-order representation.

We start by defunctionalizing the representation of state. The definitional interpreter of Figure 3.1 constructs functions with the `state` type at two distinct sites. We therefore partition the function space into two summands:

0. The initial state, the constant function mapping any location to the numeric representation of zero, which has no free variables.
1. The functional extension of state, which has three free variables: the location `l`, the value `v`, and the previous state `s'`.

Defunctionalizing the function type of state therefore gives rise to (1) the following data type:

```
datatype state
  = S0
  | S1 of loc * int * state
```

and (2) the following 'apply' function dispatching over the state data type (here renaming `s'` by `s`):

```
fun apply (S0,           l ) = 0
  | apply (S1 (l, v, s), l') = if l = l' then v else apply (s, l')
```

Renaming `S0` by `EMPTY`, `S1` by `ASSOC` and `apply` by `lookup`, this first-order representation of the state coincides with our representation of the store and its lookup judgment for the operational semantics (Figure 2.5).

Defunctionalizing the state of the direct-semantics implementation into a store yields an implementation of the big-step operational semantics. Here all structures are first-order and adequately encode the evaluation judgments of Section 2.3.1.

Returning to the derived implementation of the continuation semantics, we can defunctionalize the functions spaces of continuations. Thus for each type of continuation we introduce a data type and an 'apply' function dispatching over the data type. We reuse the same name for both the data type and the function (here `acont`, `bcont`, and `ccont`). As a final step, we η-expand and uncurry the state throughout. The result is the first-order definitional interpreter in Figure 3.2 and Figure 3.3 which has the form of an *abstract machine*. The machine has six transition functions: `aeval`, `acont`, `beval`, `bcont`, `ceval`, and `ccont`, and three data types implementing the machine stack: `acont`, `bcont`, and `ccont`. For example, the evaluation of an addition proceeds as follows:

1. In the `ADD` case of `aeval` the machine pushes a 'left-addition' stack frame, `A1`, containing `a2` onto the stack and continues evaluation of `a1`.
2. In the `A1` case of `acont` the machine has evaluated to a value, `v1`, and has a 'left-addition' frame containing `a2` on the top of stack. It replaces the top of stack by a 'right-addition' frame, `A4`, containing `v1` and continues evaluation of `a2`.

```
datatype ccont
  = C0                          (* []              *)
  | C1 of ccont * com           (* E[_; c]         *)
  | C2 of ccont * bexp * com    (* E[_; while b c] *)

datatype bcont
  = B1 of bcont                 (* E[not _]                              *)
  | B2 of bcont * bexp          (* E[_ and b2]                           *)
  | B3 of ccont * com * com     (* E[if _ then c1 else c2]               *)
  | B4 of ccont * bexp * com    (* E[if _ then (c; while b c) else skip] *)

datatype acont
  = A1 of acont * aexp          (* E[_ + a2]  *)
  | A2 of acont * aexp          (* E[_ - a2]  *)
  | A3 of acont * aexp          (* E[_ * a2]  *)
  | A4 of int * acont           (* E[v1 + _]  *)
  | A5 of int * acont           (* E[v1 - _]  *)
  | A6 of int * acont           (* E[v1 * _]  *)
  | A7 of bcont * aexp          (* E[_ <= a2] *)
  | A8 of int * bcont           (* E[v1 <= _] *)
  | A9 of loc * ccont           (* E[l := _]  *)
```

Figure 3.2: A first-order definitional interpreter for $\mathscr{I}$: contexts

3. In the `A4` case of `acont` the machine has evaluation to a value, `v2`, and has a 'right-addition' frame containing `v1` on the top of stack. It computes the sum of the values `v1` and `v2`, and continues evaluation with this value and the rest of the stack.

This abstract machine is a correct implementation of the continuation semantics and of the direct semantics because it has been derived using fully-correct program transformations.

3.3 Summary and conclusion

Starting from the implementation of a direct semantics, we have mechanically derived an abstract machine for the same language. The structural coincidence between the implicit context in a direct semantics, the explicit continuation in a continuation semantics, and the first-order stack in an abstract machine plays the key rôle in this connection. Indeed, as remarked by Stoy [201, Page 253], an uncurried reading of the state-passing continuation semantics (Figure 3.1) resembles the state of some interpreter defined as a state transition system, e.g., Landin's sharing machine [131]. Reynolds noted that after defunctionalization this interpreter has a "*machine-like character*" and implements a "*state-transition machine*" [176, Section 7 and 8], or in our terminology, an *abstract machine*.

3.4 The functional correspondence in perspective

The development of the functional correspondence followed an investigation of defunctionalization by Danvy and Nielsen [70], which prompted the question as to whether Landin's SECD machine was in defunctionalized form [58]. Indeed, the SECD machine

```
(*  aexp * acont * state -> state  *)
fun aeval (NUM n,        k, s) = acont (k, refl n, s)
  | aeval (REF l,        k, s) = acont (k, apply (s, l), s)
  | aeval (ADD (a1, a2), k, s) = aeval (a1, A1 (k, a2), s)
  | aeval (SUB (a1, a2), k, s) = aeval (a1, A2 (k, a2), s)
  | aeval (MUL (a1, a2), k, s) = aeval (a1, A3 (k, a2), s)

(*  acont * int * state -> state  *)
and acont (A1 (k, a2), v1, s) = aeval (a2, A4 (v1, k), s)
  | acont (A2 (k, a2), v1, s) = aeval (a2, A5 (v1, k), s)
  | acont (A3 (k, a2), v1, s) = aeval (a2, A6 (v1, k), s)
  | acont (A4 (v1, k), v2, s) = acont (k, v1 + v2, s)
  | acont (A5 (v1, k), v2, s) = acont (k, v1 - v2, s)
  | acont (A6 (v1, k), v2, s) = acont (k, v1 * v2, s)
  | acont (A7 (k, a2), v1, s) = aeval (a2, A8 (v1, k), s)
  | acont (A8 (v1, k), v2, s) = bcont (k, v1 <= v2, s)
  | acont (A9 (l, k),  v,  s) = ccont (k, S1 (l, v, s))

(*  bexp * bcont * state -> state  *)
and beval (TRUE,         k, s) = bcont (k, true, s)
  | beval (FALSE,        k, s) = bcont (k, false, s)
  | beval (LEQ (a1, a2), k, s) = aeval (a1, A7 (k, a2), s)
  | beval (NOT b,        k, s) = beval (b,  B1 k, s)
  | beval (AND (b1, b2), k, s) = beval (b1, B2 (k, b2), s)

(*  bcont * bool * state -> state  *)
and bcont (B1 k,           v, s) = bcont (k, not v, s)
  | bcont (B2 (k, b2),     v, s) = if v then beval (b2, k, s) else bcont (k, false, s)
  | bcont (B3 (k, c1, c2), v, s) = if v then ceval (c1, k, s) else ceval (c2, k, s)
  | bcont (B4 (k, b, c),   v, s) = if v then ceval (c, C2 (k, b, c), s) else ccont (k, s)

(*  com * ccont * state -> state  *)
and ceval (SKIP,           k, s) = ccont (k, s)
  | ceval (SEQ (c1, c2),   k, s) = ceval (c1, C1 (k, c2), s)
  | ceval (SET (l, a),     k, s) = aeval (a, A9 (l, k), s)
  | ceval (IF (b, c1, c2), k, s) = beval (b, B3 (k, c1, c2), s)
  | ceval (WHILE (b, c),   k, s) = beval (b, B4 (k, b, c), s)

(*  ccont * state -> state  *)
and ccont (C0,           s) = s
  | ccont (C1 (k, c2),   s) = ceval (c2, k, s)
  | ccont (C2 (k, b, c), s) = beval (b, B4 (k, b, c), s)

(*  com -> state  *)
fun eval c = ceval (c, C0, S0)
```

Figure 3.3: A first-order definitional interpreter for $\mathscr{I}$: evaluation

could be put in defunctionalized form and be understood by mapping it back to direct-style by the inverse transformations of defunctionalization [69] and the CPS transformation [56]. This 'backward' correspondence from abstract machine to direct-style using the left inverse of defunctionalization and CPS transformation can be found in 'forward' form in Reynolds's seminal paper on definitional interpreters [176]. Reynolds investigated the properties of functional interpreters by means of program transformations and, in retrospect, spelled out a generic correspondence between higher-order algorithms and first-

order transition systems by means of CPS transformation and defunctionalization. This functional correspondence has been shown to be widely applicable and to provide instrumental guidelines to construct new semantic artifacts and to establish the equivalence of existing ones [62]:

- Ager, Biernacki, Danvy, and Midtgaard [5] apply the correspondence to standard interpreters and abstract machines. Starting from a compositional call-by-name interpreter they derive Krivine's machine [127, 128], which in retrospect was originally derived by Schmidt [187]. They inter-derive a compositional call-by-value interpreter with Felleisen and Friedman's CEK machine [87], as originally derived by Reynolds [176]. In addition they derive a higher-order interpreter for each of Hannan and Miller's CLS machine [106], Landin's SECD machine [131], and Cousineau et al.'s CAM machine [48].
- Danvy [57] factored the compositional interpreters into the same evaluation-order-dependent one, and used the call-by-value CPS transformation to derive the CEK machine and the call-by-name CPS transformation to derive the Krivine machine. This factoring through the CPS transformations followed by defunctionalization elucidated the structural coincidence between the implicit contexts in a direct-style interpreter, the continuations in continuation-passing interpreter, and the stack in an abstract machine.
- Ager, Danvy, and Midtgaard [7] extend the correspondence to store-based call-by-need evaluation. Starting from a call-by-need interpreter that threads a store and uses updatable thunks for sharing, they derive a lazy abstract machine, in particular, they derive Crégut's lazy variant of Krivine's machine [51].
- Ager, Danvy, and Midtgaard [8] extend the correspondence to account for impure languages. Starting from a monadic interpreter they derive an abstract machines accounting for computational effects. This correspondence accounts for effects ranging from simple error handling to tail-recursive stack inspection and it provides a framework for combining monadic effects and subsequently deriving an execution environment in the form of an abstract machine.
- Biernacka, Biernacki, and Danvy [30] extend the correspondence to account for delimited control in the CPS hierarchy. For a given level of the CPS hierarchy, n, they construct a continuation-passing interpreter with $n+1$ layers of continuations and derive an abstract machine with $n+1$ control stacks accounting for the family of control operators shift_n and reset_n.
- Danvy and Millikin [68] give a rational deconstruction of Landin's J operator for the SECD machine. Their deconstruction shows how the J operator is specified in the CPS hierarchy [64], and how it relates to other uses of continuations and control operators. Based on this deconstruction, they argue for the inclusion of Landin to the list of co-discoverers of continuations.
- Biernacki and Danvy [32] extend the correspondence to connect logical inference engines for Prolog with cut. Starting from a continuation-passing interpreter they derive an abstract machine and give a direct-style interpreter using delimited control operators. These derivations pointed our differences in existing accounts of the same logical system [37], namely that some where properly tail recursive while others where not.

- Danvy and Johannsen [65, 119] extend the correspondence to inter-derive a natural semantics and an abstract machine with an explicit-substitution variant of Abadi and Cardelli's object calculus [1].
- Danvy [63] inter-derived several of the semantic artifacts for the Scheme programming language. In particular, he shows consistency between various specifications of `call/cc` by inter-deriving two abstract machines, a natural semantic, and a denotational semantics, based on Clinger's abstract machine for Core Scheme [47].
- Pirog and Biernacki [167] mechanized the derivation of the STG machine [160] in the Coq proof assistant.
- Anton and Thiemann [9, 10] extend the correspondence to derive a type system and an implementation for coroutines.
- Sergey and Clarke [192–194] extend the correspondence to inter-derive type checking algorithms. In particular, they connect reduction-based type checking [129] with traditional recursive-descent type checking [166].
- Independently, Puech [175] also uses the correspondence to inter-derive recursive and iterative algorithms for bidirectional type checking.
- Kerneis [124] uses CPS and defunctionalization to compile lightweight threads into event-based code in an imperative setting.
- Prior to the identification of this correspondence, Graunke et al. [102] used the CPS transformation and defunctionalization to compile higher-order programs into first-order programs for web programming.

In this dissertation, the techniques from the functional correspondence are employed in several places:

- Chapter 5 develops a prequel to a reduction semantics based on the functional correspondence. Starting from a big-step specification of a reduction strategy in the form of a compositional search function, we derive both the grammar of reduction contexts and the decomposition function of a reduction semantics. From this derived formulation the syntactic correspondence can be applied to obtain an abstract machine (which we later detail in Section 4.1).
- Chapter 6 extends the functional correspondence to inter-derive small-step and big-step normalization functions for boolean-propositional formulas. We inter-derive conjunctive normalization functions in the form of an abstract machine with delimited stacks, a compositional interpreter with delimited continuations, and a direct-style interpreter using the control operators shift and reset.
- Chapter 7 extends the functional correspondence to connect store-less accounts of call-by-need evaluation. We use the correspondence to inter-derive a store-less abstract machine with a store-less natural semantics and a higher-order evaluation function in the style of Cartwright and Felleisen's extensible denotational semantics [41]. This higher-order evaluation function makes use of delimited continuations and can be encoded using state or delimited control.

- Chapter 8 gives a synthetic account of call-by-need evaluation. It provides bisimulations for a series of reduction semantics for call-by-need evaluation spanning from the storeless call-by-need calculus of Ariola-al et al. [15] to a state-based reduction semantics using a store and updateable thunks. Each of these reduction semantics is then inter-derived with their corresponding abstract machine and natural semantics. Together, the bisimulations and inter-derivations give a constructive connection of Crégut's original lazy Krivine machine [51] with Ariola et al.'s call-by-need λ-calculus, and with Launchbury's natural semantics for call by need. In addition, the synthetic account systematically maps out the "spaces between" the existing specifications.
- Chapter 9 investigates whether Turner's reduction machine can be put into defunctionalization form and gives an example of a big-step graph evaluator as inter-derived by the functional correspondence.
- Chapter 11 provides a logical counterpart to the functional correspondence. Specifications are given as encodings in a logical framework. We then interpret these specifications as logic programs governed by proof search which gives them an operational reading. We identify Horn-clause logic programs a an adequate specification of a natural semantics, ordered-logic programs as an adequate specification of abstract machines, and provide a logical transformation from the first to the second.

Chapter 4

A syntactic correspondence: from reduction semantics to abstract machine

The syntactic correspondence makes it possible to inter-derive the functional implementation of a reduction semantics and the functional implementation of an abstract machine. This chapter describes each step of the syntactic correspondence as it applies to the reduction semantics for $\mathscr{I}$ introduced in Chapter 2. Our starting point is the functional implementation of the reduction semantics defined in Section 2.4.3. We show how to derive the grammar of reduction contexts and the decomposition function of the reduction semantics from a specification of its reduction strategy (Section 4.1). Next, we refocus the reduction-based evaluation of the reduction semantics into a reduction-free evaluation that implements an abstract machine (Section 4.2). This abstract machine is implemented in *small-steps* which we transform to an implementation using one *big-step* with lightweight fusion by fixed-point promotion (Section 4.3). We then hereditary compress corridor transitions yielding a more efficient abstract machine (Section 4.4). The final result of these transformations is the same implementation of an abstract machine as derived in Chapter 3.

4.1 A prequel to reduction semantics: from search to decompose

This section presents our prequel to reduction semantics. This prequel defines a method to derive a reduction semantics from a big-step specification of a reduction strategy. Specifically, this specification is given as a total and compositional function that searches for the next redex in an expression, i.e., the search function is a big-step specification. The method then derives an implementation of the decompose function that incrementally performs this search while simultaneously constructing the current context of the search, i.e., the derived decompose function is a small-step specification. In effect, we use the functional correspondence to obtain a small-step implementation of the big-step speci-

fication. The defunctionalized continuations coincide with the reduction contexts of a reduction semantics.

This prequel has been developed jointly with Olivier Danvy and Jacob Johannsen [76]. A detailed account of the prequel is given in Chapter 5. An extended version of our published work appears in Chapter 6. In addition, the prequel is applied explicitly in deriving a graph reduction machine (Chapter 9) and can be applied to our other work on syntactic theories for graph reduction (Chapter 10) and also to our work on call-by-need evaluation (Chapter 7 and Chapter 8).

4.1.1 Reduction strategy

The specification of our reduction strategy is given by three search functions defined compositionally over the structure of the three syntactic domains. The search functions map a reducible expression to the next redex according to the strategy and a non-reducible expression to its value representation.

For arithmetic expressions, the data type `aval_or_red` represents the sum type of values and redexes returned by the search function `search_aexp`:

```
datatype aval_or_red
  = AVAL  of aval
  | AREDA of ared

(*  aexp -> aval_or_red  *)
fun search_aexp (NUM n)
    = AVAL (VNUM n)
  | search_aexp (REF l)
    = AREDA (PR_REF l)
  | search_aexp (ADD (a1, a2))
    = (case search_aexp a1
        of AVAL v1
           => (case search_aexp a2
                of AVAL v2
                   => AREDA (PR_ADD (v1, v2))
                 | AREDA r
                   => AREDA r)
         | AREDA r
           => AREDA r)
  | search_aexp (SUB (a1, a2))
    = (case search_aexp a1
        of AVAL v1
           => (case search_aexp a2
                of AVAL v2
                   => AREDA (PR_SUB (v1, v2))
                 | AREDA r
                   => AREDA r)
         | AREDA r
           => AREDA r)
  | search_aexp (MUL (a1, a2))
    = (case search_aexp a1
        of AVAL v1
           => (case search_aexp a2
                of AVAL v2
                   => AREDA (PR_MUL (v1, v2))
                 | AREDA r
                   => AREDA r)
         | AREDA r
           => AREDA r)
```

This function searches for the next redex in an arithmetic expression from left to right. Any redex found in a sub-expression is mapped to a redex of the composite expression:

For boolean expressions, the data type `bval_or_red` represents the sum type of values and redexes returned by the search function `search_bexp`:

```
datatype bval_or_red
  = BVAL  of bval
  | BREDA of ared
  | BREDB of bred


(*  bexp -> bval_or_red  *)
fun search_bexp TRUE
    = BVAL VTRUE
  | search_bexp FALSE
    = BVAL VFALSE
  | search_bexp (LEQ (a1, a2))
    = (case search_aexp a1
        of AVAL v1
           => (case search_aexp a2
                of AVAL v2
                   => BREDB (PR_LEQ (v1, v2))
                 | AREDA r
                   => BREDA r)
         | AREDA r
           => BREDA r)
  | search_bexp (NOT b)
    = (case search_bexp b
        of BVAL v
           => BREDB (PR_NOT v)
         | BREDA r
           => BREDA r
         | BREDB r
           => BREDB r)
  | search_bexp (AND (b1, b2))
    = (case search_bexp b1
        of BVAL v1
           => BREDB (PR_AND (v1, b2))
         | BREDA r
           => BREDA r
         | BREDB r
           => BREDB r)
```

This function searches for the next redex in a boolean expression from left to right and in the case of the boolean-and expression only in the left sub-expression. Any redex found in a sub-expression is mapped to a redex of the composite expression. A redex can be either a reducible arithmetic expression or a reducible boolean expression.

For commands, the data type `cval_or_red` represents the sum type of values and redexes returned by the search function `search_cexp`:

```
datatype cval_or_red
  = CVAL  of cval
  | CREDA of ared
  | CREDB of bred
  | CREDC of cred


(*  com -> cval_or_red  *)
fun search_com SKIP
    = CVAL VSKIP
```

```
  | search_com (SEQ (c1, c2))
    = (case search_com c1
        of CVAL v1
           => CREDC (PR_SEQ (v1, c2))
         | CREDA r
           => CREDA r
         | CREDB r
           => CREDB r
         | CREDC r
           => CREDC r)
  | search_com (SET (l, a))
    = (case search_aexp a
        of AVAL v
           => CREDC (PR_SET (l, v))
         | AREDA r
           => CREDA r)
  | search_com (IF (b, c1, c2))
    = (case search_bexp b
        of BVAL v
           => CREDC (PR_IF (v, c1, c2))
         | BREDA r
           => CREDA r
         | BREDB r
           => CREDB r)
  | search_com (WHILE (b, c))
    = CREDC (PR_WHILE (b, c))
```

Likewise, this function searches for the next redex in commands. Any redex found in a sub-expression or sub-command is mapped to a redex of the composite command. Here a redex can either by a reducible arithmetic expression, a reducible boolean expression, or a reducible command.

The search for the next redex of a program is the search for the next redex in a command:

```
(*  com -> cval_or_red  *)
fun search t = search_com t
```

4.1.2 CPS transforming the search functions

Next we CPS transform the search functions, as described in Section 3.1. The transformation of `search_aexp` is displayed in Figure 4.1. We can simplify the CPS-transformed functions to directly return a redex instead of having it "bubble up" during search. The direct return of a final value is achieved by discarding the continuation the same way as we did for errors in Section 2.2.2. Thus we fix the answer type, denoted by the type variable `'a` in the ML code, to be either a command value or one of the three redexes. If the result is a redex, we also pair it with its continuation, which, as we will see shortly, encodes the reduction context of the redex. Thus the type of final results is:

```
datatype cval_or_red
  = CVAL  of cval
  | CREDA of ared * acont
  | CREDB of bred * bcont
  | CREDC of cred * ccont

withtype acont = aval -> cval_or_red
     and bcont = bval -> cval_or_red
     and ccont = cval -> cval_or_red
```

```
(*  aexp * (aval_or_red -> 'a) -> 'a  *)
fun search_aexp (NUM n, k)
    = k (AVAL (VNUM n))
  | search_aexp (REF l, k)
    = k (AREDA (PR_REF l))
  | search_aexp (ADD (a1, a2), k)
    = search_aexp (a1,
        fn AVAL v1
           => search_aexp (a2,
                fn AVAL v2
                   => k (AREDA (PR_ADD (v1, v2)))
                 | AREDA r
                   => k (AREDA r))
         | AREDA r
           => k (AREDA r))
  | search_aexp (SUB (a1, a2), k)
    = search_aexp (a1,
        fn AVAL v1
           => search_aexp (a2,
                fn AVAL v2
                   => k (AREDA (PR_SUB (v1, v2)))
                 | AREDA r
                   => k (AREDA r))
         | AREDA r
           => k (AREDA r))
  | search_aexp (MUL (a1, a2), k)
    = search_aexp (a1,
        fn AVAL v1
           => search_aexp (a2,
                fn AVAL v2
                   => k (AREDA (PR_MUL (v1, v2)))
                 | AREDA r
                   => k (AREDA r))
         | AREDA r
           => k (AREDA r))
```

Figure 4.1: The CPS-transformed search function for arithmetic expressions

Because redexes are returned directly, only the respective values are passed to continuations. Thus, when searching an arithmetic expression, the type of its continuation is `aval -> cval_or_red`. The simplification of `search_aexp` to directly return redexes is displayed in Figure 4.2.

4.1.3 Defunctionalizing the continuations

Next we defunctionalize the continuations of the search functions, as described in Section 3.2. The transformation of `search_aexp` together with the data types of contexts is displayed in Figure 4.3. The continuations of the search function encode a notion of context which is given a first-order representation by defunctionalization. The data type of defunctionalized continuations coincide with the reduction contexts of our reduction semantics in Figure 2.7. Furthermore, the CPS-transformed and defunctionalized search function coincides with the decomposition function in Figure 2.9, Figure 2.10, and Figure 2.11: given a command it finds (by construction) the next redex and returns both the

```
(*  aexp * (aval -> cval_or_red) -> cval_or_red  *)
fun search_aexp (NUM n, k)
    = k (VNUM n)
  | search_aexp (REF l, k)
    = CREDA (PR_REF l, k)
  | search_aexp (ADD (a1, a2), k)
    = search_aexp (a1,
        fn v1
           => search_aexp (a2,
                fn v2
                   => CREDA (PR_ADD (v1, v2), k)))
  | search_aexp (SUB (a1, a2), k)
    = search_aexp (a1,
        fn v1
           => search_aexp (a2,
                fn v2
                   => CREDA (PR_SUB (v1, v2), k)))
  | search_aexp (MUL (a1, a2), k)
    = search_aexp (a1,
        fn v1
           => search_aexp (a2,
                fn v2
                   => CREDA (PR_MUL (v1, v2), k)))
```

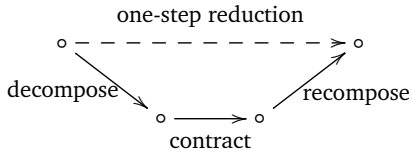
Figure 4.2: The simplified CPS-transformed search function for arithmetic expressions

redex and a representation of its reduction context.

As this prequel shows, starting from a specification of a reduction strategy given as a total and compositional function, we can mechanically extract both the grammar of reduction context and the decomposition function of a reduction semantics.

4.2 Refocusing: from reduction-based to reduction-free evaluation

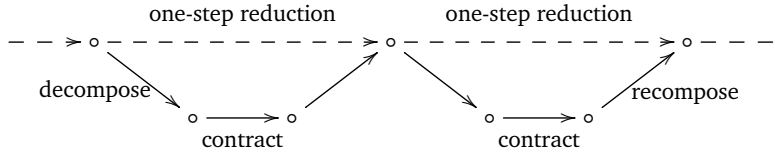
A reduction semantics defines one-step reduction as the composition of decomposition, contraction and recomposition (Figure 2.12). Diagrammatically:



```
datatype ccont = ...
datatype bcont = ...

datatype acont
  = A1 of acont * aexp
  | A2 of acont * aexp
  | A3 of acont * aexp
  | A4 of aval * acont
  | A5 of aval * acont
  | A6 of aval * acont
  | A7 of bcont * aexp  (* from: search_bexp (LEQ (a1, a2), k) *)
  | A8 of aval * bcont
  | A9 of loc * ccont   (* from: search_com (SET (l, a), k)    *)

(*  acont * aval -> cval_or_red  *)
fun search_acont (A1 (k, a2), v1) = search_aexp (a2, A4 (v1, k))
  | search_acont (A2 (k, a2), v1) = search_aexp (a2, A5 (v1, k))
  | search_acont (A3 (k, a2), v1) = search_aexp (a2, A6 (v1, k))
  | search_acont (A4 (v1, k), v2) = CREDA (PR_ADD (v1, v2), k)
  | search_acont (A5 (v1, k), v2) = CREDA (PR_SUB (v1, v2), k)
  | search_acont (A6 (v1, k), v2) = CREDA (PR_MUL (v1, v2), k)
  | search_acont (A7 (k, a2), v1) = search_aexp (a2, A8 (v1, k))
  | search_acont (A8 (v1, k), v2) = CREDB (PR_LEQ (v1, v2), k)
  | search_acont (A9 (l, k),  v ) = CREDC (PR_SET (l, v), k)

(*  aexp * acont -> cval_or_red  *)
and search_aexp (NUM n,        k) = search_acont (k, VNUM n)
  | search_aexp (REF l,        k) = CREDA (PR_REF l, k)
  | search_aexp (ADD (a1, a2), k) = search_aexp (a1, A1 (k, a2))
  | search_aexp (SUB (a1, a2), k) = search_aexp (a1, A2 (k, a2))
  | search_aexp (MUL (a1, a2), k) = search_aexp (a1, A3 (k, a2))
```

Figure 4.3: The defunctionalized and CPS-transformed search function for arithmetic expressions

Evaluation is then the iteration of one-step reduction (Figure 2.13). Diagrammatically:

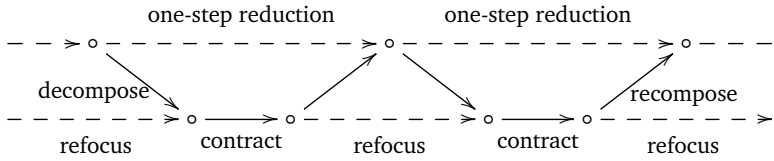


This evaluation of an expression is *reduction based* because it enumerates the successive reducts in the reduction sequence, i.e., it enumerates the successive expressions in the sequence of one-step reductions. Reduction-based evaluation is needlessly inefficient because after each contraction, the reduction context and contractum are recomposed to construct the reduced expression, only to be subsequently decomposed in search of the next redex. Danvy and Nielsen have shown how, under circumstances met here, the enumeration of intermediate expressions in the reduction sequence can be deforested away

```
(*  val_or_dec * store -> store  *)
fun iterate (VAL VSKIP, st)
    = st
  | iterate (ADEC (r, E), st)
    = iterate (decompose_aexp (contract_ared (r, st), E), st)
  | iterate (BDEC (r, E), st)
    = iterate (decompose_bexp (contract_bred r, E), st)
  | iterate (CDEC (r, E), st)
    = let val (c', st') = contract_cred (r, st)
      in iterate (decompose_com (c', E), st')
      end

(*  com -> store  *)
fun eval c = iterate (decompose_com (c, C0), EMPTY)
```

Figure 4.4: Reduction-free evaluation obtained by refocusing

by refocusing [71, 197].



This deforestation is achieved by simply *continuing the decomposition of the contractum in the current context* and is implemented in Figure 4.4.

4.3 Lightweight fusion: from small-step to big-step abstract machine

We can view the reduction-free evaluation in Section 4.2 as a state-transition system in *small steps*: the states are decompositions and the transitions are driven by the iterate function (also known as a 'driver loop' or 'trampoline' [98]). However, because we are not interested in the construction of these intermediate states when implementing evaluation, we can instead view the state-transition system as producing a final result in one *big step*: the states are functions and the transitions are function calls in tail position.

The correspondence between these two views is established by the lightweight-fusion program transformation [155]. This program transformation fuses iterate with the decomposition functions so that the resulting function is directly applied to the result of decomposition. Thus `iterate` fused with `decompose_aexp` becomes `eval_aexp`, `iterate` fused with `decompose_acont` becomes `eval_acont`, and so forth. The evaluation functions for arithmetic expressions after lightweight fusion are shown in Figure 4.5 together with the new iterate function.

```
(*  aexp * acont * store -> store  *)
fun eval_aexp (NUM n,        k, st) = eval_acont (k, VNUM n, st)
  | eval_aexp (REF l,        k, st) = iterate (ADEC (PR_REF l, k), st)
  | eval_aexp (ADD (a1, a2), k, st) = eval_aexp (a1, A1 (k, a2), st)
  | eval_aexp (SUB (a1, a2), k, st) = eval_aexp (a1, A2 (k, a2), st)
  | eval_aexp (MUL (a1, a2), k, st) = eval_aexp (a1, A3 (k, a2), st)

(*  acont * aval * store -> store  *)
and eval_acont (A1 (k, a2), v1, st) = eval_aexp (a2, A4 (v1, k), st)
  | eval_acont (A2 (k, a2), v1, st) = eval_aexp (a2, A5 (v1, k), st)
  | eval_acont (A3 (k, a2), v1, st) = eval_aexp (a2, A6 (v1, k), st)
  | eval_acont (A4 (v1, k), v2, st) = iterate (ADEC (PR_ADD (v1, v2), k), st)
  | eval_acont (A5 (v1, k), v2, st) = iterate (ADEC (PR_SUB (v1, v2), k), st)
  | eval_acont (A6 (v1, k), v2, st) = iterate (ADEC (PR_MUL (v1, v2), k), st)
  | eval_acont (A7 (k, a2), v1, st) = eval_aexp (a2, A8 (v1, k), st)
  | eval_acont (A8 (v1, k), v2, st) = iterate (BDEC (PR_LEQ (v1, v2), k), st)
  | eval_acont (A9 (l, k),  v,  st) = iterate (CDEC (PR_SET (l, v), k), st)


and eval_bexp  : bexp * bcont * store -> store = ...
and eval_bcont : bcont * bval * store -> store = ...
and eval_com   : com  * ccont * store -> store = ...
and eval_ccont : ccont * cval * store -> store = ...


(*  val_or_dec * store -> store  *)
and iterate (VAL VSKIP,   st) = st
  | iterate (ADEC (r, E), st) = eval_aexp (contract_ared (r, st), E, st)
  | iterate (BDEC (r, E), st) = eval_bexp (contract_bred r, E, st)
  | iterate (CDEC (r, E), st) = let val (c', st') = contract_cred (r, st)
                                in eval_com (c', E, st')
                                end

(*  com -> store  *)
fun eval c = eval_com (c, C0, EMPTY)
```

Figure 4.5: Evaluation of arithmetic expressions after lightweight-fusion

4.4 Hereditary transition compression

In the evaluation function derived by lightweight fusion in Section 4.3, many of the transitions are 'corridor' ones in that they yield configurations for which there is a unique further transition. For example, in the forth case of `eval_acont` we know that the next transition is the decomposition case for arithmetic expressions in iterate. Furthermore, we know that the redex is an addition redex. Thus, we can compress these transitions:

```
eval_acont (A4 (v1 as VNUM n1, k), v2 as VNUM n2, st)
  = iterate (ADEC (PR_ADD (v1, v2), k), st)
  = eval_aexp (contract_ared (PR_ADD (v1, v2), st), k, st)
  = eval_aexp (NUM (n1 + n2), k, st)
  = eval_acont (k, VNUM (n1 + n2), st)
```

We hereditary compress all of these corridor transitions. The result is the implementation of an abstract machine displayed in Figure 4.6. This implementation has a remaining inefficiency in the last case of `eval_com` pertaining to `WHILE`. Here, it constructs a `B3` context with the partially known expressions `SEQ (c, WHILE (b, c))` and `SKIP`. This structure

```
(* aexp * acont * store -> store *)
fun eval_aexp (NUM n,        k, st) = eval_acont (k, VNUM n, st)
  | eval_aexp (REF l,        k, st) = eval_acont (k, lookup (l, st), st)
  | eval_aexp (ADD (a1, a2), k, st) = eval_aexp (a1, A1 (k, a2), st)
  | eval_aexp (SUB (a1, a2), k, st) = eval_aexp (a1, A2 (k, a2), st)
  | eval_aexp (MUL (a1, a2), k, st) = eval_aexp (a1, A3 (k, a2), st)

(* acont * aval * store -> store *)
and eval_acont (A1 (k, a2),      v1,      st) = eval_aexp (a2, A4 (v1, k), st)
  | eval_acont (A2 (k, a2),      v1,      st) = eval_aexp (a2, A5 (v1, k), st)
  | eval_acont (A3 (k, a2),      v1,      st) = eval_aexp (a2, A6 (v1, k), st)
  | eval_acont (A4 (VNUM n1, k), VNUM n2, st) = eval_acont (k, VNUM (n1 + n2), st)
  | eval_acont (A5 (VNUM n1, k), VNUM n2, st) = eval_acont (k, VNUM (n1 - n2), st)
  | eval_acont (A6 (VNUM n1, k), VNUM n2, st) = eval_acont (k, VNUM (n1 * n2), st)
  | eval_acont (A7 (k, a2),      v1,      st) = eval_aexp (a2, A8 (v1, k), st)
  | eval_acont (A8 (VNUM n1, k), VNUM n2, st) = eval_bcont (k, if n1 <= n2
                                                               then VTRUE else VFALSE, st)
  | eval_acont (A9 (l, k),       v,       st) = eval_ccont (k, VSKIP, ASSOC (l, v, st))

(* bexp * bcont * store -> store *)
and eval_bexp (TRUE,         k, st) = eval_bcont (k, VTRUE, st)
  | eval_bexp (FALSE,        k, st) = eval_bcont (k, VFALSE, st)
  | eval_bexp (LEQ (a1, a2), k, st) = eval_aexp (a1, A7 (k, a2), st)
  | eval_bexp (NOT b,        k, st) = eval_bexp (b, B1 k, st)
  | eval_bexp (AND (b1, b2), k, st) = eval_bexp (b1, B2 (k, b2), st)

(* bcont * bval * store -> store *)
and eval_bcont (B1 k,           VTRUE,  st) = eval_bcont (k, VFALSE, st)
  | eval_bcont (B1 k,           VFALSE, st) = eval_bcont (k, VTRUE, st)
  | eval_bcont (B2 (k, b2),     VTRUE,  st) = eval_bexp (b2, k, st)
  | eval_bcont (B2 (k, b2),     VFALSE, st) = eval_bcont (k, VFALSE, st)
  | eval_bcont (B3 (k, c1, c2), VTRUE,  st) = eval_com (c1, k, st)
  | eval_bcont (B3 (k, c1, c2), VFALSE, st) = eval_com (c2, k, st)

(* com * ccont * store -> store *)
and eval_com (SKIP,           k, st) = eval_ccont (k, VSKIP, st)
  | eval_com (SEQ (c1, c2),   k, st) = eval_com (c1, C1 (k, c2), st)
  | eval_com (SET (l, a),     k, st) = eval_aexp (a, A9 (l, k), st)
  | eval_com (IF (b, c1, c2), k, st) = eval_bexp (b, B3 (k, c1, c2), st)
  | eval_com (WHILE (b, c),   k, st) = eval_bexp (b, B3 (k, SEQ (c, WHILE (b, c)), SKIP), st)

(* ccont * cval * store -> store *)
and eval_ccont (C0,         VSKIP, st) = st
  | eval_ccont (C1 (k, c2), VSKIP, st) = eval_com (c2, k, st)

(* com -> store *)
fun eval c = eval_com (c, C0, EMPTY)
```

Figure 4.6: Evaluation functions after hereditary transition compression

is no longer known in the case for `B3` in `eval_bcont`. To avoid loosing information, we specialize the context as `B4` and add a new case in `eval_bcont` to continue when given this specialized context:

```
  eval_com (WHILE (b, c),   k, st)
    = eval_bexp (b, B3 (k, SEQ (c, WHILE (b, c)), SKIP), st)
    = eval_bexp (b, B4 (k, b, c), st)
```

```
eval_bcont (B4 (k, b, c), VTRUE,  st)
  = eval_com (SEQ (c, WHILE (b, c)), k, st)
  = eval_com (c, C1 (k, WHILE (b, c)), st)

eval_bcont (B4 (k, b, c), VFALSE, st)
  = eval_com (SKIP, k, st)
  = eval_ccont (k, VSKIP, st)
```

This specialization gives rise to a similar construction of `C1` with the partially known expression `WHILE (b, c)`. We specialize this context as `C2` and add a new case in `eval_ccont` to continue when given this specialized context:

```
eval_bcont (B4 (k, b, c), VTRUE,  st)
  = eval_com (c, C1 (k, WHILE (b, c)), st)
  = eval_com (c, C2 (k, b, c), st)

eval_ccont (C2 (k, b, c), v, st)
  = eval_cexp (WHILE (b, c), k, st)
  = eval_bexp (b, B4 (k, b, c), st)
```

This hereditary-compressed machine coincides with the abstract machine in Figure 3.3, where truth values are represented by ML booleans, numeric values by ML integers, command values (i.e., `VSKIP`) are eliminated entirely, and a few names are renamed.

4.5 Summary and conclusion

Starting from a compositional search function, we have mechanically derived the implementation of a reduction semantics and an abstract machine for the same language. The structural coincidence between the implicit context in a search function, the explicit reduction context in a reduction semantics, and the first-order stack in an abstract machine play the key rôle in this connection.

We have mechanically derived the same implementation of an abstract machine in two ways:

1. Starting with an implementation of the direct semantics for $\mathscr{I}$ and using the functional correspondence (Chapter 3).
2. Starting with an implementation of the reduction semantics for $\mathscr{I}$ and using the syntactic correspondence (Chapter 4).

This connection is no coincidence but is witness to a striking unity of computation across big-step and small-step specifications and their implementations.

4.6 The syntactic correspondence in perspective

The development of the syntactic correspondence started with Danvy and Nielsen's work on refocusing a reduction semantics to obtain a reduction-free interpreter [59, 71], i.e., one that does not enumerate the reduction sequence during evaluation. This reduction-free interpreter is further fused into the form of an abstract machine [67], providing a syntactic companion to the functional correspondence. This syntactic correspondence has been shown to be widely applicable and to provide instrumental guidelines to construct new semantic artifacts and to establish the equivalence of existing ones [62]:

- Danvy and Nielsen [59, 71] apply the correspondence to call-by-value and call-by-name reduction. Starting from a call-by-value reduction semantics they derive Felleisen and Friedman's CK machine [87], which has no environment component. Starting from a call-by-name reduction semantics they derive a variant of Krivine's machine [127, 128] using actual substitutions.
- Biernacka and Danvy [28] extend the correspondence to calculi of explicit substitutions and abstract machines with environments. Starting from a call-by-value reduction semantics with explicit substitutions they derive Felleisen and Friedman's CEK machine [87]. Starting from a call-by-name reduction semantics with explicit substitutions they derive Krivine's machine [127, 128]. In addition, they derive Leroy's Zinc machine [136] as an applicative-order analog of Krivine's machine.
- Biernacka, Biernacki, and Danvy [30] extend the correspondence to account for delimited control in the CPS hierarchy. For a given level of the CPS hierarchy, n, they construct a reduction semantics with $n+1$ layers of evaluation contexts from which they derive an abstract machine with $n+1$ control stacks accounting for the family of control operators shift_n and reset_n.
- Biernacka and Danvy [27] extend the correspondence to account for languages with effects. Their work connects several semantics artifacts featuring control operators, stack inspection, tail recursion and store-based lazy evaluation.
- Biernacka and Danvy [29] give a calculus of explicit substitutions that is inter-derivable with Clinger's abstract machine for Core Scheme [47].
- Munk [151] shows that the correspondence systematically accounts for Felleisen and Hieb's theories of control and state [85, 88].
- Danvy and Johannsen [65, 119] extend the correspondence to inter-derive a new abstract machine with Abadi and Cardelli's object calculus [1]. In addition they provide a new variant of the calculus based on explicit substitutions together with its inter-derived abstract machine.
- Sieczkowski et al. [196, 197] mechanized the refocusing transformation and proved it correct in the Coq proof assistant.
- Anton and Thiemann [9, 10] extend the correspondence to derive a type system and an implementation for coroutines.
- Sergey and Clarke [192–194] extend the correspondence to inter-derive type checking algorithms. In particular, they connect reduction-based type checking [129] with traditional recursive-descent type checking [166].
- Recently, Ariola et al. [17] used the correspondence to derive a sequent calculus for classical call-by-need reduction.

In this dissertation, the techniques from the syntactic correspondence are employed in several places:

- Chapter 5 extends the syntactic correspondence with a prequel to reduction semantics, which is based on the functional correspondence. Starting from a big-step and specification of a reduction strategy in the form of a compositional search function,

we derive both the grammar of reduction contexts and the decomposition function of a reduction semantics. From this derived formulation the syntactic correspondence can then be applied to obtain an abstract machine.

- Chapter 6 extends the syntactic correspondence to inter-derive small-step and big-step normalization functions for boolean-propositional formulas. The immediate reduction semantics for conjunction normalization is not in the form required by refocusing. We show how to disambiguate the reduction rules using delimited reduction contexts. Refocusing the resulting system yields an abstract machine with delimited stacks which can be further inter-derived using the functional correspondence.

- Chapter 7 extends the syntactic correspondence to connect store-less accounts of call-by-need evaluation. We use the syntactic correspondence to inter-derive a store-less abstract machine directly from Ariola-al call-by-need λ-calculus [15]. In particular, this work clarifies how several types of reduction contexts are used to encode the demand-driven computation and memoization of call by need.

- Chapter 8 gives a synthetic account of call-by-need evaluation. It provides bisimulations for a series reduction semantics for call-by-need evaluation spanning from the storeless call-by-need calculus of Ariola-al et al. [15] to a state-based reduction semantics using a store and updateable thunks. Each of these reduction semantics is then inter-derived with their corresponding abstract machine and natural semantics. Together, the bisimulations and inter-derivations give a constructive connection of Crégut's original lazy Krivine machine [51] with Ariola et al.'s call-by-need λ-calculus, and with Launchbury's natural semantics for call by need. In addition, the synthetic account systematically maps out the "spaces between" the existing specifications.

- Chapter 9 and Chapter 10 connect the theory and practice of Combinatory graph reduction. The theory as embodied by Barendregt et al.'s term-graph rewriting, and the practice as embodied by Turner's original reduction machine. Chapter 9 establishes this connection in the context of an impure implementation of graph rewriting. Chapter 10 establishes this connection in the context of syntactic theories for graph rewriting.

Part II

Publications

Chapter 5

A prequel to reduction semantics

Joint work with Olivier Danvy.

Abstract

We present a methodology for deriving a reduction semantics from an equational theory. This methodology is simple and it scales to arbitrarily complex cases. Our starting point is a compositional, big-step specification of a deterministic reduction strategy. We then mechanically derive the key components of a reduction semantics: an optimal grammar of evaluation contexts, a proof of the unique decomposition property, and small-step implementations of the decomposition function and of the recomposition function. The resulting reduction semantics is "refocus-ready," i.e., it is in a form that makes it very simple to derive a corresponding abstract machine, which is traditionally perceived and presented as a major endeavor. The methodology uses two off-the-shelf program transformations: the CPS transformation and defunctionalization. We present three applications: arithmetic expressions, the lambda-calculus with exceptions, and JavaScript, which we extend with coroutines in passing.

5.1 Introduction

Since their inception in Felleisen's PhD thesis [85], witness any recent proceedings of POPL or ICFP, reduction semantics have become something of a de-facto standard for semantic engineers to specify programming languages [89]. A reduction semantics is to a small-step operational semantics [171] what a continuation semantics is to a (denotational or operational) big-step semantics in direct style: in such a semantics, each term is evaluated along with a representation of 'the rest of the evaluation' – i.e., its evaluation context – and in such a small-step semantics, each term is reduced along with a representation of 'the rest of the reduction' – i.e., its reduction context. Stating a reduction semantics therefore requires one to phrase its reduction contexts and, for deterministic languages, to prove a unique decomposition property.

There is no method to specify reduction contexts: semantic engineers are on their own to decide whether all cases are covered and whether none of them is redundant. Ditto for proving the unique decomposition property.

And then there is the issue of implementing the reduction semantics as an abstract machine and of proving the soundness of this abstract machine. Again, semantic engineers are on their own: for example, contexts are typically represented 'outside-in' at first, but then most of the time they end up being used 'inside out' shortly thereafter, on the intuitive ground that "a continuation is just an inside-out evaluation context" – an intuition that requires the semantic engineer to understand continuations. For non-trivial cases such as stack inspection [46] and call by need [99], implementing a reduction semantics as an abstract machine is a substantial endeavor. Refocusing [71, 197] has been proposed to assist this effort, but not all the reduction semantics are "refocus-ready." In any case, the crucial issue is to write a good decomposition function, but there is no method for that.

In this article, we present a method where all of the above can be obtained mechanically from the specification of a deterministic reduction strategy. The method works as follows:

1. For a given grammar of terms and values, for a given notion of contraction, and for a given reduction strategy, write a compositional function that searches a non-value term for the first potential redex according to the reduction strategy.

2. CPS transform this search function and defunctionalize the continuations. The result is the grammar of the reduction contexts along with two mutually recursive search functions acting on terms and on reduction contexts.

3. Extend this defunctionalized search function to return the first potential redex in a given term *together with the corresponding reduction context*. The result is a tail-recursive implementation of the decomposition function that is refocus-ready.

It is our experience that for its simplicity, this prequel to reduction semantics is substantially applicable. It applies not only to the λ-calculus (with actual substitutions and with explicit substitutions, and with call by value, call by name, and call by need), to combinatory logic, and to graph rewriting, but also to the λ-calculus with effects (be them undelimited or delimited). For a concrete example, it applies to all of the reduction semantics in Felleisen et al.'s textbook on semantic engineering [89].

In this article, we introduce this prequel and illustrate it with three semantics of increasing complexity: arithmetic expressions, the λ-calculus with exceptions, and Guha et al.'s semantics for JavaScript, λ_{JS}, which we extend with coroutines in passing.

Our message is that semantic notation is not random, nor is it arbitrary, and that it is also *consistent* and *inter-derivable*. This inter-derivability means that there is no need to learn separately about the possibility of reduction to be stuck (Plotkin) and of evaluation to go wrong (Milner): these two concepts are consistent and inter-derivable. Likewise for the notion of reduction strategy (normal order, applicative order, etc.) and of evaluation order (call by name, call by value, etc.): each reduction strategy is mirrored into an evaluation strategy through the inter-derivation. All in all, this inter-derivability means that semantic engineers do not need to go out of their zone of comfort to specify, characterize, or reason about programming languages. As for beginning semantic engineers, they do not need to be put off by existing semantics, or to feel helpless at the prospect of writing one: the specific contribution of the present methodology is a simple, practical, and scalable method to formulate reduction semantics. And once a beginning semantic engineer knows how to formulate a reduction semantics, (s)he is much more at ease to understand another one, for as Richard Feynman puts it, "What I cannot create, I do not understand." This methodology is now in use outside the authors' university, it has already

served as guideline to semantic engineers for obtain reduction semantics that would be daunting to write from scratch, and it has already been used to spot errors and fix them. In short, it makes it possible even for a beginning semantic engineer to formulate a correct, non-trivial, and useful reduction semantics that can then be inter-derived into an abstract machine. The variety of (necessarily concise) examples of the present article illustrates the general applicability of the methodology for deriving a reduction semantics from a simple prequel.

Overview: We illustrate the prequel with Pierce's language of conditional arithmetic expressions [166] (Section 5.2). We then turn to the call-by-value λ-calculus (Section 5.3). We extend the treatment of call-by-value to account for exceptions with context sensitive contraction rules (Section 5.4), and context insensitive contraction rules (Section 5.5). From the treatment of context insensitive contraction rules, we apply the method to Guha et al.'s semantics for JavaScript (Section 5.6).

Prerequisites: We assume the reader to be mildly familiar with the notion of reduction semantics [85, 89], the continuation-passing style transformation [94, 200] and to know that a function is defunctionalized into a data type and an apply function dispatching over this data type [70, 176].

Notations: We use $n \in \mathsf{Int}$ to denote integers, $b \in \mathsf{Bool}$ to denote the truth values true and false, and $x \in \mathsf{Ide}$ to denote identifiers. We use C to denote an abstract one-hole context which is concretely represented as elements in the function space $\mathsf{Term} \to \mathsf{Term}$ for some definition of Term. The term obtained by plugging the term t into the hole of the context C is written as $C[t]$; concretely, this plugging is achieved by applying C to t. The empty context is written as $[\,]$; concretely, it is represented as the identity function. When specifying the contraction rules for a reduction semantics, we distinguish between two types of rules: those that depend upon the context surrounding a redex – the context-sensitive rules; and those that only depend on the redex itself – the context-insensitive rules. Finally, to avoid confusion with syntactic abstraction and application, we use $\overline{\lambda}$ to indicate meta-level abstraction and $\overline{@}$ as the infix meta-level application.

5.2 Arithmetic expressions

The goal of this section is to present our prequel to reduction semantics. To this end, we consider a simple subset of Pierce's language of conditional arithmetic expressions [166]. This section serves as a point of reference for all of the later developments, so we detail it.

5.2.1 Syntax and contraction

The formulation of conditional arithmetic expressions reads as follows:

Definition 9 (conditional arithmetic expressions)**.**
Syntax:

$$\begin{aligned} \mathsf{Term} \ni t &::= n \mid b \mid t + t \mid t \Rightarrow t; t \\ \mathsf{Val} \ni v &::= n \mid b \\ \mathsf{PotRed} \ni r &::= v + v \mid v \Rightarrow t; t \end{aligned}$$

Contraction rules:

$$\begin{array}{ll} (plus) & n_0 + n_1 \to n \quad \text{where } n = n_0 + n_1 \\ (true) & \mathsf{true} \Rightarrow t_1; t_2 \to t_1 \\ (false) & \mathsf{false} \Rightarrow t_1; t_2 \to t_2 \end{array}$$

$$\mathscr{A} = (plus) \cup (true) \cup (false)$$

In words: Terms are literals, additions of two terms, and conditional expressions. We have two disjoint restrictions on terms: values which are the normal forms; and potential redexes which are the terms subject to reduction. Values are literals. There are two forms of potential redexes: the addition of two values or a conditional expression testing a value. To be an actual redex, the two values in an addition have to be integer literals, and the value being tested in a conditional expression has to be a Boolean literal. Otherwise, none of the contraction rules apply and the redex and the corresponding program are said to be stuck. We let $\mathscr{A}$ be the notion of reduction on conditional arithmetic expressions.

5.2.2 Reduction strategy

We consider the left-most outer-most reduction strategy where the branches of conditional expressions are reduced lazily. The following reduction sequence illustrates it:

$(1000 + 100) + (10 + 1) \to$
$1100 + (10 + 1) \to$
$1100 + 11 \to$
1111

This reduction strategy is obtained with a depth-first left-to-right search through the syntax tree of the term under reduction. Let us spell out this search as a function that maps a value term to itself and a non-value term to the first potential redex in the reduction sequence. The search is carried out with a straight recursive descent on terms, following the reduction strategy. Literals are mapped to values; the left sub-term of an addition is first traversed, and if it turns out to be a value, the right sub-term is then traversed; and the test sub-term of a conditional expression is traversed. To avoid notational cluttering, we leave injection tags implicit:

$$\begin{array}{rl} \mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed} & \\ term(n) = & n \\ term(b) = & b \\ term(t_0 + t_1) = & \mathit{case}\ term(t_0)\ \mathit{of} \\ & \quad \mid v_0 \to \mathit{case}\ term(t_1)\ \mathit{of} \\ & \qquad\quad \mid v_1 \to v_0 + v_1 \\ & \qquad\quad \mid r \ \to r \\ & \quad \mid r \ \to r \\ term(t_0 \Rightarrow t_1; t_2) = & \mathit{case}\ term(t_0)\ \mathit{of} \\ & \quad \mid v_0 \to v_0 \Rightarrow t_1; t_2 \\ & \quad \mid r \ \to r \end{array}$$

In general, search is a compositional and total function mapping a term to a value, potential redex, or an error. Errors are observable objects and thus distinguishable from

non-termination. Depending on the well-formedness criteria for terms, errors might be avoided as is the case for this account of arithmetic expressions.

$$\begin{aligned}
&\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed}\\
&\mathit{search}(t) = \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\
&\qquad\qquad \mid v \to v\\
&\qquad\qquad \mid r \to r
\end{aligned}$$

While straightforward to write, this search function forces one to express the essence of the reduction paraphernalia. Indeed, from this search function alone, we can mechanically extract both the grammar of reduction contexts and the decompose function, as illustrated in the next sections.

NB. This search function reflects the usual congruence rules and propagation of values as specified in a structural operational semantics [171].

5.2.3 CPS transforming the search function

Let us start by CPS transforming the search function of Section 5.2.2:

$$\begin{aligned}
&\mathsf{Term} \times (\mathsf{Val} + \mathsf{PotRed} \to \alpha) \to \alpha\\
&\qquad\qquad \mathit{term}(n,\ k) = k\,\overline{@}\,n\\
&\qquad\qquad \mathit{term}(b,\ k) = k\,\overline{@}\,b\\
&\qquad \mathit{term}(t_0 + t_1,\ k) = \mathit{term}(t_0,\ \overline{\lambda} x.\mathit{case}\ x\ \mathit{of}\\
&\qquad\qquad\qquad\qquad \mid v_0 \to \mathit{term}(t_1,\ \overline{\lambda} x.\mathit{case}\ x\ \mathit{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid v_1 \to k\,\overline{@}\,(v_0 + v_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid r \ \to k\,\overline{@}\,r)\\
&\qquad\qquad\qquad\qquad \mid r \ \to k\,\overline{@}\,r)\\
&\mathit{term}(t_0 \Rightarrow t_1; t_2,\ k) = \mathit{term}(t_0,\ \overline{\lambda} x.\mathit{case}\ x\ \mathit{of}\\
&\qquad\qquad\qquad\qquad \mid v_0 \to k\,\overline{@}\,(v_0 \Rightarrow t_1; t_2)\\
&\qquad\qquad\qquad\qquad \mid r \ \to k\,\overline{@}\,r)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed}\\
&\mathit{search}(t) = \mathit{term}(t,\ \overline{\lambda} x.\mathit{case}\ x\ \mathit{of}\\
&\qquad\qquad\qquad\qquad \mid r \to r\\
&\qquad\qquad\qquad\qquad \mid v \to v)
\end{aligned}$$

The result is a tail-recursive search function with a functional representation of "the rest of the search." This representation encodes a notion of context, as developed in the following section.

5.2.4 Defunctionalizing the continuations

In Section 5.2.3, the search function constructs continuations at three distinct sites, given an initial continuation. We therefore partition the function space into four summands:

1. The initial continuation, the identity function, which has no free variables.
2. The first continuation when searching in an addition, which has two free variables: the continuation k, and the term t_1.

3. The second continuation when searching in an addition, which has two free variables: the value v_0, and the continuation k.
4. The continuation when searching in a conditional expression, which has three free variables: the continuation k, and the terms t_1 and t_2.

Defunctionalizing the continuation of the search function therefore gives rise to (1) the following data type of reduction contexts, expressed as a grammar:

$$\mathsf{Ctx} \ni C ::= [\,] \mid C + t \mid v + C \mid C \Rightarrow t; t$$

and (2) an apply function dispatching over the data type of contexts:

$$\begin{aligned}
\mathsf{Ctx} \times (\mathsf{Val} + \mathsf{PotRed}) &\to \mathsf{Val} + \mathsf{PotRed} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}([\,],\ r) &= r \\
\mathit{context}(C + t_1,\ v_0) &= \mathit{term}(t_1,\ v_0 + C) \\
\mathit{context}(C + t_1,\ r) &= \mathit{context}(C,\ r) \\
\mathit{context}(v_0 + C,\ v_1) &= \mathit{context}(C,\ v_0 + v_1) \\
\mathit{context}(v_0 + C,\ r) &= \mathit{context}(C,\ r) \\
\mathit{context}(C \Rightarrow t_1; t_2,\ v_0) &= \mathit{context}(C,\ v_0 \Rightarrow t_1; t_2) \\
\mathit{context}(C \Rightarrow t_1; t_2,\ r) &= \mathit{context}(C,\ r)
\end{aligned}$$

After defunctionalization, the search function reads as follows:

$$\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(b,\ C) &= \mathit{context}(C,\ b) \\
\mathit{term}(t_0 + t_1,\ C) &= \mathit{term}(t_0,\ C + t_1) \\
\mathit{term}(t_0 \Rightarrow t_1; t_2,\ C) &= \mathit{term}(t_0,\ C \Rightarrow t_1; t_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \\
\mathit{search}(t) &= \mathit{term}(t,\ [\,])
\end{aligned}$$

Simplifying the apply function: In the starting definition of the search function (Section 5.2.2), a redex bubbles up through the chain of returns, reflecting the depth of the recursive descent. After CPS transformation (Section 5.2.3), the definition of the search function is tail-recursive, and so is its defunctionalized counterpart. However, in a tail-recursive definition, the chain of returns has length 1, and there is thus no need to bubble up redexes when they are found. Let us optimize the definition of the search function by short-cutting the propagation of potential redexes:

Proposition 10 (potential redexes in context)**.** $\mathit{context}(C,\ r) = r$

Proof. Induction on C. □

As a consequence, all dispatching on potential redexes can be avoided in the definition of *context*:

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\to \mathsf{Val} + \mathsf{PotRed} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}(C + t_1,\ v_0) &= \mathit{term}(t_1,\ v_0 + C) \\
\mathit{context}(v_0 + C,\ v_1) &= v_0 + v_1 \\
\mathit{context}(C \Rightarrow t_1; t_2,\ v_0) &= v_0 \Rightarrow t_1; t_2
\end{aligned}$$

5.2.5 Searching for the first potential redex and its reduction context

Let us make the search function return not just the first potential redex in a given non-value term but also the associated reduction context. Together, the potential redex and the reduction context form a *decomposition* $\langle r, C\rangle \in \mathsf{PotRed} \times \mathsf{Ctx}$. We therefore rename *search* to *decompose*:

$$\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(b,\ C) &= \mathit{context}(C,\ b) \\
\mathit{term}(t_0 + t_1,\ C) &= \mathit{term}(t_0,\ C + t_1) \\
\mathit{term}(t_0 \Rightarrow t_1; t_2,\ C) &= \mathit{term}(t_0,\ C \Rightarrow t_1; t_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}(C + t_1,\ v_0) &= \mathit{term}(t_1,\ v_0 + C) \\
\mathit{context}(v_0 + C,\ v_1) &= \langle v_0 + v_1, C\rangle \\
\mathit{context}(C \Rightarrow t_1; t_2,\ v_0) &= \langle v_0 \Rightarrow t_1; t_2, C\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\
\mathit{decompose}(t) &= \mathit{term}(t,\ [\,])
\end{aligned}$$

This definition is that of the traditional decomposition function of a non-value term into a potential redex and its context, represented as a big-step abstract machine [67]. (Value terms are fixed points.) Furthermore, the machine consists of two mutually recursive functions that respectively dispatch on terms and on reduction contexts. From such a decompose function, as illustrated elsewhere [9, 27, 28, 73, 76, 77, 194, 211, 222], a wide range of other semantic formulations can be derived.

5.2.6 Recomposing a term and a reduction context

The recomposition function is the left fold over reduction contexts. It zips each context constructor to its corresponding term constructor:

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Term} &\to \mathsf{Term} \\
\mathit{recompose}([\,],\ t) &= t \\
\mathit{recompose}(C + t_1,\ t_0) &= \mathit{recompose}(C,\ t_0 + t_1) \\
\mathit{recompose}(v_0 + C,\ t_1) &= \mathit{recompose}(C,\ v_0 + t_1) \\
\mathit{recompose}(C \Rightarrow t_1; t_2,\ t_0) &= \mathit{recompose}(C,\ t_0 \Rightarrow t_1; t_2)
\end{aligned}$$

We can state the following correctness criteria for the relation between decomposition and recomposition:

Property 11 (recomposition is a left inverse of decomposition)**.** For all non-value terms t, $\mathit{decompose}(t) = \langle r, C\rangle \Rightarrow \mathit{recompose}(C,\ r) = t$.

Proof. By mutual induction on the derivations of *term* and *context*. □

Syntax:

$$\begin{aligned} \mathsf{Term} \ni t &::= n \mid b \mid t + t \mid t \Rightarrow t ; t \\ \mathsf{Val} \ni v &::= n \mid b \\ \mathsf{PotRed} \ni r &::= v + v \mid v \Rightarrow t ; t \\ \mathsf{Ctx} \ni C &::= [\,] \mid C + t \mid v + C \mid C \Rightarrow t ; t \end{aligned}$$

Contraction rules:

$$\begin{array}{ll} (\mathit{plus}) & n_0 + n_1 \to n \quad \text{where } n = n_0 + n_1 \\ (\mathit{true}) & \mathsf{true} \Rightarrow t_1 ; t_2 \to t_1 \\ (\mathit{false}) & \mathsf{false} \Rightarrow t_1 ; t_2 \to t_2 \end{array}$$

$$\mathscr{A} = (\mathit{plus}) \cup (\mathit{true}) \cup (\mathit{false})$$

Decomposition:

$$\begin{aligned} \mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\ \mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\ \mathit{term}(b,\ C) &= \mathit{context}(C,\ b) \\ \mathit{term}(t_0 + t_1,\ C) &= \mathit{term}(t_0,\ C + t_1) \\ \mathit{term}(t_0 \Rightarrow t_1 ; t_2,\ C) &= \mathit{term}(t_0,\ C \Rightarrow t_1 ; t_2) \end{aligned}$$

$$\begin{aligned} \mathsf{Ctx} \times \mathsf{Val} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\ \mathit{context}([\,],\ v) &= v \\ \mathit{context}(C + t_1,\ v_0) &= \mathit{term}(t_1,\ v_0 + C) \\ \mathit{context}(v_0 + C,\ v_1) &= \langle v_0 + v_1,\ C \rangle \\ \mathit{context}(C \Rightarrow t_1 ; t_2,\ v_0) &= \langle v_0 \Rightarrow t_1 ; t_2,\ C \rangle \end{aligned}$$

$$\begin{aligned} \mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} \\ \mathit{decompose}(t) &= \mathit{term}(t,\ [\,]) \end{aligned}$$

Recomposition:

$$\begin{aligned} \mathsf{Ctx} \times \mathsf{Term} &\to \mathsf{Term} \\ \mathit{recompose}([\,],\ t) &= t \\ \mathit{recompose}(C + t_1,\ t_0) &= \mathit{recompose}(C,\ t_0 + t_1) \\ \mathit{recompose}(v_0 + C,\ t_1) &= \mathit{recompose}(C,\ v_0 + t_1) \\ \mathit{recompose}(C \Rightarrow t_1 ; t_2,\ t_0) &= \mathit{recompose}(C,\ t_0 \Rightarrow t_1 ; t_2) \end{aligned}$$

One-step reduction:

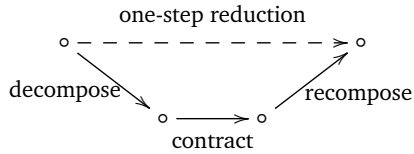
$$t \mapsto_{\mathscr{A}} t' \text{ if } \begin{cases} \mathit{decompose}(t) = \langle r, C \rangle \\ (r, t'') \in \mathscr{A} \\ \mathit{recompose}(C,\ t'') = t' \end{cases}$$

One-step reduction is a partial function since a term may already be a value or the potential redex may not be an actual one.

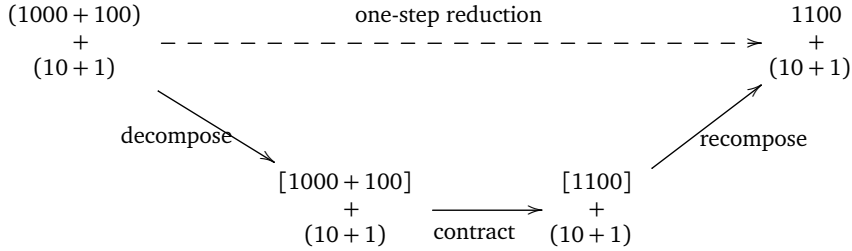
Figure 5.1: A reduction semantics for arithmetic expressions

5.2.7 A reduction semantics

We are now in position to completely state a reduction semantics for the language of arithmetic expressions and conditionals, as displayed in Figure 5.1. The syntax is that of Section 5.2.1 along with the grammar of reduction contexts derived in Section 5.2.4. The contraction rules are unchanged from Section 5.2.1. Decomposition is a tail-recursive function that, given a non-value term, yields the next potential redex and its reduction context. Its definition is that of Section 5.2.5 and was obtained by mechanically transforming the search function implementing the reduction strategy. Recomposition is a tail-recursive function that, given a reduction context and a term, plugs this term into the hole of this reduction context. The definition is that of Section 5.2.6. A single step in the reduction sequence is performed by decomposition, contraction, and recomposition. Diagrammatically:

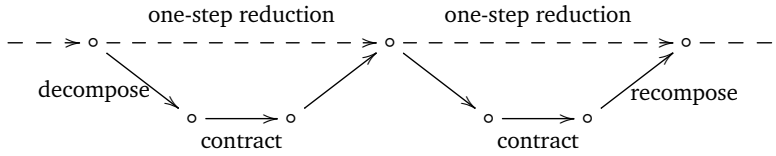


For example, to come back to the reduction sequence of Section 5.2.2, here is its first step:



5.2.8 An abstract machine

The reduction semantics of Section 5.2.7 provides a complete specification of evaluation and can be directly implemented as an evaluator that iterates one-step reduction:



However, such a reduction-based evaluator is needlessly inefficient because it enumerates the standard reduction sequence: for each contraction, the reduction context and contractum are recomposed to create the next reduct, which is subsequently decomposed in

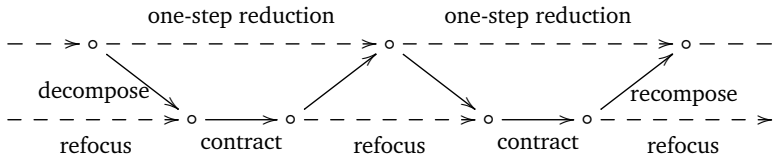
$$
\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\rightharpoonup \mathsf{Val} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(b,\ C) &= \mathit{context}(C,\ b) \\
\mathit{term}(t_0 + t_1,\ C) &= \mathit{term}(t_0,\ C + t_1) \\
\mathit{term}(t_0 \Rightarrow t_1; t_2,\ C) &= \mathit{term}(t_0,\ C \Rightarrow t_1; t_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\rightharpoonup \mathsf{Val} \\
\mathit{context}([\ ],\ v) &= v \\
\mathit{context}(C + t_1,\ v_0) &= \mathit{term}(t_1,\ v_0 + C) \\
\mathit{context}(n_0 + C,\ n_1) &= \mathit{term}(n,\ C) \quad \text{where } n = n_0 + n_1 \\
\mathit{context}(C \Rightarrow t_1; t_2,\ \mathsf{true}) &= \mathit{term}(t_1,\ C) \\
\mathit{context}(C \Rightarrow t_1; t_2,\ \mathsf{false}) &= \mathit{term}(t_2,\ C)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Term} &\rightharpoonup \mathsf{Val} \\
\mathit{eval}(t) &= \mathit{term}(t,\ [\ ])
\end{aligned}
$$

Figure 5.2: An abstract machine for arithmetic expressions

search of the next redex. Danvy and Nielsen have shown how, under circumstances met here, the enumeration of intermediate terms in the reduction sequence can be deforested away by refocusing [71, 197]:



This deforestation is achieved by simply *continuing the decomposition of the contractum in the current context*. The result is the tail-recursive evaluator of Figure 5.2, which takes the form of an abstract machine. We can optimize this machine by hereditarily compressing its "corridor" transitions. For example, the third rule of *context* has a known target and can be compressed to:

$$\mathit{context}(n_0 + C,\ n_1) = \mathit{context}(C,\ n) \quad \text{where } n = n_0 + n_1$$

5.2.9 Summary and conclusion

We have constructed a refocus-ready reduction semantics out of a total search function implementing the reduction strategy for arithmetic expressions. We have then derived the corresponding abstract machine. The following sections instantiate this construction for a variety of notions of computation.

Terminology: In the rest of these notes, we refer to the construction of a refocus-ready reduction semantics as "the prequel." Following Biernacka and Danvy [28], we refer to the subsequent derivation of an abstract machine as "the syntactic correspondence."

5.3 The call-by-value λ-calculus

In this section, we develop a call-by-value reduction semantics for the λ-calculus with integers. This semantics will serve as a starting point for the extensions treated in later sections.

5.3.1 Syntax and contraction

The call-by-value formulation reads as follows:

Definition 12 (λ-calculus for call-by-value reduction)**.**
Syntax:

$$\begin{aligned} \mathsf{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid n \mid \mathit{succ}\ t \\ \mathsf{Val} \ni v &::= \lambda x.t \mid n \\ \mathsf{PotRed} \ni r &::= v\ v \mid \mathit{succ}\ v \end{aligned}$$

Contraction rules:

$$\begin{array}{lll} (\beta_v) & (\lambda x.t)\ v \to t[v/x] & \\ (\mathit{succ}) & \mathit{succ}\ n \to n' & \text{where } n' = n+1 \end{array}$$

$$\mathscr{V} = (\beta_v) \cup (\mathit{succ})$$

In words: Terms are λ-terms with integers, and the successor function. There are two disjoint restrictions on terms: values and potential redexes. Values are λ-abstractions and integers. There are two forms of potential redexes: the application of two values or the application of the successor function to a value. To be an actual redex, the first value of an application must be a λ-abstraction, and the argument of the successor function must be an integer value; otherwise the corresponding term is stuck. We let $\mathscr{V}$ be the notion of call-by-value reduction on λ-terms.

5.3.2 Reduction strategy

For call by value, we consider the left-most inner-most reduction strategy without reducing under λ-abstractions. Let us spell out the search on terms as a function that maps a value term to itself and a non-value term to a potential redex or to an identifier that occurs free in the term. The search is carried out with a straight recursive descent on terms following the reduction strategy. Integers are mapped to integers; identifiers to identifiers; λ-abstractions to λ-abstractions; the left sub-term of an application is first traversed, and if it turns out to be a value, the right sub-term is then traversed; and the argument sub-term

of the successor function is traversed:

$$
\begin{array}{rl}
\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed} + \mathsf{Ide} & \\
\mathit{term}(n) = & n \\
\mathit{term}(x) = & x \\
\mathit{term}(\lambda x.t) = & \lambda x.t \\
\mathit{term}(t_0\ t_1) = & \mathit{case}\ \mathit{term}(t_0)\ \mathit{of} \\
& \quad \mid v_0 \to \mathit{case}\ \mathit{term}(t_1)\ \mathit{of} \\
& \qquad \mid v_1 \to v_0\ v_1 \\
& \qquad \mid r \to r \\
& \qquad \mid x \to x \\
& \quad \mid r \to r \\
& \quad \mid x \to x \\
\mathit{term}(\mathit{succ}\ t_1) = & \mathit{case}\ \mathit{term}(t_1)\ \mathit{of} \\
& \quad \mid v_1 \to \mathit{succ}\ v_1 \\
& \quad \mid r \to r \\
& \quad \mid x \to x
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed} + \mathsf{Err} & \\
\mathit{search}(t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \quad \mid v \to v \\
& \quad \mid r \to r \\
& \quad \mid x \to \mathsf{error}\ \text{“free identifier } x\text{”}
\end{array}
$$

Here the search function can map to a value, a potential redex, or an explicit error. If the result is an error or the potential redex is not an actual redex, the term is stuck.

5.3.3 A reduction semantics

Applying the prequel to the search function of Section 5.3.2 yields a grammar of reduction contexts and a decomposition function. We are then in position to state a reduction semantics for the language in complete detail, as displayed in Figure 5.3.

5.3.4 An abstract machine

Applying the syntactic correspondence (i.e., refocusing and transition compression) to the reduction semantics of Figure 5.3 yields the abstract machine displayed in Figure 5.4. The cognoscenti will recognize the classical CK machine [89]. (Had our initial calculus used explicit substitutions, the result would be an environment machine: the CEK machine [28, 87, 176].)

5.3.5 Summary and conclusion

We have constructed a refocus-ready reduction semantics out of a total search function implementing a call-by-value reduction strategy for the λ-calculus. We have then derived the corresponding abstract machine.

Syntax:

$$\begin{aligned}
\mathsf{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid n \mid \mathit{succ}\ t \\
\mathsf{Val} \ni v &::= \lambda x.t \mid n \\
\mathsf{PotRed} \ni r &::= v\ v \mid \mathit{succ}\ v \\
\mathsf{Ctx} \ni C &::= [\,] \mid C\ t \mid v\ C \mid \mathit{succ}\ C
\end{aligned}$$

Contraction rules:

$$\begin{array}{ll}
(\beta_v) & (\lambda x.t)\ v \to t[v/x] \\
(\mathit{succ}) & \mathit{succ}\ n \to n' \qquad \text{where } n' = n+1
\end{array}$$

$$\mathscr{V} = (\beta_v) \cup (\mathit{succ})$$

Decomposition:

$$\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(x,\ C) &= \text{error ``free identifier } x\text{''} \\
\mathit{term}(\lambda x.t,\ C) &= \mathit{context}(C,\ \lambda x.t) \\
\mathit{term}(t_0\ t_1,\ C) &= \mathit{term}(t_0,\ C\ t_1) \\
\mathit{term}(\mathit{succ}\ t_1,\ C) &= \mathit{term}(t_1,\ \mathit{succ}\ C)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}(C\ t_1,\ v_0) &= \mathit{term}(t_1,\ v_0\ C) \\
\mathit{context}(v_0\ C,\ v_1) &= \langle v_0\ v_1, C\rangle \\
\mathit{context}(\mathit{succ}\ C,\ v_1) &= \langle \mathit{succ}\ v_1, C\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{decompose}(t) &= \mathit{term}(t,\ [\,])
\end{aligned}$$

Recomposition:

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Term} &\to \mathsf{Term} \\
\mathit{recompose}([\,],\ t) &= t \\
\mathit{recompose}(C\ t_1,\ t_0) &= \mathit{recompose}(C,\ t_0\ t_1) \\
\mathit{recompose}(v_0\ C,\ t_1) &= \mathit{recompose}(C,\ v_0\ t_1) \\
\mathit{recompose}(\mathit{succ}\ C,\ t_1) &= \mathit{recompose}(C,\ \mathit{succ}\ t_1)
\end{aligned}$$

One-step reduction:

$$t \mapsto_{\mathscr{V}} t' \text{ if } \begin{cases} \mathit{decompose}(t) = \langle r, C\rangle \\ (r, t'') \in \mathscr{V} \\ \mathit{recompose}(C,\ t'') = t' \end{cases}$$

One-step reduction is a partial function since a term may already be a value, decomposition may yield an error, or the potential redex may not be an actual one.

Figure 5.3: A call-by-value reduction semantics for the λ-calculus with integers

$$
\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\rightharpoonup \mathsf{Val} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(x,\ C) &= \mathsf{error}\ \text{“free identifier } x\text{”} \\
\mathit{term}(\lambda x.t,\ C) &= \mathit{context}(C,\ \lambda x.t) \\
\mathit{term}(t_0\ t_1,\ C) &= \mathit{term}(t_0,\ C\ t_1) \\
\mathit{term}(\mathit{succ}\ t_1,\ C) &= \mathit{term}(t_1,\ \mathit{succ}\ C)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\rightharpoonup \mathsf{Val} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}(C\ t_1,\ v_0) &= \mathit{term}(t_1,\ v_0\ C) \\
\mathit{context}((\lambda x.t)\ C,\ v_1) &= \mathit{term}(t[v_1/x],\ C) \\
\mathit{context}(\mathit{succ}\ C,\ n_1) &= \mathit{context}(C,\ n_1') \\
&\quad \text{where } n_1' = n_1 + 1
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Term} &\rightharpoonup \mathsf{Val} \\
\mathit{eval}(t) &= \mathit{term}(t,\ [\,])
\end{aligned}
$$

Figure 5.4: A call-by-value abstract machine for the λ-calculus

5.4 The call-by-value λ-calculus with exceptions and context-sensitive contraction rules

In this section, we extend the λ-calculus of Section 5.3 with exceptions using context-sensitive contraction rules. When specifying these rules, we leave the definition of contexts abstract: as in Sections 5.2 and 5.3, their concrete definition will arise as the data type of the defunctionalized continuation of the function searching for a redex according to the reduction strategy.

5.4.1 Syntax and contraction

The context-sensitive extension reads as follows:

Definition 13 (context-sensitive λ-calculus with exceptions)**.**
Syntax (extending the syntax of Definition 12):

$$
\begin{aligned}
\mathsf{Exc} &\ni e \\
\mathsf{Term} &\ni t ::= \ldots \mid \mathsf{handle}\ (t,\ e,\ t) \mid \mathsf{raise}\ e\ t \\
\mathsf{PotRed} &\ni r ::= \ldots \mid \mathsf{handle}\ (v,\ e,\ t) \mid \mathsf{raise}\ e\ v
\end{aligned}
$$

Contraction rules (extending the rules of Definition 12):

$$
\begin{aligned}
(\mathit{handle}) \quad & C[\mathsf{handle}\ (v,\ e,\ t)] \to C[v] \\
(\mathit{raise}) \quad & C[\mathsf{raise}\ e\ v] \to C'[t\ v] \\
& \qquad \text{where } C = C'[\mathsf{handle}\ (C'',\ e,\ t)] \\
& \qquad \text{and } e \text{ is not declared in } C''
\end{aligned}
$$

$$
\mathscr{E}_s = \mathscr{V} \cup (\mathit{handle}) \cup (\mathit{raise})
$$

In words: The definition of terms is extended with two forms: a handle expression and a raise exception.

- A handle expression $\mathsf{handle}\ (t,\ e,\ t')$ has three sub-forms: a body t, an exception e, and a continuation t'. When a handle expression is evaluated, its body t is first evaluated. If no exception is raised in the course of evaluating t, the resulting value, v, is also the result of evaluating the handle expression. If an exception e is raised with a value v in the course of evaluating t, the continuation t' is applied to v in the context of the handle expression.

- A raise expression $\mathsf{raise}\ e\ t$ has two sub-forms: an exception e and an argument t. When a raise expression is evaluated, its argument t is first evaluated. If this evaluation yields a value v and the context of the raise expression contains a matching handler, then the exception e is said to be raised with the value v and it is handled by the nearest matching handler as described just above. If there is no matching handler in the context, the term is stuck.

Values remain the same. For each new form, there is a new potential redex and a new context-sensitive contraction rule: a handle expression with a value (i.e., where no exception was raised in the course of the reduction of its body) contracts to this value in the same context; and a raise expression with a value locates the nearest matching handler in the current context and contracts to the application of the handler's continuation to the raised value in the context of the corresponding handle expression. In so doing, the delimited context up to the point of the handler expression, i.e., C'', is elided. We let $\mathscr{E}_s$ be the notion of context-sensitive reduction on λ-terms with exceptions.

5.4.2 Reduction strategy

For this extension of call by value with exceptions, we consider the left-most inner-most reduction strategy where handler continuations are treated lazily, the way consequents and alternatives were treated in conditional expressions, in Section 5.2. Let us spell out the search on terms as a function that maps a value term to itself and a non-value term to a potential redex or to an identifier that occurs free in the term. The search is carried out with a straight recursive descent on terms following the reduction strategy. For brevity, instead of specifying the search function in full, we present it as an extension of the search function from Section 5.3.2. Both the body of a handle expression and the argument of a raise expression are traversed:

$$
\begin{aligned}
&\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed} + \mathsf{Ide}\\
&\quad\cdots\\
\mathit{term}(\mathsf{handle}\ (t,\ e,\ t')) &= \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\
&\qquad \mid v \to \mathsf{handle}\ (v,\ e,\ t')\\
&\qquad \mid r \to r\\
&\qquad \mid x \to x\\
\mathit{term}(\mathsf{raise}\ e\ t) &= \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\
&\qquad \mid v \to \mathsf{raise}\ e\ v\\
&\qquad \mid r \to r\\
&\qquad \mid x \to x
\end{aligned}
$$

$$
\begin{array}{rl}
\mathsf{Term} \to \mathsf{Val} + \mathsf{PotRed} + \mathsf{Err} & \\
\mathit{search}(t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid v \to v \\
& \mid r \to r \\
& \mid x \to \mathsf{error}\ \text{“free identifier } x\text{”}
\end{array}
$$

5.4.3 A reduction semantics

Applying the prequel to the call-by-value search function of Section 5.4.2 yields a grammar of reduction contexts and a decomposition function. We are then in position to state a reduction semantics for the language in complete detail, as displayed in Figure 5.5.

5.4.4 An abstract machine

Applying the syntactic correspondence (i.e., refocusing and transition compression) to the reduction semantics of Figure 5.5 yields an abstract machine. This machine is the classical CK machine extended with exceptions.

5.4.5 Summary and conclusion

We have constructed a refocus-ready reduction semantics out of a total search function implementing the call-by-value reduction strategy for the λ-calculus with exceptions. We have then derived the corresponding abstract machine.

This section illustrated how to apply the prequel in the presence of a control operator specified with context-sensitive contraction rules.

5.5 The call-by-value λ-calculus with exceptions and context-insensitive contraction rules

In this section, we incrementally extend the call-by-value λ-calculus of Section 5.3 with exceptions using context-insensitive contraction rules instead of context-sensitive rules, as in Section 5.4. Again, in Section 5.5.1, as in Section 5.4.1, we leave the definition of contexts abstract: their concrete definition will arise as the data type of the defunctionalized continuation of the function searching for a redex according to the reduction strategy.

5.5.1 Syntax and contraction

The context-insensitive extension reads as follows:

Definition 14 (context-insensitive λ-calculus with exceptions)**.**
Syntax (extending the syntax of Definition 12):

$$
\begin{array}{rl}
\mathsf{Exc} \ni e & \\
\mathsf{Term} \ni t & ::= \ldots \mid \mathsf{handle}\ (t,\ e,\ t) \mid \mathsf{raise}\ e\ t \\
\mathsf{PotRed} \ni r & ::= \ldots \mid \mathsf{handle}\ (v,\ e,\ t) \mid \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t)
\end{array}
$$

Syntax:

$$\begin{aligned}
\mathsf{Exc} &\ni e \\
\mathsf{Term} &\ni t ::= x \mid \lambda x.t \mid t\ t \mid n \mid \mathit{succ}\ t \mid \mathsf{handle}\ (t,\ e,\ t) \mid \mathsf{raise}\ e\ t \\
\mathsf{Val} &\ni v ::= \lambda x.t \mid n \\
\mathsf{PotRed} &\ni r ::= v\ v \mid \mathit{succ}\ v \mid \mathsf{handle}\ (v,\ e,\ t) \mid \mathsf{raise}\ e\ v \\
\mathsf{Ctx} &\ni C ::= [\,] \mid C\ t \mid v\ C \mid \mathit{succ}\ C \mid \mathsf{handle}\ (C,\ e,\ t) \mid \mathsf{raise}\ e\ C
\end{aligned}$$

Contraction rules:

$$\begin{array}{lrll}
(\beta_v) & C[(\lambda x.t)\ v] & \to C[t[x/v]] & \\
(\mathit{succ}) & C[\mathit{succ}\ n] & \to C[n'] & \text{where } n' = n+1 \\
(\mathit{handle}) & C[\mathsf{handle}\ (v,\ e,\ t)] & \to C[v] & \\
(\mathit{raise}) & C[\mathsf{raise}\ e\ v] & \to C'[t\ v] & \text{where } C = C'[\mathsf{handle}\ (C'',\ e,\ t)] \\
 & & & \text{and } e \text{ is not declared in } C''
\end{array}$$

$$\mathscr{E}_s = (\beta_v) \cup (\mathit{succ}) \cup (\mathit{handle}) \cup (\mathit{raise})$$

Decomposition:

$$\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(x,\ C) &= \mathsf{error}\ \text{“free identifier } x\text{”} \\
\mathit{term}(\lambda x.t,\ C) &= \mathit{context}(C,\ \lambda x.t) \\
\mathit{term}(t_0\ t_1,\ C) &= \mathit{term}(t_0,\ C\ t_1) \\
\mathit{term}(\mathit{succ}\ t_1,\ C) &= \mathit{term}(t_1,\ \mathit{succ}\ C) \\
\mathit{term}(\mathsf{handle}\ (t,\ e,\ t'),\ C) &= \mathit{term}(t,\ \mathsf{handle}\ (C,\ e,\ t')) \\
\mathit{term}(\mathsf{raise}\ e\ t,\ C) &= \mathit{term}(t,\ \mathsf{raise}\ e\ C)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Val} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{context}([\,],\ v) &= v \\
\mathit{context}(C\ t_1,\ v_0) &= \mathit{term}(t_1,\ v_0\ C) \\
\mathit{context}(v_0\ C,\ v_1) &= \langle v_0\ v_1,\ C\rangle \\
\mathit{context}(\mathit{succ}\ C,\ v_1) &= \langle \mathit{succ}\ v_1,\ C\rangle \\
\mathit{context}(\mathsf{handle}\ (C,\ e,\ t'),\ v) &= \langle \mathsf{handle}\ (v,\ e,\ t'),\ C\rangle \\
\mathit{context}(\mathsf{raise}\ e\ C,\ v) &= \langle \mathsf{raise}\ e\ v,\ C\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{decompose}(t) &= \mathit{term}(t,\ [\,])
\end{aligned}$$

Recomposition:

$$\begin{aligned}
\mathsf{Ctx} \times \mathsf{Term} &\to \mathsf{Term} \\
\mathit{recompose}([\,],\ t) &= t \\
\mathit{recompose}(C\ t_1,\ t_0) &= \mathit{recompose}(C,\ t_0\ t_1) \\
\mathit{recompose}(v_0\ C,\ t_1) &= \mathit{recompose}(C,\ v_0\ t_1) \\
\mathit{recompose}(\mathit{succ}\ C,\ t_1) &= \mathit{recompose}(C,\ \mathit{succ}\ t_1) \\
\mathit{recompose}(\mathsf{handle}\ (C,\ e,\ t'),\ t) &= \mathit{recompose}(C,\ \mathsf{handle}\ (t,\ e,\ t')) \\
\mathit{recompose}(\mathsf{raise}\ e\ C,\ t) &= \mathit{recompose}(C,\ \mathsf{raise}\ e\ t)
\end{aligned}$$

One-step reduction:

$$t \mapsto_{\mathscr{E}_s} t' \ \text{ if } \ \mathit{decompose}(t) = \langle r,\ C\rangle \ \wedge\ (C[r], C'[t'']) \in \mathscr{E}_s \ \wedge\ \mathit{recompose}(C',\ t'') = t'$$

Figure 5.5: A call-by-value reduction semantics for the λ-calculus with exceptions and context-sensitive contraction rules

Contraction rules (extending the rules of Definition 12):

$$\begin{array}{ll} (\mathit{handle}) & \mathsf{handle}\ (v,\ e,\ t) \to v \\ (\mathit{raise}) & \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t) \to t\ v \quad \text{where } e \text{ is not declared in } C \end{array}$$

$$\mathscr{E}_i = \mathscr{V} \cup (\mathit{handle}) \cup (\mathit{raise})$$

In words: The syntax is extended as in Section 5.4, i.e., with a handle expression, $\mathsf{handle}\,(t,\ e,\ t')$, and a raise expression, $\mathsf{raise}\ e\ t$. Values remain the same. For each new form, there is a new potential redex and a new context-insensitive contraction rule: a handle expression with a value (i.e., where no exception was raised in the course of the reduction of its body) contracts to this value; and a raise expression with a value occurring in the nearest body of a matching handle expression contracts to the application of the handler's continuation to the raised value. In so doing, the delimited context up to the point of the handler expression is elided. Here the abstract delimited context surrounding the raise expression should satisfy some conditions: the raise expression should be uniquely determined and the context should not contain any handle expressions for the same exception. Both of these properties will follow from the specification of the search function. We let $\mathscr{E}_i$ be the notion of context-insensitive reduction on λ-terms with exceptions.

5.5.2 Reduction strategy

For this extension of call by value with exceptions, we consider the left-most inner-most reduction strategy where handler continuations are treated lazily. Let us spell out the search on terms as a function that maps a value term to itself and a non-value term to a potential redex, an identifier that occurs free in the term, or a raise expression and its (abstract) context. We denote a raise expression and its context as:

$$\mathsf{Raise} = (\mathsf{Term} \to \mathsf{Term}) \times \mathsf{Exc} \times \mathsf{Val} \ni C[\mathsf{raise}\ e\ v]$$

The search is carried out with a straight recursive descent on terms following the same reduction strategy as the call-by-value search of Section 5.3.2:

$$\begin{array}{rl} \mathsf{Term} \to & \mathsf{Val} + \mathsf{PotRed} + \mathsf{Ide} + \mathsf{Raise} \\ \mathit{term}(x) = & x \\ \mathit{term}(v) = & v \\ \mathit{term}(t_0\ t_1) = & \mathit{case}\ \mathit{term}(t_0)\ \mathit{of} \\ & \mid v_0 \to \mathit{case}\ \mathit{term}(t_1)\ \mathit{of} \\ & \qquad \mid v_1 \to v_0\ v_1 \\ & \qquad \mid r \to r \\ & \qquad \mid x \to x \\ & \qquad \mid C[\mathsf{raise}\ e\ v] \to (v_0\ C)[\mathsf{raise}\ e\ v] \\ & \mid r \to r \\ & \mid x \to x \\ & \mid C[\mathsf{raise}\ e\ v] \to (C\ t_1)[\mathsf{raise}\ e\ v] \\ \mathit{term}(\mathit{succ}\ t_1) = & \mathit{case}\ \mathit{term}(t_1)\ \mathit{of} \\ & \mid v_1 \to \mathit{succ}\ v_1 \\ & \mid r \to r \\ & \mid x \to x \\ & \mid C[\mathsf{raise}\ e\ v] \to (\mathit{succ}\ C)[\mathsf{raise}\ e\ v] \end{array}$$

Both the body of a handle expression and the argument of a raise expression are traversed:

$$
\begin{array}{rl}
\mathit{term}(\mathsf{handle}\ (t,\ e,\ t')) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid v \rightarrow \mathsf{handle}\ (v,\ e,\ t') \\
& \mid r \rightarrow r \\
& \mid x \rightarrow x \\
& \mid C[\mathsf{raise}\ e\ v] \rightarrow \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t') \\
& \mid C[\mathsf{raise}\ e'\ v] \rightarrow (\mathsf{handle}\ (C,\ e,\ t'))[\mathsf{raise}\ e'\ v] \\
& \qquad \text{where } e \neq e' \\
\mathit{term}(\mathsf{raise}\ e\ t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid v \rightarrow [\ ][\mathsf{raise}\ e\ v] \\
& \mid r \rightarrow r \\
& \mid x \rightarrow x \\
& \mid C[\mathsf{raise}\ e'\ v] \rightarrow (\mathsf{raise}\ e\ C)[\mathsf{raise}\ e'\ v]
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{Term} \rightarrow & \mathsf{Val} + \mathsf{PotRed} + \mathsf{Err} \\
\mathit{search}(t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid v \rightarrow v \\
& \mid r \rightarrow r \\
& \mid x \rightarrow \text{error “free identifier } x\text{”} \\
& \mid C[\mathsf{raise}\ e\ v] \rightarrow \text{error “unhandled exception } e\text{”}
\end{array}
$$

Delimited contexts The search function for the context-insensitive formulation is more complex than for the context-sensitive formulation. This is because the side condition that was formerly about the context surrounding a redex (the (*raise*) rule in Definition 13) is now stated on the delimited context that is part of the redex. Thus, the search function must satisfy this condition as opposed to the contraction function. Just as for the grammar of evaluation contexts, this abstract delimited context is made concrete by the search function. We defunctionalize the function-space component of Raise to obtain the grammar of “raise” contexts:

$$\mathsf{RaiseCtx} \ni R ::= [\ ] \mid R\ t \mid v\ R \mid \mathit{succ}\ R \mid \mathsf{handle}\ (R,\ e,\ t) \mid \mathsf{raise}\ e\ R$$

and an application function that consumes each production in the grammar:

$$
\begin{array}{rl}
\mathsf{RaiseCtx} \times \mathsf{Term} \rightarrow & \mathsf{Term} \\
\mathit{apply}([\ ],\ t) = & t \\
\mathit{apply}(R\ t_1,\ t_0) = & \mathit{apply}(R,\ t_0)\ t_1 \\
\mathit{apply}(v_0\ R,\ t_1) = & v_0\ \mathit{apply}(R,\ t_1) \\
\mathit{apply}(\mathit{succ}\ R,\ t_1) = & \mathit{succ}\ \mathit{apply}(R,\ t_1) \\
\mathit{apply}(\mathsf{handle}\ (R,\ e,\ t'),\ t) = & \mathsf{handle}\ (\mathit{apply}(R,\ t),\ e,\ t') \\
\mathit{apply}(\mathsf{raise}\ e\ R,\ t) = & \mathsf{raise}\ e\ \mathit{apply}(R,\ t)
\end{array}
$$

Because a raise context is constructed at return time, i.e., from the inside out, the resulting raise context is outside in. Therefore, the apply function is the right-fold over raise contexts. It maps each raise context to its corresponding term constructor. Finally, it follows from the definition of search that the raise context is uniquely determined and that no handle context is created that binds a raised exception.

Syntax:

$$
\begin{array}{rl}
\mathsf{Exc} \ni & e \\
\mathsf{Term} \ni & t ::= x \mid \lambda x.t \mid t\ t \mid n \mid \mathit{succ}\ t \mid \mathsf{handle}\ (t,\ e,\ t) \mid \mathsf{raise}\ e\ t \\
\mathsf{Val} \ni & v ::= \lambda x.t \mid n \\
\mathsf{PotRed} \ni & r ::= v\ v \mid \mathit{succ}\ v \mid \mathsf{handle}\ (v,\ e,\ t) \mid \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t) \\
\mathsf{Ctx} \ni & C ::= [\,] \mid C\ t \mid v\ C \mid \mathit{succ}\ C \mid \mathsf{handle}\ (C,\ e,\ t) \mid \mathsf{raise}\ e\ C \\
\mathsf{Raise} = & (\mathsf{Term} \to \mathsf{Term}) \times \mathsf{Exc} \times \mathsf{Val} \ni C[\mathsf{raise}\ e\ v]
\end{array}
$$

Contraction rules:

$$
\begin{array}{lrll}
(\beta_v) & (\lambda x.t)\ v & \to & t[x/v] \\
(\mathit{succ}) & \mathit{succ}\ n & \to & n' \qquad \text{where } n' = n+1 \\
(\mathit{handle}) & \mathsf{handle}\ (v,\ e,\ t) & \to & v \\
(\mathit{raise}) & \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t) & \to & t\ v
\end{array}
$$

$$\mathscr{E}_i = (\beta_v) \cup (\mathit{succ}) \cup (\mathit{handle}) \cup (\mathit{raise})$$

Figure 5.6: A call-by-value reduction semantics for the λ-calculus with exceptions and context-insensitive contraction rules: syntax and contraction rules

5.5.3 A reduction semantics

Applying the prequel to the call-by-value search function of Section 5.5.2 yields a grammar of reduction contexts and a decomposition function. We are then in position to state a reduction semantics for the language in complete detail, as displayed in Figure 5.6 and Figure 5.7.

NB. Note how the context rules for the raise summand implement the search for a matching exception handler, much as we expect done to implement the side condition of contraction in the context-sensitive specification.

5.5.4 An abstract machine

Applying the syntactic correspondence (i.e., refocusing and transition compression) to the reduction semantics of Figure 5.6 and 5.7 yields an abstract machine. This abstract machine result is the *same* one as the one obtained in Section 5.4.4. In other words, the context-sensitive specification and the context-insensitive specification truly specify the same operational behavior.

5.5.5 Summary and conclusion

We have constructed a refocus-ready reduction semantics out of a total search function implementing the call-by-value reduction strategy for the λ-calculus with exceptions. We have then derived the corresponding abstract machine.

This section illustrated how to apply the prequel in the presence of a control operator using context-insensitive contraction rules.

Decomposition:

$$
\begin{aligned}
\mathsf{Term} \times \mathsf{Ctx} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{term}(n,\ C) &= \mathit{context}(C,\ n) \\
\mathit{term}(x,\ C) &= \text{error “free identifier } x\text{”} \\
\mathit{term}(\lambda x.t,\ C) &= \mathit{context}(C,\ \lambda x.t) \\
\mathit{term}(t_0\ t_1,\ C) &= \mathit{term}(t_0,\ C\ t_1) \\
\mathit{term}(\mathit{succ}\ t_1,\ C) &= \mathit{term}(t_1,\ \mathit{succ}\ C) \\
\mathit{term}(\mathsf{handle}\ (t,\ e,\ t'),\ C) &= \mathit{term}(t,\ \mathsf{handle}\ (C,\ e,\ t')) \\
\mathit{term}(\mathsf{raise}\ e\ t,\ C) &= \mathit{term}(t,\ \mathsf{raise}\ e\ C)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Ctx} \times (\mathsf{Val} + \mathsf{Raise}) &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{context}([\ ],\ v) &= v \\
\mathit{context}([\ ],\ C[\mathsf{raise}\ e\ v]) &= \text{error “unhandled exception } e\text{”} \\
\mathit{context}(C\ t_1,\ v_0) &= \mathit{term}(t_1,\ v_0\ C) \\
\mathit{context}(C\ t_1,\ C[\mathsf{raise}\ e\ v]) &= \mathit{context}(C,\ (C\ t_1)[\mathsf{raise}\ e\ v]) \\
\mathit{context}(v_0\ C,\ v_1) &= \langle v_0\ v_1,\ C\rangle \\
\mathit{context}(v_0\ C,\ C[\mathsf{raise}\ e\ v]) &= \mathit{context}(C,\ (v_0\ C)[\mathsf{raise}\ e\ v]) \\
\mathit{context}(\mathit{succ}\ C,\ v_1) &= \langle \mathit{succ}\ v_1,\ C\rangle \\
\mathit{context}(\mathit{succ}\ C,\ C[\mathsf{raise}\ e\ v]) &= \mathit{context}(C,\ (\mathit{succ}\ C)[\mathsf{raise}\ e\ v]) \\
\mathit{context}(\mathsf{handle}\ (C,\ e,\ t'),\ v) &= \langle \mathsf{handle}\ (v,\ e,\ t'),\ C\rangle \\
\mathit{context}(\mathsf{handle}\ (C,\ e,\ t'),\ C[\mathsf{raise}\ e\ v]) &= \langle \mathsf{handle}\ (C[\mathsf{raise}\ e\ v],\ e,\ t'),\ C\rangle \\
\mathit{context}(\mathsf{handle}\ (C,\ e,\ t'),\ C[\mathsf{raise}\ e'\ v]) &= \mathit{context}(C,\ (\mathsf{handle}\ (C,\ e,\ t'))[\mathsf{raise}\ e'\ v]) \\
&\quad \text{where } e \neq e' \\
\mathit{context}(\mathsf{raise}\ e\ C,\ v) &= \mathit{context}(C,\ [\ ][\mathsf{raise}\ e\ v]) \\
\mathit{context}(\mathsf{raise}\ e\ C,\ C[\mathsf{raise}\ e'\ v]) &= \mathit{context}(C,\ (\mathsf{raise}\ e\ C)[\mathsf{raise}\ e'\ v])
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Term} &\to \mathsf{Val} + \mathsf{PotRed} \times \mathsf{Ctx} + \mathsf{Err} \\
\mathit{decompose}(t) &= \mathit{term}(t,\ [\ ])
\end{aligned}
$$

Recomposition: As defined in Figure 5.5.

One-step reduction:

$$t \mapsto_{\mathscr{E}_s} t' \ \text{ if } \ \mathit{decompose}(t) = \langle r, C\rangle \ \wedge\ (r, t'') \in \mathscr{E}_s \ \wedge\ \mathit{recompose}(C,\ t'') = t'$$

Figure 5.7: A call-by-value reduction semantics for the λ-calculus with exceptions and context-insensitive contraction rules: decomposition and recomposition

5.6 Case study: JavaScript

In this section, we develop a reduction semantics for a full programming language, namely JavaScript. We start by applying the method to derive a complete specification of Guha et al.'s reduction semantics for λ_{JS} [104]. We then look at extending the language with new control operators in the form of coroutines based on the asynchronous coroutines found in the Lua programming language [79]. For space reasons, we display only the parts of λ_{JS} concerning exceptions, breaks and state, which we find of most interest to the derivation.

5.6.1 Syntax and contraction

The semantics of λ_{JS} makes use of both context-sensitive rules and context-insensitive rules which contain delimited contexts. The former arise from the global store while the later are used in the specification of control operators. Our partial formulation of λ_{JS}, renaming non-terminals for notational uniformity, reads as follows:

Definition 15 (λ_{JS} *(full specification in [104])*)**.**
Syntax (exceptions, breaks and state):

$$
\begin{array}{rl}
\mathsf{Label} \ni p & \\
\mathsf{Location} \ni \ell & \\
\mathsf{Store} \ni \sigma & ::= \varepsilon \mid \sigma[\ell \mapsto v] \\
\mathsf{Val} \ni v & ::= \ell \mid \cdots \\
\mathsf{Term} \ni t & ::= v \mid \cdots \mid \\
 & \quad \mathsf{try}\ \{\ t\ \}\ \mathsf{catch}(x)\ \{\ t\ \} \mid \mathsf{throw}\ t \mid \mathsf{err}\ v \mid \\
 & \quad p : t \mid \mathsf{break}\ p\ t \mid \\
 & \quad \mathsf{ref}\ t \mid \mathsf{deref}\ t \mid t = t \\
\mathsf{PotRed} \ni r & ::= \mathsf{throw}\ v \mid \\
 & \quad \mathsf{try}\ \{\ v\ \}\ \mathsf{catch}(x)\ \{\ t\ \} \mid \\
 & \quad \mathsf{try}\ \{\ C[\mathsf{err}\ v]\ \}\ \mathsf{catch}(v)\ \{\ t\ \} \mid \\
 & \quad p : v \mid p : C[\mathsf{break}\ p\ v] \mid \\
 & \quad \mathsf{ref}\ v \mid \mathsf{deref}\ v \mid v = v
\end{array}
$$

Contraction rules (context-insensitive):

$$
\begin{array}{ll}
 & \cdots \\
(\textit{throw}) & \mathsf{throw}\ v \hookrightarrow \mathsf{err}\ v \\
(\textit{catch-pop}) & \mathsf{try}\ \{\ v\ \}\ \mathsf{catch}(x)\ \{\ t\ \} \hookrightarrow v \\
(\textit{catch}) & \mathsf{try}\ \{\ C[\mathsf{err}\ v]\ \}\ \mathsf{catch}(x)\ \{\ t\ \} \hookrightarrow t[v/x] \\
(\textit{label-pop}) & p : v \hookrightarrow v \\
(\textit{break}) & p : C[\mathsf{break}\ p\ v] \hookrightarrow v \\
(\textit{break-pop}) & p : C[\mathsf{break}\ p'\ v] \hookrightarrow \mathsf{break}\ p'\ v \quad \text{where } p \neq p'
\end{array}
$$

Contraction rules (context-sensitive):

$$
\begin{array}{lll}
(\textit{insen}) & \langle \sigma, C[t] \rangle \to \langle \sigma, C[t'] \rangle & \text{where } t \hookrightarrow t' \\
(\textit{ref}) & \langle \sigma, C[\mathsf{ref}\ v] \rangle \to \langle \sigma[\ell \mapsto v], C[\ell] \rangle & \text{where } \ell \notin \mathit{dom}(\sigma) \\
(\textit{deref}) & \langle \sigma, C[\mathsf{deref}\ \ell] \rangle \to \langle \sigma, C[\sigma(\ell)] \rangle & \\
(\textit{set-ref}) & \langle \sigma, C[\ell = v] \rangle \to \langle \sigma[\ell \mapsto v], C[v] \rangle & \text{where } \ell \in \mathit{dom}(\sigma)
\end{array}
$$

In words: We have a notion of labels, locations and a store. Values include labels and locations and for brevity we omit the remaining constructions. Terms include values and for brevity we display only three groups of syntactic constructs: for exceptions, for breaks and for state. For exceptions, a try expression evaluates a body catching any signaled exceptions, a throw expression signals an exception, and an error expression represents a signaled exception created by a throw expression. For breaks, a label expression creates a break point, a break expression transfers control to the label with the value of its body. For state, a reference expression creates a new mutable cell, a dereference expression reads the contents of a mutable cell, and an assignment expression updates the contents of a mutable cell.

For consistency with Guha et al.'s presentation we have stated the context-insensitive rules and the context-sensitive rules separately. For our purposes the context-insensitive rules could just as well be inlined in the context-sensitive rules.

5.6.2 Reduction strategy

For λ_{JS} we consider the usual strict reduction strategy where the continuations of try expressions are treated lazily. Let us spell out the search on terms as a function that maps a value term to itself and a non-value term to one of: a potential redex; a variable occurrence; an error expression and its context; or a break expression and its context. Notationally we denote the latter two as:

$$\begin{aligned}\mathsf{Exc} &= (\mathsf{Term} \to \mathsf{Term}) \times \mathsf{Val} \ni C[\mathsf{err}\ v]\\ \mathsf{Break} &= (\mathsf{Term} \to \mathsf{Term}) \times \mathsf{Label} \times \mathsf{Val} \ni C[\mathsf{break}\ p\ v]\end{aligned}$$

The search is carried out with a straight recursive descent on terms where we display just the the cases for the presented syntax:

$$\begin{aligned}\mathsf{Term} &\to \mathsf{PotRed} + \mathsf{Val} + \mathsf{Exc} + \mathsf{Break}\\ \mathit{term}(\ell) &= \ell\\ \mathit{term}(\mathsf{try}\ \{\ t_1\ \}\ \mathsf{catch}(x)\ \{\ t_2\ \}) &= \mathit{case}\ \mathit{term}(t_1)\ \mathit{of}\\ &\quad \mid r \to r\\ &\quad \mid v_1 \to \mathsf{try}\ \{\ v_1\ \}\ \mathsf{catch}(x)\ \{\ t_2\ \}\\ &\quad \mid C[\mathsf{err}\ v] \to \mathsf{try}\ \{\ C[\mathsf{err}\ v]\ \}\ \mathsf{catch}(x)\ \{\ t_2\ \}\\ &\quad \mid C[\mathsf{break}\ p\ v] \to (\mathsf{try}\ \{\ C\ \}\ \mathsf{catch}(x)\ \{\ t_2\ \})[\mathsf{break}\ p\ v]\\ \mathit{term}(\mathsf{throw}\ t) &= \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\ &\quad \mid r \to r\\ &\quad \mid v \to \mathsf{throw}\ v\\ &\quad \mid C[\mathsf{err}\ v] \to (\mathsf{throw}\ C)[\mathsf{err}\ v]\\ &\quad \mid C[\mathsf{break}\ p\ v] \to (\mathsf{throw}\ C)[\mathsf{break}\ p\ v]\\ \mathit{term}(\mathsf{err}\ v) &= [\ ][\mathsf{err}\ v]\\ \mathit{term}(p:\ t) &= \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\ &\quad \mid r \to r\\ &\quad \mid v \to p:\ v\\ &\quad \mid C[\mathsf{err}\ v] \to (p:\ C)[\mathsf{err}\ v]\\ &\quad \mid C[\mathsf{break}\ p'\ v] \to p:\ C[\mathsf{break}\ p'\ v]\\ \mathit{term}(\mathsf{break}\ p\ t) &= \mathit{case}\ \mathit{term}(t)\ \mathit{of}\\ &\quad \mid r \to r\\ &\quad \mid v \to [\ ][\mathsf{break}\ p\ v]\\ &\quad \mid C[\mathsf{err}\ v] \to (\mathsf{break}\ p\ C)[\mathsf{err}\ v]\\ &\quad \mid C[\mathsf{break}\ p'\ v] \to (\mathsf{break}\ p\ C)[\mathsf{break}\ p'\ v]\end{aligned}$$

$$
\begin{array}{rl}
\mathit{term}(\mathsf{ref}\ t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid r \rightarrow r \\
& \mid v \rightarrow \mathsf{ref}\ v \\
& \mid C[\mathsf{err}\ v] \rightarrow (\mathsf{ref}\ C)[\mathsf{err}\ v] \\
& \mid C[\mathsf{break}\ p\ v] \rightarrow (\mathsf{ref}\ C)[\mathsf{break}\ p\ v] \\
\mathit{term}(\mathsf{deref}\ t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid r \rightarrow r \\
& \mid v \rightarrow \mathsf{deref}\ v \\
& \mid C[\mathsf{err}\ v] \rightarrow (\mathsf{deref}\ C)[\mathsf{err}\ v] \\
& \mid C[\mathsf{break}\ p\ v] \rightarrow (\mathsf{deref}\ C)[\mathsf{break}\ p\ v] \\
\mathit{term}(t_1 = t_2) = & \mathit{case}\ \mathit{term}(t_1)\ \mathit{of} \\
& \mid r \rightarrow r \\
& \mid v_1 \rightarrow \mathit{case}\ \mathit{term}(t_2)\ \mathit{of} \\
& \qquad \mid r \rightarrow r \\
& \qquad \mid v_2 \rightarrow v_1 = v_2 \\
& \qquad \mid C[\mathsf{err}\ v] \rightarrow (v_1 = C)[\mathsf{err}\ v] \\
& \qquad \mid C[\mathsf{break}\ p\ v] \rightarrow (v_1 = C)[\mathsf{break}\ p\ v] \\
& \mid C[\mathsf{err}\ v] \rightarrow (C = t_2)[\mathsf{err}\ v] \\
& \mid C[\mathsf{break}\ p\ v] \rightarrow (C = t_2)[\mathsf{break}\ p\ v]
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{Term} \rightarrow \mathsf{PotRed} + \mathsf{Val} + \mathsf{Err} & \\
\mathit{search}(t) = & \mathit{case}\ \mathit{term}(t)\ \mathit{of} \\
& \mid r \rightarrow r \\
& \mid v \rightarrow v \\
& \mid C[\mathsf{err}\ v] \rightarrow \text{error “uncaught exception”} \\
& \mid C[\mathsf{break}\ p\ v] \rightarrow \text{error “undelimited break”}
\end{array}
$$

Delimited contexts Defunctionalizing the delimited function-spaces of Exc and Break gives the following two restrictions on evaluation contexts:

$$
\begin{array}{rl}
\mathsf{ExcCtx} \ni F ::= & [\,] \mid \cdots \mid \\
& \mathsf{throw}\ F \mid \\
& p : F \mid \mathsf{break}\ p\ F \mid \\
& \mathsf{ref}\ F \mid \mathsf{deref}\ F \mid F = t \mid v = F \\
\mathsf{BreakCtx} \ni G ::= & [\,] \mid \cdots \mid \\
& \mathsf{try}\ \{\ G\ \}\ \mathsf{catch}(x)\ \{\ t\ \} \mid \mathsf{throw}\ G \mid \\
& \mathsf{break}\ p\ G \mid \\
& \mathsf{ref}\ G \mid \mathsf{deref}\ G \mid G = t \mid v = G
\end{array}
$$

The grammar of break contexts differs with those of Guha et al. [104, Figure 8] which does not have the non-terminal “break p G”. This context represents the ability to break out of a break expression. The absence of this context breaks unique decomposition since a term such as “$p_2 :$ break p_1 (break p_2 v)” is not a value and has no decomposition.

This omission, which has been corrected in a later revision, illustrates how difficult it is for semantic engineers to specify reduction contexts, even if they use a tool such as PLT Redex, which Guha et al. did. In contrast, using the prequel leads to no such omission.

5.6.3 A reduction semantics

Applying the transformations of Section 5.2 to the search function for λ_{JS} yields the grammar of reduction contexts and the decomposition function. Thus we have reconstructed Guha et al.'s revised reduction semantics for λ_{JS}. For space reasons, we omit displaying it here.

5.6.4 An abstract machine

Applying the syntactic correspondence to the reduction semantics, we can directly derive an abstract machine to run λ_{JS} programs. In contrast, Guha et al. could not directly run the Mozilla test suite on the evaluator made available by PLT Redex [104, Section 3]. Instead, they had to design yet another evaluator to run the test suite, creating yet another proof obligation.

We are in the process of adapting their front end for our abstract machine.

5.6.5 Coroutines for JavaScript

In this section, we show how to add support for coroutines to JavaScript based on the existing semantics for λ_{JS}. This extension is inspired by de Moura et al.'s semantics for asynchronous coroutines in Lua [79]. We have adapted their semantics as a minimal extension to the existing semantics for λ_{JS}. More concretely, we generalize the semantics of break expressions in λ_{JS} so that instead of discarding the escaped context, the context is saved as a functional abstraction in the store:

$$\begin{array}{ll} (gbreak) & \langle \sigma, C[p:\ C'[\mathsf{break}\ p\ v]]\rangle \to \langle \sigma', C[v]\rangle \\ & \text{where } \sigma' = \sigma[p \mapsto \mathsf{func}(x)\ \{\ \mathsf{return}\ p:\ C'[x]\ \}] \end{array}$$

By subsequently invoking this abstraction, the captured context is restored with the break delimiter in place.

We can now specify coroutine operators as syntactic sugar. Here desugaring is parameterized by the label of the lexically enclosing coroutine:

$$\begin{array}{rl} [\![\mathsf{create}\ t]\!]_p &= p':\ [\![t]\!]_{p'}\ (\mathsf{break}\ p'\ p') \quad \text{where } p' \text{ is fresh} \\ [\![\mathsf{resume}\ t_1\ t_2]\!]_p &= (\mathsf{deref}\ [\![t_1]\!]_p)\ [\![t_2]\!]_p \\ [\![\mathsf{yield}\ t]\!]_p &= \mathsf{break}\ p\ [\![t]\!]_p \end{array}$$

Compared to the coroutines in Lua, the ones here can resume themselves. We conjecture that the same generalization provides support for the generators of JavaScript 1.5.

5.6.6 Summary and conclusion

We have constructed a refocus-ready reduction semantics out of a total search function implementing the reduction strategy for Guha et al.'s λ_{JS}. This section illustrates how the prequel can provide assistance and support for an ongoing effort in semantic engineering.

5.7 Related work

We are aware of two tools that support the development of syntactic theories and provide automatic property checking and interpreter construction: SL [221] and PLT Redex [89]. SL automates the construction of unique decomposition proofs. PLT Redex visualizes reduction. Neither uses refocusing to derive efficient abstract machines. One reason could be that, in general, it is difficult to derive a decompose function that can be used for refocusing. As we have shown here, this is not the case when starting from the search strategy. We therefore hope that the present work can be used to further assist semantic engineers.

5.8 Conclusion and perspectives

We have shown how the specification of a deterministic reduction strategy, in the form of a compositional search function, provides all of the information needed to mechanically derive a complete specification of a reduction semantics where unique decomposition follows as a corollary of the search function. Furthermore, the decompose function obtained can be refocused to mechanically derive an abstract machine. We have illustrated the method for several languages, including context-sensitive and context-insensitive specifications using delimited contexts and non-trivial control operators. Finally, we have evaluated the method by reconstructing a semantics for an actual programming language. In so doing, the method mechanically uncovered several design issues and provided constructive solutions.

The present work thus provides a practical prequel for automating the construction of abstract machines via a reduction semantics. Future work includes formalizing the metalanguage for program transformations and developing tools that automate the derivation of abstract machines and provide mechanically verifiable correctness proofs.

Chapter 6

Normalization functions for Boolean propositional formulas

This chapter is joint work with Olivier Danvy and Jacob Johannsen. This chapter is an extended version of [76]: Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011)*, pages 1–12, Austin, Texas, January 2011. ACM Press. Invited talk.

Abstract

This article stems from the 20th anniversary of PEPM. It illustrates semantics-based program manipulation by inter-deriving reduction-based and reduction-free normalization functions for Boolean terms. We first consider negational normal forms and then conjunctive normal forms. The reduction-based normalizers proceed in many reduction steps: they explicitly implement de Morgan's laws and the distributivity of disjunctions over conjunctions by enumerating the reduction sequence according to a given reduction strategy. The reduction-free normalizers are compositional and in one pass: they internalize the reduction strategy into an evaluation order and carry out de Morgan's laws and the distributivity of disjunctions over conjunctions implicitly, in passing. The reduction-free negational normalizer is in direct style. The reduction-free conjunctive normalizer uses delimited continuations and can be expressed in direct style with the delimited-control operators shift and reset. Each of these semantic artifacts is usually designed by hand, on a case-by-case basis. Our overarching message here is that they can all be seamlessly inter-derived.

6.1 Introduction

This work presents the first derivation of a reduction-free algorithm from a reduction-based algorithm that is not refocusable: conjunctive normalization of Boolean terms.

The normalization of Boolean terms into negational normal form and subsequent normalization into conjunctive normal form can be equivalently viewed as *reduction-based, small-step processes*, where the reduction rules are repeatedly applied until a normal form

is obtained, and as *reduction-free, big-step processes*, where a given Boolean term is recursively traversed in one fell swoop. Occasionally, the normalization is also specified with an abstract machine, which can itself be equally viewed as a small-step process and as a big-step one [67].

The goal of this article is to inter-derive these normalization processes using the program transformations used in the Reynolds functional correspondence between evaluators and big-step abstract machines [5, 176] and in the syntactic correspondence between calculi and small-step abstract machines [27], to which we add a new prequel.

Overview In Section 6.2, we specify the abstract syntax of Boolean terms, of negational normal forms, and of conjunctive normal forms. The derivation is then organized in two steps. In Section 6.3, we inter-derive the normalization of Boolean terms into negational normal form, listing each intermediate step in full detail. In Section 6.4, we inter-derive the normalization of negational normal forms into conjunctive normal forms, succinctly reusing the same presentation as in Section 6.3. For emphasis, the presentations of Sections 6.3.1 to 6.3.4 and 6.4.1 to 6.4.4 are deliberately parallel, so that the reader can easily identify what is generic to the methodology and what is specific to each example. Throughout we use pure Standard ML as a functional meta-language and additionally make use of the delimited control operators shift and reset for the direct-style normalizer in Section 6.4.5. This article is meant to be self-contained, but in case of doubt, the reader should consult the first author's lecture notes at the Sixth International School on Advanced Functional Programming [61].

6.2 Domain of discourse

We consider Boolean terms of conjunctions, disjunctions and negations:

$$t ::= x \mid \neg t \mid t \wedge t \mid t \vee t$$

De Morgan's laws provide *conversion rules* between Boolean terms, where negations float up or down an abstract syntax tree:

$$\begin{aligned} \neg(\neg t) &\leftrightarrow t \\ \neg(t_1 \wedge t_2) &\leftrightarrow (\neg t_1) \vee (\neg t_2) \\ \neg(t_1 \vee t_2) &\leftrightarrow (\neg t_1) \wedge (\neg t_2) \end{aligned}$$

These conversion rules can be oriented into *reduction rules*. For example, the following reduction rules make negations float down the abstract syntax tree of a given term:

$$\begin{aligned} \neg(\neg t) &\rightarrow t \\ \neg(t_1 \wedge t_2) &\rightarrow (\neg t_1) \vee (\neg t_2) \\ \neg(t_1 \vee t_2) &\rightarrow (\neg t_1) \wedge (\neg t_2) \end{aligned}$$

Any Boolean term can be reduced into a *negational normal form*, where only variables are negated:

$$\begin{aligned} l &::= x \mid \neg x \\ t_{nnf} &::= l \mid t_{nnf} \wedge t_{nnf} \mid t_{nnf} \vee t_{nnf} \end{aligned}$$

A negational normal form is thus a mixed tree of conjunctions and disjunctions of literals.

Likewise, the distributivity laws provide conversion rules between negational normal forms, where disjunctions float up or down an abstract-syntax tree:

$$t_{nnf1} \vee (t_{nnf2} \wedge t_{nnf3}) \leftrightarrow (t_{nnf1} \vee t_{nnf2}) \wedge (t_{nnf1} \vee t_{nnf3})$$
$$(t_{nnf1} \wedge t_{nnf2}) \vee t_{nnf3} \leftrightarrow (t_{nnf1} \vee t_{nnf3}) \wedge (t_{nnf2} \vee t_{nnf3})$$

These conversion rules can be oriented into reduction rules. For example, the following reduction rules make disjunctions float down the abstract syntax tree of a given term in negational normal form:

$$t_{nnf1} \vee (t_{nnf2} \wedge t_{nnf3}) \rightarrow (t_{nnf1} \vee t_{nnf2}) \wedge (t_{nnf1} \vee t_{nnf3})$$
$$(t_{nnf1} \wedge t_{nnf2}) \vee t_{nnf3} \rightarrow (t_{nnf1} \vee t_{nnf3}) \wedge (t_{nnf2} \vee t_{nnf3})$$

Any Boolean term in negational normal form can be reduced into a *conjunctive normal form*, where conjunctions, disjunctions and literals are stratified:

$$\begin{aligned} l &::= x \mid \neg x \\ t_{dnf} &::= t_{dnf} \vee t_{dnf} \mid l \\ t_{cnf} &::= t_{cnf} \wedge t_{cnf} \mid t_{dnf} \end{aligned}$$

A conjunctive normal form is thus a stratified tree of conjunctions of disjunctions of literals.

Terms A Boolean term is either a variable, a negated term, a conjunction of two terms, or a disjunction of two terms. We implement Boolean terms with the following ML data type:

```
datatype term = VAR  of ide
              | NEG  of term
              | CONJ of term * term
              | DISJ of term * term
```

The fold functional associated to this data type abstracts its recursive descent by parameterizing what to do in each case:

```
(*  (ide -> 'a) * ('a -> 'a) * ('a * 'a -> 'a) * ('a * 'a -> 'a) -> term -> 'a  *)
fun term_foldr (var, neg, conj, disj) t
    = let fun visit (VAR x)         = var x
            | visit (NEG t)         = neg (visit t)
            | visit (CONJ (t1, t2)) = conj (visit t1, visit t2)
            | visit (DISJ (t1, t2)) = disj (visit t1, visit t2)
      in visit t
      end
```

The ML encoding of terms is an adequate representation of t:

Proposition 16 (Adequacy of Boolean terms)**.** *There is a bijection between Boolean terms in the grammar of* t *and values in the data type* `term`*.*

Since we have no structural properties on Boolean terms, e.g., substitution, a bijection is sufficient to ensure adequacy of the representation.

Negational normal forms A Boolean term is in negational normal form when only its variables are negated. Since ML does not support subtyping, we implement negational normal forms with the following specialized data types:

```
datatype literal = POSVAR of ide
                 | NEGVAR of ide

datatype term_nnf = LIT_nnf  of literal
                  | CONJ_nnf of term_nnf * term_nnf
                  | DISJ_nnf of term_nnf * term_nnf
```

The fold functional associated to this data type abstracts its recursive descent by parameterizing what to do in each case:

```
(*  (ide -> 'a) * (ide -> 'a) * ('a -> 'a) * ('a * 'a -> 'a) * ('a * 'a -> 'a)  *)
(*  -> term_nnf -> 'a  *)
fun term_nnf_foldr (posvar, negvar, lit, conj, disj) t
    = let fun visit_l (POSVAR x)          = posvar x
            | visit_l (NEGVAR x)          = negvar x
          fun visit_n (LIT_nnf l)         = lit (visit_l l)
            | visit_n (CONJ_nnf (n1, n2)) = conj (visit_n n1, visit_n n2)
            | visit_n (DISJ_nnf (n1, n2)) = disj (visit_n n1, visit_n n2)
      in visit_n t
      end
```

For example, a negational normal form is dualized by recursively mapping positive occurrences of variables to negative ones, negative occurrences of variables to positive ones, conjunctions to disjunctions, and disjunctions to conjunctions:

```
(*  term_nnf -> term_nnf  *)
fun dualize t
    = term_nnf_foldr (NEGVAR, POSVAR, LIT_nnf, DISJ_nnf, CONJ_nnf) t
```

For another example, a negational normal form is embedded into a Boolean term by mapping every specialized constructor into the corresponding original constructor(s):

```
(*  term_nnf -> term  *)
fun embed_nnf t
    = term_nnf_foldr (VAR, fn x => NEG (VAR x), fn t => t, CONJ, DISJ) t
```

The ML encoding of negational normal forms is an adequate representation of t_{nnf}:

Proposition 17 (Adequacy of negational normal forms)**.** *There is a bijection between negational normal forms in the grammar of t_{nnf} and values in the data type* `term_nnf`*.*

Conjunctive normal forms A Boolean term is in conjunctive normal form when it is stratified as conjunctions of disjunctions of literals. Again, since ML does not support subtyping, we implement normal forms with the following specialized data types:

```
datatype disj_cnf = DISJ_leaf of literal
                  | DISJ_node of disj_cnf * disj_cnf

datatype conj_cnf = CONJ_leaf of disj_cnf
                  | CONJ_node of conj_cnf * conj_cnf

type term_cnf = conj_cnf
```

The fold functional associated to this data type abstracts its recursive descent by parameterizing what to do in each case:

```
(*  (ide -> 'b) * (ide -> 'b) * ('b * 'b -> 'a) * ('a * 'a -> 'a) -> term_cnf -> 'a *)
fun term_cnf_foldr (posvar, negvar, lit, conj, disj) t
    = let fun visit_l (POSVAR x)           = posvar x
            | visit_l (NEGVAR x)           = negvar x
          fun visit_d (DISJ_leaf l)        = lit (visit_l l)
            | visit_d (DISJ_node (d1, d2)) = disj (visit_d d1, visit_d d2)
          fun visit_c (CONJ_leaf d)        = visit_d d
            | visit_c (CONJ_node (c1, c2)) = conj (visit_c c1, visit_c c2)
      in visit_c t
      end
```

A conjunctive normal form is embedded into a negational normal form by mapping every specialized constructor into the corresponding original constructor:

```
(*  term_cnf -> term_nnf  *)
fun embed_cnf t
    = term_cnf_foldr (POSVAR, NEGVAR, LIT_nnf, CONJ_nnf, DISJ_nnf) t
```

The ML encoding of conjunctive normal forms is an adequate representation of t_{cnf}:

Proposition 18 (Adequacy of conjunctive normal forms)**.** *There is a bijection between conjunctive normal forms in the grammar of t_{cnf} and values in the data type* `term_cnf`*.*

6.3 Leftmost outermost negational normalization

In this section, we consider negational normal forms. We go from a leftmost-outermost *reduction* strategy to the corresponding leftmost-outermost *evaluation* strategy. We first implement the reduction strategy (Section 6.3.1) as a prequel to implementing the corresponding reduction semantics (Section 6.3.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 6.3.3) and to the functional correspondence between abstract machines and normalization functions (Section 6.3.4).

6.3.1 Prequel to a reduction semantics

The reduction strategy induces a notion of value and of potential redex (i.e., of a term that is an actual redex or that is stuck); we are then in position to state a compositional search function that implements the reduction strategy and maps a given term either to the corresponding value, if it is in normal form, or to a potential redex (Section 6.3.1). From this search function, we derive a decomposition function mapping a given term either to the corresponding value, if it is in normal form, or to a potential redex and its reduction context (Section 6.3.1). As a corollary, we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 6.3.1).

The reduction strategy

The reduction strategy consists in locating the leftmost-outermost negation of a term which is not a variable. A *value* therefore is a term where only variables are negated, i.e., a negational normal form:

```
type value = term_nnf
```

A *potential redex* is the negation of a term that is not a variable:

```
datatype potential_redex = PR_NEG  of term
                         | PR_CONJ of term * term
                         | PR_DISJ of term * term
```

The following compositional search function implements the reduction strategy. It searches a potential redex *depth-first and from left to right*, using an auxiliary function for negated subterms:

```
datatype found = VAL    of value
               | POTRED of potential_redex

(*  term -> found  *)
fun search_term_neg (VAR x)         = VAL (LIT_nnf (NEGVAR x))
  | search_term_neg (NEG t)         = POTRED (PR_NEG t)
  | search_term_neg (CONJ (t1, t2)) = POTRED (PR_CONJ (t1, t2))
  | search_term_neg (DISJ (t1, t2)) = POTRED (PR_DISJ (t1, t2))

(*  term -> found  *)
fun search_term (VAR x)
    = VAL (LIT_nnf (POSVAR x))
  | search_term (NEG t)
    = search_term_neg t
  | search_term (CONJ (t1, t2))
    = (case search_term t1
        of (VAL v1)
           => (case search_term t2
                of (VAL v2)
                   => VAL (CONJ_nnf (v1, v2))
                 | (POTRED pr)
                   => POTRED pr)
         | (POTRED pr)
           => POTRED pr)
  | search_term (DISJ (t1, t2))
    = (case search_term t1
        of (VAL v1)
           => (case search_term t2
                of (VAL v2)
                   => VAL (DISJ_nnf (v1, v2))
                 | (POTRED pr)
                   => POTRED pr)
         | (POTRED pr)
           => POTRED pr)

(*  term -> found  *)
fun search t = search_term t
```

When a negation is encountered, the auxiliary function `search_term_neg` is called to decide whether this negation is a value or a potential redex.

By adequacy of our representation, the search function defines a reduction strategy:

Definition 19. The search function finds the leftmost outermost redex in the input term.

From searching to decomposing

Let us transform the search function of Section 6.3.1 into a decomposition function for the reduction semantics of Section 6.3.2. The only difference between searching and decomposing is that given a non-value term, searching yields a potential redex whereas

decomposing yields a potential redex and its reduction context, i.e., a *decomposition*. This reduction context is the defunctionalized continuation of the search function, and we construct it as such, by (1) CPS-transforming the search function (and simplifying it one bit) and (2) defunctionalizing its continuation.

CPS transformation The search function is CPS-transformed by naming its intermediate results, sequentializing their computation, and introducing an extra functional argument, the continuation, that maps an intermediate result to a final answer:

```
(*  term * (found -> 'a) -> 'a  *)
fun search1_term_neg (VAR x,         k) = k (VAL (LIT_nnf (NEGVAR x)))
  | search1_term_neg (NEG t,         k) = k (POTRED (PR_NEG t))
  | search1_term_neg (CONJ (t1, t2), k) = k (POTRED (PR_CONJ (t1, t2)))
  | search1_term_neg (DISJ (t1, t2), k) = k (POTRED (PR_DISJ (t1, t2)))

(*  term * (found -> 'a) -> 'a  *)
fun search1_term (VAR x, k)
    = k (VAL (LIT_nnf (POSVAR x)))
  | search1_term (NEG t, k)
    = search1_term_neg (t, k)
  | search1_term (CONJ (t1, t2), k)
    = search1_term (t1,
        fn (VAL v1)
           => search1_term (t2,
                fn (VAL v2)
                   => k (VAL (CONJ_nnf (v1, v2)))
                 | (POTRED pr)
                   => k (POTRED pr))
         | (POTRED pr)
           => k (POTRED pr))
  | search1_term (DISJ (t1, t2), k)
    = search1_term (t1,
        fn (VAL v1)
           => search1_term (t2,
                fn (VAL v2)
                   => k (VAL (DISJ_nnf (v1, v2)))
                 | (POTRED pr)
                   => k (POTRED pr))
         | (POTRED pr)
           => k (POTRED pr))

(*  term -> found  *)
fun search1 t = search1_term (t, fn f => f)
```

Simplifying the CPS-transformed search The search is completed as soon as a potential redex is found. It can thus be simplified by only applying the continuation when a value is found:

```
(*  term * (value -> found) -> found  *)
fun search2_term_neg (VAR x,         k) = k (LIT_nnf (NEGVAR x))
  | search2_term_neg (NEG t,         k) = POTRED (PR_NEG t)
  | search2_term_neg (CONJ (t1, t2), k) = POTRED (PR_CONJ (t1, t2))
  | search2_term_neg (DISJ (t1, t2), k) = POTRED (PR_DISJ (t1, t2))

(*  term * (value -> found) -> found  *)
fun search2_term (VAR x, k)
    = k (LIT_nnf (POSVAR x))
```

```
  | search2_term (NEG t, k)
    = search2_term_neg (t, k)
  | search2_term (CONJ (t1, t2), k)
    = search2_term (t1, fn v1 =>
        search2_term (t2, fn v2 =>
          k (CONJ_nnf (v1, v2))))
  | search2_term (DISJ (t1, t2), k)
    = search2_term (t1, fn v1 =>
        search2_term (t2, fn v2 =>
          k (DISJ_nnf (v1, v2))))

(*  term -> found  *)
fun search2 t = search2_term (t, fn v => VAL v)
```

Potential redexes are now returned directly and the `VAL` constructor is relegated to the initial continuation.

Defunctionalization To defunctionalize the continuation, we first enumerate the inhabitants of its function space. These inhabitants arise from the initial continuation in the definition of `search2` and in the 4 intermediate continuations in the definition of `search2_term`. We therefore partition the continuation with these 5 functional abstractions, 4 of which have free variables. We then represent this partition as

- a data type with 5 constructors that are parameterized with the free variables of the corresponding function abstraction, together with
- a function `apply3_cont` dispatching upon these 5 summands and mapping them to the corresponding function abstractions:

```
datatype cont = C0
              | C1 of value * cont
              | C2 of cont * term
              | C3 of value * cont
              | C4 of cont * term

(*  cont -> value -> found  *)
fun apply3_cont  C0          = (fn v  => VAL v)
  | apply3_cont (C1 (v1, k)) = (fn v2 => apply3_cont k (CONJ_nnf (v1, v2)))
  | apply3_cont (C2 (k, t2)) = (fn v1 => search3_term (t2, C1 (v1, k)))
  | apply3_cont (C3 (v1, k)) = (fn v2 => apply3_cont k (DISJ_nnf (v1, v2)))
  | apply3_cont (C4 (k, t2)) = (fn v1 => search3_term (t2, C3 (v1, k)))

(*  term * cont -> found  *)
and search3_term_neg (VAR x,         k) = apply3_cont k (LIT_nnf (NEGVAR x))
  | search3_term_neg (NEG t,         k) = POTRED (PR_NEG t)
  | search3_term_neg (CONJ (t1, t2), k) = POTRED (PR_CONJ (t1, t2))
  | search3_term_neg (DISJ (t1, t2), k) = POTRED (PR_DISJ (t1, t2))

(*  term * cont -> found  *)
and search3_term (VAR x,         k) = apply3_cont k (LIT_nnf (POSVAR x))
  | search3_term (NEG t,         k) = search3_term_neg (t, k)
  | search3_term (CONJ (t1, t2), k) = search3_term (t1, C2 (k, t2))
  | search3_term (DISJ (t1, t2), k) = search3_term (t1, C4 (k, t2))

(*  term -> found  *)
fun search3 t = search3_term (t, C0)
```

This data type of defunctionalized continuations is that of reduction contexts [57].

We have defined `apply3_cont` in curried form to emphasize that it maps each summand to a continuation. In the following, we consider its uncurried definition.

Decomposition We are now in position to extend the search function to not only return a potential redex (if one exists) *but also its reduction context*. The result is the decomposition function of a reduction semantics, where `value_or_decomposition`, `decompose`, `decompose_term`, `decompose_term_neg`, and `decompose_cont` are the respective clones of `found`, `search3`, `search3_term`, `search3_term_neg`, and `apply3_cont`:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * cont

(*  cont * value -> value_or_decomposition  *)
fun decompose_cont (C0,         v ) = VAL v
  | decompose_cont (C1 (v1, k), v2) = decompose_cont (k, CONJ_nnf (v1, v2))
  | decompose_cont (C2 (k, t2), v1) = decompose_term (t2, C1 (v1, k))
  | decompose_cont (C3 (v1, k), v2) = decompose_cont (k, DISJ_nnf (v1, v2))
  | decompose_cont (C4 (k, t2), v1) = decompose_term (t2, C3 (v1, k))

(*  term * cont -> value_or_decomposition  *)
and decompose_term_neg (VAR x, k) = decompose_cont (k, LIT_nnf (NEGVAR x))
  | decompose_term_neg (NEG t,         k) = DEC (PR_NEG t, k)
  | decompose_term_neg (CONJ (t1, t2), k) = DEC (PR_CONJ (t1, t2), k)
  | decompose_term_neg (DISJ (t1, t2), k) = DEC (PR_DISJ (t1, t2), k)

(*  term * cont -> value_or_decomposition  *)
and decompose_term (VAR x, k) = decompose_cont (k, LIT_nnf (POSVAR x))
  | decompose_term (NEG t,         k) = decompose_term_neg (t, k)
  | decompose_term (CONJ (t1, t2), k) = decompose_term (t1, C2 (k, t2))
  | decompose_term (DISJ (t1, t2), k) = decompose_term (t1, C4 (k, t2))

(*  term -> value_or_decomposition  *)
fun decompose t = decompose_term (t, C0)
```

Recomposing

A reduction context is recomposed around a term with a left fold over this context:

```
(*  cont * term -> term  *)
fun recompose (C0,         t ) = t
  | recompose (C1 (v1, k), t2) = recompose (k, CONJ (embed_nnf v1, t2))
  | recompose (C2 (k, t2), t1) = recompose (k, CONJ (t1, t2))
  | recompose (C3 (v1, k), t2) = recompose (k, DISJ (embed_nnf v1, t2))
  | recompose (C4 (k, t2), t1) = recompose (k, DISJ (t1, t2))
```

Proposition 20 (left inverseness)**.** *Given a function `pr2t` mapping a potential redex to the corresponding term,*

```
forall t,
  decompose t = DEC (pr, k)
  =>
  recompose (k, pr2t pr) = t
```

6.3.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for negational normalization.

Notion of reduction

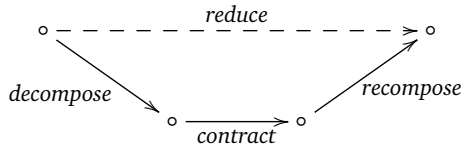
The reduction rules implement the De Morgan laws:

```
datatype contractum_or_error = CONTRACTUM of term
                             | ERROR      of string

(*  potential_redex -> contractum_or_error  *)
fun contract (PR_NEG t)         = CONTRACTUM t
  | contract (PR_CONJ (t1, t2)) = CONTRACTUM (DISJ (NEG t1, NEG t2))
  | contract (PR_DISJ (t1, t2)) = CONTRACTUM (CONJ (NEG t1, NEG t2))
```

In the present case, all potential redexes are actual ones, i.e., no terms are stuck.

One-step reduction

Given a non-value term, a one-step reduction function (1) decomposes this non-value term into a potential redex and a reduction context, (2) contracts the potential redex if it is an actual one, and (3) recomposes the reduction context with the contractum.



If the potential redex is not an actual one, reduction is stuck. Given a value term, reduction is also stuck:

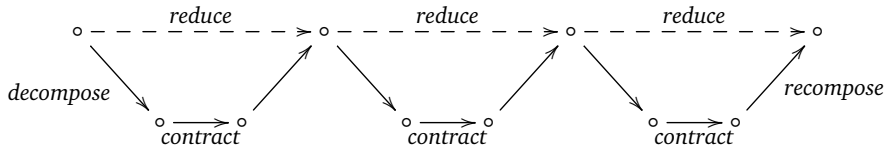
```
datatype reduct_or_stuck = REDUCT of term
                         | STUCK  of string


(*  term -> reduct_or_stuck  *)
fun reduce t
    = (case decompose t
        of (VAL v)
           => STUCK "irreducible term"
         | (DEC (pr, k))
           => (case contract pr
                of (CONTRACTUM t') => REDUCT (recompose (k, t'))
                 | (ERROR s)       => STUCK s))
```

This one-step reduction function is the hallmark of a reduction semantics [85, 89]:

Property 21 (soundness)**.** Let `t` represent t and `t'` represent t'. If `reduce t` evaluates to `REDUCT t'` then t reduces in one step by leftmost outermost reduction to t'.

Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value or becomes stuck:



If it yields a value, this value is the result of evaluation, and if it becomes stuck, evaluation goes wrong:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string
```

The following definition uses `decompose` to distinguish between value and non-value terms:

```
(*  value_or_decomposition -> result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM t') => iterate (decompose (recompose (k, t')))
         | (ERROR s)       => WRONG s)

(*  term -> result_or_wrong  *)
fun normalize t = iterate (decompose t)
```

Property 22 (soundness)**.** Let `t` represent t and `v` represent v. If `normalize t` evaluates to `RESULT v` then t reduces to v by leftmost outermost reduction.

6.3.3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 6.3.2 into a family of reduction-free normalization functions, i.e., functions that do not enumerate the reduction sequence and where no intermediate reduct is ever constructed. We first refocus the reduction-based normalization function to deforest the intermediate reducts [71, 197], and we obtain a small-step abstract machine implementing the iteration of the refocus function (Section 6.3.3). After inlining the contraction function (Section 6.3.3), we transform this small-step abstract machine into a big-step one [67] (Section 6.3.3). This machine exhibits a number of corridor transitions, and we compress them (Section 6.3.3). We also opportunistically specialize its contexts (Section 6.3.3). The resulting abstract machine is in defunctionalized form [70], and we refunctionalize it [69] (Section 6.3.4). The result is in continuation-passing style and we re-express it in direct style [56] (Section 6.3.4). The resulting direct-style function is a traditional conversion function for Boolean formulas; in particular, it is compositional. We express it with one recursive descent using `term_foldr` (Section 6.3.4).

Modus operandi In each of the following subsections, we derive successive versions of the normalization function, indexing its components with the number of the subsection.

Refocusing

The normalization function of Section 6.3.2 is reduction-based because it constructs every intermediate term in the reduction sequence. In its definition, `decompose` is always applied to the result of `recompose` after the first decomposition. In fact, a vacuous initial call to `recompose` ensures that in all cases, `decompose` is applied to the result of `recompose`:

```
fun normalize t = iterate (decompose (recompose (C0, t)))
```

We can factor out these composite calls in a function, `refocus0`, that maps a contractum and its reduction context to the next potential redex and the next reduction context, if such a pair exists, in the reduction sequence:

```
(* term * cont -> value_or_decomposition *)
fun refocus0 (t, k) = decompose (recompose (k, t))

(* value_or_decomposition -> result_or_wrong *)
fun iterate0 (VAL v)
    = RESULT v
  | iterate0 (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM t') => iterate0 (refocus0 (t', k))
         | (ERROR s)       => WRONG s)

(* term -> result_or_wrong *)
fun normalize0 t = iterate0 (refocus0 (t, C0))
```

Refocusing, extensionally The refocus function goes from a redex site to the next redex site, if there is one.

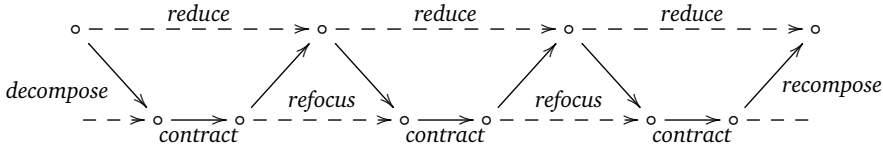
Refocusing, intensionally As investigated by Nielsen and the first author [71], the refocus function can be deforested to avoid constructing any intermediate reduct [214]. Such a deforestation makes the normalization function reduction-free. The deforested version of `refocus` is optimally defined as *continuing the decomposition of the contractum in the current context*, i.e., as `decompose_term`:

```
(* term * cont -> value_or_decomposition *)
fun refocus1 (t, k) = decompose_term (t, k)
```

The refocused evaluation function therefore reads as follows:

```
(* value_or_decomposition -> result_or_wrong *)
fun iterate1 (VAL v)
    = RESULT v
  | iterate1 (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM t') => iterate1 (refocus1 (t', k))
         | (ERROR s)       => WRONG s)

(* term -> result_or_wrong *)
fun normalize1 t = iterate1 (refocus1 (t, C0))
```

This refocused normalization function is reduction-free because it is no longer based on a (one-step) reduction function and it no longer enumerates the successive reducts in the reduction sequence:



In the rest of this section, we mechanically transform this reduction-free normalization function into an abstract machine.

Inlining the contraction function

We first unfold the call to `contract` in the definition of `iterate1`, and name the resulting function `iterate2`. Reasoning by inversion, there are three potential redexes and therefore the `DEC` clause in the definition of `iterate1` is replaced by three `DEC` clauses in the definition of `iterate2`:

```
(*  term * cont -> value_or_decomposition  *)
fun refocus2 (t, k) = decompose_term (t, k)

(*  value_or_decomposition -> result_or_wrong  *)
fun iterate2 (VAL v)
    = RESULT v
  | iterate2 (DEC (PR_NEG t, k))
    = iterate2 (refocus2 (t, k))
  | iterate2 (DEC (PR_CONJ (t1, t2), k))
    = iterate2 (refocus2 (DISJ (NEG t1, NEG t2), k))
  | iterate2 (DEC (PR_DISJ (t1, t2), k))
    = iterate2 (refocus2 (CONJ (NEG t1, NEG t2), k))

(*  term -> result_or_wrong  *)
fun normalize2 t = iterate2 (refocus2 (t, C0))
```

Lightweight fusion: from small-step abstract machine to big-step abstract machine

The refocused normalization function is a small-step abstract machine in the sense that `refocus2` (i.e., `decompose_term`, `decompose_term_neg` and `decompose_cont`) acts as an inner transition function and `iterate2` as an outer transition function. The outer transition function (also known as a 'driver loop' and as a 'trampoline' [98]) keeps activating the inner transition function until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [67, 155], we fuse `iterate2` and `refocus2` (i.e., `decompose_term`, `decompose_term_neg` and `decompose_cont`) so that the resulting function `iterate3` is *directly* applied to the result of `decompose_term`, `decompose_term_neg` and `decompose_cont`. The result is a big-step abstract machine [171] consisting of four (mutually tail-recursive) state-transition functions:

- `normalize3_term` is the composition of `iterate2` and `decompose_term` and a clone of `decompose_term`;
- `normalize3_term_neg` is the composition of `iterate2` and `decompose_term_neg` and a clone of `decompose_term_neg`;
- `normalize3_cont` is the composition of `iterate2` and `decompose_cont` that directly calls `iterate3` over a value or a decomposition instead of returning it to `iterate2` as `decompose_cont` did;
- `iterate3` is a clone of `iterate2` that calls the fused function `normalize3_term`.
- `normalize3` is a clone of `normalize` that calls the fused function `normalize3_term`.

```
(*  cont * value -> result_or_wrong  *)
fun normalize3_cont (C0, v)
    = iterate3 (VAL v)
  | normalize3_cont (C1 (v1, k), v2)
    = normalize3_cont (k, CONJ_nnf (v1, v2))
  | normalize3_cont (C2 (k, t2), v1)
    = normalize3_term (t2, C1 (v1, k))
  | normalize3_cont (C3 (v1, k), v2)
    = normalize3_cont (k, DISJ_nnf (v1, v2))
  | normalize3_cont (C4 (k, t2), v1)
    = normalize3_term (t2, C3 (v1, k))

(*  term * cont -> result_or_wrong  *)
and normalize3_term_neg (VAR x, k)
    = normalize3_cont (k, LIT_nnf (NEGVAR x))
  | normalize3_term_neg (NEG t, k)
    = iterate3 (DEC (PR_NEG t, k))
  | normalize3_term_neg (CONJ (t1, t2), k)
    = iterate3 (DEC (PR_CONJ (t1, t2), k))
  | normalize3_term_neg (DISJ (t1, t2), k)
    = iterate3 (DEC (PR_DISJ (t1, t2), k))

(*  term * cont -> result_or_wrong  *)
and normalize3_term (VAR x, k)
    = normalize3_cont (k, LIT_nnf (POSVAR x))
  | normalize3_term (NEG t, k)
    = normalize3_term_neg (t, k)
  | normalize3_term (CONJ (t1, t2), k)
    = normalize3_term (t1, C2 (k, t2))
  | normalize3_term (DISJ (t1, t2), k)
    = normalize3_term (t1, C4 (k, t2))

(*  value_or_decomposition -> result_or_wrong  *)
and iterate3 (VAL v)
    = RESULT v
  | iterate3 (DEC (PR_NEG t, k))
    = normalize3_term (t, k)
  | iterate3 (DEC (PR_CONJ (t1, t2), k))
    = normalize3_term (DISJ (NEG t1, NEG t2), k)
  | iterate3 (DEC (PR_DISJ (t1, t2), k))
    = normalize3_term (CONJ (NEG t1, NEG t2), k)

(*  term -> result_or_wrong  *)
fun normalize3 t = normalize3_term (t, C0)
```

Hereditary transition compression

In the abstract machine of Section 6.3.3, many of the transitions are 'corridor' ones in that they yield configurations for which there is a unique further transition. Let us hereditarily compress these transitions. To this end, we cut-and-paste the transition functions above, renaming their indices from 3 to 4. We consider each of their clauses in turn:

Clause `normalize4_cont (C0, v)`:

```
normalize4_cont (C0, v)
= (* by inlining normalize4_cont *)
iterate4 (VAL v)
= (* by inlining iterate4 *)
RESULT v
```

Clause `normalize4_term_neg (NEG t, k)`:

```
normalize4_term_neg (NEG t, k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_NEG t, k))
= (* by inlining iterate4 *)
normalize3_term (t, k)
```

Clause `normalize4_term_neg (CONJ (t1, t2), k)`:

```
normalize4_term_neg (CONJ (t1, t2), k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_CONJ (t1, t2), k))
= (* by inlining iterate4 *)
normalize4_term (DISJ (NEG t1, NEG t2), k)
= (* by inlining normalize4_term *)
normalize4_term (NEG t1, C4 (k, NEG t2))
= (* by inlining normalize4_term *)
normalize4_term_neg (t1, C4 (k, NEG t2))
```

Clause `normalize4_term_neg (DISJ (t1, t2), k)`:

```
normalize4_term_neg (DISJ (t1, t2), k)
= (* by inlining normalize4_term_neg *)
iterate4 (DEC (PR_DISJ (t1, t2), k))
= (* by inlining iterate4 *)
normalize4_term (CONJ (NEG t1, NEG t2), k)
= (* by inlining normalize4_term *)
normalize4_term (NEG t1, C2 (k, NEG t2))
= (* by inlining normalize4_term *)
normalize4_term_neg (t1, C2 (k, NEG t2))
```

As a corollary of the compressions, the definition of `iterate3` is now unused and can be omitted. The resulting abstract machine reads as follows:

```
(*  cont * value -> result_or_wrong  *)
fun normalize4_cont (C0, v)
    = RESULT v
  | normalize4_cont (C1 (v1, k), v2)
    = normalize4_cont (k, CONJ_nnf (v1, v2))
  | normalize4_cont (C2 (k, t2), v1)
    = normalize4_term (t2, C1 (v1, k))
  | normalize4_cont (C3 (v1, k), v2)
```

```
    = normalize4_cont (k, DISJ_nnf (v1, v2))
  | normalize4_cont (C4 (k, t2), v1)
    = normalize4_term (t2, C3 (v1, k))

(*  term * cont -> result_or_wrong  *)
and normalize4_term_neg (VAR x, k)
    = normalize4_cont (k, LIT_nnf (NEGVAR x))
  | normalize4_term_neg (NEG t, k)
    = normalize4_term (t, k)
  | normalize4_term_neg (CONJ (t1, t2), k)
    = normalize4_term_neg (t1, C4 (k, NEG t2))
  | normalize4_term_neg (DISJ (t1, t2), k)
    = normalize4_term_neg (t1, C2 (k, NEG t2))

(*  term * cont -> result_or_wrong  *)
and normalize4_term (VAR x, k)
    = normalize4_cont (k, LIT_nnf (POSVAR x))
  | normalize4_term (NEG t, k)
    = normalize4_term_neg (t, k)
  | normalize4_term (CONJ (t1, t2), k)
    = normalize4_term (t1, C2 (k, t2))
  | normalize4_term (DISJ (t1, t2), k)
    = normalize4_term (t1, C4 (k, t2))

(*  term -> result_or_wrong  *)
fun normalize4 t = normalize4_term (t, C0)
```

Context specialization

To symmetrize the definitions of `normalize4_term` and `normalize4_term_neg`, we introduce two specialized contexts for `C2` and `C4`, and we specialize `normalize4_cont` to directly call `normalize5_term_neg` for the new contexts `C2NEG` and `C4NEG`:

```
datatype cont = C0
              | C1 of value * cont
              | C2 of cont * term
              | C2NEG of cont * term
              | C3 of value * cont
              | C4 of cont * term
              | C4NEG of cont * term

(*  cont * value -> result_or_wrong  *)
fun normalize5_cont (C0, v)
    = RESULT v
  | normalize5_cont (C1 (v1, k), v2)
    = normalize5_cont (k, CONJ_nnf (v1, v2))
  | normalize5_cont (C2 (k, t2), v1)
    = normalize5_term (t2, C1 (v1, k))
  | normalize5_cont (C2NEG (k, t2), v1)
    = normalize5_term_neg (t2, C1 (v1, k))
  | normalize5_cont (C3 (v1, k), v2)
    = normalize5_cont (k, DISJ_nnf (v1, v2))
  | normalize5_cont (C4 (k, t2), v1)
    = normalize5_term (t2, C3 (v1, k))
  | normalize5_cont (C4NEG (k, t2), v1)
    = normalize5_term_neg (t2, C3 (v1, k))

(*  term * cont -> result_or_wrong  *)
and normalize5_term_neg (VAR x, k)
```

```
    = normalize5_cont (k, LIT_nnf (NEGVAR x))
  | normalize5_term_neg (NEG t, k)
    = normalize5_term (t, k)
  | normalize5_term_neg (CONJ (t1, t2), k)
    = normalize5_term_neg (t1, C4NEG (k, t2))
  | normalize5_term_neg (DISJ (t1, t2), k)
    = normalize5_term_neg (t1, C2NEG (k, t2))

(*  term * cont -> result_or_wrong  *)
and normalize5_term (VAR x, k)
    = normalize5_cont (k, LIT_nnf (POSVAR x))
  | normalize5_term (NEG t, k)
    = normalize5_term_neg (t, k)
  | normalize5_term (CONJ (t1, t2), k)
    = normalize5_term (t1, C2 (k, t2))
  | normalize5_term (DISJ (t1, t2), k)
    = normalize5_term (t1, C4 (k, t2))

(*  term -> result_or_wrong  *)
fun normalize5 t = normalize5_term (t, C0)
```

6.3.4 From abstract machines to normalization functions

In this section, we transform the abstract machine of Section 6.3.3 into two compositional normalization functions, one in continuation-passing style (Section 6.3.4) and one in direct style (Section 6.3.4).

Refunctionalization

Like many other big-step abstract machines [5, 61], the abstract machine of Section 6.3.3 is in defunctionalized form [70]: the reduction contexts, together with `normalize5_cont`, are the first-order counterpart of a function. This function is introduced with the data-type constructors `C0`, etc., and eliminated with calls to the dispatching function `normalize5_cont`. The higher-order counterpart of this abstract machine reads as follows:

```
(*  term * (value -> 'a) -> 'a  *)
fun normalize6_term_neg (VAR x, k)
    = k (LIT_nnf (NEGVAR x))
  | normalize6_term_neg (NEG t, k)
    = normalize6_term (t, k)
  | normalize6_term_neg (CONJ (t1, t2), k)
    = normalize6_term_neg (t1, fn v1 =>
        normalize6_term_neg (t2, fn v2 =>
          k (DISJ_nnf (v1, v2))))
  | normalize6_term_neg (DISJ (t1, t2), k)
    = normalize6_term_neg (t1, fn v1 =>
        normalize6_term_neg (t2, fn v2 =>
          k (CONJ_nnf (v1, v2))))

(*  term * (value -> 'a) -> 'a  *)
and normalize6_term (VAR x, k)
    = k (LIT_nnf (POSVAR x))
  | normalize6_term (NEG t, k)
    = normalize6_term_neg (t, k)
  | normalize6_term (CONJ (t1, t2), k)
    = normalize6_term (t1, fn v1 =>
        normalize6_term (t2, fn v2 =>
```

```
          k (CONJ_nnf (v1, v2))))
  | normalize6_term (DISJ (t1, t2), k)
    = normalize6_term (t1, fn v1 =>
        normalize6_term (t2, fn v2 =>
          k (DISJ_nnf (v1, v2))))

(* term -> result_or_wrong *)
fun normalize6 t = normalize6_term (t, fn v => RESULT v)
```

This normalization function is compositional over source terms: all recursive calls are over a proper subpart of the left-hand side.

Back to direct style

The refunctionalized definition of Section 6.3.4 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [56]. Its direct-style counterpart reads as follows:

```
(* term -> value *)
fun normalize7_term_neg (VAR x)
    = LIT_nnf (NEGVAR x)
  | normalize7_term_neg (NEG t)
    = normalize7_term t
  | normalize7_term_neg (CONJ (t1, t2))
    = DISJ_nnf (normalize7_term_neg t1, normalize7_term_neg t2)
  | normalize7_term_neg (DISJ (t1, t2))
    = CONJ_nnf (normalize7_term_neg t1, normalize7_term_neg t2)

(* term -> value *)
and normalize7_term (VAR x)
    = LIT_nnf (POSVAR x)
  | normalize7_term (NEG t)
    = normalize7_term_neg t
  | normalize7_term (CONJ (t1, t2))
    = CONJ_nnf (normalize7_term t1, normalize7_term t2)
  | normalize7_term (DISJ (t1, t2))
    = DISJ_nnf (normalize7_term t1, normalize7_term t2)

(* term -> result_or_wrong *)
fun normalize7 t = RESULT (normalize7_term t)
```

This normalization function is compositional over source terms.

Catamorphic normalizers

The compositional normalizer of Section 6.3.4 features two mutually recursive functions from terms to values. These two functions can be expressed as one, using the following type isomorphism:

$$(A \to B) \times (A \to B) \simeq A \to B^2$$

Representationally, this isomorphism can be exploited in two ways: by representing B^2 as $2 \to B$ and by representing B^2 as $B \times B$. Let us review each of these representations.

Representing B^2 as $2 \to B$ We first need a two-element type to account for the "polarity" of the current sub-term, i.e., whether the number of negations between the root of the given term and the current sub-term is even (in which case the polarity is positive) or it is odd (in which case the polarity is negative):

```
datatype polarity = PLUS
                  | MINUS
```

We are now in position to express the normalizer with one recursive descent over the given term, threading the current polarity in an inherited fashion, and returning a term in normal form:

```
(*  term -> (polarity -> value)  *)
fun normalize8_term (VAR x)
    = (fn PLUS  => LIT_nnf (POSVAR x)
        | MINUS => LIT_nnf (NEGVAR x))
  | normalize8_term (NEG t)
    = let val c = normalize8_term t
      in fn PLUS  => c MINUS
          | MINUS => c PLUS
      end
  | normalize8_term (CONJ (t1, t2))
    = let val c1 = normalize8_term t1
          val c2 = normalize8_term t2
      in fn PLUS  => CONJ_nnf (c1 PLUS , c2 PLUS)
          | MINUS => DISJ_nnf (c1 MINUS, c2 MINUS)
      end
  | normalize8_term (DISJ (t1, t2))
    = let val c1 = normalize8_term t1
          val c2 = normalize8_term t2
      in fn PLUS  => DISJ_nnf (c1 PLUS , c2 PLUS)
          | MINUS => CONJ_nnf (c1 MINUS, c2 MINUS)
      end

(*  term -> result_or_wrong  *)
fun normalize8 t = RESULT (normalize8_term t  PLUS)
```

Initially, the given term has a positive polarity.

To make it manifest that this normalizer is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `term_foldr`:

```
(*  term -> (polarity -> value)  *)
val normalize9_term
    = term_foldr
        (fn x => (fn PLUS  => LIT_nnf (POSVAR x)
                   | MINUS => LIT_nnf (NEGVAR x)),
         fn c => (fn PLUS  => c MINUS
                   | MINUS => c PLUS),
         fn (c1, c2) => (fn PLUS  => CONJ_nnf (c1 PLUS , c2 PLUS)
                          | MINUS => DISJ_nnf (c1 MINUS, c2 MINUS)),
         fn (c1, c2) => (fn PLUS  => DISJ_nnf (c1 PLUS,  c2 PLUS)
                          | MINUS => CONJ_nnf (c1 MINUS, c2 MINUS)))

(*  term -> result_or_wrong  *)
fun normalize9 t = RESULT (normalize9_term t PLUS)
```

Representing B^2 as $B \times B$ We use a pair holding a term in normal form and its dual. This pair puts us in position to express the normalizer with one recursive descent over the given term, returning a pair of terms in normal form in a synthesized fashion:

```
(*  term -> value * value *)
fun normalize10_term (VAR x)
    = (LIT_nnf (POSVAR x), LIT_nnf (NEGVAR x))
  | normalize10_term (NEG t)
```

```
    = let val (tp, tm) = normalize10_term t
      in (tm, tp)
      end
  | normalize10_term (CONJ (t1, t2))
    = let val (t1p, t1m) = normalize10_term t1
          val (t2p, t2m) = normalize10_term t2
      in (CONJ_nnf (t1p, t2p), DISJ_nnf (t1m, t2m))
      end
  | normalize10_term (DISJ (t1, t2))
    = let val (t1p, t1m) = normalize10_term t1
          val (t2p, t2m) = normalize10_term t2
      in (DISJ_nnf (t1p, t2p), CONJ_nnf (t1m, t2m))
      end

(* term -> result_or_wrong *)
fun normalize10 t
    = let val (tp, tm) = normalize10_term t
      in RESULT tp
      end
```

The final result is the positive component of the resulting pair.

To make it manifest that this normalization function is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `term_foldr`:

```
(* term -> value * value *)
val normalize11_term
    = term_foldr
        (fn x
            => (LIT_nnf (POSVAR x), LIT_nnf (NEGVAR x)),
         fn (tp, tm)
            => (tm, tp),
         fn ((t1p, t1m), (t2p, t2m))
            => (CONJ_nnf (t1p, t2p), DISJ_nnf (t1m, t2m)),
         fn ((t1p, t1m), (t2p, t2m))
            => (DISJ_nnf (t1p, t2p), CONJ_nnf (t1m, t2m)))

(* term -> result_or_wrong *)
fun normalize11 t
    = let val (tp, tm) = normalize11_term t
      in RESULT tp
      end
```

6.3.5 Summary and conclusion

We have refocused the reduction-based normalization function of Section 6.3.2 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions that all are inter-derivable.

We wish to emphasize that the starting point (the small-step reduction semantics) and the ending point (the big-step compositional normalization functions) are natural ones, i.e., undergraduate teaching material in functional programming. Proving that they are equivalent, on the other hand, is non trivial. In fact, the proof techniques involved to establish this equivalence form the core of Nielson and Nielson's popular textbook *Semantics with Applications* [154]. An additional advantage of the present calculational method is that it scales. To substantiate this claim, we invite the reader to pause and sketch a negational normalizer that implements a *leftmost innermost* strategy instead of the leftmost

outermost strategy considered in this section. Rendez-vous then with the accompanying Standard ML code for a complete derivation of such negational normalizers.

6.4 Leftmost outermost conjunctive normalization

In this section, we consider conjunctive normal forms. We go from a leftmost-outermost *reduction* strategy to the corresponding leftmost-outermost *evaluation* strategy. Unlike the reduction rules for negational normalization, the rules for conjunctive normalization do not allow for a straightforward transformation. We therefore begin this section by adjusting the reduction rules (Section 6.4.0). We then implement the reduction strategy for the new rules (Section 6.4.1) as a prequel to implementing the corresponding reduction semantics (Section 6.4.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 6.4.3) and to the functional correspondence between abstract machines and normalization functions (Section 6.4.4).

6.4.0 Generalized reduction

Consider the reduction rules provided by the distributivity laws presented in Section 6.1:

$$t_{nnf1} \vee (t_{nnf2} \wedge t_{nnf3}) \rightarrow (t_{nnf1} \vee t_{nnf2}) \wedge (t_{nnf1} \vee t_{nnf3})$$
$$(t_{nnf1} \wedge t_{nnf2}) \vee t_{nnf3} \rightarrow (t_{nnf1} \vee t_{nnf3}) \wedge (t_{nnf2} \vee t_{nnf3})$$

Unlike the reduction rules for negational normalization, these two rules overlap in two ways:

1. Redexes overlap because a term can be an instance of several redexes. For example, the term $(t_{nnf1} \wedge t_{nnf2}) \vee (t_{nnf3} \wedge t_{nnf4})$ can be reduced by either rule. In the present case, this critical pair results in a non-confluent rewrite system where normal forms are not unique: the order in which the reduction rules are applied matters, and depending on this order, distinct normal forms can be obtained.

2. Redexes and contractums overlap because when a contractum is a conjunction and occurs as an immediate subterm of a disjunction, this disjunction forms a new redex. For example, the term $t_{nnf1} \vee (t_{nnf2} \vee (t_{nnf3} \wedge t_{nnf4}))$ reduces in one step to $t_{nnf1} \vee ((t_{nnf2} \vee t_{nnf3}) \wedge (t_{nnf2} \vee t_{nnf4}))$ where the contractum (i.e., the conjunction) and the top-most disjunction now form the next redex to be contracted.

   This situation is due to an overlap between the left-hand sides and right-hand sides of the reduction rules where a right-hand side can occur as a sub-pattern of a left-hand side. Such overlaps can make the next redex be spread across the current context and the current contractum. This next redex would be found by recomposing the current reduction context around the contractum and decomposing the resulting reduct. It would, however, *not be found* by decomposing the contractum in the current context.

   Overlaps of this kind therefore make refocusing unapplicable, unlike in Sections 6.3 and in all the case studies of refocusing that the authors are aware of.

We eliminate these overlaps as follows:

1. To eliminate overlapping redexes, we restrict the lefthand side of the first reduction rule to $t_{cnf1} \vee (t_{nnf2} \wedge t_{nnf3})$, thereby making the reduction strategy go from left to right.

2. To eliminate overlaps between left-hand sides and right-hand sides, we first note that the form of a conjunction in the context of disjunctions is $D[t_{nnf1} \wedge t_{nnf2}]$, where D is a context of (restricted) disjunctions defined by:

$$D ::= [\,] \mid D \vee t_{nnf} \mid t_{dnf} \vee D$$

   Using this observation, we can generalize the reduction rules to eliminate the construction of new redexes:

$$D[t_{nnf1} \wedge t_{nnf2}] \rightarrow D[t_{nnf1}] \wedge D[t_{nnf2}] \quad \text{where } D \neq [\,]$$

In effect, the generalized contraction hereditarily pulls out a conjunction in one single step, thereby avoiding the construction of intermediate redexes.

We now prove that leftmost outermost reduction by this generalized reduction rule is sound with respect to leftmost outermost reduction by the original two reduction rules. We use the notation $t \mapsto t'$ to denote that t reduces to t' by reducing the leftmost outermost redex of t using the original reduction rules, and $\mapsto^*$ to denote the reflexive and transitive closure of $\mapsto$. Also, we use $t \rightarrowtail t'$ to denote that t reduces to t' by reducing the leftmost outermost redex of t using the generalized rule:

Proposition 23 (soundness)**.** *Let $r = D[t_{nnf1} \wedge t_{nnf2}]$, and let $t = C[r]$, where C is any context such that r is the leftmost outermost redex of t by the generalized reduction rule, i.e., $t \rightarrowtail C[D[t_{nnf1}] \wedge D[t_{nnf2}]]$. Then $t \mapsto^* C[D[t_{nnf1}] \wedge D[t_{nnf2}]]$.*

Proof. We first note that since r is the leftmost outermost redex in t, we can limit our proof to the case where $C = [\,]$ without loss of generality. We proceed by induction on D:

Case $D = [\,]$ By reflexivity of $\mapsto^*$.

Case $D = D_0[[\,] \vee t_{nnf}]$

$$\begin{array}{ll} & t \\ = & D_0[(t_{nnf1} \wedge t_{nnf2}) \vee t_{nnf}] \\ \mapsto & D_0[(t_{nnf1} \vee t_{nnf}) \wedge (t_{nnf2} \vee t_{nnf})] \\ \mapsto^* & D_0[t_{nnf1} \vee t_{nnf}] \wedge D_0[t_{nnf2} \vee t_{nnf}] \\ = & C[D[t_{nnf1}] \wedge D[t_{nnf2}]] \end{array}$$

Case $D = D_0[t_{dnf} \vee [\,]]$

$$\begin{array}{ll} & t \\ = & D_0[t_{cnf} \vee (t_{nnf1} \wedge t_{nnf2})] \\ \mapsto & D_0[(t_{dnf} \vee t_{nnf1}) \wedge (t_{dnf} \vee t_{nnf2})] \\ \mapsto^* & D_0[t_{dnf} \vee t_{nnf1}] \wedge D_0[t_{dnf} \vee t_{nnf2}] \\ = & C[D[t_{nnf1}] \wedge D[t_{nnf2}]] \end{array}$$

Note that we only need to apply the induction hypothesis when D_0 is non-empty, in which case it is indeed applied to the leftmost outermost redex of the term. □

With this adjusted reduction rule, we can proceed as in Sections 6.3.

6.4.1 Prelude to a reduction semantics

The reduction strategy induces a notion of value and of potential redex (i.e., of a term that is an actual redex or that is stuck); we are then in position to state a compositional search function that implements the reduction strategy and maps a given term either to the corresponding value, if it is in normal form, or to a potential redex. Reflecting the stratification of conjunctive normal forms described in Sections 6.1 and 6.2, we specify the search for a potential redex in two stages. First, we search for the leftmost-outermost conjunction inside a disjunction. From this search function, we derive a decomposition function mapping a given term either to the corresponding value, if it is in the grammar of t_{dnf}, or to the leftmost-outermost conjunction and its context of disjunctions (Section 6.4.1). Second, we search for the leftmost-outermost disjunction containing a conjunction. From this search function, we derive a decomposition function mapping a given term either to the corresponding value, if it is in the grammar of t_{cnf}, or to a potential redex and its reduction context (Section 6.4.1). As a corollary, we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 6.4.1).

Prequel to the reduction strategy

Under the assumption that there is a surrounding disjunction, we must locate the leftmost-outermost conjunction. A value therefore is a term where there are no conjunctions, i.e., a term in the grammar t_{dnf}:

```
datatype found_d = VAL_d     of disj_cnf
                 | LMOM_CONJ of term_nnf * term_nnf
```

The following implements the reduction strategy as a compositional search function. It searches for a conjunction depth-first and from left to right:

```
(*  term_nnf -> found_d  *)
fun search_term_d (LIT_nnf x)
    = VAL_d (DISJ_leaf x)
  | search_term_d (CONJ_nnf (t1, t2))
    = LMOM_CONJ (t1, t2)
  | search_term_d (DISJ_nnf (t1, t2))
    = (case search_term_d t1
        of (VAL_d d1)
           => (case search_term_d t2
                of (VAL_d d2)
                   => VAL_d (DISJ_node (d1, d2))
                 | (LMOM_CONJ conj)
                   => LMOM_CONJ conj)
         | (LMOM_CONJ conj)
           => LMOM_CONJ conj)

(*  term_nnf -> found_d *)
fun search_d t = search_term_d t
```

From searching to decomposing As in Section 6.3.1, we transform the search function into a decomposition function. We do so by (1) CPS-transforming the search function, (2) defunctionalizing its continuation,

```
datatype cont_d = D0
                | D1 of cont_d * term_nnf
                | D2 of disj_cnf * cont_d
```

and (3) returning a conjunction (if one exists) and its context of disjunctions:

```
datatype value_or_decomposition_d = VAL_d of disj_cnf
                                  | DEC_d of term_nnf * term_nnf * cont_d

(*  cont_d * disj_cnf -> value_or_decomposition_d  *)
fun decompose_cont_d (D0, d)
    = VAL_d d
  | decompose_cont_d (D1 (k, t2), d1)
    = decompose_term_d (t2, D2 (d1, k))
  | decompose_cont_d (D2 (d1, k), d2)
    = decompose_cont_d (k, DISJ_node (d1, d2))

(*  term_nnf * cont_d -> value_or_decomposition_d  *)
and decompose_term_d (LIT_nnf x, k)
    = decompose_cont_d (k, DISJ_leaf x)
  | decompose_term_d (CONJ_nnf (t1, t2), k)
    = DEC_d (t1, t2, k)
  | decompose_term_d (DISJ_nnf (t1, t2), k)
    = decompose_term_d (t1, D1 (k, t2))

(*  term_nnf -> value_or_decomposition_d  *)
fun decompose_d t = decompose_term_d (t, D0)
```

The reduction strategy

The reduction strategy consists in locating the leftmost-outermost conjunction that is directly below a disjunction. A value therefore is a term where disjunctions contain no conjunctions, i.e., a conjunctive normal form:

```
type value = term_cnf
```

A potential redex is the distribution of disjunctions over a conjunction:

```
datatype potential_redex = PR_DISTR_DISJ of term_nnf * term_nnf * cont_d
```

The following compositional search function implements the reduction strategy. It searches for a potential redex depth-first and from left to right:

```
datatype found_c = VAL_c    of value
                 | POTRED_c of potential_redex

(*  term_nnf -> found_c  *)
fun search_term_c (LIT_nnf x)
    = VAL_c (CONJ_leaf (DISJ_leaf x))
  | search_term_c (CONJ_nnf (t1, t2))
    = (case search_term_c t1
        of (VAL_c c1)
           => (case search_term_c t2
                of (VAL_c c2)
                   => VAL_c (CONJ_node (c1, c2))
                 | (POTRED_c pr)
                   => POTRED_c pr)
         | (POTRED_c pr)
           => POTRED_c pr)
  | search_term_c (DISJ_nnf (t1, t2))
```

```
    = (case decompose_d (DISJ_nnf (t1, t2))
        of (VAL_d d)
           => VAL_c (CONJ_leaf d)
         | (DEC_d (t1, t2, D))
           => POTRED_c (PR_DISTR_DISJ (t1, t2, D)))

(*  term_nnf -> found_c  *)
fun search_c t = search_term_c t
```

From searching to decomposing As in Section 6.3.1, we transform the search function into a decomposition function. We do so by (1) CPS-transforming the search function, (2) defunctionalizing its continuation,

```
datatype cont_c = C0
                | C1 of cont_c * term_nnf
                | C2 of conj_cnf * cont_c
```

and (3) returning a potential redex (if one exists) and its reduction context:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * cont_c

(*  cont_d * disj_cnf * cont_c -> value_or_decomposition  *)
fun decompose_cont_d (D0, d, C)
    = decompose_cont_c (C, CONJ_leaf d)
  | decompose_cont_d (D1 (D, t2), d1, C)
    = decompose_term_d (t2, D2 (d1, D), C)
  | decompose_cont_d (D2 (d1, D), d2, C)
    = decompose_cont_d (D, DISJ_node (d1, d2), C)

(*  term_nnf * cont_d * cont_c -> value_or_decomposition  *)
and decompose_term_d (LIT_nnf x, D, C)
    = decompose_cont_d (D, DISJ_leaf x, C)
  | decompose_term_d (CONJ_nnf (t1, t2), D, C)
    = DEC (PR_DISTR_DISJ (t1, t2, D), C)
  | decompose_term_d (DISJ_nnf (t1, t2), D, C)
    = decompose_term_d (t1, D1 (D, t2), C)

(*  cont_c * conj_cnf -> value_or_decomposition  *)
and decompose_cont_c (C0, c)
    = VAL c
  | decompose_cont_c (C1 (C, t2), c1)
    = decompose_term_c (t2, C2 (c1, C))
  | decompose_cont_c (C2 (c1, C), c2)
    = decompose_cont_c (C, CONJ_node (c1, c2))

(*  term_nnf * cont_c -> value_or_decomposition  *)
and decompose_term_c (LIT_nnf x, C)
    = decompose_cont_c (C, CONJ_leaf (DISJ_leaf x))
  | decompose_term_c (CONJ_nnf (t1, t2), C)
    = decompose_term_c (t1, C1 (C, t2))
  | decompose_term_c (t as DISJ_nnf (t1, t2), C)
    = decompose_term_d (t, D0, C)

(*  term_nnf -> value_or_decomposition  *)
fun decompose t = decompose_term_c (t, C0)
```

Recomposition

A reduction context is recomposed around a term with a left fold over this context:

```
(*  cont_d * term_nnf -> term_nnf  *)
fun recompose_d (D0, t)
    = t
  | recompose_d (D1 (D, t2), t1)
    = recompose_d (D, DISJ_nnf (t1, t2))
  | recompose_d (D2 (d1, D), t2)
    = recompose_d (D, DISJ_nnf (embed_cnf (CONJ_leaf d1), t2))

(*  cont_c * term_nnf -> term_nnf  *)
fun recompose_c (C0, t)
    = t
  | recompose_c (C1 (C, t2), t1)
    = recompose_c (C, CONJ_nnf (t1, t2))
  | recompose_c (C2 (c1, C), t2)
    = recompose_c (C, CONJ_nnf (embed_cnf c1, t2))

(*  cont_c * cont_d * term_nnf -> term_nnf  *)
fun recompose (C, D, t) = recompose_c (C, recompose_d (D, t))
```

6.4.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for conjunctive normalization.

Notion of reduction

The reduction rule is implemented as:

```
datatype contractum_or_error = CONTRACTUM of term_nnf
                             | ERROR of string

(*  potential_redex -> contractum_or_error  *)
fun contract (PR_DISTR_DISJ (t1, t2, D))
    = CONTRACTUM (CONJ_nnf (recompose_d (D, t1), recompose_d (D, t2)))
```

In the present case, all potential redexes are actual ones, i.e., no terms are stuck.

One-step reduction

Given a non-value term, a one-step reduction function (1) decomposes this non-value term into a potential redex and a reduction context, (2) contracts the potential redex if it is an actual one, and (3) recomposes the reduction context with the contractum. If the potential redex is not an actual one, reduction is stuck. Given a value term, reduction is also stuck:

```
datatype reduct_or_stuck = REDUCT of term_nnf
                         | STUCK  of string

(*  term_nnf -> reduct_or_stuck *)
fun reduce t
    = (case decompose t
```

```
       of (VAL v)
          => STUCK "irreducible term"
        | (DEC (pr, C))
          => (case contract pr
               of (CONTRACTUM t') => REDUCT (recompose_c (C, t'))
                | (ERROR s)       => STUCK s))
```

Reduction-based normalization

A reduction-based normalization function is one that iterates the one-step reduction function until it yields a value or becomes stuck. If it yields a value, this value is the result of evaluation, and if it becomes stuck, evaluation goes wrong:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string
```

The following definition uses `decompose` to distinguish between value and non-value terms:

```
(*  value_or_decomposition -> result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, C))
    = (case contract pr
        of (CONTRACTUM t') => iterate (decompose (recompose_c (C, t')))
         | (ERROR s)       => WRONG s)

(*  term_nnf -> result_or_wrong  *)
fun normalize t = iterate (decompose t)
```

6.4.3 From reduction-based to reduction-free normalization

This section follows the steps of Section 6.3.3: we refocus the reduction-based normalization function of Section 6.4.2, inline the contraction function, fuse the resulting small-step abstract machine into a big-step one, and compress its corridor transitions. The result reads as follows:

```
datatype cont_d = D0
                | D1 of cont_d * term_nnf
                | D2 of disj_cnf * cont_d

datatype cont_c = C0
                | C1     of cont_c * term_nnf
                | C2     of conj_cnf * cont_c
                | C1_rec of cont_c * cont_d * term_nnf

(*  cont_d * disj_cnf * cont_c -> result_or_wrong  *)
fun normalize4_cont_d (D0, d, C)
    = normalize4_cont_c (C, CONJ_leaf d)
  | normalize4_cont_d (D1 (D, t2), d1, C)
    = normalize4_term_d (t2, D2 (d1, D), C)
  | normalize4_cont_d (D2 (d1, D), d2, C)
    = normalize4_cont_d (D, DISJ_node (d1, d2), C)

(*  term_nnf * cont_d * cont_c -> result_or_wrong  *)
and normalize4_term_d (LIT_nnf x, D, C)
    = normalize4_cont_d (D, DISJ_leaf x, C)
```

```
  | normalize4_term_d (CONJ_nnf (t1, t2), D, C)
    = normalize4_term_d (t1, D, C1_rec (C, D, t2))
  | normalize4_term_d (DISJ_nnf (t1, t2), D, C)
    = normalize4_term_d (t1, D1 (D, t2), C)

(*  cont_c * conj_cnf -> result_or_wrong *)
and normalize4_cont_c (C0, c)
    = RESULT c
  | normalize4_cont_c (C1 (C, t2), c1)
    = normalize4_term_c (t2, C2 (c1, C))
  | normalize4_cont_c (C2 (c1, C), c2)
    = normalize4_cont_c (C, CONJ_node (c1, c2))
  | normalize4_cont_c (C1_rec (C, D, t2), c1)
    = normalize4_term_d (t2, D, C2 (c1, C))

(*  term_nnf * cont_c -> result_or_wrong  *)
and normalize4_term_c (LIT_nnf x, C)
    = normalize4_cont_c (C, CONJ_leaf (DISJ_leaf x))
  | normalize4_term_c (CONJ_nnf (t1, t2), C)
    = normalize4_term_c (t1, C1 (C, t2))
  | normalize4_term_c (DISJ_nnf (t1, t2), C)
    = normalize4_term_d (t1, D1 (D0, t2), C)

(*  term_nnf -> result_or_wrong  *)
fun normalize4 t = normalize4_term_c (t, C0)
```

6.4.4 From abstract machines to normalization functions

The big-step abstract machine of Section 6.4.3 is in defunctionalized form. Its higher-order counterpart reads as follows.

```
(*  term_nnf * (disj_cnf * (value -> 'a) -> 'a) * (value -> 'a) -> 'a  *)
fun normalize5_term_d (LIT_nnf x, k, mk)
    = k (DISJ_leaf x, mk)
  | normalize5_term_d (CONJ_nnf (t1, t2), k, mk)
    = normalize5_term_d (t1, k, fn c1 =>
        normalize5_term_d (t2, k, fn c2 =>
          mk (CONJ_node (c1, c2))))
  | normalize5_term_d (DISJ_nnf (t1, t2), k, mk)
    = normalize5_term_d (t1, fn (d1, mk') =>
        normalize5_term_d (t2, fn (d2, mk'') =>
           k (DISJ_node (d1, d2), mk''), mk'), mk)

(*  term_nnf * (value -> 'a) -> 'a  *)
fun normalize5_term_c (LIT_nnf x, mk)
    = mk (CONJ_leaf (DISJ_leaf x))
  | normalize5_term_c (CONJ_nnf (t1, t2), mk)
    = normalize5_term_c (t1, fn c1 =>
        normalize5_term_c (t2, fn c2 =>
          mk (CONJ_node (c1, c2))))
  | normalize5_term_c (DISJ_nnf (t1, t2), mk)
    = normalize5_term_d (DISJ_nnf (t1, t2), fn (d, mk') =>
        mk' (CONJ_leaf d), mk)

(*  term_nnf -> result_or_wrong  *)
fun normalize5 t = normalize5_term_c (t, RESULT)
```

This normalization function is in continuation-passing style. Its direct-style counterpart reads as follows:

```
(*  term_nnf * (disj_cnf -> conj_cnf) -> conj_cnf  *)
fun normalize6_term_d (LIT_nnf x, k)
    = k (DISJ_leaf x)
  | normalize6_term_d (CONJ_nnf (t1, t2), k)
    = CONJ_node (normalize6_term_d (t1, k), normalize6_term_d (t2, k))
  | normalize6_term_d (DISJ_nnf (t1, t2), k)
    = normalize6_term_d (t1, fn d1 =>
        normalize6_term_d (t2, fn d2 =>
          k (DISJ_node (d1, d2))))

(*  term_nnf -> value  *)
fun normalize6_term_c (LIT_nnf x)
    = CONJ_leaf (DISJ_leaf x)
  | normalize6_term_c (CONJ_nnf (t1, t2))
    = CONJ_node (normalize6_term_c t1, normalize6_term_c t2)
  | normalize6_term_c (DISJ_nnf (t1, t2))
    = normalize6_term_d (DISJ_nnf (t1, t2), fn d => CONJ_leaf d)

(*  term_nnf -> result_or_wrong  *)
fun normalize6 t = RESULT (normalize6_term_c t)
```

In this normalization function, `normalize_term_d` is expressed in the "continuation-composing style" characteristic of functional backtracking. It is called in the clause for disjunctions, in the definition of `normalize_term_c`, with an initial continuation. In the clauses for literals and disjunctions, `normalize_term_d` is in the ordinary continuation-passing style, where all calls are tail calls. In the clause for conjunctions, however, there are two non-tail calls to `normalize_term_d`. In direct style, this programming idiom is captured with the delimited control operators shift and reset [64].

6.4.5 Delimited continuations in direct style

In this section, we use Filinski's encoding of shift and reset in ML [90]:

```
val shift : (('a -> value) -> value) -> 'a
val reset : (unit -> value) -> value
```

The control delimiter `reset` is used to initialize the continuation. The control operator `shift` is used to capture the current continuation, as delimited by a surrounding occurrence of `reset`.

The direct-style counterpart of the normalization function of Section 6.4.4 reads as follows:

```
(*  term_nnf/value -> disj_cnf/value  *)
fun normalize7_term_d (LIT_nnf x)
    = DISJ_leaf x
  | normalize7_term_d (CONJ_nnf (t1, t2))
    = shift (fn k =>
        CONJ_node (reset (fn () => k (normalize7_term_d t1)),
                   reset (fn () => k (normalize7_term_d t2))))
  | normalize7_term_d (DISJ_nnf (t1, t2))
    = DISJ_node (normalize7_term_d t1, normalize7_term_d t2)

(*  term_nnf/'a -> value/'a  *)
fun normalize7_term_c (LIT_nnf x)
    = CONJ_leaf (DISJ_leaf x)
  | normalize7_term_c (CONJ_nnf (t1, t2))
    = CONJ_node (normalize7_term_c t1, normalize7_term_c t2)
  | normalize7_term_c (DISJ_nnf (t1, t2))
```

```
    = reset (fn () => CONJ_leaf (normalize7_term_d (DISJ_nnf (t1, t2))))

(* term_nnf -> result_or_wrong *)
fun normalize7 t = RESULT (normalize7_term_c t)
```

In this normalization function, `normalize_term_d` is now expressed in direct style. Its call in the clause for disjunctions, in the definition of `normalize_term_c`, is now delimited with an occurrence of `reset`. In the clauses for literals and disjunctions, `normalize_term_d` is in ordinary direct style. In the clause for conjunctions, however, an occurrence of `shift` captures the current (delimited) continuation and duplicates it by applying it twice inside the conjunction, thereby realizing the duplication of contexts in the generalized distributivity law.

6.4.6 Summary and conclusion

We have first identified that the reduction rules implementing the distribution of disjunctions over conjunctions suffer from overlaps that make it impossible to refocus the corresponding reduction-based normalization function. We have therefore adjusted them. Then, taking the very same steps as in Section 6.3, we have refocused the reduction-based normalization function of Section 6.4.2 into a small-step abstract machine, and we have exhibited a family of corresponding reduction-free normalization functions that all are inter-derivable. It is our observation that the resulting higher-order normalization functions use delimited continuations, which we have exemplified by using the delimited-control operators shift and reset.

Ever since Wand's foundational article on continuation-based program-transformation strategies [217], converting a formula into conjunctive normal form is a classic among continuation aficionados. We are adding two stones to this monument:

1. our big-step, reduction-free normalization function is not designed per se; it is systematically calculated from a small-step, reduction-based normalization function; and

2. it shows that delimited continuations form a natural expressive medium to carry out the distribution law in the big-step normalization function.

6.5 Conclusion and perspectives

The inter-derivations illustrated here witness a striking unity of computation across reduction semantics, abstract machines, and normalization functions: they all truly define the same elephant, so to speak. The structural coincidence between reduction contexts and evaluation contexts as defunctionalized continuations, in particular, plays a key rôle to connect reduction strategies and evaluation strategies, a connection that was first established by Plotkin [169] and that scales to delimited continuations. As for Ohori and Sasano's lightweight fusion [155], it provides the linchpin between the functional representations of small-step and big-step operational semantics [67]. Overall, the inter-derivations illustrate the conceptual value of semantics-based program manipulation, as promoted at PEPM ever since its inception.

Acknowledgements. We are grateful to Jeremy Siek and Siau-Cheng Khoo for their invitation to present a preliminary version of this work at the 20th anniversary of PEPM [76].

The example of negational normalization originates in a joint work of the first and second authors [65]. The two prequels to a reduction semantics originate in a joint work of the first and third authors.

Chapter 7

Storeless call-by-need evaluation

This chapter appeared in [77]: Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science*, 435:21–42, 2012.

An earlier version appeared in [75]: Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer.

Abstract

Starting from the standard call-by-need reduction for the λ-calculus that is common to Ariola, Felleisen, Maraist, Odersky, and Wadler, we inter-derive a series of hygienic semantic artifacts: a reduction-free storeless abstract machine, a continuation-passing evaluation function, and what appears to be the first heapless natural semantics for call-by-need evaluation. Furthermore we observe that the evaluation function implementing this natural semantics is in defunctionalized form. The refunctionalized counterpart of this evaluation function implements an extended direct semantics in the sense of Cartwright and Felleisen.

Overall, the semantic artifacts presented here are simpler than many other such artifacts that have been independently worked out, and which require ingenuity, skill, and independent soundness proofs on a case-by-case basis. They are also simpler to inter-derive because the inter-derivational tools (e.g., refocusing and defunctionalization) already exist.

7.1 Introduction

A famous functional programmer once was asked to give an overview talk. He began with "This talk is about lazy functional programming and call by need." and paused. Then, quizzically looking at the audience, he quipped: "Are there any questions?" There were some, and so he continued: "Now listen very carefully, I shall say this only once."

This apocryphal story illustrates *demand-driven computation* and *memoization of intermediate results*, two key features that have elicited a fascinating variety of semantic specifications and implementation techniques over the years, ranging from purely syntactic treatments to mutable state, and featuring small-step operational semantics [12, 141], a range of abstract machines [96, 99, 195], big-step operational semantics [7, 134], as well as evaluation functions [113, 122].

In this article, we extract the computational content of the standard call-by-need reduction for the λ-calculus that is common to both Ariola and Felleisen [12] and Maraist, Odersky, and Wadler [141]. This computational content takes the forms of a one-step reduction relation, an abstract machine, and a natural semantics that are mutually compatible and all abide by Barendregt's variable convention [22, page 26]. Traditionally, one could either handcraft each of these semantic artifacts from scratch and then prove a series of soundness theorems, or invent a calculation to go from artifact to artifact and prove the correctness of the calculation on the way. We depart from these two traditions by going from artifact to artifact using a pre-defined series of fully correct transformations, following the programme outlined in the first author's invited talk at ICFP 2008 [62]. To this programme, though, we add one new refunctionalization step that is specific to call by need. The inter-derivation is itemized as follows:

0. for starters, we make the contraction rules explicitly hygienic to make the standard one-step reduction preserve Barendregt's variable convention;

1. iterating this hygienic standard one-step reduction yields a standard reduction-based evaluation, which we refocus [71] to obtain a reduction-free evaluation with the same built-in hygiene; this reduction-free evaluation takes the form of an abstract machine and is correct by construction; we simplify this hygienic abstract machine by hereditarily compressing its corridor transitions.

We then change perspective and instead of considering this abstract machine as a small-step entity defining a relation, we consider it as a big-step entity defining a function:

2. we refunctionalize [69] the simplified hygienic abstract machine of Item 1 into a continuation-passing evaluation function, which we write back to direct style, obtaining a functional program that is correct by construction and that implements a heapless natural semantics with the same built-in hygiene;

3. in addition, we observe that the evaluation function implementing this hygienic natural semantics is in defunctionalized form [70], and we present the corresponding higher-order evaluation function.

Overview We start with a call-by-name semantics of the λ_{let}-calculus (Section 7.2). This reduction semantics provides a syntactic account of demand-driven computation. Extending this syntactic account with the memoization of intermediate results yields Ariola et al.'s call-by-need semantics of the λ_{let}-calculus (Section 7.3). This reduction semantics is deceivingly concise: in the first half of this article (Section 7.4), we methodically analyze it, considering in turn its potential redexes (Section 7.4.1), its (lack of) hygiene (Section 7.4.2), its evaluation contexts (Section 7.4.3), the recomposition of its evaluation contexts around a term (Section 7.4.4), its decomposition of a non-answer term into a potential redex and its evaluation context according to the reduction strategy (Section 7.4.5), its contraction rules (Section 7.4.6), its standard one-step reduction

(Section 7.4.7), and its standard reduction-based evaluation (Section 7.4.8). The extensional properties such as unique decomposition, standardization, and hygiene ensure the existence of a deterministic evaluator extensionally. However, it is our thesis that they also provide precious intensional guidelines. We illustrate this thesis in the second half of this article (Sections 7.5 to 7.8): from the reduction semantics, we mechanically derive an abstract machine (Section 7.5), from this abstract machine, we mechanically derive a natural semantics (Sections 7.7.1 and 7.7.2), and from this natural semantics we mechanically derive a higher-order evaluation function (Section 7.7.3).

The ML code of the entire derivation is available from the last author's web page.[1]

Prerequisites We assume a degree of familiarity with the formats of operational semantics – specifically reduction semantics, abstract machines, and natural semantics – though no more as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [61].

7.2 The standard call-by-name reduction for the λ-calculus

Let us start with demand-driven computation and the standard reduction corresponding to call by name. The call-by-name reduction semantics for the λ_{let}-calculus reads as follows:

Definition 24 (call-by-name λ_{let}-calculus)**.**

Syntax:

$$\begin{array}{rl} \text{Var} \ni & x \\ \text{Term} \ni T ::= & x \mid \lambda x.T \mid T\ T \mid \text{let } x \text{ be } T \text{ in } T \\ \text{Value} \ni V ::= & \lambda x.T \\ \text{Answer} \ni A ::= & V \mid \text{let } x \text{ be } T \text{ in } A \\ \text{Evaluation Context} \ni E ::= & [\,] \mid E\ T \mid \text{let } x \text{ be } T \text{ in } E \end{array}$$

Contraction rules:

$$\begin{array}{ll} (I) & (\lambda x.T)\ T_1 \to \text{let } x \text{ be } T_1 \text{ in } T \\ (N) & \text{let } x \text{ be } T \text{ in } E[x] \to \text{let } x \text{ be } T \text{ in } E[T] \\ (C) & (\text{let } x \text{ be } T_1 \text{ in } A)\ T_2 \to \text{let } x \text{ be } T_1 \text{ in } A\ T_2 \end{array}$$

In words:

- Programs are closed λ-terms with no let expressions.
- Terms are pure λ-terms with non-recursive let expressions. (We follow the tradition of referring to λ-declared and let-declared denotables as "variables" even though they do not vary.)
- Values are λ-abstractions.
- Answers are let expressions nested around a value.

[1] http://www.zerny.dk/def-int-for-call-by-need.html

- Evaluation contexts are terms with a hole that are constructed inductively. The notation "$E[T]$" stands for a term that decomposes into an evaluation context E and a term T. Evaluation contexts specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-name reduction strategy.

Each contraction rule maps a redex to a contractum:

- Rule (I) introduces a let binding from an application, in a way akin to let insertion in partial evaluation [36].
- Rule (N) hygienically substitutes a definiens (here: a term) for the occurrence of a let-declared variable arising in an evaluation context. There may be more than one occurrence of the variable in the context. These other occurrences are not substituted.
- Rule (C) allows let bindings to commute with applications, hygienically, i.e., renaming what needs to be renamed so that no free variable is captured.

Reduction is then defined in terms of evaluation contexts and contraction. A term T_0 reduces to T_1 if there exists an evaluation context E, a redex T_0' and a contractum T_1' such that $T_0 = E[T_0']$, $(T_0', T_1') \in (I) \cup (N) \cup (C)$, and $T_1 = E[T_1']$. The following reduction sequence (one reduct per line) illustrates the demand-driven aspect of call by name as well as the duplication of work it entails. We note one-step reduction with $\mapsto_{\text{name}}$ and annotate each reduction step with the name of the corresponding contraction rule:

$$
\begin{array}{ll}
(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x)) \mapsto_{\text{name}} & (I) \\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \boxed{z}\ z} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \underline{((\lambda y.y)\ (\lambda x.x))\ z} \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ (\underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{y}})\ z \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \underline{(\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)\ z} \mapsto_{\text{name}} & (C) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\lambda x.x)\ z} \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \boxed{x}} \mapsto_{\text{name}} & (N) \\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \boxed{z}} \mapsto_{\text{name}} & (N) \\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ (\lambda y.y)\ (\lambda x.x)} \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{y}} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x} &
\end{array}
$$

At every step, we have explicitly decomposed each reduct into a redex (underlined) and its evaluation context (not underlined). Each (N) contraction is triggered by a demand over a variable: we have shaded the occurrence of this variable. Each of the two shaded occurrences of z forces the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is not memoized.

7.3 The standard call-by-need reduction for the λ-calculus

Let us supplement demand-driven computation with the memoization of intermediate results to obtain the standard reduction corresponding to call by need. The following call-by-need reduction semantics for the λ_{let}-calculus is common to Ariola, Felleisen, Maraist,

Odersky, and Wadler's articles [12, 15, 141], renaming non-terminals for notational uniformity:

Definition 25 (call-by-need λ_{let}-calculus [15, Figure 3])**.**

Syntax:

$$\begin{aligned}
\text{Var} &\ni x \\
\text{Term} &\ni T ::= x \mid \lambda x.T \mid T\ T \mid \text{let } x \text{ be } T \text{ in } T \\
\text{Value} &\ni V ::= \lambda x.T \\
\text{Answer} &\ni A ::= V \mid \text{let } x \text{ be } T \text{ in } A \\
\text{Evaluation Context} &\ni E ::= [\,] \mid E\ T \mid \text{let } x \text{ be } T \text{ in } E \mid \text{let } x \text{ be } E \text{ in } E[x]
\end{aligned}$$

Contraction rules:

$$\begin{array}{ll}
(I) & (\lambda x.T)\ T_1 \to \text{let } x \text{ be } T_1 \text{ in } T \\
(V) & \text{let } x \text{ be } V \text{ in } E[x] \to \text{let } x \text{ be } V \text{ in } E[V] \\
(C) & (\text{let } x \text{ be } T_1 \text{ in } A)\ T_2 \to \text{let } x \text{ be } T_1 \text{ in } A\ T_2 \\
(A) & \text{let } x \text{ be let } y \text{ be } T_1 \text{ in } A \text{ in } E[x] \to \text{let } y \text{ be } T_1 \text{ in let } x \text{ be } A \text{ in } E[x]
\end{array}$$

In words:

- Programs are closed λ-terms with no let expressions.
- Terms are pure λ-terms with non-recursive let expressions.
- Values are λ-abstractions.
- Answers are let expressions nested around a value.
- Evaluation contexts are terms with a hole that are constructed inductively. They specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-need reduction strategy The notation "$E[T]$" stands for a term that decomposes into an evaluation context E and a term T. Evaluation contexts specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-need reduction strategy.

Each contraction rule maps a redex to a contractum:

- Rule (I) introduces a let binding from an application.
- Rule (V) hygienically substitutes a definiens (here: a value) for the occurrence of a let-declared variable arising in an evaluation context. There may be more than one occurrence of the variable in the context. These other occurrences are not substituted.
- Rule (C) allows let bindings to commute with applications.
- Rule (A) re-associates let bindings.

Where call by name uses Rule (N), call by need uses Rule (V), ensuring that only values are duplicated. The reduction strategy thus also differs, so that the definiens of a needed variable is first reduced and this variable is henceforth declared to denote this reduct.

The following reduction sequence (one reduct per line) illustrates the demand-driven aspect of call by need as well as the memoization of intermediate results it enables. We note one-step reduction with $\mapsto_{\text{need}}$ (and specify it precisely in Section 7.4.7) and annotate each reduction step with the name of the corresponding contraction rule:

$$\begin{array}{ll}
\underline{(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x))} \mapsto_{\text{need}} & (I)\\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)}\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (I)\\
\mathsf{let}\ z\ \mathsf{be}\ \underline{(\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{y})}\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (V)\\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)}\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (A)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{z}\ z} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{(\lambda x.x)\ z} \mapsto_{\text{need}} & (I)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \boxed{z}\ \mathsf{in}\ \boxed{x}} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{x}} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x &
\end{array}$$

At every step, we have explicitly decomposed each reduct into a redex (underlined) and its evaluation context (not underlined). We have shaded the occurrences of the variables whose value is needed in the course of the reduction. Only the first shaded occurrence of z forces the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is memoized in the let expression that declares z. It is thus reused when z triggers the two subsequent (V) contractions. This let expression is needed as long as z occurs free in its body; thereafter it can be elided with a garbage-collection rule [34].

This enumeration of successive call-by-need reducts is shorter than the call-by-name reduction sequence in Section 7.2: call by need is an optimization of call by name [12, 141].

To add computational intuition and also to make it easier to test our successive implementations, we take the liberty of extending the calculus of Definition 25 with integers and the (strict) successor function:

Definition 26 (call-by-need λ_{let}-calculus applied to integers)**.**

Syntax:

$$\begin{array}{rl}
\text{Term} \ni T ::= & \boxed{\ulcorner n \urcorner} \mid \boxed{\mathit{succ}\ T} \mid x \mid \lambda x.T \mid T\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ T\\
\text{Value} \ni V ::= & \boxed{\ulcorner n \urcorner} \mid \lambda x.T\\
\text{Answer} \ni A ::= & V \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A\\
\text{Evaluation Context} \ni E ::= & [\,] \mid \boxed{\mathit{succ}\ E} \mid E\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E[x]
\end{array}$$

Contraction rules:

$$\begin{array}{lrl}
(I) & (\lambda x.T)\ T_1 & \to \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T\\
\boxed{(I')} & \mathit{succ}\ \ulcorner n \urcorner & \to \ulcorner n' \urcorner \quad \text{where } n' = n+1\\
(V) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] & \to \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V]\\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \to \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2\\
\boxed{(C')} & \mathit{succ}\ (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A) & \to \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \mathit{succ}\ A\\
(A) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1 & \to \mathsf{let}\ y\ \mathsf{be}\ T_1\\
& \quad \mathsf{in}\ A & \quad \mathsf{in let}\ x\ \mathsf{be}\ A\\
& \mathsf{in}\ E[x] & \quad \mathsf{in}\ E[x]
\end{array}$$

Compared to Definition 25, the shaded parts are new.

This definition is our starting point.

7.4 Some exegesis

Definition 26 packs a lot of information. Let us methodically spell it out:

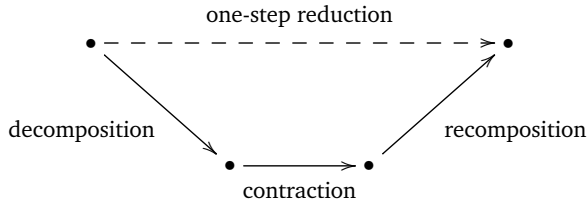
- The contraction rules are a mouthful, and so in Section 7.4.1, we identify their underlying structure by stating a grammar for potential redexes.
- In reduction semantics, evaluation is defined as iterated one-step reduction. However, one-step reduction assumes Barendregt's variable convention, i.e., that all declared variables are distinct, but not all the contraction rules preserve this convention: naive iteration is thus unsound. Rather than subsequently ensuring hygiene as in Garcia et al.'s construction of a lazy abstract machine [99], we make the contraction rules explicitly hygienic in Section 7.4.2 to make one-step reduction preserve Barendregt's variable convention upfront.
- The evaluation contexts are unusual in that they involve terms that are uniquely decomposable into a delimited evaluation context and a variable. In Section 7.4.3, we restate their definition to clearly distinguish between ordinary evaluation contexts and delimited evaluation contexts.
- The one-step reduction of a reduction semantics is implicitly structured in three parts: given a non-answer term,

  (1, decomposition): locate the next potential redex according to the reduction strategy;

  (2, contraction): if the potential redex is an actual one, i.e., if the non-answer term is not stuck, contract this actual redex as specified by the contraction rules; and

  (3, recomposition): fill the surrounding context with the contractum to construct the next term in the reduction sequence.

  Diagrammatically:



Based on Sections 7.4.1, 7.4.2, and 7.4.3, we specify decomposition, hygienic contraction, and recomposition in Sections 7.4.4, 7.4.5, and 7.4.6. We then formalize hygienic one-step reduction in Section 7.4.7 and hygienic evaluation as iterated one-step reduction in Section 7.4.8.

7.4.1 Potential redexes

To bring out the underlying structure of the contraction rules, let us state a grammar for potential redexes:

$$\text{Potential Redex} \ni R ::= \mathit{succ}\ A \mid A\ T \mid \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x]$$

where $E[x]$ stands for a non-answer term.

The two forms of answers – value and let expression – give rise to one contraction rule for each production in the grammar of potential redexes:

- (I') arises from the application of the successor function to a value; (C') arises from the application of the successor function to a let expression; likewise,
- (I) and (C) arise from the application of an answer; and
- (V) and (A) arise from the binding of an answer to a variable whose value is needed.

Not all potential redexes are actual ones: a non-answer term may be stuck due to a type error.

7.4.2 Barendregt's variable convention

The definition of evaluation as iterated one-step reduction assumes Barendregt's variable convention, i.e., that all bound variables are distinct. Indeed the rules (V), (C) and (A) assume the variable convention when they move a term in the scope of a binding. A reduction step involving (V), however, yields a term where the variable convention does not hold, since V is duplicated and it may contain λ-abstractions and therefore bound variables.

There are many ways to ensure variable hygiene, if not the variable convention, at all times. We choose to allow λ-declared (not let-declared) variables to overlap, since no reduction can take place inside a λ-abstraction prior to its application, and to ensure that all let-declared variables are distinct. To this end, in Rule (I), we make each let expression explicitly hygienic by declaring a globally fresh variable and renaming the corresponding λ-declared variable in passing:

$$(I) \quad (\lambda x.T)\ T_1 \to \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x] \quad \text{where } x' \text{ is fresh}$$

This explicit hygiene ensures Barendregt's variable convention for let-declared variables.

Other alternatives exist for ensuring variable hygiene. We have explored several of them, and in our experience they lead to semantic artifacts that are about as simple and understandable as the ones presented here. The alternative we chose here, i.e., making Rule (I) explicitly hygienic corresponds to, and is derived into the same renaming side condition as in Maraist, Odersky, and Wadler's natural semantics [141, Figure 11]. We also observe that this alternative is at the heart of the renaming mechanism in Garcia et al.'s lazy abstract machine [99, Section 4.5]. Across small-step semantics (the present work), abstract machines (Garcia et al.), and big-step semantics (Maraist et al.), there is therefore a genuine consensus about what befits hygienic reduction best in call by need. We have characterized this consensus in Rule (I) here.

With Rule (I) explicitly hygienic, every contraction and thus every reduction step requires at most one fresh variable. Every finite reduction sequence (say, of length n) therefore requires at most n fresh variables. In fact, this notion of fresh variables is coinductive since programs may diverge and thus reduction sequences may be infinite. We thus materialize the freshness condition by threading a stream of fresh variables throughout successive contractions:

$$X \in \mathsf{FreshVars} = \nu X.\mathsf{Var} \times X$$

This stream is used to implement Rule (I):

$$(I) \quad ((x', X), (\lambda x.T)\ T_1) \to (X, \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x])$$

In all the other rules, X is threaded passively. Threading such a stream matches implementational practice, where the so-called "gensym" procedure yields a fresh variable. Here, this fresh variable is the next one in the stream.

7.4.3 The evaluation contexts

The grammars of contexts for call by need, in Definitions 25 and 26, are unusual compared to the one for call by name given in Definition 24. Call-by-need evaluation contexts have an additional constructor involving the term "$E[x]$" for which there exists a variable x in the eye of a delimited context E. Spelling out decomposition (see Section 7.4.5 and Figure 7.3) shows that these delimited contexts are inductively constructed *outside in* whereas all the others are constructed *inside out*. To emphasize the computational difference we make it explicit which are which by adopting two isomorphic representations of contexts as a list of frames:

$$\begin{array}{rl} \mathsf{Context\ Frame} \ni F & ::= succ\ \Box \mid \Box\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \Box \mid \mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x] \\ \mathsf{Outside\text{-}in\ Context} \ni E^{oi} & ::= \varepsilon^{oi} \mid E^{oi} :: F \\ \mathsf{Inside\text{-}out\ Context} \ni E^{io} & ::= \varepsilon^{io} \mid F :: E^{io} \end{array}$$

Here $\Box$ is the hole in a context frame, ε^{oi} is the empty outside-in context, ε^{io} is the empty inside-out context, and $::$ is the (overloaded) context constructor. For example, the context $E = ([\ ]\ T_1)\ T_2$ is equivalent to the outside-in context $E^{oi} = \varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2)$ and to the inside-out context $E^{io} = (\Box\ T_1) :: (\Box\ T_2) :: \varepsilon^{io}$ in the sense that for a term T_0 they all recompose to $(T_0\ T_1)\ T_2$, as defined in Section 7.4.4. Outside-in contexts hang to the left and inside-out contexts hang to the right. They are composed by concatenation to the left or to the right:

$$\begin{array}{ll} E^{oi} \circ_{oi} \varepsilon^{io} = E^{oi} & \varepsilon^{oi} \circ_{io} E^{io} = E^{io} \\ E^{oi} \circ_{oi} (F :: E^{io}) = (E^{oi} :: F) \circ_{oi} E^{io} & (E^{oi} :: F) \circ_{io} E^{io} = E^{oi} \circ_{io} (F :: E^{io}) \end{array}$$

NB. In this BNF of context frames, as pointed out in Section 7.2 and 7.3, the notation "$E^{oi}[x]$" represents a term that uniquely decomposes into an outside-in evaluation context E^{oi} and a variable x. In Section 7.5.2 and onwards, we take notational advantage of this paired representation to short-cut any subsequent decomposition of this term into E^{oi} and x.

$$\frac{}{\langle T,\, \varepsilon^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T} \qquad \frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_1}{\langle T,\, E^{oi} :: (succ\ \Box)\rangle_{oi} \Uparrow_{\mathrm{rec}} succ\ T_1} \qquad \frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_0}{\langle T,\, E^{oi} :: (\Box\ T_1)\rangle_{oi} \Uparrow_{\mathrm{rec}} T_0\ T_1}$$

$$\frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, E^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box)\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

$$\frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_1 \qquad \langle x,\, E_1^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, E^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E_1^{oi}[x])\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

Figure 7.1: Recomposition of outside-in contexts

$$\frac{}{\langle T,\, \varepsilon^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T} \qquad \frac{\langle succ\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_1}{\langle T,\, (succ\ \Box) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_1} \qquad \frac{\langle T_0\ T_1,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T_0,\, (\Box\ T_1) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

$$\frac{\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

$$\frac{\langle x,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T \qquad \langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

Figure 7.2: Recomposition of inside-out contexts

7.4.4 Recomposition

Outside-in contexts and inside-out contexts are recomposed (or again are 'plugged' or 'filled') as follows:

Definition 27 (recomposition of outside-in contexts)**.** An outside-in context E^{oi} is recomposed around a term T into a term T' whenever $\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 7.1.)

Definition 28 (recomposition of inside-out contexts)**.** An inside-out context E^{io} is recomposed around a term T into a term T' whenever $\langle T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 7.2.)

For example, let us recompose the term $\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2$ in the inside-out context $(\Box\ T_3) :: \varepsilon^{io}$:

$$\begin{array}{c} \langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2,\, (\Box\ T_3) :: \varepsilon^{io}\rangle_{io} \\ \text{recomposition} \Big\downarrow \Uparrow_{\mathrm{rec}} \\ (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2)\ T_3 \end{array}$$

Proposition 29 (unique recomposition of outside-in contexts)**.** *For any term T and outside-in context E^{oi} such that $\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T'$ holds, the term T' is unique.*

$$\begin{aligned}
\langle \ulcorner n \urcorner, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle E^{io}, \ulcorner n \urcorner \rangle_{context} \\
\langle succ\ T, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle T, (succ\ \Box) :: E^{io} \rangle_{term} \\
\langle x, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle E^{io}, (\varepsilon^{oi}, x) \rangle_{reroot} \\
\langle \lambda x.T, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle E^{io}, \lambda x.T \rangle_{context} \\
\langle T_0\ T_1, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle T_0, (\Box\ T_1) :: E^{io} \rangle_{term} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T, E^{io} \rangle_{term} &\downarrow_{\mathrm{dec}} \langle T, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io} \rangle_{term}
\end{aligned}$$

$$\begin{aligned}
\langle \varepsilon^{io}, A \rangle_{context} &\downarrow_{\mathrm{dec}} \langle A \rangle_{answer} \\
\langle (succ\ \Box) :: E^{io}, A \rangle_{context} &\downarrow_{\mathrm{dec}} \langle succ\ A, E^{io} \rangle_{redex} \\
\langle (\Box\ T_1) :: E^{io}, A \rangle_{context} &\downarrow_{\mathrm{dec}} \langle A\ T_1, E^{io} \rangle_{redex} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io}, A \rangle_{context} &\downarrow_{\mathrm{dec}} \langle E^{io}, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io}, A \rangle_{context} &\downarrow_{\mathrm{dec}} \langle \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E^{oi}[x], E^{io} \rangle_{redex}
\end{aligned}$$

$$\begin{aligned}
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io}, (E^{oi}, x) \rangle_{reroot} &\downarrow_{\mathrm{dec}} \langle T_1, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io} \rangle_{term} \\
\langle F :: E^{io}, (E^{oi}, x) \rangle_{reroot} &\downarrow_{\mathrm{dec}} \langle E^{io}, (E^{oi} :: F, x) \rangle_{reroot} \\
&\qquad \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \Box
\end{aligned}$$

Figure 7.3: Decomposition of an answer term into itself
and of a non-answer term into a potential redex and its context

Proof. Induction on E^{oi}. □

Proposition 30 (unique recomposition of inside-out contexts)**.** *For any term T and inside-out context E^{io} such that $\langle T, E^{io} \rangle_{io} \Uparrow_{\mathrm{rec}} T'$ holds, the term T' is unique.*

Proof. Induction on E^{io}. □

7.4.5 Decomposition

Decomposing a non-answer term into a potential redex and its evaluation context according to the reduction strategy is at the heart of a reduction semantics, but outside of the authors' publications, it seems never to be spelled out. Let us do so.

There are many ways to specify decomposition. In our experience, a convenient one is the abstract machine displayed in Figure 7.3. This machine starts in the configuration $\langle T, \varepsilon^{io} \rangle_{term}$, for a given term T. It halts in an answer state if the given term contains no potential redex, and in a decomposition state $\langle R, E^{io} \rangle_{redex}$ otherwise, where R is a potential redex in T and E^{io} its evaluation context according to the reduction strategy specified by the grammar of evaluation contexts.

Definition 31 (decomposition)**.** The decomposition relation, $\downarrow^*_{\mathrm{dec}}$, is the transitive closure of $\downarrow_{\mathrm{dec}}$. (See Figure 7.3.)

For example, let us decompose the non-answer term $(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3$:

$$\begin{array}{c}
\langle (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3, \varepsilon^{io} \rangle_{term} \\
\text{decomposition} \Big\downarrow \downarrow^*_{\mathrm{dec}} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2))[x], (\Box\ T_3) :: \varepsilon^{io} \rangle_{redex}
\end{array}$$

The *term* and *context* transitions are traditional: one dispatches on a term and the other on the top context frame. The *reroot* transitions locate the let-binder for a variable while maintaining the outside-in context from the binder to its occurrence, zipper-style [115].[2] In effect, the transitions reverse the prefix of an inside-out context into an outside-in context. For the example above, this reversal is carried out in the following sub-steps of decomposition:

$$\langle (\Box\ T_1) :: (\Box\ T_2) :: (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ \Box) :: (\Box\ T_3) :: \varepsilon^{io},\ (\varepsilon^{io},\ x)\rangle_{reroot}$$
$$\Big\downarrow \downarrow^*_{\mathrm{dec}}$$
$$\langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2))[x],\ (\Box\ T_3) :: \varepsilon^{io}\rangle_{redex}$$

Proposition 32 (vacuous decomposition of an answer term)**.** *An answer term is vacuously decomposed into itself: for any answer term A,* $\langle A, \varepsilon^{io}\rangle_{term} \downarrow^*_{\mathrm{dec}} \langle A\rangle_{answer}$ *holds.*

Proof. By induction: the transitions over *term*-configurations turn the answer term inside-out into a context until its innermost value is reached, and the transitions over *context*-configurations turn back this context inside-out into the answer term until the empty context is reached. □

Proposition 33 (unique decomposition of a non-answer term)**.** *For any non-answer term* T *such that* $\langle T, \varepsilon^{io}\rangle_{term} \downarrow^*_{\mathrm{dec}} \langle R, E^{io}\rangle_{redex}$ *holds, the potential redex* R *and evaluation context* E^{io} *are unique.*

Proof. The $\downarrow_{\mathrm{dec}}$ relation is uniquely determined and $\langle R, E^{io}\rangle_{redex}$ is a terminal state, thus by transitivity $\langle R, E^{io}\rangle_{redex}$ is unique. □

7.4.6 The contraction rules

In accordance with the new BNF of contexts, the contraction rules of Definition 26 are hygienically stated as follows:

$$\begin{array}{lll}
(I) & ((x', X), (\lambda x.T)\ T_1) \to (X, \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x]) & \\
(I') & (X, succ\ \ulcorner n \urcorner) \to (X, \ulcorner n' \urcorner) & \text{where } n' = n+1 \\
(V) & (X, \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E^{oi}[x]) \to (X, \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ T) & \text{where } \langle V, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T \\
(C) & (X, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2) \to (X, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2) & \\
(C') & (X, succ\ (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A)) \to (X, \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ succ\ A) & \\
(A) & (X, \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1 \to (X, \mathsf{let}\ y\ \mathsf{be}\ T_1 & \\
 & \quad \mathsf{in}\ A \quad \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A & \\
 & \quad \mathsf{in}\ E^{oi}[x]) \quad \mathsf{in}\ E^{oi}[x]) &
\end{array}$$

Definition 34 (notion of reduction)**.** $\mathscr{R} = (I) \cup (I') \cup (V) \cup (C) \cup (C') \cup (A)$ and a redex R contracts to T, denoted $R \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} T$, iff $((X, R), (X', T)) \in \mathscr{R}$.

[2] Decomposition could be stuck for terms containing free variables, but we assume programs to be closed.

For example, let us contract the actual redex $\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2))[x]$ with the stream of fresh variables X:

$$\begin{array}{c} \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2))[x] \\ \text{contraction of } (V) \Big\downarrow \overset{X;X}{\rightsquigarrow}_{\mathscr{R}} \\ \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2 \end{array}$$

Proposition 35 (unique contraction)**.** *For any redex R and stream of fresh variables X such that $R \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} T$ holds, T and X' are unique.*

Proof. By case analysis on R. (See Section 7.4.1.) □

7.4.7 Standard one-step reduction

The standard one-step reduction performs one contraction in a non-answer term and is defined as

1. locating a potential redex and its evaluation context in the non-answer term through a number of decomposition steps,
2. contracting this potential redex if it is an actual one (otherwise the non-answer term is stuck), and
3. recomposing the resulting contractum into the evaluation context:

Definition 36 (standard one-step reduction)**.**

$$(X,\ T) \mapsto_{\text{need}} (X',\ T'') \text{ iff } \begin{cases} \langle T,\ \varepsilon^{oi} \rangle_{term} \downarrow^{*}_{\text{dec}} \langle R,\ E^{io} \rangle_{redex} \\ R \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} T' \\ \langle T',\ E^{io} \rangle_{io} \Uparrow_{\text{rec}} T'' \end{cases}$$

Note that the standard one-step reduction is not the compatible closure of $\mathscr{R}$. The compatible closure, $\rightarrow_{\mathscr{R}}$, is closed over general contexts (i.e., terms with a hole), whereas the standard one-step reduction is closed over the restricted grammar of evaluation contexts.

For example, given a stream of fresh variables X, let us illustrate standard one-step reduction for the term $(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3$:

$$\begin{array}{c} \langle (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3,\ \varepsilon^{io} \rangle_{term} \\ \text{decomposition} \Big\downarrow \downarrow^{*}_{\text{dec}} \\ \langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2))[x],\ (\Box\ T_3) :: \varepsilon^{io} \rangle_{redex} \\ \text{contraction of } (V) \Big\downarrow \langle \overset{X;X}{\rightsquigarrow}_{\mathscr{R}}, id \rangle \\ \langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2,\ (\Box\ T_3) :: \varepsilon^{io} \rangle_{io} \\ \text{recomposition} \Big\downarrow \Uparrow_{\text{rec}} \\ (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2)\ T_3 \end{array}$$

Proposition 37 (unique standard one-step reduction)**.** *For any term T and stream of fresh variables X such that $(X, T) \mapsto_{\text{need}} (X', T')$ holds, T' and X' are unique.*

Proof. Corollary of unique decomposition (Proposition 33), unique contraction (Proposition 35), and unique recomposition (Proposition 30). □

7.4.8 Standard reduction-based evaluation

The standard reduction-based evaluation is defined as the iteration of the standard one-step reduction. It thus enumerates the successive call-by-need reducts, i.e., the standard reduction sequence, of any given term:

Definition 38 (standard reduction-based evaluation)**.** Standard reduction-based evaluation, $\mapsto^*_{\text{need}}$, is the reflexive, transitive closure of standard one-step reduction, $\mapsto_{\text{need}}$.

Proposition 39 (unique standard reduction-based evaluation to answers)**.** *For any term T and stream of fresh variables X such that $(X, T) \mapsto^*_{\text{need}} (X', A)$ holds, A and X' are unique.*

Proof. Corollary of unique standard reduction (Proposition 37). □

7.4.9 Conclusion and perspectives

As illustrated here, there is substantially more than meets the eye in a reduction semantics.

In addition, extensional properties such as unique decomposition, standardization, and hygiene do not only ensure the existence of a deterministic evaluator extensionally, but it is our thesis that they also provide precious intensional guidelines. Indeed, after exegetically spelling out what does not readily meet the eye, things become compellingly simple:

- refocusing the standard reduction-based evaluation immediately gives a reduction-free abstract machine (Section 7.5.1) and compressing the corridor transitions of this abstract machine improves the efficiency of its execution (Section 7.5.2);
- we can then move from the relational view of small-step abstract machines to the functional view of big-step abstract machines (Section 7.6);
- refunctionalizing the compressed big-step abstract machine with respect to the evaluation contexts gives a reduction-free evaluation function in continuation-passing style (Section 7.7.1). Mapping this evaluation function back to direct style gives a functional implementation of a natural semantics (Section 7.7.2).[3]

All of these semantic artifacts are correct by construction, and their operational behaviors rigorously mirror each other in a lock-step sort of way. For one example, the semantic artifacts agree not only up to α-equivalence but up to syntactic equality. For another example, should one be tempted to fine-tune either of these semantic artifacts, one is then in position to adjust the others to keep their operational behaviors in line, or to understand why this alignment is not possible and where coherence got lost in the fine-tuning [61].

[3] Recently [167], Pirog and Biernacki have used the CPS transformation and defunctionalization to connect Launchbury and Sestoft's natural semantics for lazy evaluation [134, 195] and Peyton Jones's spineless tagless G-machine [160].

$$\langle T, E^{io}\rangle_{term} \rightarrow_{\text{refocus}} D \text{ iff } \langle T, E^{io}\rangle_{io} \Uparrow_{\text{rec}} T' \wedge \langle T', \varepsilon^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$$

Figure 7.4: Reduction-based refocusing

7.5 From reduction semantics to abstract machine

This section implements the first half of the programme outlined in Section 7.4.9. We first go from the standard reduction-based evaluation of Definition 38 (that enumerates all the successive reducts in the standard reduction sequence) to a reduction-free evaluation (that does not perform this enumeration because all the reducts are deforested away). This reduction-free evaluation takes the form of an abstract machine.

7.5.1 Refocusing: from reduction semantics to abstract machine

By recomposing and then immediately decomposing, a reduction-based evaluator takes a detour from a redex site, up to the top of the term, and back down again to the next redex site. The steps that make up this detour can be eliminated by refocusing [71]. Refocusing the reduction-based evaluation of a reduction semantics yields a reduction-free evaluation that directly navigates in a term from redex site to redex site without any detour via the top of the term.

Refocusing replaces successive recompositions and decompositions by a 'refocus' relation that associates a contractum and its (inside-out) evaluation context to an answer or a decomposition consisting of the next potential redex and associated evaluation context:

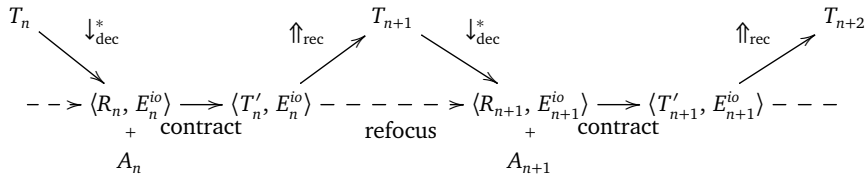


Figure 7.4 displays the naive, reduction-based definition of refocusing: an evaluation context is recomposed around a contractum and the resulting reduct is decomposed either into an answer or into another potential redex and its evaluation context. This definition is 'reduction-based' because the intermediate reduct is constructed.

Surprisingly, *optimal refocusing consists of simply continuing with decomposition from the contractum and its associated evaluation context, according to the standard reduction strategy* [71], here as well as in all the reduction semantics in Felleisen and Flatt's lecture notes on programming languages and lambda calculi [86]. (This is another reason why we place such store in the decomposition function of a reduction semantics.)

Figure 7.5 displays the optimal, reduction-free definition of refocusing: a contractum and an evaluation context are directly associated with an answer or another potential redex and its evaluation context simply by decomposing the contractum in the evaluation context. This definition is 'reduction-free' because no intermediate reduct is constructed.

$$\langle T, E^{io}\rangle_{term} \rightarrow_{\text{refocus}} D \text{ iff } \langle T, E^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$$

Figure 7.5: Reduction-free refocusing

Reduction-free evaluation is defined, after an initial decomposition of the input term, as the iteration of contraction and reduction-free refocusing (i.e., term decomposition in the current evaluation context):

Definition 40 (standard reduction-free evaluation)**.** Let $\rightarrow_{\text{step}}$ be one-step contraction and refocusing: $\xrightarrow{X;X'}_{\text{step}} = \downarrow^{*}_{\text{dec}} \cup \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} \downarrow^{*}_{\text{dec}}$, where $\langle R, E^{io}\rangle_{redex} \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} \downarrow^{*}_{\text{dec}} D$ iff $R \overset{X;X'}{\rightsquigarrow}_{\mathscr{R}} T \ \wedge \ \langle T, E^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$. Standard reduction-free evaluation, $\rightarrow^{*}_{\text{step}}$, is the transitive closure of $\rightarrow_{\text{step}}$. Notationally we use $\xrightarrow{X;X'}{}^{*}_{\text{step}}$ to express that X is the input stream and X' is a suffix of X obtained after iterating $\rightarrow_{\text{step}}$.

Evaluation is thus defined with the decomposition transitions from Figure 7.3 plus, for each contraction rule from Section 7.4.6, one transition towards decomposing the contractum in the current evaluation context. Like decomposition in Figure 7.3, evaluation therefore takes the form of an abstract machine.[4] This abstract machine is displayed in Figure 7.6.

Proposition 41 (full correctness)**.** *For the abstract machine of Figure 7.6,*

$$(X, T) \mapsto^{*}_{\text{need}} (X', A) \iff \langle T, \varepsilon^{io}\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\text{step}} \langle A\rangle_{answer}.$$

Proof. Corollary of the correctness of refocusing [71, 197]. □

7.5.2 Transition compression: from abstract machine to abstract machine

In the abstract machine of Figure 7.6, some of the transitions yield a configuration for which there unconditionally exists another transition: all transitions to a *term*-configuration with a known term, all transitions to a *context*-configuration with a known context, and all transitions to a *redex*-configuration with a known redex (i.e., all transitions to *redex*). For example, the application of any let expression, which is a redex, gives rise to the following unconditional transitions:

$$\begin{aligned}\langle(\text{let } x \text{ be } T_1 \text{ in } A)\ T_2, E^{io}\rangle_{redex} &\xrightarrow{X;X}_{\text{step}} \langle\text{let } x \text{ be } T_1 \text{ in } A\ T_2, E^{io}\rangle_{term}\\ &\xrightarrow{X;X}_{\text{step}} \langle A\ T_2, (\text{let } x \text{ be } T_1 \text{ in } \square) :: E^{io}\rangle_{term}\\ &\xrightarrow{X;X}_{\text{step}} \langle A, (\square\ T_2) :: (\text{let } x \text{ be } T_1 \text{ in } \square) :: E^{io}\rangle_{term}\end{aligned}$$

These so-called "corridor transitions" from one configuration to another can be hereditarily compressed so that the first configuration yields the last one in one transition.

[4] Giving decomposition another format would make evaluation inherit this format.

$$\begin{array}{rcl}
\langle \ulcorner n \urcorner, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle E^{io}, \ulcorner n \urcorner \rangle_{context} \\
\langle succ\ T, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle T, (succ\ \square) :: E^{io} \rangle_{term} \\
\langle x, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle E^{io}, (\varepsilon^{oi}, x) \rangle_{reroot} \\
\langle \lambda x.T, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle E^{io}, \lambda x.T \rangle_{context} \\
\langle T_0\ T_1, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle T_0, (\square\ T_1) :: E^{io} \rangle_{term} \\
\langle \text{let } x \text{ be } T_1 \text{ in } T, E^{io} \rangle_{term} & \xrightarrow{X;X}_{\text{step}} & \langle T, (\text{let } x \text{ be } T_1 \text{ in } \square) :: E^{io} \rangle_{term} \\
\\
\langle \varepsilon^{io}, A \rangle_{context} & \xrightarrow{X;X}_{\text{step}} & \langle A \rangle_{answer} \\
\langle (succ\ \square) :: E^{io}, A \rangle_{context} & \xrightarrow{X;X}_{\text{step}} & \langle succ\ A, E^{io} \rangle_{redex} \\
\langle (\square\ T_1) :: E^{io}, A \rangle_{context} & \xrightarrow{X;X}_{\text{step}} & \langle A\ T_1, E^{io} \rangle_{redex} \\
\langle (\text{let } x \text{ be } T_1 \text{ in } \square) :: E^{io}, A \rangle_{context} & \xrightarrow{X;X}_{\text{step}} & \langle E^{io}, \text{let } x \text{ be } T_1 \text{ in } A \rangle_{context} \\
\langle (\text{let } x \text{ be } \square \text{ in } E^{oi}[x]) :: E^{io}, A \rangle_{context} & \xrightarrow{X;X}_{\text{step}} & \langle \text{let } x \text{ be } A \text{ in } E^{oi}[x], E^{io} \rangle_{redex} \\
\\
\langle (\text{let } x \text{ be } T_1 \text{ in } \square) :: E^{io}, (E^{oi}, x) \rangle_{reroot} & \xrightarrow{X;X}_{\text{step}} & \langle T_1, (\text{let } x \text{ be } \square \text{ in } E^{oi}[x]) :: E^{io} \rangle_{term} \\
\langle F :: E^{io}, (E^{oi}, x) \rangle_{reroot} & \xrightarrow{X;X}_{\text{step}} & \langle E^{io}, (E^{oi} :: F, x) \rangle_{reroot} \\
 & & \text{where } F \neq \text{let } x \text{ be } T \text{ in } \square \\
\\
\langle succ\ \ulcorner n \urcorner, E^{io} \rangle_{redex} & \xrightarrow{X;X}_{\text{step}} & \langle \ulcorner n' \urcorner, E^{io} \rangle_{term} \\
 & & \text{where } n' = n+1 \\
\langle succ\ (\text{let } x \text{ be } T \text{ in } A), E^{io} \rangle_{redex} & \xrightarrow{X;X}_{\text{step}} & \langle \text{let } x \text{ be } T \text{ in } succ\ A, E^{io} \rangle_{term} \\
\langle (\lambda x.T)\ T_1, E^{io} \rangle_{redex} & \xrightarrow{(x',X);X}_{\text{step}} & \langle \text{let } x' \text{ be } T_1 \text{ in } T[x'/x], E^{io} \rangle_{term} \\
\langle (\text{let } x \text{ be } T_1 \text{ in } A)\ T_2, E^{io} \rangle_{redex} & \xrightarrow{X;X}_{\text{step}} & \langle \text{let } x \text{ be } T_1 \text{ in } A\ T_2, E^{io} \rangle_{term} \\
\langle \text{let } x \text{ be } V \text{ in } E^{oi}[x], E^{io} \rangle_{redex} & \xrightarrow{X;X}_{\text{step}} & \langle \text{let } x \text{ be } V \text{ in } T, E^{io} \rangle_{term} \\
 & & \text{where } \langle V, E^{oi} \rangle_{oi} \Uparrow_{\text{rec}} T \\
\langle \text{let } x \text{ be let } y \text{ be } T_1 \text{ in } A \text{ in } E^{oi}[x], E^{io} \rangle_{redex} & \xrightarrow{X;X}_{\text{step}} & \langle \text{let } y \text{ be } T_1 \text{ in let } x \text{ be } A \text{ in } T, E^{io} \rangle_{term} \\
 & & \text{where } \langle x, E^{oi} \rangle_{oi} \Uparrow_{\text{rec}} T
\end{array}$$

Figure 7.6: Storeless abstract machine for call-by-need evaluation

Other transition compressions are determined by the structure of the term or of the evaluation context, and proceed over several steps. For example, analogously to what happens in Proposition 32, a *term*-configuration with an answer in a context always yields a *context*-configuration with this context and this answer:

$$\begin{array}{l}
\langle \text{let } x_1 \text{ be } T_1 \text{ in let } x_2 \text{ be } T_2 \text{ in } \cdots \text{let } x_n \text{ be } T_n \text{ in } V, E^{io} \rangle_{term} \\
\xrightarrow{X;X}_{\text{step}} \langle \text{let } x_2 \text{ be } T_2 \text{ in } \cdots \text{let } x_n \text{ be } T_n \text{ in } V, (\text{let } x_1 \text{ be } T_1 \text{ in } \square) :: E^{io} \rangle_{term} \\
\cdots \\
\xrightarrow{X;X}_{\text{step}} \langle V, (\text{let } x_n \text{ be } T_n \text{ in } \square) :: \cdots :: (\text{let } x_2 \text{ be } T_2 \text{ in } \square) :: (\text{let } x_1 \text{ be } T_1 \text{ in } \square) :: E^{io} \rangle_{term} \\
\xrightarrow{X;X}_{\text{step}} \langle (\text{let } x_n \text{ be } T_n \text{ in } \square) :: \cdots :: (\text{let } x_2 \text{ be } T_2 \text{ in } \square) :: (\text{let } x_1 \text{ be } T_1 \text{ in } \square) :: E^{io}, V \rangle_{context} \\
\cdots \\
\xrightarrow{X;X}_{\text{step}} \langle E^{io}, \text{let } x_1 \text{ be } T_1 \text{ in let } x_2 \text{ be } T_2 \text{ in } \cdots \text{let } x_n \text{ be } T_n \text{ in } V \rangle_{context}
\end{array}$$

So, rather than turning the answer inside-out into the prefix of a context (with transitions over *term*-configurations) until its innermost value is reached, and then turning this prefix inside-out back into the answer (with transitions over *context*-configurations), we can directly refocus the original *term*-configuration into the final *context*-configuration:

Proposition 42 (refocusing over answers)**.** *For any A, E^{io} and X,*

$$\langle A, E^{io}\rangle_{term} \xrightarrow{X;X}{}^{*}_{\text{step}} \langle E^{io}, A\rangle_{context}$$

Proof. Induction on A. □

Likewise, we can compress the transitions from a *term*-configuration over any term $E^{oi}[x]$ to a *term*-configuration over x, using the reverse concatenation "$\diamond$" defined in Section 7.4.3:

Proposition 43 (restoring outside-in evaluation contexts)**.** *For any T, E^{io}, E^{oi} and X such that $\langle T, E^{oi}\rangle_{oi} \Uparrow_{\text{rec}} T'$,*

$$\langle T', E^{io}\rangle_{term} \xrightarrow{X;X}{}^{*}_{\text{step}} \langle T, E^{oi} \circ_{io} E^{io}\rangle_{term}$$

Proof. Induction on E^{oi}. □

Finally, we can short-cut the search for the definiens of a needed variable:

Proposition 44 (locating a definiens)**.** *For any x, T, E^{oi}, E_1^{io}, E_2^{io}, and X, where x is not declared in E_1^{io},*

$$\langle E_1^{io} :: (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E^{oi}[x]) :: E_2^{io}, (E^{oi}, x)\rangle_{reroot} \xrightarrow{X;X}{}^{*}_{\text{step}} \langle T, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ (E^{oi} \circ_{oi} E_1^{io})[x]) :: E_2^{io}\rangle_{term}$$

Proof. Induction on E_1^{io}. □

The resulting abstract machine is displayed in Figure 7.7. No occurrences of "$\diamond$" appear in the final abstract machine because in the course of compression all occurrences introduced by Proposition 43 are subsequently eliminated by Proposition 44.

Proposition 45 (full correctness)**.** *For the abstract machine of Figure 7.7,*

$$(X, T) \mapsto^{*}_{\text{need}} (X', A) \iff \langle T, \varepsilon^{io}\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\text{step}} \langle A\rangle_{answer}.$$

Proof. By Proposition 41 and calculation using Propositions 42, 43, and 44. □

$$\langle \ulcorner n \urcorner, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle E^{io}, \ulcorner n \urcorner \rangle_{context}$$
$$\langle succ\ T, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle T, (succ\ \square) :: E^{io} \rangle_{term}$$
$$\langle x, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle E^{io}, (\varepsilon^{oi}, x) \rangle_{reroot}$$
$$\langle \lambda x.T, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle E^{io}, \lambda x.T \rangle_{context}$$
$$\langle T_0\ T_1, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle T_0, (\square\ T_1) :: E^{io} \rangle_{term}$$
$$\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T, E^{io} \rangle_{term} \xrightarrow[\text{step}]{X;X} \langle T, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{term}$$

$$\langle \varepsilon^{io}, A \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle A \rangle_{answer}$$
$$\langle (succ\ \square) :: E^{io}, \ulcorner n \urcorner \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle E^{io}, \ulcorner n' \urcorner \rangle_{context} \quad \text{where } n' = n + 1$$
$$\langle (succ\ \square) :: E^{io}, \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle (succ\ \square) :: (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square) :: E^{io}, A \rangle_{context}$$
$$\langle (\square\ T_1) :: E^{io}, \lambda x.T \rangle_{context} \xrightarrow[\text{step}]{(x',X);X} \langle T[x'/x], (\mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{term}$$
$$\langle (\square\ T_2) :: E^{io}, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle (\square\ T_2) :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io}, A \rangle_{context}$$
$$\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io}, A \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle E^{io}, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context}$$
$$\langle (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io}, V \rangle_{context} \xrightarrow[\text{step}]{X;X} \langle T, (\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \square) :: E^{io} \rangle_{term} \quad \text{where } \langle V, E^{oi} \rangle_{oi} \Uparrow_{\text{rec}} T$$
$$\left\langle \begin{pmatrix} \mathsf{let}\ x\ \mathsf{be}\ \square \\ \mathsf{in}\ E^{oi}[x] \end{pmatrix} :: E^{io}, \begin{pmatrix} \mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \mathsf{in}\ A \end{pmatrix} \right\rangle_{context} \xrightarrow[\text{step}]{X;X} \left\langle \begin{pmatrix} \mathsf{let}\ x\ \mathsf{be}\ \square \\ \mathsf{in}\ E^{oi}[x] \end{pmatrix} :: \begin{pmatrix} \mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \mathsf{in}\ \square \end{pmatrix} :: E^{io}, A \right\rangle_{context}$$

$$\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io}, (E^{oi}, x) \rangle_{reroot} \xrightarrow[\text{step}]{X;X} \langle T_1, (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io} \rangle_{term}$$
$$\langle F :: E^{io}, (E^{oi}, x) \rangle_{reroot} \xrightarrow[\text{step}]{X;X} \langle E^{io}, (E^{oi} :: F, x) \rangle_{reroot} \quad \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square$$

Figure 7.7: The storeless abstract machine of Figure 7.6 after transition compression

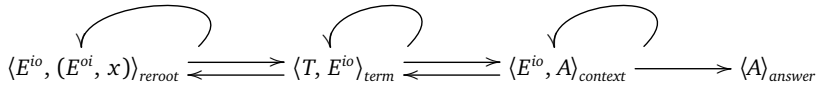
7.6 Small-step abstract machines define relations, big-step abstract machines define functions

A deterministic small-step abstract machine is characterized by a single-step state-transition system that associates a machine configuration with the next and is iterated toward a final state, if there is one. This characterization is aptly formalized by a *relation* that associates any non-final state to its successive states. The transitive closure of this relation then describes the transition sequence of any given term as well as its final state, if there is one. In contrast, a big-step abstract machine is characterized by a collection of mutually tail-recursive transitions mapping a configuration to a final state, if there is one. This characterization is aptly formalized by a *function* that maps any non-final state to a final state, if there is one. Here we have no interest in the actual reduction sequences towards a final state.

The difference between the two styles of abstract machines is not typically apparent in the abstract-machine specifications found in programming-language semantics. A ma-

chine specification is normally presented as a small-step abstract machine given by reading the transition arrow as the definition of a single-step transition relation to be iterated and with the configuration labels as passive components of the configurations. However, the same specification can equally be seen as a big-step abstract machine if the transition labels are interpreted as tail-recursive functions, with the transition arrow connecting left- and right-hand sides of their definitions.

The following diagram depicts the states and transitions of the abstract machine in Figure 7.7:



States can be viewed as a sum type of labeled components, and the transition arrows as a relation that maps any non-final state to its successive states. Alternatively, the states can be viewed as a set of mutually (tail-)recursive functions and the transition arrows as tail calls between the functions. By Proposition 39 we know that final states are unique, and thus we can model the big-step abstract machine as a partial function mapping any term to its unique final state, if there is one.

These two views (of small steps and of big steps) are relevant to transform an abstract machine implementing an operational semantics into an interpreter implementing a natural semantics. Such interpreters operate in big steps, and it is for this reason that we now shift gears and view the abstract machine of Figure 7.7 as a big-step one with evaluation defined by a partial function. These two views on an abstract machine are illustrated in 7.A.2 and 7.A.3 with a simpler example. From a programming perspective [67], the correctness of these two views is established by the lightweight fusion program transformation [155].

7.7 From abstract machine to evaluation functions

This section implements the second half of the programme outlined in Section 7.4.9. We start from the big-step abstract machine of Figure 7.7 and refunctionalize it into a continuation-passing interpreter (Section 7.7.1), which we then map back to direct style (Section 7.7.2). Observing that a component of the resulting direct-style interpreter is in defunctionalized form, we refunctionalize it (Section 7.7.3). Refunctionalization and the direct-style transformation are illustrated in 7.A.3, 7.A.4 and 7.A.5 with a simpler example.

7.7.1 Refunctionalization: from abstract machine to continuation-passing interpreter

Defunctionalization and refunctionalization: Reynolds introduced defunctionalization [70, 176] to derive first-order evaluators from higher-order ones. Defunctionalization turns a function type into a sum type, and function application into the application of an apply function dispatching on the sum type. Its left inverse, *refunctionalization* [69], can transform first-order abstract machines into higher-order interpreters. It specifically works on programs that are in defunctionalized form, i.e., in the image of Reynolds's defunctionalization.

Towards refunctionalizing the big-step abstract machine of Figure 7.7: The big-step abstract machine of Figure 7.7 is not in defunctionalized form with respect to the inside-out evaluation contexts. Indeed these contexts are consumed by the two transition functions corresponding to $\langle E^{io}, A\rangle_{context}$ and $\langle E^{io}, (E^{oi}, x)\rangle_{reroot}$ rather than by the single apply function demanded for refunctionalization. This mismatch can be fixed by introducing a sum type discriminating between the (non-context) arguments to the two transition functions and combining them into a single transition function [69]. The left summand (tagged "*ans*") holds an answer, and the right summand (tagged "*var*") pairs a variable whose value is needed and an incrementally-constructed outside-in context used to get back to the place in the term where the value was needed.

Three of the context constructors occur on the right-hand sides of their own apply function clauses. When refunctionalized, these correspond to recursive functions and therefore appear as named functions.

The refunctionalized abstract machine is an interpreter for lazy evaluation in continuation-passing style, where the continuations are the functional representation of the inside-out contexts.

7.7.2 Back to direct style: from continuation-passing interpreter to natural semantics

It is a simple matter to transform the continuation-passing interpreter described in Section 7.7.1 into direct style [56]. The continuations do not represent any control effect other than non-tail calls, so the resulting direct-style interpreter does not require first-class control operators [66].

In the present case, the interpreter of Section 7.7.1 implements a natural semantics (i.e., a big-step operational semantics) for lazy evaluation. This semantics is displayed in Figure 7.8. In reference to Figure 7.7,

- there is one *term* transition and one $\Downarrow_{\text{eval}}$ judgement for each syntactic construct;
- for every *context* transition, there is a corresponding judgment over the *ans* injection tag:
  - two $\Downarrow_{\text{succ}}$ judgments for the two transitions on the frame "*succ* $\square$",
  - two $\Downarrow_{\text{apply}}$ judgments for the two transitions on the frame "$\square\ T$",
  - one $\Downarrow_{\text{bind}}$ judgement for the transition on the frame "let x be T in $\square$", and
  - two $\Downarrow_{\text{force}}$ judgements for the two transitions on the frame "let x be $\square$ in $E^{oi}[x]$"; and
- for every *reroot* transition, there is a corresponding judgment over the *var* injection tag: one for each context frame, plus one for when there is a match.

Proposition 46 (full correctness)**.**

$$\langle T, \varepsilon^{io}\rangle_{term} \xrightarrow[\text{step}]{X;X'\ *} \langle A\rangle_{answer} \iff T\ {}^{X}\!\Downarrow^{X'}_{\text{eval}} ans(A).$$

Proof. Corollary of the correctness of defunctionalization and the CPS transformation. □

As illustrated in 7.A.6 and 7.A.5, the natural semantics of Figure 7.8 is implemented as an interpreter in direct style. Following Reynolds's functional correspondence, it can be CPS transformed and defunctionalized towards the abstract machine of Figure 7.7.

$$\frac{}{\ulcorner n \urcorner \ {}^{X}\Downarrow^{X}_{\text{eval}} \ ans(\ulcorner n \urcorner)} \qquad \frac{T \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad r \ {}^{X'}\Downarrow^{X''}_{\text{succ}} \ r'}{succ \ T \ {}^{X}\Downarrow^{X''}_{\text{eval}} \ r'} \qquad \frac{}{x \ {}^{X}\Downarrow^{X}_{\text{eval}} \ var(x, \varepsilon^{oi})}$$

$$\frac{}{\lambda x.T \ {}^{X}\Downarrow^{X}_{\text{eval}} \ ans(\lambda x.T)} \qquad \frac{T_0 \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad r \ T_1 \ {}^{X'}\Downarrow^{X''}_{\text{apply}} \ r'}{T_0 \ T_1 \ {}^{X}\Downarrow^{X''}_{\text{eval}} \ r'}$$

$$\frac{T \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad (x, T_1, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{\text{let } x \text{ be } T_1 \text{ in } T \ {}^{X}\Downarrow^{X''}_{\text{eval}} \ r'} \qquad \frac{}{ans(\ulcorner n \urcorner) \ {}^{X}\Downarrow^{X}_{\text{succ}} \ ans(\ulcorner n' \urcorner)} \text{where } n' = n + 1$$

$$\frac{ans(A) \ {}^{X}\Downarrow^{X'}_{\text{succ}} \ r \qquad (x, T, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{ans(\text{let } x \text{ be } T \text{ in } A) \ {}^{X}\Downarrow^{X''}_{\text{succ}} \ r'} \qquad \frac{}{var(x, E^{oi}) \ {}^{X}\Downarrow^{X}_{\text{succ}} \ var(x, E^{oi} :: (succ \ \Box))}$$

$$\frac{T[x'/x] \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad (x', T_1, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{(ans(\lambda x.T)) \ T_1 \ {}^{(x',X)}\Downarrow^{X''}_{\text{apply}} \ r'} \qquad \frac{(ans(A)) \ T_2 \ {}^{X}\Downarrow^{X'}_{\text{apply}} \ r \qquad (x, T_1, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{(ans(\text{let } x \text{ be } T_1 \text{ in } A)) \ T_2 \ {}^{X}\Downarrow^{X''}_{\text{apply}} \ r'}$$

$$\frac{}{(var(x, E^{oi})) \ T_1 \ {}^{X}\Downarrow^{X}_{\text{apply}} \ var(x, E^{oi} :: (\Box \ T_1))}$$

$$\frac{}{(x, T_1, ans(A)) \ {}^{X}\Downarrow^{X}_{\text{bind}} \ ans(\text{let } x \text{ be } T_1 \text{ in } A)} \qquad \frac{T_1 \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad (x, r, E^{oi}) \ {}^{X'}\Downarrow^{X''}_{\text{force}} \ r'}{(x, T_1, var(x, E^{oi})) \ {}^{X}\Downarrow^{X''}_{\text{bind}} \ r'}$$

$$\frac{}{(x, T_1, var(y, E^{oi})) \ {}^{X}\Downarrow^{X}_{\text{bind}} \ var(y, E^{oi} :: (\text{let } x \text{ be } T_1 \text{ in } \Box))} \text{where } x \neq y$$

$$\frac{\langle V, E^{oi} \rangle_{oi} \Uparrow_{\text{rec}} T \qquad T \ {}^{X}\Downarrow^{X'}_{\text{eval}} \ r \qquad (x, V, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{(x, ans(V), E^{oi}) \ {}^{X}\Downarrow^{X''}_{\text{force}} \ r'}$$

$$\frac{(x, ans(A), E^{oi}) \ {}^{X}\Downarrow^{X'}_{\text{force}} \ r \qquad (y, T_1, r) \ {}^{X'}\Downarrow^{X''}_{\text{bind}} \ r'}{(x, ans(\text{let } y \text{ be } T_1 \text{ in } A), E^{oi}) \ {}^{X}\Downarrow^{X''}_{\text{force}} \ r'}$$

$$\frac{}{(x, var(y, E_1^{oi}), E^{oi}) \ {}^{X}\Downarrow^{X}_{\text{force}} \ var(y, E_1^{oi} :: (\text{let } x \text{ be } \Box \text{ in } E^{oi}[x]))}$$

Figure 7.8: Heapless natural semantics for call-by-need evaluation

7.7.3 Refunctionalization: from natural semantics to higher-order evaluation function

The natural-semantics implementation of Section 7.7.2 is already in defunctionalized form with respect to the first-order outside-in contexts. Indeed, as already mentioned in Section 7.4.4, the recomposition function of Definition 27 and Figure 7.1 is the corresponding apply function.

An outside-in context acts as an accumulator recording the path from a variable whose

$$\mathsf{Result} = (\mu X.\mathsf{Answer} + (\mathsf{Var} \times (X \to X))) \times \mathsf{FreshVars}$$

$$\begin{aligned}
\mathrm{eval} &: \mathsf{Term} \times \mathsf{FreshVars} \to \mathsf{Result} \\
\mathrm{eval}(\ulcorner n \urcorner, X) &= (\mathit{ans}(\ulcorner n \urcorner), X) \\
\mathrm{eval}(x, X) &= (\mathit{var}(x, \overline{\lambda} r.r), X) \\
\mathrm{eval}(\lambda x.T, X) &= (\mathit{ans}(\lambda x.T), X) \\
\mathrm{eval}(T_0\ T_1, X) &= \mathrm{apply}(\mathrm{eval}(T_0, X), T_1) \\
\mathrm{eval}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T, X) &= \mathrm{bind}(x, T_1, \mathrm{eval}(T, X)) \\
\mathrm{eval}(\mathit{succ}\ T, X) &= \mathrm{succ}(\mathrm{eval}(T, X))
\end{aligned}$$

$$\begin{aligned}
\mathrm{apply} &: \mathsf{Result} \times \mathsf{Term} \to \mathsf{Result} \\
\mathrm{apply}((\mathit{ans}(\lambda x.T), (x', X)), T_1) &= \mathrm{bind}(x', T_1, \mathrm{eval}(T[x'/x], X)) \\
\mathrm{apply}((\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A), X), T_2) &= \mathrm{bind}(x, T_1, \mathrm{apply}((\mathit{ans}(A), X), T_2)) \\
\mathrm{apply}((\mathit{var}(x, h), X), T_1) &= (\mathit{var}(x, \overline{\lambda} r.\mathrm{apply}(h\ \overline{@}\ r, T_1)), X)
\end{aligned}$$

$$\begin{aligned}
\mathrm{bind} &: \mathsf{Var} \times \mathsf{Term} \times \mathsf{Result} \to \mathsf{Result} \\
\mathrm{bind}(x, T_1, (\mathit{ans}(A), X)) &= (\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A), X) \\
\mathrm{bind}(x, T_1, (\mathit{var}(x, h), X)) &= \mathrm{force}(x, \mathrm{eval}(T_1, X), h) \\
\mathrm{bind}(x, T_1, (\mathit{var}(y, h), X)) &= (\mathit{var}(y, \overline{\lambda} r.\mathrm{bind}(x, T_1, h\ \overline{@}\ r)), X) \\
&\qquad \text{where } x \neq y
\end{aligned}$$

$$\begin{aligned}
\mathrm{force} &: \mathsf{Var} \times \mathsf{Result} \times (\mathsf{Result} \to \mathsf{Result}) \to \mathsf{Result} \\
\mathrm{force}(x, (\mathit{ans}(V), X), h) &= \mathrm{bind}(x, V, h\ \overline{@}\ (\mathit{ans}(V), X)) \\
\mathrm{force}(x, (\mathit{ans}(\mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A), X), h) &= \mathrm{bind}(y, T_1, \mathrm{force}(x, (\mathit{ans}(A), X), h)) \\
\mathrm{force}(x, (\mathit{var}(y, h'), X), h) &= (\mathit{var}(y, \overline{\lambda} r.\mathrm{force}(x, h'\ \overline{@}\ r, h)), X)
\end{aligned}$$

$$\begin{aligned}
\mathrm{succ} &: \mathsf{Result} \to \mathsf{Result} \\
\mathrm{succ}((\mathit{ans}(\ulcorner n \urcorner), X)) &= (\mathit{ans}(\ulcorner n' \urcorner), X) \quad \text{where } n' = n + 1 \\
\mathrm{succ}((\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A), X)) &= \mathrm{bind}(x, T, \mathrm{succ}((\mathit{ans}(A), X))) \\
\mathrm{succ}((\mathit{var}(x, h), X)) &= (\mathit{var}(x, \overline{\lambda} r.\mathrm{succ}(h\ \overline{@}\ r)), X)
\end{aligned}$$

Figure 7.9: The heapless natural semantics of Figure 7.8 after refunctionalization

value is needed to its binding site. The recomposition function turns this accumulator inside-out again when the variable's value is found. The refunctionalized outside-in contexts are functional representations of these accumulators.

The resulting refunctionalized evaluation function is displayed in Figure 7.9. Notationally, higher-order functions are introduced with $\overline{\lambda}$ and eliminated with $\overline{@}$, which is infix.

Proposition 47 (full correctness)**.**

$$T\ {}^{X}\!\Downarrow^{X'}_{\mathrm{eval}}\ \mathit{ans}(A) \iff \mathrm{eval}(T, X) = (\mathit{ans}(A), X').$$

Proof. Corollary of the correctness of defunctionalization. □

This higher-order evaluation function exhibits a computational pattern that we find striking because it also occurs in Cartwright and Felleisen's work on extensible denota-

tional language specifications [41]: each valuation function yields either a (left-injected with "*ans*") value or a (right-injected with "*var*") variable together with a higher-order function. For each call, this higher-order function may yield another right-injected higher-order function that, when applied, restores this current call. As illustrated in 7.B, this computational pattern is typical of control: the left inject stands for an expected result, while the right inject acts as an exceptional return that incrementally captures the current continuation. This observation also points at a structural commonality in Ariola and Felleisen's small-step semantics [12], which uses delimited control, and in Cartwright and Felleisen's big-step semantics [41], which uses undelimited control. At any rate, for undelimited control, this computational pattern was subsequently re-invented by Fünfrocken to implement process migration [97, 191, 205], and then put to use to implement first-class continuations [140, 158]. In the present case, this pattern embodies two distinct computational aspects—one intensional and the other extensional:

How: The computational pattern is one of delimited control, from the point of use of a let-declared variable to its point of declaration.

What: The computational effect is one of a write-once state since once the delimited context is captured, it is restored with the value of the let-declared variable.

These two aspects were instrumental in Cartwright and Felleisen's design of extensible denotational semantics for (undelimited) Control Scheme and for State Scheme [41]. For let insertion in partial evaluation, these control and state aspects were re-discovered and put to use by Sumii and Kobayashi [203], and for let insertion in type-directed partial evaluation, by Grobauer and Yang [103]. For normalization by evaluation, this control aspect was also re-discovered and put to use by Balat et al. [20], who abstract delimited control from the use site of a lambda-declared variable to its definition site. For call by need, this control aspect was recently identified and put to new use by Garcia, Lumsdaine and Sabry [99], and this store aspect was originally envisioned by Vuillemin [213], Wadsworth [215], and initially Landin [131].

These observations put us in position to write the evaluation function of Figure 7.9 in direct style, either with delimited control operators (one control delimiter for each let declaration, and one control abstraction for each occurrence of a let-declared variable whose value is needed), or with a state monad, as illustrated in 7.B with a simpler example.

7.8 Deterministic abstract machines define functions

Up to Section 7.6, we scrupulously described small-step computation with relations, before shifting to functions for describing big-step computation. However, for deterministic programming languages, functions are sufficient to describe small-step computation, as done throughout the first author's lecture notes at AFP 2008 [61]. For example, in the present work, the decomposition and recomposition functions of Section 7.4, together with the data type of contexts, are in defunctionalized form. They can therefore easily be refunctionalized, as illustrated in 7.A. This representational flexibility indicates a large and friendly degree of expressive freedom for implementing reduction semantics and one-step reduction functions in a functional programming language, not just for the call-by-need λ-calculus, but in general.

7.9 Conclusion

Semantics should be call by need.
– Rod Burstall

Over the years, the two key features of lazy evaluation – demand-driven computation and memoization of intermediate results – have elicited a fascinating variety of semantic artifacts, each with its own originality and elegance. It is our overarching thesis that spelling out the methodical search for the next potential redex that is implicit in a reduction semantics paves the way towards other semantic artifacts that not only are uniformly inter-derivable and sound by construction *but also match what a programming-language semanticist crafts by hand*. Elsewhere, we have already shown that refocusing, etc. do not merely apply to purely syntactic theories such as, e.g., Felleisen and Hieb's syntactic theories of sequential control and state [88, 151]: the methodology also applies to call by need with a global heap of memo-thunks [7, 27, 167] and to combinatory graph reduction, connecting term graph rewriting systems à la Barendregt et al. and graph-reduction machines à la Turner [73, 222]. Here, we have shown that the methodology also applies to Ariola et al.'s purely syntactic account of call by need.

Acknowledgments: Thanks are due to the anonymous reviewers. We are also grateful to Zena Ariola, Kenichi Asai, Ronald Garcia, Oleg Kiselyov, Kristoffer Rose, Ilya Sergey and Chung-chieh Shan for discussions and comments.

The first author heard Rod Burstall's quote in Section 7.9 from Neil Jones in the late 1980s, but was unable to locate it in writing. In May 2009, he asked Rod Burstall about it: Rod Burstall made that statement at Edinburgh at the occasion of a seminar by Christopher Wadsworth in the course of the 1970s.

7.A On refunctionalizing and going back to direct style

The goal of this appendix is to illustrate refunctionalization and the direct-style transformation. Our running example is an evaluator for a simple language of arithmetic expressions.

7.A.1 Abstract machine for evaluating arithmetic expressions

Our language of arithmetic expressions reads as follows:

$$\begin{aligned} \mathsf{Term} \ni T &::= \ulcorner n \urcorner \mid T + T \mid T \times T \\ \mathsf{Value} \ni V &::= \ulcorner n \urcorner \\ \mathsf{Evaluation\ Context} \ni E &::= [\,] \mid E + T \mid V + E \mid E \times T \mid V \times E \end{aligned}$$

In words: terms are integers, additions, and multiplications; values are integers; and evaluation contexts specify a left-most inner-most reduction order.

Here is an abstract machine for this language:

$$\begin{aligned}
\langle \ulcorner n \urcorner, E\rangle_{term} &\to_{\text{step}} \langle E, \ulcorner n \urcorner\rangle_{context} \\
\langle T_1 + T_2, E\rangle_{term} &\to_{\text{step}} \langle T_1, E + T_2\rangle_{term} \\
\langle T_1 \times T_2, E\rangle_{term} &\to_{\text{step}} \langle T_1, E \times T_2\rangle_{term}
\end{aligned}$$

$$\begin{aligned}
\langle [\,], \ulcorner n \urcorner\rangle_{context} &\to_{\text{step}} \langle \ulcorner n \urcorner\rangle_{value} \\
\langle E + T_2, \ulcorner n_1 \urcorner\rangle_{context} &\to_{\text{step}} \langle T_2, \ulcorner n_1 \urcorner + E\rangle_{term} \\
\langle \ulcorner n_1 \urcorner + E, \ulcorner n_2 \urcorner\rangle_{context} &\to_{\text{step}} \langle E, \ulcorner n \urcorner\rangle_{context} \\
&\qquad \text{where } n = n_1 + n_2 \\
\langle E \times T_2, \ulcorner n_1 \urcorner\rangle_{context} &\to_{\text{step}} \langle T_2, \ulcorner n_1 \urcorner \times E\rangle_{term} \\
\langle \ulcorner n_1 \urcorner \times E, \ulcorner n_2 \urcorner\rangle_{context} &\to_{\text{step}} \langle E, \ulcorner n \urcorner\rangle_{context} \\
&\qquad \text{where } n = n_1 \times n_2
\end{aligned}$$

This abstract machine consists of three states: the first defines the relation on terms, the second defines the relation on evaluation contexts, and the third is the terminal state of values. A term T evaluates to a value V iff $\langle T, [\,]\rangle_{term} \to^*_{\text{step}} \langle V\rangle_{value}$.

7.A.2 Small-step implementation of the abstract machine

The abstract machine of Section 7.A.1 can be regarded as a small-step abstract machine defining a relation between any non-final state and its successive state. Let us implement it in Haskell.

Terms, values and evaluation contexts read as follows:

```
data Term = Num Int | Add Term Term | Mul Term Term
type Val  = Int
data Cont = Empty
          | EAddL Cont Term | EAddR Val Cont
          | EMulL Cont Term | EMulR Val Cont
```

States are represented with a data type:

```
data State = TERM Term Cont | CONT Cont Val | VAL Val
```

The `TERM` constructor is used to represent *term* states, `CONT` to represent *context* states, and `VAL` to represent the final state.

Transitions are implemented with a function associating each non-final state to its successive state:

```
step :: State → State
step (TERM (Num n)       e) = CONT e n
step (TERM (Add t1 t2)   e) = TERM t1 (EAddL e t2)
step (TERM (Mul t1 t2)   e) = TERM t1 (EMulL e t2)
step (CONT Empty         n) = VAL n
step (CONT (EAddL e t2) n1) = TERM t2 (EAddR n1 e)
step (CONT (EAddR n1 e) n2) = CONT e (n1 + n2)
step (CONT (EMulL e t2) n1) = TERM t2 (EMulR n1 e)
step (CONT (EMulR n1 e) n2) = CONT e (n1 * n2)
```

Evaluating a term is done by starting in the initial *term* state with the term and the empty context and iterating the transition sequence towards a final state:

```
iterate :: State → Val
iterate (VAL n) = n
iterate state   = iterate (step state)

main0 :: Term → Val
main0 t = iterate (TERM t Empty)
```

7.A.3 Big-step implementation of the abstract machine

The abstract machine of Section 7.A.1 can be equally regarded as a big-step abstract machine defining a function from terms to values [67]. Let us implement it in Haskell.

Terms, values and evaluation contexts read as in Section 7.A.2.

Transitions are implemented with a set of mutually tail-recursive functions:

```
term :: Term → Cont → Val
term (Num n)     e = cont e n
term (Add t1 t2) e = term t1 (EAddL e t2)
term (Mul t1 t2) e = term t1 (EMulL e t2)

cont :: Cont → Val → Val
cont Empty        n  = n
cont (EAddL e t2) n1 = term t2 (EAddR n1 e)
cont (EAddR n1 e) n2 = cont e (n1 + n2)
cont (EMulL e t2) n1 = term t2 (EMulR n1 e)
cont (EMulR n1 e) n2 = cont e (n1 * n2)

main1 :: Term → Val
main1 t = term t Empty
```

The `term` function represents transitions from *term* states; the `cont` function represents transitions from *context* states; and the final return value represents the final *value* states. Evaluating a term is done by invoking the `term`-transition with the term and the empty context.

This implementation is in defunctionalized form with respect to the data type of evaluation contexts, `Cont`, and the function, `cont`, dispatching on that data type: each data constructor of `Cont` is consumed by `cont` which implements how to continue evaluation. Refunctionalization replaces each call to a data constructor of `Cont` by the introduction of a function that implements how to continue evaluation, and each call to `cont` by the elimination of this function, i.e., its application. For example, `cont` maps the data constructor `Empty` to the identity function; `Empty` is thus refunctionalized as the identity function. The function implementing how to continue evaluation is of course the continuation of an evaluator.

7.A.4 Continuation-passing evaluator

Refunctionalizating the abstract machine of Section 7.A.3 yields the following evaluator, which is in continuation-passing style (CPS):

```
evalc :: Term → (Val → a) → a
evalc (Num n)     k = k n
evalc (Add t1 t2) k = evalc t1 (λn1 → evalc t2 (λn2 → k (n1 + n2)))
evalc (Mul t1 t2) k = evalc t1 (λn1 → evalc t2 (λn2 → k (n1 * n2)))

main2 :: Term → Val
main2 t = evalc t (λn → n)
```

This evaluator is in CPS since all calls are tail calls and the second parameter is a continuation.

7.A.5 Direct-style evaluator

Applying the direct-style transformation, i.e., the left inverse of the CPS transformation [56], to the continuation-passing evaluator of Section 7.A.4, we obtain the following evaluator, which is in direct style:

```
eval :: Term → Val
eval (Num n)     = n
eval (Add t1 t2) = eval t1 + eval t2
eval (Mul t1 t2) = eval t1 * eval t2

main3 :: Term → Val
main3 t = eval t
```

CPS-transforming this direct-style evaluator yields the continuation-passing evaluator of Section 7.A.4. Defunctionalizing this continuation-passing evaluator yields the abstract machine of Section 7.A.3. This sequence of program transformations was introduced in Reynolds's work on definitional interpreters 4 decades ago [176]. It was put in the limelight, together with the converse sequence, in the past decade [5, 62].

7.A.6 Natural semantics

The direct-style interpreter of Section 7.A.5 implements the following (big-step) natural semantics:

$$\frac{}{\ulcorner n \urcorner \Downarrow_{\text{eval}} \ulcorner n \urcorner} \qquad \frac{T_1 \Downarrow_{\text{eval}} \ulcorner n_1 \urcorner \quad T_2 \Downarrow_{\text{eval}} \ulcorner n_2 \urcorner}{T_1 + T_2 \Downarrow_{\text{eval}} \ulcorner n \urcorner} \text{ where } n = n_1 + n_2$$

$$\frac{T_1 \Downarrow_{\text{eval}} \ulcorner n_1 \urcorner \quad T_2 \Downarrow_{\text{eval}} \ulcorner n_2 \urcorner}{T_1 \times T_2 \Downarrow_{\text{eval}} \ulcorner n \urcorner} \text{ where } n = n_1 \times n_2$$

A term T evaluates to a value V iff $T \Downarrow_{\text{eval}} V$.

The present natural semantics and the abstract machine of Section 7.A.1 are thus uniformly inter-derivable, and they match what a programming-language semanticist would craft by hand (see Footnote 3, page 138 for a non-trivial recent example).

7.B On the control pattern underlying call by need

The goal of this appendix is to illustrate the control pattern of Figure 7.9. Our running example counts the number of occurrences of each bound variable in a λ-term. More precisely, we define a function mapping a closed λ-term of type `Term1` into a new λ-term of type `Term2` where each binder $\lambda x.T$ has been tagged with the number of free occurrences of x in T.

```
data Term1 = Var1 String
           | Lam1 String Term1
           | App1 Term1 Term1
```

```
data Term2 = Var2 String
           | Lam2 String Int Term2
           | App2 Term2 Term2
```

We present three definitions: one with the control pattern of Figure 7.9 (Appendix 7.B.1), one with its direct-style counterpart using control operators (Appendix 7.B.2), and one in state-passing style (Appendix 7.B.3). All three are implemented in Haskell. We have tested them with the Glasgow Haskell Compiler.

7.B.1 Function-based encoding

The main function, $\texttt{count}_1$, calls an auxiliary function, `visit`, that returns the characteristic sum type of intermediate results: the current answer, left-injected with "`Ans`", or a function resuming the computation of the current intermediate answer, right-injected with "`Var`" and tagged with the variable under consideration:

```
data Intermediate
  = Ans Term2
  | Var String (() → Intermediate)

count1 t = case visit t of
    Ans t' → t'
    Var x h → error "open term"
  where
    visit :: Term1 → Intermediate
    visit (Var1 x)     = var x
    visit (Lam1 x t)   = lam x 0 (visit t)
    visit (App1 t0 t1) = app (visit t0) (visit t1)

    var x = Var x (λ() → Ans (Var2 x))

    lam x n (Ans t) = Ans (Lam2 x n t)
    lam x n (Var y h)
      | x == y      = lam x (n + 1) (h ())
      | otherwise   = Var y (lam x n ∘ h)

    app (Ans t0) (Ans t1) = Ans (App2 t0 t1)
    app (Var x h) r = Var x ((λs → app s r) ∘ h)
    app r (Var x h) = Var x ((λs → app r s) ∘ h)
```

Each time a variable is visited, its continuation is captured from its point of use to its point of definition, its count is incremented, and the captured continuation is restored. The capture is realized by bubbling up with `Var`, as it were, from a point of use to its lexical point of definition while accumulating a delimited continuation by function composition. The restoration is realized by applying this delimited continuation.

7.B.2 Continuation-based encoding

The main function, $\texttt{count}_2$, calls an auxiliary function, `visit`, that delimits control for each variable definition, and abstracts control for each variable use, using Dybvig, Peyton-Jones and Sabry's monadic framework for subcontinuations [83]:

```
import Control.Monad.CC

count2 t = runCC (visit t [])
  where
```

```
visit :: MonadDelimitedCont p s m ⇒
         Term1 → [(String, Term2 → m Term2)] → m Term2
visit (Var1 x) ms =
  (mark x ms) (Var2 x)

visit (Lam1 x t) ms = do
  h ← reset (λp → do
    let k t = shift p (λk → do
                h ← k (return t)
                return (λn → h (n + 1)))
    t' ← visit t ((x, k) : ms)
    shift p (λk → return (λn → Lam2 x n t')))
  return (h 0)
visit (App1 t1 t2) ms = do
  t1' ← visit t1 ms
  t2' ← visit t2 ms
  return (App2 t1' t2')

mark x ms =
  case lookup x ms of
    Just p  → p
    Nothing → error "open term"
```

This implementation reflects the control pattern in 7.B.1 in that the computation incrementing the counter is defined at the point of variable definition. However, since the control abstraction has no free variables, we can unfold it from its definition to its use:

```
count3 t = runCC (visit t [])
  where
    visit :: MonadDelimitedCont p s m ⇒
             Term1 → [(String, p (Int → Term2))] → m Term2
    visit (Var1 x) ms =
      shift (mark x ms) (λk → do
        h ← k (return (Var2 x))
        return (λn → h (n + 1)))
    visit (Lam1 x t) ms = do
      h ← reset (λp → do
        t' ← visit t ((x, p) : ms)
        shift p (λk → return (λn → Lam2 x n t')))
      return (h 0)
    visit (App1 t0 t1) ms = do
      t0' ← visit t0 ms
      t1' ← visit t1 ms
      return (App2 t0' t1')

    mark x ms =
      case lookup x ms of
        Just p  → p
        Nothing → error "open term"
```

Each time a variable is visited, its continuation is captured from its point of use to its point of definition, its count is incremented, and the captured continuation is restored. The capture is realized by using the control operator `shift`, which abstracts control into a delimited continuation. The restoration is realized by applying this delimited continuation.

7.B.3 State-based encoding

The main function, $\texttt{count}_4$, calls an auxiliary function, `visit`, that threads an association list of declared variables and numbers of their occurrences by use of a state monad:

```
import Control.Monad.State

count₄ t = evalState (visit t) []
  where
    visit :: Term1 → State [(String, Int)] Term2
    visit (Var1 x) = do
      modify (incr x)
      return (Var2 x)
    visit (Lam1 x t) = do
      modify ((x, 0):)
      t' ← visit t
      n  ← gets (snd ∘ head)
      modify tail
      return (Lam2 x n t')
    visit (App1 t0 t1) = do
      t0' ← visit t0
      t1' ← visit t1
      return (App2 t0' t1')

    incr x [] = error "open term"
    incr x ((y, n) : ys)
      | x == y    = (y, n + 1) : ys
      | otherwise = (y, n    ) : incr x ys
```

Each time a variable is visited, its association is accessed in the current state, the count in this association is incremented, which yields a new state, and the computation is resumed in this new state.

Chapter 8

A synthetic operational account of call-by-need evaluation

Joint work with Olivier Danvy.

Abstract

We present the first operational account of call by need that connects theory and practice. Theory: the storeless operational semantics using syntax rewriting to account for demand-driven computation and for caching intermediate results. And practice: the store-based operational technique using memo-thunks to implement demand-driven computation and to cache intermediate results for subsequent sharing. The practice was initiated by Landin and Wadsworth and is prevalent today to implement lazy programming languages such as Haskell. The theory was initiated by Launchbury and by Ariola, Felleisen, Maraist, Odersky and Wadler and is prevalent today to reason equationally about lazy programs, on par with Barendregt et al.'s term graphs. Nobody knows, however, how the theory of call by need compares to the practice of call by need: all that is known is that the theory of call by need agrees with the theory of call by name, and that the practice of call by need optimizes the practice of call by name.

Our operational account takes the form of three new calculi for lazy evaluation of lambda-terms and our synthesis takes the form of three lock-step equivalences. The first calculus is a hereditarily compressed variant of Ariola et al.'s call-by-need lambda-calculus and makes "neededness" syntactically explicit. The second calculus distinguishes between strict bindings (which are induced by demand-driven computation) and non-strict bindings (which are used for caching intermediate results). The third calculus uses memo-thunks and an algebraic store. The first calculus syntactically corresponds to a storeless abstract machine, the second to an abstract machine with local stores, and the third to a lazy Krivine machine, i.e., a traditional store-based abstract machine implementing lazy evaluation. The machines are intensionally compatible with extensional reasoning about lazy programs and they are lock-step equivalent. Each machine functionally corresponds to a natural semantics for call by need in the style of Launchbury.

Our results reveal a genuine and principled unity of computational theory and computational practice, one that readily applies to variations on the general theme of call by need.

8.1 Introduction

Seen in the historical HAKMEM report [25, Item 101B]:

```
Let x be a continued fraction
  p0 + q0/(p1 + q1/(...)) = p0 + q0/x'
where x' is again a continued fraction
and the p's and q's are integers. [...]
Instead of a list of p's and q's,
let x be a subroutine
producing its next p and q each time it is called.
Thus on its first usage, x will "output" p0 and q0
and, in effect, change itself into x'.
```

Lovely example of a lazy list and of its evaluation, isn't it? And to think it was programmed in assembly language too...

But what is lazy evaluation? Lazy evaluation is an embodiment of *computation on demand* and of *memoization of intermediate results* for subsequent reuse, in case of subsequent similar demands. In the λ-calculus, lazy evaluation improves the standard reduction of λ-terms. In functional programming languages, lazy evaluation is implemented by passing actual parameters "by need" both to user-defined functions and to data constructors. Call by need is traditionally implemented with memo-thunks, as in Gosper's procedural representation of continued fractions above: parameterless procedures that delay computation and which, when forced, memoize their result in the store. And indeed lazy abstract machines canonically use a store to manage memo-thunks. Alternatively, they use updateable graphs rather than abstract-syntax trees, in which case these graphs play the rôle of the store [81, 82]. Today, the best-known such store-based abstract machine is probably Peyton Jones's Spineless Tagless G-Machine [160], which is the run-time system of the Glasgow Haskell compiler.

Does this mean that a store is inherently necessary to account for lazy evaluation? Perhaps surprisingly, the answer is no: in the mid-1990's [15], in a simultaneous tour de force, Ariola and Felleisen [12] and Maraist, Odersky and Wadler [141] provided a purely syntactic, storeless operational account of call by need.

So what has happened since? Somewhat unexpectedly, the store-based implementation technique of using memo-thunks and the storeless operational account of call by need have remained disconnected.[1] Instead, for example, the control aspect of the storeless operational account has most recently been sought to emulate this storage effect using delimited continuations [99] or more generally a computational monad [77].

So what else has happened instead? Sestoft [195] and Maraist et al. [141] have continued to investigate store-based natural semantics for lazy evaluation, following Launchbury's [134]. Recently, Nakata and Hasegawa have shown the equivalence of variants of the calculus and Launchbury's natural semantics [152] and Chang and Felleisen have designed a new calculus with one axiom and shown it to correspond with Launchbury's natural semantics [43]. Other lazy abstract machines have been designed as well [96, 99]. All of these results provide extensional equivalence between the semantics, which yield equivalent values on identical terminating terms. Also, some of the machines are ostensibly derived but none of the derivation methods seem to have been subsequently reused.

[1] Indeed, the only known property of Ariola et al.'s call-by-need λ-calculus is its completeness with respect to the standard reduction of the λ-calculus (i.e., call by name).

Does this mean that the long-emerged investigation of lazy evaluation is still an ever-expanding and disconnected exploratory process? Actually, no. Ager et al. [7] have inter-derived a store-based call-by-need evaluator and a lazy version of the Krivine machine with a store (Figure 8.2, page 162), using the functional correspondence between evaluators and abstract machines summarized in Appendix 8.A.2 and originally due to Reynolds [5, 176]. Biernacka and Danvy [27, Section 9] have inter-derived a reduction semantics of a λ-calculus with explicit substitutions and a store, $\lambda\widehat{\rho}_{\mathrm{l}}$, and the *same* lazy version of the Krivine machine with a store (Figure 8.2), using the syntactic correspondence between reduction semantics and abstract machines summarized in Appendix 8.A.1. Pirog and Biernacki [167] have inter-derived Launchbury and Sestoft's natural semantics for lazy evaluation and Peyton Jones's Spineless Tagless G-Machine, using Reynolds's functional correspondence and formalizing this correspondence in Coq. We have inter-derived combinatory graph reduction over term graphs à la Barendregt et al. [24] and Turner's graph-reduction machine [209], including the Y combinator [73]. And Ariola, Downen, Herbelin, Nakata and Saurin have put this inter-derivational unity of semantic artifacts to use for call-by-need sequent calculi [17].

So what is the contribution of the present article? We report a similar major unified progress in the investigation of lazy evaluation in the λ-calculus. Macroscopically, as depicted in Figure 8.13, page 178, we present a connection between the two major accounts of lazy evaluation—store-based and storeless—across the three major styles of operational semantics: reduction semantics, abstract machines and natural semantics. And microscopically, as defined in Section 8.4, our connection is based on a *lock-step equivalence* between the small-step semantics.

In what does our synthetic account differ from others? Our account is dual to Hardin, Maranget and Pagano's [107], who seek invariant structures in existing abstract machines using a calculus of explicit substitutions, and to Douence and Fradet's [82], who seek common structures in existing abstract machines to establish a taxonomy. Indeed we *end* with abstract machines as semantic artifacts that are in the common range of the syntactic correspondence (Appendix 8.A.1) and of the functional correspondence (Appendix 8.A.2). We also illustrate how tuning one semantics mechanically gives rise to another, e.g., Launchbury's natural semantics and Sestoft's abstract machine.

Keeping in mind the answers to the series of questions above, the rest of this article is organized as follows:[2] Section 8.2 first contrasts call by name and call by need. Section 8.3 presents our first starting point: a store-based lazy version of the Krivine abstract machine. Section 8.4 presents our notion of lock-step equivalence. Section 8.5 presents our second starting point: a reduction semantics for call-by-need reduction. Section 8.6 then presents abstract machines for call-by-need evaluation, and Section 8.7 natural semantics for call-by-need evaluation. Figure 8.1 depicts our overall story.

Prerequisites and notations: We expect the reader to be acquainted with the formats of a (small-step) reduction semantics, an abstract machine, and a (big-step) natural semantics, as can be gathered, e.g., in Felleisen et al.'s recent textbook [89] and in Danvy's lecture notes at AFP 2008 [61]. We use x and y to range over names and use subscripts

[2] In the wake of the previous 700 papers about lazy evaluation, it is something of a challenge to write an original introduction for yet another article about lazy evaluation. In the 701st paper [77], Danvy et al. started with an unashamedly apocryphal anecdote illustrating computation on demand and memoization of intermediate results. In the present paper, and as no doubt spotted by the zealous reader already, we *wrote* the present introduction in call-by-need style with a series of Socratic questions concluded with a request to keep their answers in mind.

as well as primes to distinguish meta-variables in a syntactic category, e.g., x_0, x_1, x', x''. We assume all initial terms to be closed λ-terms defined by the usual BNF of the pure λ-calculus:

$$\Lambda \ni t ::= x \mid \lambda x.t \mid t\ t$$

and let $fv(t)$ denote the set of free variables in t. When unambiguous, we allow meta-variables from different semantic specifications to overlap. When ambiguous, we superscript the meta-variables with the notion of reduction of the particular semantics, e.g., $t^{\mathcal{R}}$.

8.2 Call by need vs. call by name

Lazy evaluation is an optimization of the standard reduction of λ-terms and as such we expect any two strategies for lazy evaluation to assign identical meanings to identical λ-terms. However, the operational behaviors specified by the two strategies can be readily observed on even small programs, such as this fully applied expression using Church numbers:

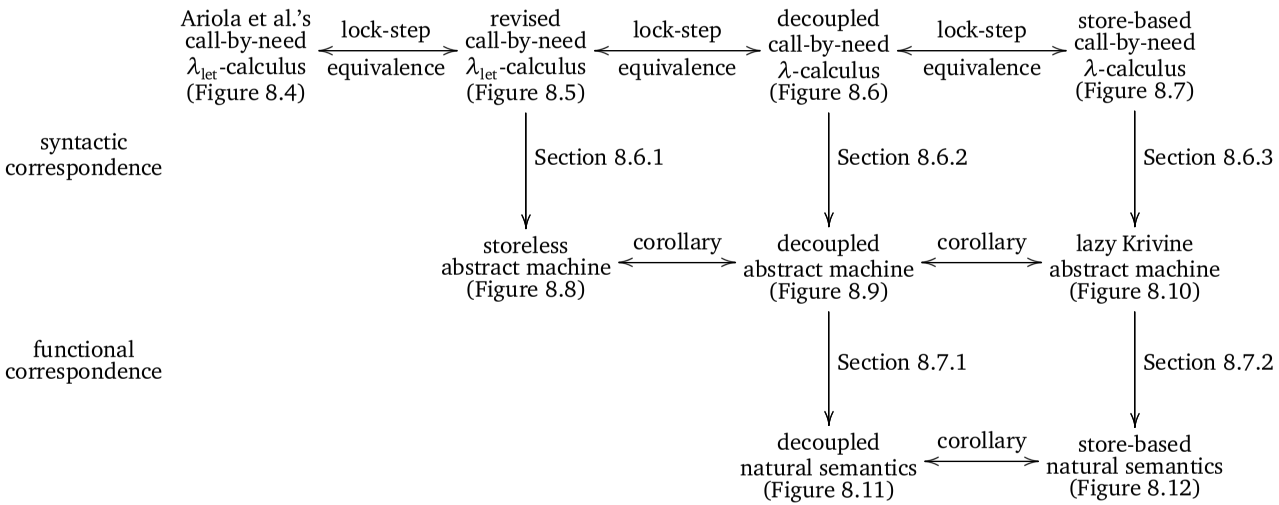
$$\overbrace{c_m\ (c_m\ (\cdots(c_m}^{n}\ id\ id)\cdots)\ id)\ id$$

where $c_n = \lambda s.\lambda z.\overbrace{s\ (s\ \cdots(s}^{n}\ z)\cdots)$. Under call by name, the above expression takes an exponential number of steps to reduce, whereas under call by need, the number of steps is polynomial.

In this regard, call-by-need evaluation is a particular optimization and it is this optimization we want to characterize (so we are not interested in optimal reduction here [19, 138]). As language users and implementors alike, we are interested in reasoning about the time and space complexity of programs under call by need. In other words, *we are concerned with the operational behavior of its specification as opposed to the equational theory it enables*, which is already aptly covered by call by name. We therefore wish to precisely relate the operational behavior specified by the call-by-need λ-calculus to the established method of implementing the call-by-need evaluation strategy. To this end, we first argue for what is a canonical specification of the call-by-need reduction strategy (Section 8.3). We then define a notion of lock-step equivalence by which we can prove the operational behavior of reduction strategies equivalent up to a fine-grained notion of steps (Section 8.4).

8.3 Store-based call-by-need evaluation and reduction

The call-by-need evaluation strategy originates with Wadsworth [215] and was subsequently and independently used for programming languages by Henderson and Morris [113] and by Friedman and Wise [95]. All of these specifications have one idea in common: to delay the evaluation of actual parameters with thunks allocated in a global store, and to force this evaluation on demand and memoize the resulting value in the thunk. In this section, we specify a canonical machine for call-by-need evaluation of pure λ-terms which reflects this implementational practice. We then inter-derive the corresponding notion of one-step reduction.



Small-step to big-step accounts of call by need are shown from top to bottom.
Storeless to store-based accounts of call by need are shown from left to right.

Figure 8.1: Local picture: semantics for call-by-need evaluation

Syntax:

$$\begin{aligned} \mathsf{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \\ \mathsf{Value} \ni v &::= \lambda x.t \\ \mathsf{Context} \ni E &::= \Box \mid E\ t \mid x := E \\ \mathsf{Store} \ni \sigma &::= \varepsilon \mid \sigma[x = t] \end{aligned}$$

Transition rules:

$$\begin{aligned} \langle \lambda x.t,\, E,\, \sigma\rangle_{term} &\to_{\mathscr{S}} \langle E,\, \lambda x.t,\, \sigma\rangle_{cont} \\ \langle t_0\ t_1,\, E,\, \sigma\rangle_{term} &\to_{\mathscr{S}} \langle t_0,\, E[\Box\ t_1],\, \sigma\rangle_{term} \\ \langle x,\, E,\, \sigma\rangle_{term} &\to_{\mathscr{S}} \langle t,\, E[x := \Box],\, \sigma\rangle_{term} \\ &\qquad \text{where } t = \sigma(x) \\ &\qquad\quad \text{and } t \notin \mathsf{Value} \\ \langle x,\, E,\, \sigma\rangle_{term} &\to_{\mathscr{S}} \langle E,\, v,\, \sigma\rangle_{cont} \\ &\qquad \text{where } v = \sigma(x) \\ \\ \langle \Box,\, v,\, \sigma\rangle_{cont} &\to_{\mathscr{S}} \langle v,\, \sigma\rangle_{ans} \\ \langle E[\Box\ t_1],\, \lambda x.t,\, \sigma\rangle_{cont} &\to_{\mathscr{S}} \langle t[x'/x],\, E,\, \sigma[x' = t_1]\rangle_{term} \\ &\qquad \text{where } x' \notin \mathrm{dom}(\sigma) \\ \langle E[x := \Box],\, v,\, \sigma\rangle_{cont} &\to_{\mathscr{S}} \langle E,\, v,\, \sigma[x = v]\rangle_{cont} \end{aligned}$$

Execution starts in a *term*-configuration with an empty context and an empty store, and proceeds through successive transitions. In the second *cont*-transition, an actual parameter is delayed in a thunk. In the second-to-last *term*-transition, a thunk is forced. In the last *cont*-transition, a thunk has completed and its result is memoized.

Figure 8.2: Lazy version of the Krivine abstract machine

8.3.1 A machine for call-by-need evaluation

Our starting point is the properly tail-recursive and lazy variant of the Krivine machine displayed in Figure 8.2 [51]. This machine features memo-thunks in a global store. In words: Terms are pure λ-terms. Values are λ-abstractions. Evaluation contexts consist of the empty context, an application context, and an update context. The machine uses two transition functions: *term* dispatching on terms and *cont* dispatching on evaluation contexts. Here demand-driven computation is implemented by the third *term*-transition and the second *cont*-transition, while memoization is implemented by the last *cont*-transition.

This machine represents a canonical implementation of the call-by-need evaluation strategy using actual substitutions. The update contexts represent "update markers" in the sense of Fairbairn and Wray's Three Instruction Machine [84]. Furthermore, this machine can be inter-derived with traditional store-based call-by-need evaluators, as shown by Ager et al. [7].

Definition 48 (reduction-free evaluation)**.** A term $t \in \Lambda$ evaluates to a value v iff

$$\langle t,\, \Box,\, \varepsilon\rangle_{term} \to^*_{\mathscr{S}} \langle v,\, \sigma\rangle_{ans}$$

holds, where $\to^*_{\mathscr{S}}$ is the transitive closure of $\to_{\mathscr{S}}$. (See Figure 8.2.)

Syntax:

$$\begin{aligned}
\text{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid x := t \\
\text{Value} \ni v &::= \lambda x.t \\
\text{Context} \ni E &::= \Box \mid E\ t \mid x := E \\
\text{Store} \ni \sigma &::= \varepsilon \mid \sigma[x = t] \\
\text{Redex} \ni r &::= v\ t \mid x := v \mid x
\end{aligned}$$

Contraction rules:

$$\begin{array}{lll}
(I) & \langle E[(\lambda x.t)\ t_1], \sigma\rangle \to \langle E[t[x'/x]], \sigma[x' = t_1]\rangle & \text{where } x' \notin \mathrm{dom}(\sigma) \\
(V) & \langle E[x := v], \sigma\rangle \to \langle E[v], \sigma[x = v]\rangle & \\
(L) & \langle E[x], \sigma\rangle \to \langle E[x := t], \sigma\rangle & \text{where } t = \sigma(x) \text{ and } t \notin \mathsf{Value} \\
(\widetilde{V}) & \langle E[x], \sigma\rangle \to \langle E[v], \sigma\rangle & \text{where } v = \sigma(x)
\end{array}$$

$$\mathscr{S} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t, \sigma\rangle \mapsto_{\mathscr{S}} \langle t', \sigma'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ (\langle E[r], \sigma\rangle, \langle E[t''], \sigma'\rangle) \in \mathscr{S} & \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 8.3: The reduction semantics corresponding to Figure 8.2

8.3.2 A semantics for call-by-need reduction

Using Biernacka and Danvy's syntactic correspondence between reduction semantics and abstract machines [27], we can inter-derive the lazy Krivine machine of Section 8.3.1 with a reduction-based counterpart. This reduction-based semantics takes the form of a reduction semantics and is displayed in Figure 8.3. Compared to the machine, an update context is given a term representation in the form of an update expression. The grammar of potential redexes gives rise to the four contraction rules. Here, demand-driven computation is captured by Rule (I) and Rule (L), and memoization is captured by Rule (V).

This reduction semantics is the closure-free counterpart of $\lambda\widehat{\rho}_1$ [27, Section 9].

Definition 49 (reduction-based evaluation)**.** A term $t \in \Lambda$ reduces to a value v iff $\langle t, \varepsilon\rangle \mapsto^*_{\mathscr{S}} \langle v, \sigma\rangle$ holds, where $\mapsto^*_{\mathscr{S}}$ is the reflexive-transitive closure of $\mapsto_{\mathscr{S}}$. (See Figure 8.3.)

8.3.3 From evaluation to reduction

In summary, we have defined a reduction semantics for call-by-need reduction. Through the syntactic correspondence, this semantics corresponds to a store-based lazy version of the Krivine abstract machine which is inter-derivable with a traditional store-based lazy

evaluator. Therefore, these semantics specify the canonical evaluation strategy of call by need.

Proposition 50 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \Box, \varepsilon\rangle_{term} \rightarrow^*_{\mathscr{S}} \langle v_1, \sigma\rangle_{ans} \iff \langle t, \varepsilon\rangle \mapsto^*_{\mathscr{S}} \langle v_2, \sigma\rangle$$

where $v_1 =_\alpha v_2$*.*

Proof. Corollary of the full correctness of refocusing [71, 197]. □

8.4 Lock-step equivalence

Our notion of lock-step equivalence is based on Milner's weak bisimulation or observable equivalence of processes [145, Chapter 5]. In contrast to bisimulation where one equates objects within the same system, we are here interested in relating objects between separately defined systems. Also, we are not concerned with labeling.

We understand a process to be the steps defined by a transition system, in our case the standard reduction sequence as defined by the standard one-step reduction of a reduction semantics. Two processes are lock-step equivalent if any step taken by one can be mirrored by the other modulo the steps considered internal to the other process:

Definition 51 (Lock-step equivalence)**.** Let $\rightarrow_\alpha$ be a transition system with internal transitions $\rightarrow_{\hat{\alpha}}$ and let $\rightarrow_\beta$ be a transition system with internal transitions $\rightarrow_{\hat{\beta}}$. A binary relation $\mathbf{R}$ between the states of $\rightarrow_\alpha$ and $\rightarrow_\beta$ is a lock-step relation if for all $a\,\mathbf{R}\,b$:

$$a \rightarrow_\alpha a' \Rightarrow \exists b' :\ b \rightarrow^*_{\hat{\beta}} \rightarrow_\beta \rightarrow^*_{\hat{\beta}} b' \ \wedge\ a'\,\mathbf{R}\,b'$$

and

$$b \rightarrow_\beta b' \Rightarrow \exists a' :\ a \rightarrow^*_{\hat{\alpha}} \rightarrow_\alpha \rightarrow^*_{\hat{\alpha}} a' \ \wedge\ a'\,\mathbf{R}\,b'$$

Two states a and b are lock-step equivalent, $a \approx_{\alpha\beta} b$, iff there exists a lock-step relation $\mathbf{R}$ such that $a\,\mathbf{R}\,b$. If both sets of internal transitions are empty, then the lock-step relation is akin to a strong bisimulation.

8.5 Reduction semantics for call-by-need reduction

This section presents four reduction semantics for call-by-need evaluation together with their standard reduction. The first one is the λ_{let}-calculus and is due to Ariola, Felleisen, Maraist, Odersky, and Wadler (Section 8.5.1). We then calculate a revised semantics where (1) we hereditarily contract potential redexes, and (2) we make explicit when denotables are needed from their point of use to their point of declaration (Section 8.5.2). This latter distinction leads us to introducing a new term to represent this def-use chain: a strict let expression. A more uniform distinction between strict and non-strict let expressions leads us to 'decoupling' the reduction semantics into one with two contexts (Section 8.5.3). Interpreting one of these two contexts as a global store yields the reduction semantics of Figure 8.3 (see Section 8.5.4) which specifies the canonical evaluation strategy of call by need: Ariola et al.'s reduction semantics therefore also specifies the canonical evaluation strategy of call by need.

Syntax:

$$\begin{aligned}
\text{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \\
\text{Value} \ni v &::= \lambda x.t \\
\text{Answer} \ni a &::= \lambda x.t \mid \mathsf{let}\ x = t\ \mathsf{in}\ a \\
\text{Context} \ni E &::= \Box \mid E\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E[x] \\
\text{Redex} \ni r &::= a\ t \mid \mathsf{let}\ x = a\ \mathsf{in}\ E[x]
\end{aligned}$$

Contraction rules:

$$\begin{array}{ll}
(I) & (\lambda x.t)\ t_1 \to \mathsf{let}\ x = t_1\ \mathsf{in}\ t \\
(C) & (\mathsf{let}\ x = t_1\ \mathsf{in}\ a)\ t_2 \to \mathsf{let}\ x = t_1\ \mathsf{in}\ a\ t_2 \\
(V) & \mathsf{let}\ x = v\ \mathsf{in}\ E[x] \to \mathsf{let}\ x = v\ \mathsf{in}\ E[v] \\
(A) & \mathsf{let}\ x = \mathsf{let}\ y = t_1 \to \mathsf{let}\ y = t_1 \\
 & \qquad\quad \mathsf{in}\ a \qquad\quad\ \mathsf{in let}\ x = a \\
 & \mathsf{in}\ E[x] \qquad\quad\ \mathsf{in}\ E[x]
\end{array}$$

$$\mathscr{R} = (I) \cup (C) \cup (V) \cup (A)$$

Standard one-step reduction:

$$t \mapsto_{\mathscr{R}} t' \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ \text{contraction:} & (r, t'') \in \mathscr{R} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 8.4: The call-by-need λ_{let}-calculus

8.5.1 Storeless reduction semantics

Our starting point is the standard call-by-need reduction for the λ_{let}-calculus that is common to Ariola, Felleisen, Maraist, Odersky, and Wadler's articles [12, 15, 141], renaming non-terminals for notational uniformity. This calculus and its standard reduction are displayed in Figure 8.4. In words: Terms are pure λ-terms with let expressions declaring denotables. Values are λ-abstractions. Answers are let expressions nested around a value. Evaluation contexts are terms with a single hole and are constructed according to the standard call-by-need reduction strategy. Redexes come in two forms: the application of an answer, or the binding of an answer to a denotable whose value is needed. Each gives rise to a pair of contraction rules:

- Rules (I) and (C) arise from the application of an answer. (I) introduces a let binding, while (C) allows let bindings to commute with applications.
- Rules (V) and (A) arise from the binding of an answer to a variable whose value is needed. (V) hygienically substitutes a definiens (here: a value) for a variable occurrence, while (A) re-associates let bindings.

The standard one-step reduction is the compatible closure over the contraction rules with respect to the evaluation contexts.

Definition 52 (reduction-based evaluation [15])**.** A term $t \in \Lambda$ reduces to an answer a iff $t \mapsto^*_{\mathscr{R}} a$ holds, where $\mapsto^*_{\mathscr{R}}$ is the reflexive-transitive closure of $\mapsto_{\mathscr{R}}$. (See Figure 8.4.)

8.5.2 Revised storeless reduction semantics

In this section, we develop a revised version of the storeless reduction semantics with contraction rules that more closely match the rules of the store-based reduction semantics in Section 8.3.2.

Hereditary contraction: Examining the contraction rules of Figure 8.4, we see that a contractum of Rule (C) contains a redex of the form $a\ t$. This redex can thus be further contracted by either Rule (C) or Rule (I). Likewise, a contractum of Rule (A) contains a redex of the form $\mathsf{let}\ x = a\ \mathsf{in}\ E[x]$. This redex can thus be further contracted by either Rule (A) or Rule (V). More precisely, by straightforward induction on the following definition of answer contexts,

$$\mathsf{Ans\ Ctx} \ni A ::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A$$

we relate the following terms under the reflexive transitive closure of the standard one-step reduction relation:

$$\begin{array}{c} A[\lambda x.t]\ t_1 \mapsto^*_{\mathscr{R}} A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t] \\ \mathsf{let}\ x = A[v]\ \mathsf{in}\ E[x] \mapsto^*_{\mathscr{R}} A[\mathsf{let}\ x = v\ \mathsf{in}\ E[v]] \end{array}$$

Strict let expressions: The reduction semantics of Section 8.5.1 cleverly specifies what it means for a denotable to be "needed." Specifically, a denotable x is needed in a term t if t can be uniquely decomposed into $E[x]$. Therefore, the let expression "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t$" can be in one of two states: if $t = E[x]$, we say that the let is *strict*, forcing evaluation of the definiens; and if $t \neq E[x]$, we say that the let is non-strict, postponing the evaluation of the definiens.

Let us make this property syntactically explicit in terms, using a strict let expression "$\mathsf{let}\ x := t\ \mathsf{in}\ E[x]$" where x is needed in the body $E[x]$. In contrast, the original let expression "$\mathsf{let}\ x = t\ \mathsf{in}\ t$" is a non-strict let expression. Strict let expressions are then introduced and eliminated with the following rules:

$$\begin{array}{c} \mathsf{let}\ x = t_1\ \mathsf{in}\ E[x] \to \mathsf{let}\ x := t_1\ \mathsf{in}\ E[x] \\ \mathsf{let}\ x := v\ \mathsf{in}\ E[x] \to \mathsf{let}\ x = v\ \mathsf{in}\ E[v] \end{array}$$

In the case where t_1 is a value, this introduction rule and this elimination rule are applied consecutively in the reduction sequence. To cater for that case, we fuse these two rules into a new one, $(\widetilde{V})$.

Applying the two changes (hereditary contraction and introduction of strict let expressions) to the reduction semantics of Section 8.5.1 we obtain the semantics displayed in Figure 8.5.

Definition 53 (reduction-based evaluation)**.** A term $t \in \Lambda$ reduces to a value v in an answer context A iff $t \mapsto^*_{\mathscr{C}} A[v]$ holds, where $\mapsto^*_{\mathscr{C}}$ is the reflexive-transitive closure of $\mapsto_{\mathscr{C}}$. (See Figure 8.5.)

Proposition 54 (full correctness)**.** *For any closed* $t \in \Lambda$, $t\ {}_{\mathscr{R}}{\approx}_{\mathscr{C}}\ t$.

Proof. There exists a lock-step relation over $\mathscr{R}$ and $\mathscr{C}$, where (C), (A) and (L) are internal transitions. (See Figure 8.15.) □

Syntax:

$$\begin{aligned}
\text{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid \mathsf{let}\ x := t\ \mathsf{in}\ E[x] \\
\text{Value} \ni v &::= \lambda x.t \\
\text{Ans Ctx} \ni A &::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A \\
\text{Context} \ni E &::= \Box \mid E\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ E \mid \mathsf{let}\ x := E\ \mathsf{in}\ E[x] \\
\text{Redex} \ni r &::= A[v]\ t \mid \mathsf{let}\ x := A[v]\ \mathsf{in}\ E[x] \mid \mathsf{let}\ x = t\ \mathsf{in}\ E[x]
\end{aligned}$$

Contraction rules:

$$\begin{aligned}
&(I) && A[\lambda x.t]\ t_1 \to A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t] \\
&(V) && \mathsf{let}\ x := A[v]\ \mathsf{in}\ E[x] \to A[\mathsf{let}\ x = v\ \mathsf{in}\ E[v]] \\
&(L) && \mathsf{let}\ x = t\ \mathsf{in}\ E[x] \to \mathsf{let}\ x := t\ \mathsf{in}\ E[x] \\
& && \qquad\qquad \text{where } t \notin \text{Value} \\
&(\widetilde{V}) && \mathsf{let}\ x = v\ \mathsf{in}\ E[x] \to \mathsf{let}\ x = v\ \mathsf{in}\ E[v]
\end{aligned}$$

$$\mathscr{C} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$t \mapsto_{\mathscr{C}} t' \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ \text{contraction:} & (r,\ t'') \in \mathscr{C} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 8.5: The revised call-by-need λ_{let}-calculus

8.5.3 Decoupled reduction semantics

The reduction semantics of Section 8.5.1 distinguishes strict and non-strict forms only in its specification of evaluation contexts. In Section 8.5.2, strictness is made explicit in terms, yet strict and non-strict forms remain coupled during contraction. Examining the contraction rules of Figure 8.5, we see that strict let expressions are introduced to guide computation while non-strict let expressions are introduced to store intermediate results. In this section, we decouple strict and non-strict contexts, thereby separating the concerns of computation from the concerns of mere storage of intermediate results. The resulting semantics is displayed in Figure 8.6. The defining difference is that non-strict let expressions (in the form of non-strict contexts) are no longer interleaved in the terms. Therefore:

- Terms no longer include non-strict let expressions and strict let expressions delimit just the non-strict context.
- Strict contexts are terms with a hole in strict position: the operand of an application, or the term bound by a strict let expression.
- Non-strict contexts are nested non-strict let expressions.
- Rules (I) and (V) directly place non-strict let bindings in the non-strict context.
- Rules (L) and $(\widetilde{V})$ directly look up denotables in the non-strict context.

Syntax:

$$\begin{aligned}
\text{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x := t\ \mathsf{in}\ A \\
\text{Value} \ni v &::= \lambda x.t \\
\text{Non-strict Context} \ni A &::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A \\
\text{Strict Context} \ni E &::= \Box \mid E\ t \mid \mathsf{let}\ x := E\ \mathsf{in}\ A \\
\text{Redex} \ni r &::= v\ t \mid \mathsf{let}\ x := v\ \mathsf{in}\ A \mid x
\end{aligned}$$

Contraction rules:

$$\begin{array}{lll}
(I) & \langle E[(\lambda x.t)\ t_1], A\rangle \to \langle E[t], A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box]\rangle & \\
(V) & \langle E[\mathsf{let}\ x := v\ \mathsf{in}\ A_1], A\rangle \to \langle E[v], A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle & \\
(L) & \langle E[x], A\rangle \to \langle E[\mathsf{let}\ x := t\ \mathsf{in}\ A_2], A_1\rangle & \\
 & \qquad \text{where } A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] & \\
 & \qquad\quad \text{and } t \notin \mathsf{Value} & \\
(\widetilde{V}) & \langle E[x], A\rangle \to \langle E[v], A\rangle & \\
 & \qquad \text{where } A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2] &
\end{array}$$

$$\mathscr{D} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t, A\rangle \mapsto_{\mathscr{D}} \langle t', A'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ & (\langle E[r], A\rangle, \langle E[t''], A'\rangle) \in \mathscr{D} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 8.6: The decoupled call-by-need λ-calculus

The standard one-step reduction is now defined over a term and a non-strict context:

Definition 55 (reduction-based evaluation)**.** A term $t \in \Lambda$ reduces to a value v and a non-strict context A iff $\langle t, \Box\rangle \mapsto^*_{\mathscr{D}} \langle v, A\rangle$ holds, where $\mapsto^*_{\mathscr{D}}$ is the reflexive-transitive closure of $\mapsto_{\mathscr{D}}$. (See Figure 8.6.)

Proposition 56 (full correctness)**.** *For any closed $t \in \Lambda$,*

$$t \ {}_{\mathscr{C}}{\approx}_{\mathscr{D}} \langle t, \Box\rangle$$

Proof. There exists a lock-step relation over $\mathscr{C}$ and $\mathscr{D}$, with no internal transitions. (See Figure 8.17.) □

8.5.4 Store-based reduction semantics

The reduction semantics of Section 8.5.3 distinguishes between strict and non-strict contexts—strict contexts for guiding computation and non-strict contexts for storing intermediate results. In this section, we accentuate this distinction by interpreting strict contexts as evaluation contexts and by representing non-strict contexts—which, by hygiene, all declare distinct denotables—with a global store. The result is a syntactic theory of the traditional implementation technique for lazy evaluation, the one using memo-thunks in a

Syntax:

$$\begin{aligned} \text{Term} \ni t &::= x \mid \lambda x.t \mid t\ t \mid x := t \\ \text{Value} \ni v &::= \lambda x.t \\ \text{Context} \ni E &::= \Box \mid E\ t \mid x := E \\ \text{Store} \ni \sigma &::= \varepsilon \mid \sigma[x = t] \\ \text{Redex} \ni r &::= v\ t \mid x := v \mid x \end{aligned}$$

Contraction rules:

$$\begin{array}{ll} (I) & \langle E[(\lambda x.t)\ t_1], \sigma\rangle \to \langle E[t], \sigma[x = t_1]\rangle \\ (V) & \langle E[x := v], \sigma\rangle \to \langle E[v], \sigma[x = v]\rangle \\ (L) & \langle E[x], \sigma\rangle \to \langle E[x := t], \sigma\rangle \\ & \qquad \text{where } t = \sigma(x) \\ & \qquad \text{and } t \notin \text{Value} \\ (\widetilde{V}) & \langle E[x], \sigma\rangle \to \langle E[v], \sigma\rangle \\ & \qquad \text{where } v = \sigma(x) \end{array}$$

$$\mathscr{S} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t, \sigma\rangle \mapsto_{\mathscr{S}} \langle t', \sigma'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ & (\langle E[r], \sigma\rangle, \langle E[t''], \sigma'\rangle) \in \mathscr{S} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 8.7: The store-based call-by-need λ-calculus

global store. This semantics is displayed in Figure 8.7. The defining difference is that the global store is never delimited. Therefore:

- Undelimiting update expressions (contexts) replace the delimiting strict let expressions (contexts) in the decoupled semantics.

The standard one-step reduction is defined over a term and a store:

Definition 57 (reduction-based evaluation)**.** A term $t \in \Lambda$ reduces to a value v with a store σ iff $\langle t, \varepsilon\rangle \mapsto^*_{\mathscr{S}} \langle v, \sigma\rangle$ holds, where $\mapsto^*_{\mathscr{S}}$ is the reflexive-transitive closure of $\mapsto_{\mathscr{S}}$. (See Figure 8.7.)

Proposition 58 (full correctness)**.** *For any closed $t \in \Lambda$,*

$$\langle t, \Box\rangle_{\mathscr{D}} \approx_{\mathscr{S}} \langle t, \varepsilon\rangle$$

Proof. There exists a lock-step relation over $\mathscr{D}$ and $\mathscr{S}$, with no internal transitions. (See Figure 8.19.) □

8.5.5 Summary and conclusions

We have presented four lock-step equivalent reduction semantics for call by need:

Corollary 59 (full correctness)**.** *For any closed* $t \in \Lambda$*,*

$$t \;_{\mathscr{R}}\!\approx_{\mathscr{C}} t \;_{\mathscr{C}}\!\approx_{\mathscr{D}} \langle t, \Box\rangle \;_{\mathscr{D}}\!\approx_{\mathscr{S}} \langle t, \varepsilon\rangle$$

Each of these reduction semantics captures a descriptive aspect of call by need: the first one, which is due to Ariola et al., is storeless and the fourth one is store-based and accounts for the traditional implementation technique of using memo-thunks: Figure 8.7 is the implicitly hygienic version of Figure 8.3, which syntactically corresponds to the lazy Krivine machine of Figure 8.2.

8.6 Abstract machines for call-by-need evaluation

In this section, we derive abstract machines from each of the reduction semantics presented in Sections 8.5.2, 8.5.3, and 8.5.4. To this end, we use the syntactic correspondence developed by Biernacka and Danvy [27]. The method consists of a series of program transformations between a reduction-based evaluation function, as typically specified by a reduction semantics, and a reduction-free evaluation function, as typically specified by an abstract machine (see Appendix 8.A.1). We do not display the abstract machine corresponding to the reduction semantics presented in Section 8.5.1 (Ariola et al.'s) because it has already been derived by Danvy et al. [77, Figure 7] and is shown in Figure 7.7.

Following common practice, the reduction semantics of Section 8.5 implicitly assume hygiene in the contraction rules. However, when specifying an abstract machine as the basis for an implementation, the method of ensuring hygiene should be explicit. Mirroring implementation practice, we thread a stream of fresh names with the reduction sequence:

$$X \in \mathsf{FreshNames} = \nu X.\mathsf{Name} \times X.$$

Each contraction rule therefore inherits and synthesizes this stream.

Following Danvy et al.'s hygiene strategy [77, Section 4.2] as detailed in Section 7.4.2, we choose to rename λ-bound names when introducing let expressions, thereby enforcing that all let-bound names are distinct. To this end, we modify the (I)-rule of each reduction semantics:

$$\begin{array}{ll}
(I)^{\mathscr{C}} & \langle E[(\lambda x.t)\ t_1], (x', X)\rangle \to \langle E[\mathsf{let}\ x' = t_1\ \mathsf{in}\ t[x'/x]], X\rangle \\
(I)^{\mathscr{D}} & \langle E[(\lambda x.t)\ t_1], A, (x', X)\rangle \to \langle E[t[x'/x]], A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \Box], X\rangle \\
(I)^{\mathscr{S}} & \langle E[(\lambda x.t)\ t_1], \sigma, (x', X)\rangle \to \langle E[t[x'/x]], \sigma[x' = t_1], X\rangle
\end{array}$$

For each of our reduction semantics, the syntactic correspondence mechanically yields a reduction-free abstract machine where intermediate steps in the reduction sequence have been deforested away.

8.6.1 Storeless abstract machine

Figure 8.8 displays the abstract machine derived from the storeless reduction semantics of Section 8.5.2. It uses the same definition of terms, values, answer contexts and evaluation contexts. (See Figure 8.5.)

We note that the abstract machine of Figure 8.8 is essentially the same as the abstract machines of Garcia et al. [99] and Danvy et al. [77], which differ only with respect to

$$
\begin{aligned}
\langle \lambda x.t,\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle E,\, \lambda x.t\rangle_{cont} \\
\langle t_0\ t_1,\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle t_0,\, E[\Box\ t_1]\rangle_{term} \\
\langle \mathsf{let}\ x = t_1\ \mathsf{in}\ t,\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle t,\, E[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box]\rangle_{term} \\
\langle \mathsf{let}\ x := t\ \mathsf{in}\ E_1[x],\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle t,\, E[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_1[x]]\rangle_{term} \\
\langle x,\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle t,\, E_1[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_2[x]]\rangle_{term} \\
&\qquad \text{where } E = E_1[\mathsf{let}\ x = t\ \mathsf{in}\ E_2] \\
&\qquad\quad \text{and } t \notin \mathsf{Value} \\
\langle x,\, E\rangle_{term} &\xrightarrow{X;X}_{\mathscr{C}} \langle E,\, v\rangle_{cont} \\
&\qquad \text{where } E = E_1[\mathsf{let}\ x = v\ \mathsf{in}\ E_2] \\
\\
\langle \Box,\, A[v]\rangle_{cont} &\xrightarrow{X;X}_{\mathscr{C}} \langle A[v]\rangle_{ans} \\
\langle E[\Box\ t_1],\, A[\lambda x.t]\rangle_{cont} &\xrightarrow{(x',X);X}_{\mathscr{C}} \langle t[x'/x],\, E[A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \Box]]\rangle_{term} \\
\langle E[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box],\, A[v]\rangle_{cont} &\xrightarrow{X;X}_{\mathscr{C}} \langle E,\, \mathsf{let}\ x = t_1\ \mathsf{in}\ A[v]\rangle_{cont} \\
\langle E[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_1[x]],\, A[v]\rangle_{cont} &\xrightarrow{X;X}_{\mathscr{C}} \langle E[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1]],\, v\rangle_{cont}
\end{aligned}
$$

Execution starts in a *term*-configuration with an empty context and proceeds through successive transitions. The stream of fresh names X is threaded through. In the second *cont*-transition, an actual parameter is delayed in a non-strict let expression. In the second-to-last *term*-transition, a non-strict let expression is replaced by a strict let expression, thereby forcing the evaluation of its definiens. In the last *cont*-transition, the evaluation of the definiens has completed and the strict let expression is replaced by a non-strict let expression declaring the resulting value.

Figure 8.8: Storeless machine for call by need

their handling of hygiene. This equivalence arises from two facts: (1) the hereditary compression in the revised semantics is superseded by transition compression of the abstract machine, and (2) even without introducing strict let expressions, their context counterpart must still be represented to guide evaluation in the abstract machine.

Definition 60 (reduction-free evaluation)**.** A term $t \in \Lambda$ evaluates to a value in an answer context $A[v]$ iff

$$\langle t,\, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{C}} \langle A[v]\rangle_{ans}$$

holds, where $\rightarrow^{*}_{\mathscr{C}}$ is the transitive closure of $\rightarrow_{\mathscr{C}}$. (See Figure 8.8.)

Notationally we use $\xrightarrow{X;X'}{}^{*}_{\mathscr{C}}$ to express that X is the input stream and X' is a suffix of X obtained after iterating $\rightarrow_{\mathscr{C}}$.

NB. Assuming the initial term to be a pure λ-term (i.e., to contain no let expressions), we can omit the third and fourth *term*-transitions. Starting from a pure λ-term, the forms $\mathsf{let}\ x = t\ \mathsf{in}\ t$ and $\mathsf{let}\ x := t\ \mathsf{in}\ E[x]$ are indeed never constructed in the course of execution.

Proposition 61 (full correctness)**.** *For any term $t \in \Lambda$,*

$$t \mapsto^{*}_{\mathscr{C}} A_1[v_1] \iff \langle t,\, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{C}} \langle A_2[v_2]\rangle_{ans}$$

where $v_1 =_\alpha v_2$. *(In fact,* $v_1 = v_2$ *if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)*

8.6.2 Decoupled abstract machine

Figure 8.9 displays the abstract machine derived from the decoupled reduction semantics of Section 8.5.3. It uses the same definition of terms, values, strict contexts and non-strict contexts. (See Figure 8.6.)

Definition 62 (reduction-free evaluation)**.** A term $t \in \Lambda$ evaluates to a value v with a non-strict context A iff

$$\langle t, \Box, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{D}} \langle v, A\rangle_{ans}$$

holds, where $\rightarrow^*_{\mathscr{D}}$ is the transitive closure of $\rightarrow_{\mathscr{D}}$. (See Figure 8.9.)

As in Section 8.6.1, assuming the initial term to be a pure λ-term, we can omit the third *term*-transition.

Proposition 63 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \Box\rangle \mapsto^*_{\mathscr{D}} \langle v_1, A_1\rangle \iff \langle t, \Box, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{D}} \langle v_2, A_2\rangle_{ans}$$

where $v_1 =_\alpha v_2$. *(Again,* $v_1 = v_2$ *if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)*

8.6.3 Store-based abstract machine

Figure 8.10 displays the abstract machine derived from the store-based reduction semantics of Section 8.5.4. It uses the same definition of terms, values, evaluation contexts and stores. (See Figure 8.7.)

Definition 64 (reduction-free evaluation)**.** A term $t \in \Lambda$ evaluates to a value v with a store σ iff

$$\langle t, \Box, \varepsilon\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{S}} \langle v, \sigma\rangle_{ans}$$

holds, where $\rightarrow^*_{\mathscr{S}}$ is the transitive closure of $\rightarrow_{\mathscr{S}}$. (See Figure 8.10.)

As in Sections 8.6.1 and 8.6.2, assuming the initial term to be a pure λ-term, we can omit the third *term*-transition. The abstract machine then coincides with the lazy Krivine machine in Figure 8.2.

Proposition 65 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \varepsilon\rangle \mapsto^*_{\mathscr{S}} \langle v_1, \sigma_1\rangle \iff \langle t, \Box, \varepsilon\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{S}} \langle v_2, \sigma_2\rangle_{ans}$$

where $v_1 =_\alpha v_2$. *(One more time,* $v_1 = v_2$ *if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)*

$$
\begin{array}{rl}
\langle \lambda x.t,\, E,\, A\rangle_{term} & \xrightarrow{X;X}_{\mathscr{D}} \langle E,\, \lambda x.t,\, A\rangle_{cont} \\
\langle t_0\ t_1,\, E,\, A\rangle_{term} & \xrightarrow{X;X}_{\mathscr{D}} \langle t_0,\, E[\Box\ t_1],\, A\rangle_{term} \\
\langle \mathsf{let}\ x := t\ \mathsf{in}\ A_1,\, E,\, A\rangle_{term} & \xrightarrow{X;X}_{\mathscr{D}} \langle t,\, E[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_1],\, A\rangle_{term} \\
\langle x,\, E,\, A\rangle_{term} & \xrightarrow{X;X}_{\mathscr{D}} \langle t,\, E[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_2],\, A_1\rangle_{term} \\
 & \quad \text{where } A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] \\
 & \quad \text{and } t \notin \mathsf{Value} \\
\langle x,\, E,\, A\rangle_{term} & \xrightarrow{X;X}_{\mathscr{D}} \langle E,\, v,\, A\rangle_{cont} \\
 & \quad \text{where } A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]
\end{array}
$$

$$
\begin{array}{rl}
\langle \Box,\, v,\, A\rangle_{cont} & \xrightarrow{X;X}_{\mathscr{D}} \langle v,\, A\rangle_{ans} \\
\langle E[\Box\ t_1],\, \lambda x.t,\, A\rangle_{cont} & \xrightarrow{(x',X);X}_{\mathscr{D}} \langle t[x'/x],\, E,\, A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \Box]\rangle_{term} \\
\langle E[\mathsf{let}\ x := \Box,\, v,\, A\rangle_{cont} & \xrightarrow{X;X}_{\mathscr{D}} \langle E,\, v,\, A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle_{cont} \\
\quad \mathsf{in}\ A_1[x]] &
\end{array}
$$

Execution starts in a *term*-configuration with two empty contexts and proceeds through successive transitions. The stream of fresh names X is threaded through. In the second *cont*-transition, an actual parameter is delayed in a non-strict context. In the second-to-last *term*-transition, a non-strict let expression is replaced by a strict let expression, thereby forcing the evaluation of its definiens. In the last *cont*-transition, the evaluation of the definiens has completed and the strict let expression is replaced back by a non-strict let expression declaring the resulting value.

Figure 8.9: Decoupled machine for call by need

8.6.4 Summary and conclusions

We have presented three equivalent abstract machines for call by need:

Corollary 66 (full correctness)**.** *For any $t \in \Lambda$,*

$$
\begin{array}{rl}
 & \langle t,\, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{C}}\ \langle A_1[v_1]\rangle_{ans} \\
\Updownarrow & \\
 & \langle t,\, \Box,\, \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{D}}\ \langle v_2,\, A_2\rangle_{ans} \\
\Updownarrow & \\
 & \langle t,\, \Box,\, \varepsilon\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{S}}\ \langle v_3,\, \sigma\rangle_{ans}
\end{array}
$$

where $v_1 = v_2 = v_3$.

Each of these abstract machines captures a descriptive aspect of call by need: the two first ones are storeless and the third one is store-based and accounts for the traditional implementation technique of using memo-thunks.

8.7 Natural semantics for call-by-need evaluation

In this section, we derive natural semantics from each of the abstract machines derived in Sections 8.6.2 and 8.6.3. To this end, we use the functional correspondence initiated by Reynolds [176] and developed by Danvy et al. [5, 7, 61]. The method consists of a series

$$
\begin{aligned}
\langle \lambda x.t,\, E,\, \sigma\rangle_{term} &\xrightarrow{X;X}_{\mathscr{S}} \langle E,\, \lambda x.t,\, \sigma\rangle_{cont} \\
\langle t_0\ t_1,\, E,\, \sigma\rangle_{term} &\xrightarrow{X;X}_{\mathscr{S}} \langle t_0,\, E[\Box\ t_1],\, \sigma\rangle_{term} \\
\langle x := t,\, E,\, \sigma\rangle_{term} &\xrightarrow{X;X}_{\mathscr{S}} \langle t,\, E[x := \Box],\, \sigma\rangle_{term} \\
\langle x,\, E,\, \sigma\rangle_{term} &\xrightarrow{X;X}_{\mathscr{S}} \langle t,\, E[x := \Box],\, \sigma\rangle_{term} \\
&\qquad \text{where } t = \sigma(x) \\
&\qquad\quad \text{and } t \notin \mathsf{Value} \\
\langle x,\, E,\, \sigma\rangle_{term} &\xrightarrow{X;X}_{\mathscr{S}} \langle E,\, v,\, \sigma\rangle_{cont} \\
&\qquad \text{where } v = \sigma(x)
\end{aligned}
$$

$$
\begin{aligned}
\langle \Box,\, v,\, \sigma\rangle_{cont} &\xrightarrow{X;X}_{\mathscr{S}} \langle v,\, \sigma\rangle_{ans} \\
\langle E[\Box\ t_1],\, \lambda x.t,\, \sigma\rangle_{cont} &\xrightarrow{(x',X);X}_{\mathscr{S}} \langle t[x'/x],\, E,\, \sigma[x' = t_1]\rangle_{term} \\
\langle E[x := \Box],\, v,\, \sigma\rangle_{cont} &\xrightarrow{X;X}_{\mathscr{S}} \langle E,\, v,\, \sigma[x = v]\rangle_{cont}
\end{aligned}
$$

Execution starts in a *term*-configuration with an empty context and an empty store, and proceeds through successive transitions. The store σ and the stream of fresh names X are threaded through. In the second *cont*-transition, an actual parameter is delayed in a thunk. In the second-to-last *term*-transition, a thunk is forced. In the last *cont*-transition, a thunk has completed and its result is memoized.

Figure 8.10: Store-based machine for call by need

of program transformations between an abstract machine and a natural semantics, as typically specified by a recursive evaluation function in direct style (see Appendix 8.A.2). We do not display the natural semantics corresponding to the storeless abstract machine of Section 8.6.1 because it has already been derived by Danvy et al. [77, Figure 8] and is shown in Figure 7.8.

8.7.1 Decoupled natural semantics

Figure 8.11 displays the natural semantics derived from the abstract machine of Section 8.6.2. It uses the same definition of terms, values and non-strict contexts. (See Figure 8.6.)

We note that the natural semantics of Figure 8.11 is essentially the same as Nakata and Hasegawa's instrumented natural semantics [152, Figure 7]. The most notable—and yet superficial—difference is that Nakata and Hasegawa retain the entire structure of the evaluation context as part of their structured heaps whereas we only maintain the structure of the non-strict let expressions.

Definition 67 (evaluation)**.** A term $t \in \Lambda$ evaluates to a value v with a non-strict context A iff $\langle t, \Box\rangle\ {}^{X}\Downarrow^{X'}_{\mathscr{D}}\ \langle v, A\rangle$ holds. (See Figure 8.11.)

NB. Assuming the initial term to be a pure λ-term (i.e., to contain no let expressions), we can omit the third rule. Indeed, starting from a pure λ-term, the form $\mathsf{let}\ x := t\ \mathsf{in}\ A[x]$ can never be constructed by a derivation.

$$\frac{}{\langle \lambda x.t, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X} \ \langle \lambda x.t, A\rangle}$$

$$\frac{\langle t_0, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{(x',X')} \ \langle \lambda x.t, A'\rangle \quad \langle t[x'/x], A'[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \square]\rangle \ {}^{X'}\Downarrow_{\mathscr{G}}^{X''} \ \langle v, A''\rangle}{\langle t_0\ t_1, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X''} \ \langle v, A''\rangle}$$

$$\frac{\langle t, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A'\rangle}{\langle \mathsf{let}\ x := t\ \mathsf{in}\ A_1[x], A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A'[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle}$$

$$\frac{A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] \quad \langle t, A_1\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A_1'\rangle}{\langle x, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A_1'[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]\rangle}\ \text{where } t \notin \mathsf{Value}$$

$$\frac{A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]}{\langle x, A\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]\rangle}$$

Figure 8.11: Decoupled natural semantics for call by need

$$\frac{}{\langle \lambda x.t, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X} \ \langle \lambda x.t, \sigma\rangle}$$

$$\frac{\langle t_0, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{(x',X')} \ \langle \lambda x.t, \sigma'\rangle \quad \langle t[x'/x], \sigma'[x' = t_1]\rangle \ {}^{X'}\Downarrow_{\mathscr{S}}^{X''} \ \langle v, \sigma''\rangle}{\langle t_0\ t_1, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X''} \ \langle v, \sigma''\rangle}$$

$$\frac{\langle t, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X'} \ \langle v, \sigma'\rangle}{\langle x := t, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X'} \ \langle v, \sigma'[x = v]\rangle}$$

$$\frac{t = \sigma(x) \quad \langle t, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X'} \ \langle v, \sigma'\rangle}{\langle x, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X'} \ \langle v, \sigma'[x = v]\rangle}\ \text{where } t \notin \mathsf{Value}$$

$$\frac{v = \sigma(x)}{\langle x, \sigma\rangle \ {}^{X}\Downarrow_{\mathscr{S}}^{X} \ \langle v, \sigma\rangle}$$

Figure 8.12: Store-based natural semantics for call by need

Proposition 68 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \square, \square\rangle_{term} \xrightarrow{X;X'}{}_{\mathscr{G}}^{*} \langle v, A\rangle_{ans} \iff \langle t, \square\rangle \ {}^{X}\Downarrow_{\mathscr{G}}^{X'} \ \langle v, A\rangle.$$

8.7.2 Store-based natural semantics

Figure 8.12 displays the natural semantics derived from the abstract machine of Section 8.10. It uses the same definition of terms, values and stores. (See Figure 8.7.)

Compared to Launchbury's [134] and to Maraist et al.'s [141], the natural semantics in Figure 8.11 explicitly handles name hygiene. Compared to Sestoft's [195], its handling

of name hygiene reflects implementational practice. Also, both Launchbury and Sestoft use preprocessed terms, which prevents a direct syntactic comparison.

Definition 69 (evaluation)**.** A term $t \in \Lambda$ evaluates to a value v with a store σ iff $\langle t, \varepsilon\rangle \; {}^{X}\Downarrow^{X'}_{\mathscr{S}} \; \langle v, \sigma\rangle$ holds. (See Figure 8.12.)

As in Section 8.7.1, assuming the initial term to be a pure λ-term, we can omit the third rule.

Proposition 70 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \Box, \varepsilon\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathscr{S}} \langle v, \sigma\rangle_{ans} \iff \langle t, \varepsilon\rangle \; {}^{X}\Downarrow^{X'}_{\mathscr{S}} \; \langle v, \sigma\rangle.$$

8.7.3 Summary and conclusions

We have presented two equivalent natural semantics for call-by-need evaluation:

Corollary 71 (full correctness)**.** *For any term* $t \in \Lambda$*,*

$$\langle t, \Box\rangle \; {}^{X}\Downarrow^{X'}_{\mathscr{D}} \; \langle v_1, A\rangle \iff \langle t, \varepsilon\rangle \; {}^{X}\Downarrow^{X'}_{\mathscr{S}} \; \langle v_2, \sigma\rangle$$

where $v_1 =_\alpha v_2$*.*

Each of these natural semantics captures a descriptive aspect of call by need: the first one is storeless and the second one is store-based and accounts for the traditional implementation technique of using memo-thunks.

Perhaps surprisingly, Launchbury's natural semantics is to be found *between* the decoupled and store-based natural semantics presented here. The store-based semantics differs since it uses a global store whereas Launchbury's semantics does not. The decoupled semantics more closely connects with Launchbury's semantics. The fourth rule of Figure 8.11 mirrors Launchbury's *Variable* rule in that the binding of a denotable is not visible when reducing its definiens to a value. In addition, the decoupled semantics extends this restriction to all of the bindings not lexically visible according to the call-by-need λ_{let}-calculus thereby exposing inherent structure of store. The same can be said about Sestoft's abstract machine and the abstract machines presented in Section 8.6.2 and 8.6.3.

8.8 Extensions

In this section we briefly describe a few common extensions on the λ-calculus. Adding these extensions and applying the equivalence developed here is at the level of an exercise.

8.8.1 Alias optimization

The introduction of a let expression by Rule (I) corresponds to the dynamic allocation of a delayed application frame. In the case of a denotable, this frame can be eliminated akin to tail-call optimization. We refine Rule (I) as:

$$\begin{aligned} \langle E[(\lambda x.t)\; x_1], X\rangle &\to \langle E[t[x_1/x]], X\rangle \\ \langle E[(\lambda x.t)\; t_1], (x', X)\rangle &\to \langle E[\mathsf{let}\; x' = t_1\; \mathsf{in}\; t[x'/x]], X\rangle \\ &\qquad \text{where } t_1 \notin \mathsf{Name} \end{aligned}$$

This optimization leads to a well-known space optimization of the lazy abstract machine [28, 51, 96].

8.8.2 Generalized contraction

Often, a contraction of Rule (I) directly gives rise to a new (I) redex. In this case, the series of applications can be done in one step by generalizing Rule (I):

$$(I)^{\mathscr{R}} \quad \begin{array}{l} \langle E[(\lambda x_1.\cdots\lambda x_n.t)\ t_1\ \cdots t_n], (x'_1, (\cdots, (x'_n, X)))\rangle \\ \rightarrow \\ \langle E[\mathsf{let}\ x'_1 = t_1, \cdots, x'_n = t_n\ \mathsf{in}\ t[x'_1/x_1, \cdots, x'_n/x_n]], X\rangle \end{array}$$

The resulting abstract machine is a lazy version of Krivine's original abstract machine [128]. (Indeed Krivine's machine implements generalized beta-reduction whereas what is known as the Krivine machine [51] implements ordinary beta-reduction.)

8.8.3 Preprocessing

Alias optimization and generalized contraction can be further exploited if we split the reduction in two phases: a compile-time notion of reduction $\mathscr{R}_0$:

$$\langle E[t_0\ t_1], (x, X)\rangle \rightarrow \langle E[\mathsf{let}\ x = t_1\ \mathsf{in}\ t_0\ x], X\rangle \quad \text{where } t_1 \notin \mathsf{Name}$$

and a run-time notion of reduction $\mathscr{R}_1$ which specializes the rules of $\mathscr{R}$ to the sub-grammar of $\mathscr{R}_0$-normal forms. Indeed, these preprocessed terms are those used by Launchbury [134, Section 3.1] and such preprocessing can be viewed as compiling to a term-graph representation of terms [24, 73].

The global preprocessing of terms invalidates the assumption used to ensure hygiene in Section 8.6. Since preprocessing occurs under λ-binders, the introduced let-bound names might be duplicated during reduction. Proper hygiene must therefore be ensured by another method, e.g., using explicit substitutions or global renaming.

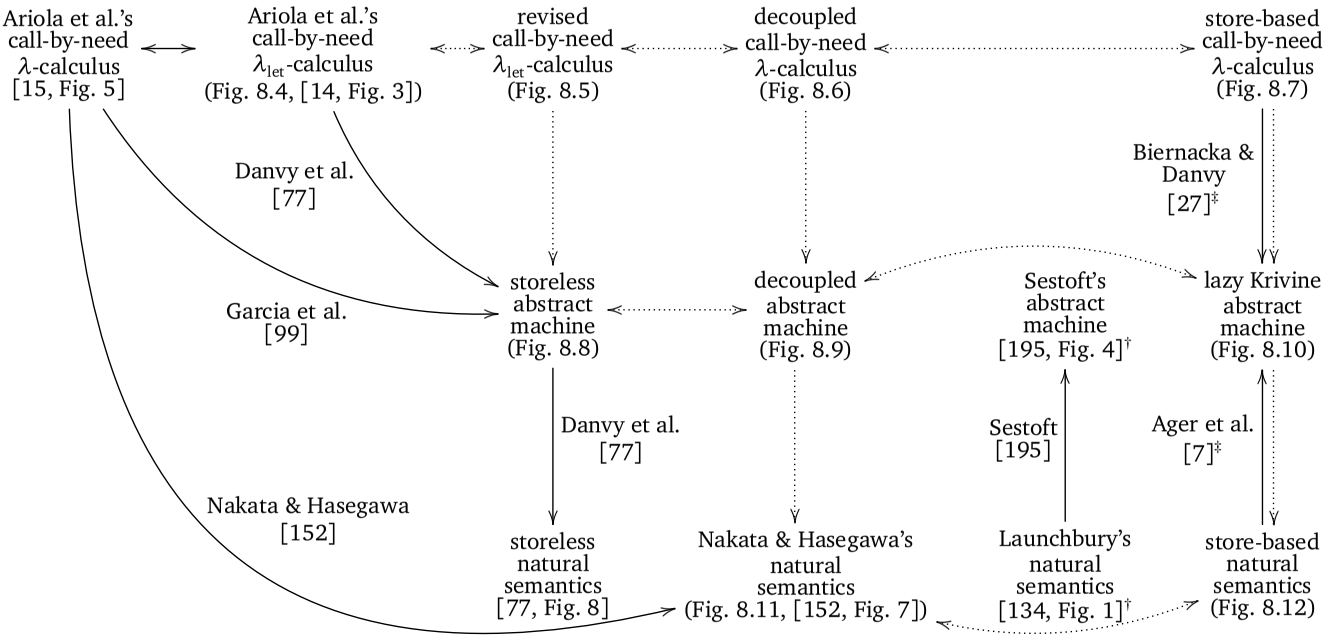
8.8.4 Cyclic terms

All the inter-derivations of Figure 8.1 scale to cyclic structures starting with the mutually recursive letrec as defined by Ariola and Felleisen [12]. We are in the process of proving the lock-step equivalences extended for cyclic terms.

8.9 Conclusion and perspectives

We have presented the first operational account of lazy evaluation that connects theory and practice, where theory stands for a purely syntactic account and practice stands for the traditional implementation technique of imperative memo-thunks. This connection reveals a genuine unity of purpose among theoreticians and implementors and opens the door to more interaction.

Our account is simple, structured and systematic (lock-step equivalence, syntactic correspondence, and functional correspondence). As depicted in Figure 8.13, it also connects



Small-step to big-step accounts of call by need are shown from top to bottom.
Storeless to store-based accounts of call by need are shown from left to right.

† For preprocessed terms.

‡ For explicit substitutions / environments

Figure 8.13: Global picture: semantics for call-by-need evaluation

independent forays, discoveries, and inventions. It however does not readily account for issues pertaining to stackability [21, 44], duality [16], and the factorization of an abstract machine into a byte-code compiler and the corresponding virtual machine [6, 84]—a future work.

8.A Outline of the correspondences

This appendix briefly summarizes the syntactic correspondence and the functional correspondence used in the body of this article to connect reduction semantics, abstract machines and natural semantics. This summary is based on Danvy's lecture notes at AFP 2008 [61].

8.A.1 The syntactic correspondence

The syntactic correspondence makes it possible to inter-derive the representation of a reduction semantics and the representation of an abstract machine as pure functional programs.

A reduction semantics is defined with a grammar of terms, a notion of normal form, a collection of potential redexes, a partial contraction function (this function is partial because not all potential redexes are actual ones: terms may be stuck), and a reduction strategy that determines a grammar of reduction contexts. The reduction strategy is implemented with a decomposition function that maps a term in normal form to itself and a term not in normal form to a potential redex and its reduction context. The recomposition function is a left fold over a reduction context. One-step reduction of a term which is not in normal form (1) locates the first potential redex in this term according to the reduction strategy by decomposing this term into a potential redex and its reduction context, (2) contracts this redex if it is an actual one, and (3) recomposes the contractum over the reduction context, yielding a reduct. Evaluation is defined as the iteration of one-step reduction: this iteration enumerates the reduction sequence; it is thus reduction-based. Evaluation can become stuck, or yield a term in normal form, or diverge.

From reduction-based evaluation to reduction-free evaluation: The goal of refocusing is to deforest, rather than enumerate, the reducts of a reduction sequence. To this end, each consecutive call to decompose over the result of a call to recompose is replaced by one call to a refocus function that optimally navigates from a reduction site to the next reduction site. The result is a small-step abstract machine.

From small-step to big-step abstract machine: The small-step abstract machine is transformed into a big-step abstract machine using Ohori and Sasano's lightweight fusion [67, 155].

Transition compression: The corridor transitions of the big-step abstract machine are compressed.

8.A.2 The functional correspondence

The functional correspondence makes it possible to inter-derive the representation of an evaluation function and the representation of an abstract machine as pure functional programs.

Lambda-lifting: If the evaluation function contains scope-sensitive local functions, their free variables become parameters, and the resulting scope-insensitive functions float up to the top-level lexical scope, yielding recursive equations. Lambda-dropping is the left inverse of lambda-lifting.

Closure conversion: If the recursive equations are higher-order, i.e., use functions as values, these functions are represented as closures, i.e., pairs of terms and lexical environments. Closure unconversion is the left inverse of closure conversion.

Given a compositional evaluation function implementing a denotational semantics, the result of lambda-lifting and closure conversion is a typical functional program implementing a natural semantics.

CPS transformation: All intermediate results are named, their computation is sequentialized (which yields 'A-normal forms'), and all functions are passed an extra function representing the rest of the computation: the continuation. The result of the transformation is in the eponymous Continuation-Passing Style. The direct-style transformation is the left inverse of the CPS transformation.

Defunctionalization: The function space of continuations is partitioned into a sum type. Each introduction of a continuation is transformed into an injection into this sum type, and each elimination of a continuation is transformed into a call to a function dispatching over the sum type. Refunctionalization is the left inverse of defunctionalization.

8.A.3 Synergy

The functional correspondence and the syntactic correspondence synergize because of the concrete coincidence between the data type of evaluation contexts (obtained by defunctionalizing the continuation of the big-step evaluation function) and the data type of reduction contexts (obtained by defunctionalizing the continuation of the small-step reduction function). The abstract connection between reduction order and evaluation order was first pointed out by Plotkin [169].

8.B Lock-step equivalence proofs

This appendix proves lock-step equivalence, as defined in Definition 51, between each of the reduction semantics of Section 8.5.1, 8.5.2, 8.5.3 and 8.5.4. In each of these proofs we define two relations: the first relates the contexts of two semantics, and the second relates the terms of two semantics. The relation on contexts it built such that the relation on terms can be defined in terms of *closing contexts*, i.e., a context that when plugged with a suitable term will produce a closed term. To this end, the context relation includes sets of names that ensure this property. These sets are orthogonal to the proofs and can

freely be ignored. Since decomposition is deterministic, a reducible term can have only one active redex. Thus, each proof proceeds by case analysis on the grammar of redexes.

8.B.1 Original and revised

The difference between the reduction sequences described by the storeless λ_{let}-calculus (Section 8.5.1) and the revised calculus (Section 8.5.2) is that (A) and (C) contractions are done hereditarily and if a denotable is needed during evaluation this is explicitly visible in the syntax of terms. Therefore, any let expression of the form $\mathsf{let}\ x = t_1\ \mathsf{in}\ t$ where $t = E[x]$ is converted to a strict let expression $\mathsf{let}\ x := t_1\ \mathsf{in}\ E[x]$ where we know the decomposition property to always hold for the body of the strict let. We characterize this difference in the fourth rule of **C** in Figure 8.14 by relating let bindings that have the decomposition property with strict let bindings. The remaining rules simply relate the remaining evaluation-context constructions:

Definition 72 (Related contexts)**.** An original context E is related to a revised context E' iff $E\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\ E'$ where $\mathbf{X}$ is the set of denotables lexically visible from the hole of the contexts and $\mathbf{Y}$ is the set of free variables of the contexts. (See Figure 8.14.)

Using the relation on contexts we state the relation on terms as being any pure term in two related contexts, such that the resulting terms are closed:

Definition 73 (Related terms)**.** An original term t is related to a revised term t' iff $t\ \mathbf{B}\ t'$. (See Figure 8.15.)

Proposition 74 (Lock-step relation)**.** *The relation* $\mathbf{B}$ *in Figure 8.15 is a lock-step relation over* $\mathscr{R}$ *and* $\mathscr{C}$ *where* (A)*,* (C) *and* (L) *are internal transitions.*

Proof. Let $t^{\mathscr{R}}\ \mathbf{B}\ t^{\mathscr{C}}$. Assuming that $t^{\mathscr{R}}$ and $t^{\mathscr{C}}$ are reducible terms, we proceed by case analysis on redexes.

Case $a\ t_1$ **and** $A[v]\ t_1$**:** By inversion on $\mathbf{B}$:

$$E^{\mathscr{R}}[A[v]\ t_1]\ \mathbf{B}\ E^{\mathscr{C}}[A[v]\ t_1]$$

The left-hand side contracts by a series of internal (C) transitions followed by a single (I) transition:

$$\begin{aligned} E^{\mathscr{R}}[A[\lambda x.t]\ t_1] &\mapsto^*_{(C)} E^{\mathscr{R}}[A[(\lambda x.t)\ t_1]] \\ &\mapsto_{(I)} E^{\mathscr{R}}[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t]] \end{aligned}$$

and the right-hand side contracts by a single (I) transition:

$$E^{\mathscr{C}}[A[\lambda x.t]\ t_1] \mapsto_{(I)} E^{\mathscr{C}}[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t]]$$

Here both reducts are related by $\mathbf{B}$. (In this case the contractums are even syntactically the same).

Case let $x = a$ in $E_1^{\mathscr{R}}[x]$ **and** let $x := A[v]$ in $E_1^{\mathscr{C}}[x]$**:**
By inversion on **B**:

$$E^{\mathscr{R}}[\mathsf{let}\ x = A[v]\ \mathsf{in}\ E_1^{\mathscr{R}}[x]]\ \mathbf{B}\ E^{\mathscr{C}}[\mathsf{let}\ x := A[v]\ \mathsf{in}\ E_1^{\mathscr{C}}[x]]$$

The left-hand side contracts by a series of internal (A) transitions followed by a single (V) transition:

$$\begin{aligned} E^{\mathscr{R}}[\mathsf{let}\ x = A[v]\ \mathsf{in}\ E_1^{\mathscr{R}}[x]] &\mapsto_{(A)}^{*} E^{\mathscr{R}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{R}}[x]]] \\ &\mapsto_{(V)} E^{\mathscr{R}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{R}}[v]]] \end{aligned}$$

and the right-hand side contracts by a single (V) transition:

$$E^{\mathscr{C}}[\mathsf{let}\ x := A[v]\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \mapsto_{(V)} E^{\mathscr{R}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}[v]]]$$

Here both reducts are related by **B**.

Case let $x = t_1$ in $E_1^{\mathscr{C}}[x]$**:** By inversion on **B**:

$$E^{\mathscr{R}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{R}}[x]]\ \mathbf{B}\ E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]]$$

If $t_1 \in \mathsf{Value}$ then we have the two contractions:

$$\begin{aligned} E^{\mathscr{R}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{R}}[x]] &\mapsto_{(V)} E^{\mathscr{R}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{R}}[t_1]] \\ E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] &\mapsto_{(\tilde{V})} E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[t_1]] \end{aligned}$$

with reducts in **B**.

If $t_1 \notin \mathsf{Value}$ then we have a single internal transition:

$$E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \mapsto_{(L)} E^{\mathscr{C}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]]$$

where

$$E^{\mathscr{R}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1^{\mathscr{R}}[x]]\ \mathbf{B}\ E^{\mathscr{C}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]]$$

□

8.B.2 Revised and decoupled

The difference between the reduction sequences described by the revised calculus (Section 8.5.2) and the decoupled semantics (Section 8.5.3) is that the decoupled semantics has completely separated the non-strict let expressions into a non-strict context. Therefore, in the decoupled semantics, any non-strict let expression is immediately placed outside the evaluation context while preserving the same ordering of non-strict let expressions as in the context-sensitive semantics (Rule 3, Figure 8.16). Strict let expressions can delimit a context in which the bound denotable is needed. To preserve ordering in the decoupled case, the strict let expressions must delimit the non-strict part of the delimited context (Rule 4, Figure 8.16).

Definition 75 (Related contexts)**.** A revised context E is related to a decoupled context $\langle E', A\rangle$ iff $E\ \mathbf{C}_{\mathbf{Y}}^{\mathbf{X}}\ \langle E', A\rangle$ where $\mathbf{X}$ is the set of denotables lexically visible from the hole of the contexts and $\mathbf{Y}$ is the set of free variables of the contexts. (See Figure 8.16.)

$$\frac{}{\Box\ \mathbf{C}^{\emptyset}_{\emptyset}\ \Box}$$

$$\frac{E\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\ E'}{E[\Box\ t]\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\ E'[\Box\ t]}\text{where } t \in \Lambda$$

$$\frac{E\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\ E'}{E[\mathsf{let}\ x = t\ \mathsf{in}\ \Box]\ \mathbf{C}^{\mathbf{X}\cup\{x\}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\ E'[\mathsf{let}\ x = t\ \mathsf{in}\ \Box]}\text{where } t \in \Lambda$$

$$\frac{E_1\ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\ E'_1 \qquad E_2\ \mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\ E'_2 \qquad \text{where } x \notin \mathbf{X}_2}{E_1[\mathsf{let}\ x = \Box\ \mathsf{in}\ E_2[x]]\ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1\cup\{x\})}\ E'_1[\mathsf{let}\ x := \Box\ \mathsf{in}\ E'_2[x]]}$$

Figure 8.14: A lock-step relation on contexts of $\mathscr{R}$ and $\mathscr{C}$

$$\frac{E\ \mathbf{C}^{\mathbf{X}}_{\emptyset}\ E'}{E[t]\ \mathbf{B}\ E'[t]}\text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 8.15: A lock-step relation on terms of $\mathscr{R}$ and $\mathscr{C}$

$$\frac{}{\Box\ \mathbf{C}^{\emptyset}_{\emptyset}\ \langle\Box, \Box\rangle}$$

$$\frac{E\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\ \langle E', A\rangle}{E[\Box\ t]\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\ \langle E'[\Box\ t], A\rangle}\text{where } t \in \Lambda$$

$$\frac{E\ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\ \langle E', A\rangle}{E[\mathsf{let}\ x = t\ \mathsf{in}\ \Box]\ \mathbf{C}^{\mathbf{X}\cup\{x\}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\ \langle E', A[\mathsf{let}\ x = t\ \mathsf{in}\ \Box]\rangle}\text{where } t \in \Lambda$$

$$\frac{E_1\ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\ \langle E'_1, A_1\rangle \qquad E_2\ \mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\ \langle E'_2, A_2\rangle \qquad \text{where } x \notin \mathbf{X}_2}{E_1[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_2[x]]\ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1\cup\{x\})}\ \langle E'_1[E'_2[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_2]], A_1\rangle}$$

Figure 8.16: A lock-step relation on contexts of $\mathscr{C}$ and $\mathscr{D}$

Definition 76 (Related terms)**.** A revised term t is related to a decoupled term and non-strict context $\langle t', A\rangle$ iff $t\ \mathbf{B}\ \langle t', A\rangle$. (See Figure 8.17.)

Lemma 77 (Context composition)**.**
If $E_1^{\mathscr{C}}\ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\ \langle E_1^{\mathscr{D}}, A_1\rangle$ *and* $E_2^{\mathscr{C}}\ \mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\ \langle E_2^{\mathscr{D}}, A_2\rangle$ *then*

$$E_1^{\mathscr{C}}[E_2^{\mathscr{C}}]\ \mathbf{C}^{\mathbf{X}_1\cup\mathbf{X}_2}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1)}\ \langle E_1^{\mathscr{D}}[E_2^{\mathscr{D}}], A_1[A_2]\rangle$$

Proof. By induction on the (second) derivation of $\mathbf{C}$. □

$$\frac{E\ \mathbf{C}_{\emptyset}^{\mathbf{X}}\ \langle E', A\rangle}{E[t]\ \mathbf{B}\ \langle E'[t], A\rangle} \text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 8.17: A lock-step relation on terms of $\mathscr{C}$ and $\mathscr{D}$

Proposition 78 (Lock-step relation)**.** *The relation* $\mathbf{B}$ *in Figure 8.17 is a lock-step relation over* $\mathscr{C}$ *and* $\mathscr{D}$ *with no internal transitions.*

Proof. Let $t^{\mathscr{C}}\ \mathbf{B}\ \langle t^{\mathscr{D}}, A\rangle$. Assuming that $t^{\mathscr{C}}$ and $\langle t^{\mathscr{D}}, A\rangle$ are reducible terms, we proceed by case analysis on redexes.

Case $A[v]\ t_1$ **and** $v\ t_1$**:** By inversion on $\mathbf{B}$:

$$E^{\mathscr{C}}[A[\lambda x.t]\ t_1]\ \mathbf{B}\ \langle E^{\mathscr{D}}[(\lambda x.t)\ t_1], A'[A]\rangle$$

where $E^{\mathscr{C}}\ \mathbf{C}_{\emptyset}^{\mathbf{X}'}\ \langle E^{\mathscr{D}}, A'\rangle$. Thus, we can construct

$$\begin{array}{cc} E^{\mathscr{C}}[A] & \mathbf{C}_{\emptyset}^{\mathbf{X}}\ \langle E^{\mathscr{D}}, A'[A]\rangle \\ E^{\mathscr{C}}[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \square]] & \mathbf{C}_{\emptyset}^{\mathbf{X}\cup\{x\}}\ \langle E^{\mathscr{D}}, A'[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \square]]\rangle \\ E^{\mathscr{C}}[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t]] & \mathbf{B}\ \ \langle E^{\mathscr{D}}[t], A'[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \square]]\rangle \end{array}$$

which accounts for the pair of (I) reductions:

$$\begin{array}{c} E^{\mathscr{C}}[A[\lambda x.t]\ t_1] \mapsto_{(I)} E^{\mathscr{C}}[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t]] \\ \langle E^{\mathscr{D}}[(\lambda x.t)\ t_1], A'[A]\rangle \mapsto_{(I)} \langle E^{\mathscr{D}}[t], A'[A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \square]]\rangle \end{array}$$

and these reducts are in $\mathbf{B}$.

Case $\mathsf{let}\ x := A[v]\ \mathsf{in}\ E_1[x]$ **and** $\mathsf{let}\ x := v\ \mathsf{in}\ A_1$**:** By inversion on $\mathbf{B}$:

$$E^{\mathscr{C}}[\mathsf{let}\ x := A[v]\ \mathsf{in}\ E_1^{\mathscr{C}}[x]]\ \mathbf{B}\ \langle E^{\mathscr{D}}[\mathsf{let}\ x := v\ \mathsf{in}\ A_1], A'[A]\rangle$$

where there exists a $E_1^{\mathscr{D}}$ and a $E_2^{\mathscr{D}}$ such that $E^{\mathscr{D}} = E_1^{\mathscr{D}}[E_2^{\mathscr{D}}]$ and $E^{\mathscr{C}}\ \mathbf{C}_{\emptyset}^{\mathbf{X}}\ \langle E_1^{\mathscr{D}}, A'\rangle$ and $E_1^{\mathscr{C}}\ \mathbf{C}_{\mathbf{Y}'}^{\mathbf{X}'}\ \langle E_2^{\mathscr{D}}, A_1\rangle$ where $\mathbf{Y}' \subseteq \mathbf{X}$. Thus, we can construct:

$$E^{\mathscr{C}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ \square]]\ \mathbf{C}_{\emptyset}^{\mathbf{X}''}\ \langle E_1^{\mathscr{D}}, A'[A[\mathsf{let}\ x = v\ \mathsf{in}\ \square]]\rangle$$

and by Lemma 77:

$$\begin{array}{cc} E^{\mathscr{C}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}]] & \mathbf{C}_{\emptyset}^{\mathbf{X}'''}\ \langle E_1^{\mathscr{D}}[E_2^{\mathscr{D}}], A'[A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]]\rangle \\ E^{\mathscr{C}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}[v]]] & \mathbf{B}\ \ \langle E_1^{\mathscr{D}}[E_2^{\mathscr{D}}[v]], A'[A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]]\rangle \end{array}$$

which accounts for the pair of (V) reductions:

$$\begin{array}{c} E^{\mathscr{C}}[\mathsf{let}\ x := A[v]\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \mapsto_{(V)} E^{\mathscr{C}}[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}[v]]] \\ \langle E^{\mathscr{D}}[\mathsf{let}\ x := v\ \mathsf{in}\ A_1], A'[A]\rangle \mapsto_{(V)} \langle E^{\mathscr{D}}[v], A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle \end{array}$$

and these reducts are in $\mathbf{B}$:

$$\begin{array}{c} E^{\mathscr{C}}[E_a[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}]]\ \mathbf{C}_{\emptyset}^{\mathbf{X}}\ \langle E^{\mathscr{D}}, A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle \\ E^{\mathscr{C}}[E_a[\mathsf{let}\ x = v\ \mathsf{in}\ E_1^{\mathscr{C}}[v]]]\ \mathbf{B}\ \ \langle E^{\mathscr{D}}[v], A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle \end{array}$$

$$\frac{}{\langle \Box, \Box \rangle \ \mathbf{C}^{\emptyset}_{\emptyset} \ \langle \Box, \varepsilon \rangle}$$

$$\frac{\langle E, A \rangle \ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}} \ \langle E', \sigma \rangle}{\langle E[\Box\ t], A \rangle \ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y} \cup (fv(t) \setminus \mathbf{X})} \ \langle E'[\Box\ t], \sigma \rangle} \text{where } t \in \Lambda$$

$$\frac{\langle E, A \rangle \ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}} \ \langle E', \sigma \rangle}{\langle E', A[\mathsf{let}\ x = t\ \mathsf{in}\ \Box] \rangle \ \mathbf{C}^{\mathbf{X} \cup \{x\}}_{\mathbf{Y} \cup (fv(t) \setminus \mathbf{X})} \ \langle E', \sigma[x = t] \rangle} \begin{array}{l} \text{where } t \in \Lambda \\ \text{and } x \notin \mathrm{dom}(\sigma) \end{array}$$

$$\frac{\langle E_1, A_1 \rangle \ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1} \ \langle E_1', \sigma_1 \rangle \qquad \langle E_2, A_2 \rangle \ \mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2} \ \langle E_2', \sigma_2 \rangle}{\langle E_1[E_2[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_2]], A_1 \rangle \ \mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1 \cup (\mathbf{Y}_2 \setminus \mathbf{X}_1 \cup \{x\})} \ \langle E_1'[E_2'[x := \Box]], \sigma_1[\sigma_2[x = t]] \rangle} \begin{array}{l} \text{where } t \in \Lambda \\ \text{and } x \notin \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2) \\ \text{and } \emptyset = \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) \end{array}$$

Figure 8.18: A lock-step relation on contexts of $\mathscr{D}$ and $\mathscr{S}$

Case $\mathsf{let}\ x = t\ \mathsf{in}\ E_1[x]$ **and** x**:** By inversion on **B**:

$$E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1[x]] \ \mathbf{B} \ \langle E^{\mathscr{D}}[x], A[\mathsf{let}\ x = t_1\ \mathsf{in}\ A_1] \rangle$$

where there exists an $E_1^{\mathscr{D}}$, $E_2^{\mathscr{D}}$ such that $E^{\mathscr{D}} = E_1^{\mathscr{D}}[E_2^{\mathscr{D}}]$,
$E^{\mathscr{C}} \ \mathbf{C}^{\mathbf{X}}_{\emptyset} \ \langle E_1^{\mathscr{D}}, A \rangle$, and $E_1^{\mathscr{C}} \ \mathbf{C}^{\mathbf{X}'}_{\mathbf{Y}'} \ \langle E_2^{\mathscr{D}}, A_1 \rangle$ where $\mathbf{Y}' \subseteq \mathbf{X}$.

If $t_1 \in \mathsf{Value}$ then we have a pair of $(\widetilde{V})$ contractions:

$$\begin{array}{c} E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1[x]] \mapsto_{(\widetilde{V})} E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1[t_1]] \\ \langle E^{\mathscr{D}}[x], A[\mathsf{let}\ x = t_1\ \mathsf{in}\ A_1] \rangle \mapsto_{(\widetilde{V})} \langle E^{\mathscr{D}}[t_1], A[\mathsf{let}\ x = t_1\ \mathsf{in}\ A_1] \rangle \end{array}$$

with reducts in **B**.

If $t_1 \notin \mathsf{Value}$ then we have a pair of (L) contractions:

$$\begin{array}{c} E^{\mathscr{C}}[\mathsf{let}\ x = t_1\ \mathsf{in}\ E_1[x]] \mapsto_{(L)} E^{\mathscr{C}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \\ \langle E^{\mathscr{D}}[x], A[\mathsf{let}\ x = t_1\ \mathsf{in}\ A_1] \rangle \mapsto_{(L)} \langle E^{\mathscr{D}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ A_1], A \rangle \end{array}$$

and by the above properties we have that the reducts are in **B**:

$$\begin{array}{c} E^{\mathscr{C}}[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \ \mathbf{C}^{\mathbf{X}}_{\emptyset} \ \langle E_1^{\mathscr{D}}[E_2^{\mathscr{D}}[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_1]], A \rangle \\ E^{\mathscr{C}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ E_1^{\mathscr{C}}[x]] \ \mathbf{B} \ \langle E^{\mathscr{D}}[\mathsf{let}\ x := t_1\ \mathsf{in}\ A_1], A \rangle \end{array}$$

□

8.B.3 Decoupled and store-based

The difference between the reduction sequences described by the decoupled semantics (Section 8.5.3) and the store-based semantics (Section 8.5.4) is that the store-based semantics "forgets" the internal structure of non-strict contexts, i.e., non-strict contexts become an unstructured store. Therefore, in the store-based semantics, bindings in the store must have unique names (Rule 3 and 4, Figure 8.18), and since they are never removed (as is done for non-strict lets) any update expression must have a corresponding binding in the store (Rule 4, Figure 8.18).

$$\frac{\langle E, A\rangle \ \mathbf{C}^{\mathbf{X}}_{\emptyset} \ \langle E', \sigma\rangle}{\langle E[t], A\rangle \ \mathbf{B} \ \langle E'[t], \sigma\rangle} \text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 8.19: A lock-step relation on terms of $\mathscr{D}$ and $\mathscr{S}$

Definition 79 (Related contexts)**.** A decoupled context $\langle E, A\rangle$ is related to a store-based context $\langle E', \sigma\rangle$ iff $\langle E, A\rangle \ \mathbf{C}^{\mathbf{X}}_{\mathbf{Y}} \ \langle E', \sigma\rangle$ where $\mathbf{X}$ is the set of denotables lexically visible from the hole of the contexts and $\mathbf{Y}$ is the set of free variables of the contexts. (See Figure 8.18.)

Definition 80 (Related terms)**.** A decoupled term and non-strict context $\langle t, A\rangle$ is related to a store-based term and store $\langle t', \sigma\rangle$ iff $\langle t, A\rangle \ \mathbf{B} \ \langle t', \sigma\rangle$. (See Figure 8.19.)

Proposition 81 (Lock-step relation)**.** *The relation* $\mathbf{B}$ *in Figure 8.19 is a lock-step relation over* $\mathscr{D}$ *and* $\mathscr{S}$ *with no internal transitions.*

Proof. Let $\langle t^{\mathscr{D}}, A\rangle \ \mathbf{B} \ \langle t^{\mathscr{S}}, \sigma\rangle$. Assuming that $\langle t^{\mathscr{D}}, A\rangle$ and $\langle t^{\mathscr{S}}, \sigma\rangle$ are reducible terms, we proceed by case analysis on redexes.

Case $v \ t_1$ **and** $v \ t_1$**:** By inversion on $\mathbf{B}$:

$$\langle E^{\mathscr{D}}[(\lambda x.t) \ t_1], A\rangle \ \mathbf{B} \ \langle E^{\mathscr{S}}[(\lambda x.t) \ t_1], \sigma\rangle$$

which gives rise to a pair of (I) contractions with reducts in $\mathbf{B}$.

Case $\mathsf{let}\ x := v\ \mathsf{in}\ A_1$ **and** $x := v$**:** By inversion on $\mathbf{B}$:

$$\langle E^{\mathscr{D}}[\mathsf{let}\ x := v\ \mathsf{in}\ A_1], A\rangle \ \mathbf{B} \ \langle E^{\mathscr{S}}[x := v], \sigma\rangle$$

which gives rise to a pair of (V) contractions with reducts in $\mathbf{B}$.

Case x **and** x**:** By inversion on $\mathbf{B}$:

$$\langle E^{\mathscr{D}}[x], A\rangle \ \mathbf{B} \ \langle E^{\mathscr{S}}[x], \sigma\rangle$$

which gives rise to a pair of (L) or $(\widetilde{V})$ contractions with reducts in $\mathbf{B}$.

□

Chapter 9

The inter-derivation of graph rewriting, reduction, and evaluation

This chapter will appear in [223]: Ian Zerny. On graph rewriting, reduction, and evaluation in the presence of cycles. *Higher-Order and Symbolic Computation*, 2013.

An earlier version appeared in [222]: Ian Zerny. On graph rewriting, reduction and evaluation. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Trends in Functional Programming, Volume 10*, pages 81–112, Komárno, Slovakia, June 2009. Intellect Books. Best student-paper award of TFP 2009.

Abstract

We inter-derive two prototypical styles of graph reduction: reduction machines à la Turner and graph rewriting systems à la Barendregt et al. To this end, we adapt Danvy et al.'s mechanical program derivations from the world of terms to the world of cyclic graphs. We also outline how to inter-derive a third style of graph reduction: a graph evaluator.

9.1 Introduction

Graph reduction [215] is a key subject in the specification and implementation of functional programming languages. As such, there is a need for models of graph reduction to reason about the semantics of functional languages. To this end, two general approaches have been developed: the theory of *term-graph rewriting* to specify functional languages as described by Barendregt et al. [24] and the practice of *graph-reduction machines* to implement functional languages as pioneered by Turner [209]. Both approaches give rise to semantic descriptions of their own. The semantics are constructed separately, possess different properties and are used for different purposes. Often, however, the language theoretician has the need for an efficient implementation and the language implementer

has the use of a more abstract model. The problem then arises of relating such semantics, as exemplified for λ-terms and for Landin's SECD machine by Plotkin [68, 131, 169].

We are motivated to consider this problem in the setting of graphs since graph reduction is closer than term reduction to the actual implementations of modern functional languages [210]. Despite this closer relation, the two approaches have yet to be fully connected. Our approach to this problem is to mechanically inter-derive graph rewriting à la Barendregt and reduction machines à la Turner. In doing so we maintain an explicit connection between the two semantic artifacts.

Our domain of discourse is the applicative language of S, K, I and Y combinators with no extensions as defined by the equations:

$$\mathrm{S}\,x\,y\,z = x\,z\,(y\,z), \quad \mathrm{K}\,x\,y = x, \quad \mathrm{I}\,x = x, \quad \text{and} \quad \mathrm{Y}\,x = x\,(\mathrm{Y}\,x).$$

where application is left-associative juxtaposition. Making application explicit with the binary operator A, the left-hand side of the equation for S is: $\mathrm{A}(\mathrm{A}(\mathrm{A}(\mathrm{S},x),y),z)$. The series of applications from the root of such a term to the leftmost combinator, in this case S, is called the *spine* of the term.

This article extends our TFP article [222] to account for cyclic terms. Cyclic terms enable sharing of locally-recursive subgraphs. In the case of Y, this sharing is due to the self-reference of the contractum.

9.1.1 Rewriting à la Barendregt

The work of Barendregt et al. [24] provides an account of term-graph rewriting as an adaptation of term rewriting that includes the notion of graph reduction. Among other things, this work is used to model sharing [11, 100] and to aid language implementation [118, 126], and is part of the foundational work on graph reduction [168].

We briefly exemplify Barendregt et al.'s work with a graph rewriting system for the language of S, K, I and Y. For a more elaborate presentation, we refer readers to the original work [24]. An expression is given by a labelled directed graph over the function symbols $F = \{\mathrm{S},\mathrm{K},\mathrm{I},\mathrm{Y},\mathrm{A}\}$. A graph is defined by a set of nodes N; a label function $lab : N \to F$ mapping nodes to labels; and a partial successor function $succ : N \rightharpoonup N \times N$ from nodes to child nodes. In our case, *succ* is defined on exactly the nodes with label A, which denotes an application, and it produces the operator and operand of the application. A rewrite rule is a triple $\langle g, r, r' \rangle$, where the first component is a graph, and the second and third are nodes, named respectively the 'left root' and the 'right root'. A rewrite will in part consist of redirecting the left root to the right root. The rewrite rules for I, K, S and Y are written below using Barendregt et al.'s notation for graphs and rewriting rules [24, Section 4.4]. Note that + is used to combine graphs, possibly resulting in a disconnected graph.

$$\begin{array}{rlll}
\text{I-rule:} & \langle\ r : \mathrm{A}(\mathrm{I},x), & r, & x\ \rangle \\
\text{K-rule:} & \langle\ r : \mathrm{A}(\mathrm{A}(\mathrm{K},x),y), & r, & x\ \rangle \\
\text{S-rule:} & \langle\ r : \mathrm{A}(\mathrm{A}(\mathrm{A}(\mathrm{S},x),y),z) + r' : \mathrm{A}(\mathrm{A}(x,z),\mathrm{A}(y,z)), & r, & r'\ \rangle \\
\text{Y-rule:} & \langle\ r : \mathrm{A}(\mathrm{Y},x) + r' : \mathrm{A}(x,r'), & r, & r'\ \rangle
\end{array}$$

A rule of the form $\langle g, r, r' \rangle$ is said to be a redex of a graph $\hat{g}$ rooted at $\hat{r}$ if there is a morphism, f, that preserves both labels and successors and maps the subgraph of g rooted at r to $\hat{g}$. In other words, f must be able to construct $\hat{g}$ by filling in placeholders in r and appending the result to some other graph (possibly the empty graph). Rewriting is

performed in three steps: given a redex $\langle g, r, r' \rangle$, (1) build a copy of r', sharing any nodes contained in r; (2) redirect all occurrences of r to the copy of r'; and (3) garbage-collect all nodes that are no longer accessible from $\hat{r}$, which could itself be redirected from r to r'. As usual, a graph is said to be in normal form if no redex exists.

9.1.2 Reduction à la Turner

By adapting graph reduction to Combinatory Logic, Turner created a convenient and efficient target for the implementation of functional languages [209]. He did so by cleverly combining the simple reduction mechanism of Combinatory Logic with graph reduction. A considerable body of work has since followed this path in the form of alternative translation techniques [40], different sets of combinators [116], concurrent and parallel extensions [38, 139] and alternative reduction machines [39, 120, 126, 159, 160], representing the current state of the art in functional-language implementations.

Turner's scheme operated by first translating λ-terms to a graph built with a set of basic combinators amounting to the assembly language of the reduction machine. The reduction machine executes by unwinding the spine of the graph while maintaining an 'ancestor stack'. When a left-hanging atom (a leaf node) is encountered at the end of the spine, the machine applies the contraction rule for this atom. The contraction itself is implemented in terms of a graph transformation, where the arguments are made available through the ancestor stack. By unwinding to the left, the machine implements normal-order reduction, where combinators are reduced with possibly unevaluated arguments. However, for primitive operations, such as addition, the arguments must first be fully evaluated. This requires unwinding to the right while taking special care in representing and manipulating the ancestor stack. Since the language considered here does not have such primitive operations, we instead normalize to full normal form. In contrast, functional-language implementations will typically reduce to weak head normal form. Full normalization ensures reduction under right branches, which incidentally accounts for the operations needed to support primitive operations that are strict in their arguments.

9.1.3 On term rewriting, reduction, and evaluation

Languages defined by terms[1] in the form of abstract-syntax trees have received considerable attention with respect to specifications, implementations and their interconnections. Interconnecting such semantic artifacts is often done by methods tailored to the concrete semantics under consideration. Such methods may provide elegant calculational connections [117], but these connections are provided on a case-by-case basis. An alternative approach is to mechanically inter-derive semantic artifacts by program transformations [62]. The correctness of the connection then follows as a corollary of the correctness of the individual transformations, where the calculations have been done generically once and for all. The derivational approach we consider here has successfully been used to connect a wide range of artifacts and to reflect changes made to one in the other [29, 63, 65, 68, 76, 77]

[1] Terms are directed acyclic graphs where a node can have at most one parent.

9.1.4 On graph rewriting, reduction, and evaluation

Previous work has developed a systematic method to derive abstract machines from calculi for languages defined by terms.[1] In this article we consider languages defined by graphs.[2] The use of graphs raises two issues: the representation includes self-referential graphs; and the interpretation involves sharing-preserving modification by means of copying or assignment. In consideration of these issues, we present the derivation of a reduction machine from a graph rewriting system. We do so by adapting the methods used in the setting of term rewriting and evaluation to the new setting of graph reduction. Our work hinges on the fact that each inter-derived semantic artifact gives rise to the same trace of successive contractions [68]: the inter-derivation acts only on the 'functional glue' between each contraction. We illustrate the method by deriving a reduction machine for the language of S, K, I and Y combinators as the direct result of a series of mechanical program transformations on a graph rewriting system for the same language. Our presentation focuses on the foundational aspects of graph reduction, and we will not be treating any issues related to high-performance reduction machines and language implementation. Our contribution is to connect the graph rewriting system for S, K, I and Y à la Barendregt with the reduction machine for S, K, I and Y à la Turner.

Prerequisites. All of the semantic specifications are presented as implementations in Core Standard ML with additional use of references to account for the implementation of the formal rewriting axioms. Readers acquainted with Standard ML [147] or a related functional language are equipped to follow the presentation. The reader should be familiar with the derivational approach of Danvy et al. [61, 62, 76], in particular refocusing, lightweight fusion, defunctionalization and the CPS transformation. Some prior experience with Combinatory Logic, term rewriting, graph rewriting and reduction machines would come handy and can be obtained from many sources [23, 53, 159, 168].

Overview. The rest of this article is structured as follows. In Section 9.2, we begin by presenting a reduction machine for the language of S, K, I and Y combinators that closely matches the original presentation of Turner. Then we take the calculus of S, K, I and Y combinators in the style of Barendregt et al. and implement a full graph rewriting system in Section 9.3. To this end, we start from a simple specification of the rewriting strategy and derive a full rewriting system implementation where we have made explicit all of the implicit operations of such a system, using the same data types as in Section 9.2. With the graph rewriting system as a starting point, in Section 9.4, we systematically derive the reduction machine of Section 9.2, presenting all intermediate transformations. Finally, in Section 9.5, we consider further transformations to the reduction machine and compare the results with other known artifacts. We conclude in Section 9.6. The complete derivation, along with tests, can be found on the author's home page ⟨`www.zerny.dk/on-graph-rewriting.html`⟩.

9.2 Graph reduction

In this section, we present a reduction machine à la Turner in the form of a state transition system. We start with the type for graphs. A graph is a reference to a node that can either

[2] Graphs are directed cyclic graphs where a node can have more than one parent.

```
(*  stack * graph -> unit  *)
fun setptr (EMPTY, g)
    = ()
  | setptr (PUSH (r as ref (A (_, g1)), _), g)
    = r := A (g, g1)
  | setptr (MARK (r as ref (A (g0, _)), _), g)
    = r := A (g0, g)

(*  graph * stack -> graph  *)
fun unwind (g as ref (A (g0, g1)), gs)
    = unwind (g0, PUSH (g, gs))
  | unwind (g as ref (C a), gs)
    = apply (a, g, gs)

(*  atom * graph * stack -> graph  *)
and apply (I, _, PUSH (r as ref (A (_, x)), gs))
    = (setptr (gs, x);
       unwind (x, gs))
  | apply (K, _, PUSH (      ref (A (_, x)),
                       PUSH (r as ref (A (_, y)),
                             gs)))
    = (r := A (ref (C I), x);
       setptr (gs, x);
       unwind (x, gs))
  | apply (S, _, PUSH (       ref (A (_, x)),
                       PUSH (      ref (A (_, y)),
                             PUSH (r as ref (A (_, z)),
                                   gs))))
    = (r := A (ref (A (x, z)), ref (A (y, z)));
       unwind (r, gs))
  | apply (Y, _, PUSH (r as ref (A (_, x)), gs))
    = (r := A (x, r);
       unwind (r, gs))
  | apply (_, g, gs)
    = continue (gs, g)

(*  stack * graph -> graph  *)
and continue (EMPTY, g)
    = g
  | continue (PUSH (g as ref (A (_, g1)), gs), _)
    = unwind (g1, MARK (g, gs))
  | continue (MARK (g, gs), _)
    = continue (gs, g)

(*  graph -> graph  *)
fun normalize g
    = unwind (g, EMPTY)
```

Figure 9.1: Reduction machine for S, K, I and Y à la Turner

be an atom, corresponding to a basic combinator, or an application of two graphs. We define graphs with the following ML data types, where the label of a node is represented by an ML reference, i.e., a location in memory:

```
datatype atom = I | K | S | Y
datatype node = C of atom | A of graph * graph
withtype graph = node ref
```

In order to reduce the right branch of a graph, we need to store information such that the state can be re-established after completing the reduction of the branch. We choose to specify the abstract reduction machine as a traditional state-transition system and thus cannot recursively unwind on the right branch as Turner does. Instead, we mark the ancestor stack, and when completing reduction on the right branch we pop the mark and re-establish the previous ancestor stack. By marking the stack, we separate the arguments of each function application. More on the various techniques for managing the spine stack can be found in Peyton Jones's treatise [159]. Our stack scheme gives rise to the following data type:

```
datatype stack
  = EMPTY
  | PUSH of graph * stack
  | MARK of graph * stack
```

Figure 9.1 displays the entire reduction machine as a state transition system with three functions `unwind`, `apply` and `continue` that implement the operations described in Section 9.1.2. The auxiliary procedure `setptr` rewires the top of stack pointer. One issue arises when reducing K. The machine cannot globally replace the top most application node with the graph given by x. Only the top of stack can be rewired and any other reference will continue to point to the same K-redex. Instead, following Turner [209, p. 43], an indirection node is installed in the form of an application of I to x, and the machine proceeds to rewire the top of stack to point at x. It is the process of reducing $\mathrm{Y}\,x$ that produces cyclic graphs. The result of the reduction is the application of the argument to the application itself. This cyclic graph represents the infinite expansion of applications of the argument: $x(x(x(\ldots)))$.

9.3 Graph rewriting

In this section we develop an implementation that is faithful to the graph rewriting system à la Barendregt as described in Section 9.1.1. We do so by investigating each of the required steps in turn, making explicit the algorithms and data structures involved following the style and terminology of Danvy et al. [61, 62].

We start by reusing the data type of graphs from the previous section. For any instance of type `graph`, we can construct unique maps for

$$lab : \texttt{graph} \to F$$
$$succ : \texttt{graph} \rightharpoonup \texttt{graph} \times \texttt{graph}$$

and ML references aptly account for a set of unique node identifiers. Thus, the implementation faithfully accounts for a graph.

The rewriting system consists of two parts: the rewriting rules specify how to rewrite the graph (Section 9.3.1); and the rewriting strategy specifies what to rewrite (Section 9.3.2).

9.3.1 Rewriting rules

In this section we implement the rewriting rules of the system. Remember that graph rewriting consisted of three phases: building, redirecting and garbage collection. We omit the treatment of garbage collection and simply rely on the underlying run-time system of Standard ML to collect unreachable graphs.

As described in Section 9.1.1, we have four rules: one for each of I, K, S and Y. For each rule, we have a corresponding constructor in the data type `redex`:

```
datatype redex
  = RED_I of graph * graph                  (* (r, x)        *)
  | RED_K of graph * graph * graph          (* (r, x, y)     *)
  | RED_S of graph * graph * graph * graph  (* (r, x, y, z)  *)
  | RED_Y of graph * graph                  (* (r, x)        *)
```

Each constructor provides the structure reachable from the 'left root' of the redex triple. Having found a redex, it is to be redirected, or contracted, to form the resulting graph. For redirecting nodes we use two auxiliary procedures: `replace` that in-place updates the contents of an application node, and `rewire` that redirects a single reference in the immediate context. Their definitions are:

```
(*  graph * graph * graph -> graph  *)
fun replace (g, gl, gr)
    = (g := A (gl, gr); g)

(*  context * graph -> graph  *)
fun rewire (CTX_MT, g)
    = g
  | rewire (CTX_L (g, gr, k), gl)
    = (g := A (gl, gr); gl)
  | rewire (CTX_R (g, gl, k), gr)
    = (g := A (gl, gr); gr)
```

The rewriting axioms are implemented by the `contract` function:

```
(*  redex -> graph  *)
fun contract (RED_I (r, x), k)
    = rewire (k, x)
  | contract (RED_K (r, x, y), k)
    = (replace (r, ref (C I), x); rewire (k, x))
  | contract (RED_S (r, x, y, z), k)
    = replace (r, ref (A (x, z)), ref (A (y, z)))
  | contract (RED_Y (r, x), k)
    = replace (r, x, r)
```

For an I-redex, we have nothing to build and simply rewire the root to x. For a K-redex, we likewise have nothing to build and proceed to rewire the root to x. As noted in Section 9.2, the root of K is an application node and we cannot simply overwrite it with x. Instead, we replace it with an indirection node and only rewire the immediate reference to the root. For an S-redex, we construct nodes for the parts not shared between the left and right root of the rewrite rule and then replace the root application with the newly constructed nodes. For a Y-redex, we replace the root application with the application of the argument to the root itself, thereby constructing a cyclic term. Note that `contract`, which implements the rewriting axioms, is an impure and total function: impure since effects are used for in-place rewriting, and total since contraction is defined on all redexes and as such does not give rise to stuck graphs.

9.3.2 Rewriting strategy

In this section we implement the rewriting strategy for the system. Following recent work by Danvy et al. [76], we define the rewriting strategy via a compositional search function over the term language, in our case graphs. We then derive a decomposition algorithm by which we can specify the rewriting system in the style of a reduction semantics [89].

Compositional search. The leftmost-outermost rewriting strategy considered here is a depth-first left-to-right search through the graph. We implement this strategy as a function that maps a reducible graph to the first potential redex in the reduction sequence or, if the graph is in normal form, to itself:

```
datatype redex_or_nf
  = REDEX of redex
  | NF    of graph
```

The search is carried out with a straight recursive descent on the graph according to the rewriting strategy. While descending the search maintains an ancestor list containing ancestors and right-siblings. Applications are first searched to the left, pushing the ancestor and sibling on the list. If no redex is found, the search continues to the right. When a combinator is found, its arguments are found in the ancestor list. If the combinator is fully applied a redex is returned. Otherwise, the combinator is not reducible and its subgraph is therefore in normal from:

```
(*  graph * (graph * graph) list -> redex_or_nf  *)
fun search_graph1 (g as ref (C a), gs)
    = (case (a, gs)
        of (I, (g0, x) :: gs)
           => REDEX (RED_I (g0, x))
         | (K, (g0, x) :: (g1, y) :: gs)
           => REDEX (RED_K (g1, x, y))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs)
           => REDEX (RED_S (g2, x, y, z))
         | (Y, (g0, x) :: gs)
           => REDEX (RED_Y (g0, x))
         | (a, gs)
           => NF g)
  | search_graph1 (g as ref (A (gl, gr)), gs)
    = (case search_graph1 (gl, (g, gr) :: gs)
        of REDEX red
           => REDEX red
         | NF gl
           => (case search_graph1 (gr, [])
                 of REDEX red
                    => REDEX red
                  | NF gr
                    => NF g))

(*  graph -> redex_or_nf  *)
fun search1 g = search_graph1 (g, [])
```

Due to the Y combinator, graphs can contain cycles. The search function is therefore partial.

Continuation-passing style transformation. Next we CPS transform the search function to obtain a tail-recursive search function with a higher-order encoding of the search context:

```
(*  graph * (graph * graph) list * (redex_or_nf -> 'a) -> 'a  *)
fun search_graph2 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs)
           => k (REDEX (RED_I (g0, x)))
         | (K, (g0, x) :: (g1, y) :: gs)
           => k (REDEX (RED_K (g1, x, y)))
```

```
          | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs)
            => k (REDEX (RED_S (g2, x, y, z)))
          | (Y, (g0, x) :: gs)
            => k (REDEX (RED_Y (g0, x)))
          | (a, gs)
            => k (NF g))
    | search_graph2 (g as ref (A (gl, gr)), gs, k)
      = search_graph2 (gl, (g, gr) :: gs,
          fn REDEX red
             => k (REDEX red)
           | NF gl
             => search_graph2 (gr, [],
                  fn REDEX red
                     => k (REDEX red)
                   | NF gr
                     => k (NF g)))

(*  graph -> redex_or_nf  *)
fun search2 g = search_graph2 (g, [], fn x => x)
```

Defunctionalization. We then defunctionalize the continuations to obtain a first-order representation of the search context. The CPS search function constructs continuations at three distinct sites:

1. The initial continuation, with no free variables.
2. The continuation when searching to the left, with free variables for the current graph, its right subgraph and the current continuation.
3. The continuation when searching to the right, with free variables for the current graph and the current continuation.

We represent defunctionalized continuations by the data type `cont` with a constructor for each of the continuations:

```
datatype cont
  = C0
  | C1 of graph * graph * cont
  | C2 of graph * cont
```

A defunctionalized continuation is now applied with $\texttt{search\_cont}_3$ where the free variables are supplied by the `cont` data type:

```
(*  cont -> redex_or_nf -> redex_or_nf  *)
fun search_cont3 C0
    = (fn x => x)
  | search_cont3 (C1 (g, gr, k))
    = (fn REDEX red
           => search_cont3 k (REDEX red)
         | NF gl
           => search_graph3 (gr, [], C2 (g, k)))
  | search_cont3 (C2 (g, k))
    = (fn REDEX red
          => search_cont3 k (REDEX red)
        | NF gr
          => search_cont3 k (NF g))
(*  graph * (graph * graph) list * cont -> redex_or_nf  *)
and search_graph3 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs)
```

```
            => search_cont3 k (REDEX (RED_I (g0, x)))
          | (K, (g0, x) :: (g1, y) :: gs)
            => search_cont3 k (REDEX (RED_K (g1, x, y)))
          | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs)
            => search_cont3 k (REDEX (RED_S (g2, x, y, z)))
          | (Y, (g0, x) :: gs)
            => search_cont3 k (REDEX (RED_Y (g0, x)))
          | (a, gs)
            => search_cont3 k (NF g))
  | search_graph3 (g as ref (A (gl, gr)), gs, k)
    = search_graph3 (gl, (g, gr) :: gs, C1 (g, gr, k))

(* graph -> redex_or_nf *)
fun search3 g = search_graph3 (g, [], C0)
```

Notice how each case of `search_cont3` is defined simply as the corresponding continuation from the CPS search function, replacing the nested continuation in `C1` by the defunctionalized continuation `C2`.

Uncurrying. The defunctionalized search function is in curried form and each call site is fully applied. We uncurry it to eliminate higher-order functions entirely:

```
(* cont * redex_or_nf -> redex_or_nf *)
fun search_cont4 (C0, x)
    = x
  | search_cont4 (C1 (g, gr, k), REDEX red)
    = search_cont4 (k, REDEX red)
  | search_cont4 (C1 (g, gr, k), NF gl)
    = search_graph4 (gr, [], C2 (g, k))
  | search_cont4 (C2 (g, k), REDEX red)
    = search_cont4 (k, REDEX red)
  | search_cont4 (C2 (g, k), NF gr)
    = search_cont4 (k, NF g)

(* graph * (graph * graph) list * cont -> redex_or_nf *)
and search_graph4 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs)
           => search_cont4 (k, REDEX (RED_I (g0, x)))
         | (K, (g0, x) :: (g1, y) :: gs)
           => search_cont4 (k, REDEX (RED_K (g1, x, y)))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs)
           => search_cont4 (k, REDEX (RED_S (g2, x, y, z)))
         | (Y, (g0, x) :: gs)
           => search_cont4 (k, REDEX (RED_Y (g0, x)))
         | (a, gs)
           => search_cont4 (k, NF g))
  | search_graph4 (g as ref (A (gl, gr)), gs, k)
    = search_graph4 (gl, (g, gr) :: gs, C1 (g, gr, k))

(* graph -> redex_or_nf *)
fun search4 g = search_graph4 (g, [], C0)
```

Simplification. We notice that redexes are final results of the search function:

Property 82 (redexes are final)**.** For an any continuation `k : cont` and redex `red : redex`, `search_cont4(k, REDEX red)` = `REDEX red`.

Proof. By induction on `k : cont`. □

With this property we simplify the search function:

```
(*  cont * graph -> redex_or_nf  *)
fun search_cont5 (C0, g)
    = NF g
  | search_cont5 (C1 (g, gr, k), gl)
    = search_graph5 (gr, [], C2 (g, k))
  | search_cont5 (C2 (g, k), gr)
    = search_cont5 (k, g)

(*  graph * (graph * graph) list * cont -> redex_or_nf  *)
and search_graph5 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs)
           => REDEX (RED_I (g0, x))
         | (K, (g0, x) :: (g1, y) :: gs)
           => REDEX (RED_K (g1, x, y))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs)
           => REDEX (RED_S (g2, x, y, z))
         | (Y, (g0, x) :: gs)
           => REDEX (RED_Y (g0, x))
         | (a, gs)
           => search_cont5 (k, g))
  | search_graph5 (g as ref (A (gl, gr)), gs, k)
    = search_graph5 (gl, (g, gr) :: gs, C1 (g, gr, k))

(*  graph -> redex_or_nf  *)
fun search5 g = search_graph5 (g, [], C0)
```

Also, at any point of the search, the ancestor list can be constructed from the defunctionalized continuations:

Property 83 (ancestor list from continuation)**.** For any `gs : (graph * graph) list` and `k : cont`, at any point in the search: `gs` = `k2ls(k)`, given by the total translation on defunctionalized continuations:

```
(*  cont -> (graph * graph) list  *)
fun k2ls (C0)            = []
  | k2ls (C1 (g, gr, k)) = (g, gr) :: k2ls k
  | k2ls (C2 (g, k))     = []
```

Proof. Holds for initial calls to $\texttt{search\_graph}_5$ and is preserved by $\texttt{search\_graph}_5$. □

Indeed, the ancestor list is used last-in first-out and thus behaves like Turner's ancestor stack. With this property we further simplify the search, replacing all uses of the ancestor list with corresponding uses of the defunctionalized continuations:

```
(*  cont * graph -> redex_or_nf  *)
fun search_cont6 (C0, g)
    = NF g
  | search_cont6 (C1 (g, gr, k), gl)
    = search_graph6 (gr, C2 (g, k))
  | search_cont6 (C2 (g, k), gr)
    = search_cont6 (k, g)
(*  graph * cont -> redex_or_nf  *)
and search_graph6 (g as ref (C a), k)
    = (case (a, k)
        of (I, C1 (g0, x, k'))
```

```
            => REDEX (RED_I (g0, x))
          | (K, C1 (g0, x, C1 (g1, y, k')))
            => REDEX (RED_K (g1, x, y))
          | (S, C1 (g0, x, C1 (g1, y, C1 (g2, z, k'))))
            => REDEX (RED_S (g2, x, y, z))
          | (Y, C1 (g0, x, k'))
            => REDEX (RED_Y (g0, x))
          | (a, k)
            => search_cont6 (k, g))
  | search_graph6 (g as ref (A (gl, gr)), k)
    = search_graph6 (gl, C1 (g, gr, k))

(*  graph -> redex_or_nf  *)
fun search6 g = search_graph6 (g, C0)
```

Redex in context. Finally, let us make the search function return not just the redex of a reducible graph but also the associated context of that redex, represented by the defunctionalized continuation. Together, the redex and its associated context form a *decomposition* of the graph changing the return type of search:

```
datatype decomposition_or_nf
  = DEC of redex * cont
  | NF  of graph
```

The context we are interested in is that at the root of the redex, therefore we pair the redex with `k'` and not `k`:

```
(*  cont * graph -> decomposition_or_nf  *)
fun search_cont7 (C0, g)
    = NF g
  | search_cont7 (C1 (g, gr, k), gl)
    = search_graph7 (gr, C2 (g, k))
  | search_cont7 (C2 (g, k), gr)
    = search_cont7 (k, g)

(*  graph * cont -> decomposition_or_nf  *)
and search_graph7 (g as ref (C a), k)
    = (case (a, k)
        of (I, C1 (g0, x, k'))
           => DEC (RED_I (g0, x), k')
         | (K, C1 (g0, x, C1 (g1, y, k')))
           => DEC (RED_K (g1, x, y), k')
         | (S, C1 (g0, x, C1 (g1, y, C1 (g2, z, k'))))
           => DEC (RED_S (g2, x, y, z), k')
         | (Y, C1 (g0, x, k'))
           => DEC (RED_Y (g0, x), k')
         | (a, k)
           => search_cont7 (k, g))
  | search_graph7 (g as ref (A (gl, gr)), k)
    = search_graph7 (gl, C1 (g, gr, k))

(*  graph -> decomposition_or_nf  *)
fun search7 g = search_graph7 (g, C0)
```

Decomposition. The derived search function implements the decomposition of a reducible graph into a redex and its reduction context. The defunctionalized continuations

coincide with the reduction contexts in the sense of a reduction semantics. To make this clear, we rename `cont` to `context` and give the constructors descriptive names:

```
datatype context
  = CTX_MT
  | CTX_L of graph * graph * context
  | CTX_R of graph * graph * context
```

Here, `CTX_L` marks a traversal on the left subgraph, where we store current graph (the ancestor) along with its right subgraph (the sibling). Likewise, we store the current graph for `CTX_R` but not the right subgraph since it is not used when deconstructing the context. We could dispense with the right subgraph in `CTX_L`, since it is indirectly accessible through the ancestor graph. This context representation and its use is reminiscent of the zipper technique for traversing data structures in functional programs [115].

To reflect its purpose, we rename search to decompose:

```
(*  context * graph -> decomposition_or_nf  *)
fun decompose_context (CTX_MT, g)
    = NF g
  | decompose_context (CTX_L (g, gr, k), gl)
    = decompose_graph (gr, CTX_R (g, gl, k))
  | decompose_context (CTX_R (g, gl, k), gr)
    = (g := A (gl, gr); decompose_context (k, g))

(*  graph * context -> decomposition_or_nf  *)
and decompose_graph (g as ref (C a), k)
    = (case (a, k)
        of (I, CTX_L (g0, x, k'))
           => DEC (RED_I (g0, x), k')
         | (K, CTX_L (g0, x, CTX_L (g1, y, k')))
           => DEC (RED_K (g1, x, y), k')
         | (S, CTX_L (g0, x, CTX_L (g1, y, CTX_L (g2, z, k'))))
           => DEC (RED_S (g2, x, y, z), k')
         | (Y, CTX_L (g0, x, k'))
           => DEC (RED_Y (g0, x), k')
         | (a, k)
           => decompose_context (k, g))
  | decompose_graph (g as ref (A (gl, gr)), k)
    = decompose_graph (gl, CTX_L (g, gr, k))

(*  graph -> decomposition_or_nf  *)
fun decompose g = decompose_graph (g, CTX_MT)
```

Recall that the search function, and thus this derived decomposition function, is a pure partial function.

Recomposition. After decomposing a graph and contracting the redex, we need to recreate the graph. Since contraction actually modifies the graph in place, recomposing is simply finding the root of the graph:

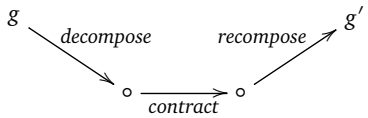
```
(*  context * graph -> graph  *)
fun recompose (CTX_MT, g)
    = g
  | recompose (CTX_L (g, gr, k), gl)
    = recompose (k, g)
  | recompose (CTX_R (g, gl, k), gr)
    = recompose (k, g)
```

Recomposition is a pure total function.

One-step reduction. We implement one-step reduction as the process of decomposing, contracting and recomposing:

```
(*  graph -> graph  *)
fun reduce g
    = (case decompose g
        of NF g'
           => g'
         | DEC (red, k)
           => recompose (k, contract (red, k)))
```

This reduction maps a normal form to itself and a reducible graph to its reduct:

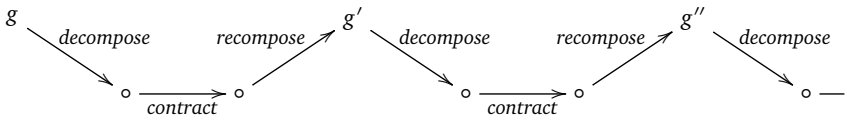


One-step reduction is an impure partial function, since contraction is effectful and decomposition can diverge on circular graphs.

Normalization. We implement normalization as the iteration of one-step reduction. The iteration implements normal-order reduction, which is known to terminate with a normal form should one exist. This result generalizes to the setting of graphs since normal-order reduction in Combinatory Logic is hypernormalizing [24]. Normalization is implemented by the following definitions:

```
(*  decomposition_or_nf -> graph  *)
fun iterate0 (NF g)
    = g
  | iterate0 (DEC (red, k))
    = iterate0 (decompose (recompose (k, contract (red, k))))

(*  graph -> graph  *)
fun normalize0 g
    = iterate0 (decompose g)
```

This reduction-based normalization proceeds by enumerating each reduct in the reduction sequence:



Normalization is an impure partial function, since contraction is effectful, decomposition can diverge on circular graphs, and furthermore, the normalization process might diverge if the graph has no normal form. In other accounts of graph reduction these two forms of partiality can be distinguished. The partiality of decomposition corresponds to "black holes" or observable self-reference, whereas the partiality of normalization corresponds to the usual notion of diverging computation.

This concludes our implementation of the graph rewriting system. For each step, we have done no more than make explicit the operations that are implicit in the abstract account – remaining faithful to the calculus à la Barendregt.

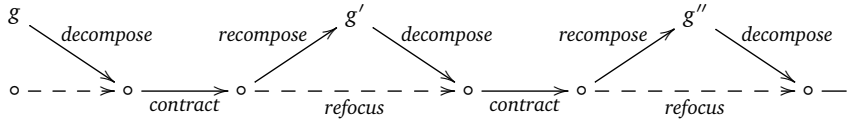
```
(*  graph * context -> decomposition_or_nf  *)
fun refocus1 (g, k)
    = decompose_graph (g, k)

(*  decomposition_or_nf -> graph  *)
fun iterate1 (NF g)
    = g
  | iterate1 (DEC (red, k))
    = iterate1 (refocus1 (contract (red, k), k))

(*  graph -> graph  *)
fun normalize1 g
    = iterate1 (refocus1 (g, CTX_MT))
```

Figure 9.2: Small-step abstract machine obtained by refocusing

9.4 Connecting graph rewriting and graph reduction

With the graph rewriting system of Section 9.3 as our starting point, we successively submit it to the program transformations of Biernacka and Danvy's syntactic correspondence [28, 29] lifted to the level of graphs. For an overview of the program transformations we refer to the work of Danvy et al. [61, 62, 76].

Refocusing. Our first step is to refocus the reduction-based normalization from the previous section. Refocusing avoids repeated decomposition and recomposition, in effect deforesting the intermediate results. The result is a reduction-free normalization function that directly finds the next redex without first navigating to the root of the graph.



As mentioned in the previous section, recomposition is a pure total function and decomposition is a pure partial function. Thus their composition is a pure partial function. In particular, so is their deforested composition: side effects are only used for the formal rewriting axioms of the system and are confined to `contract`. Thus the refocusing transformation acts only on the 'functional glue' between each contraction. The deforested composition we choose is due to Danvy and Nielsen [71], which simply consists in continuing the decomposition at the contraction site. In other words, $\texttt{refocus}_1$ is an alias for `decompose_graph`. The result is an abstract machine iterating contraction and refocusing. More precisely, as displayed in Figure 9.2, it is a small-step abstract machine with $\texttt{refocus}_1$ (which is pure) and then `contract` (which is impure) as the transition function and a 'driver loop', $\texttt{iterate}_1$.

Contraction unfolding. We then unfold `contract` into $\texttt{iterate}_1$, resulting in the function $\texttt{iterate}_2$, which dispatches on the redex of a decomposition with one case per contraction rule:

```
(*  context * graph -> graph  *)
fun refocus_context3 (CTX_MT, g)
    = iterate3 (NF g)
  | refocus_context3 (CTX_L (g, gr, k), gl)
    = refocus_graph3 (gr, CTX_R (g, gl, k))
  | refocus_context3 (CTX_R (g, gl, k), gr)
    = refocus_context3 (k, g)

(*  graph * context -> graph  *)
and refocus_graph3 (g as ref (C a), k)
    = (case (a, k)
        of (I, CTX_L (g0, x, k'))
           => iterate3 (DEC (RED_I (g0, x), k'))
         | (K, CTX_L (g0, x, CTX_L (g1, y, k')))
           => iterate3 (DEC (RED_K (g1, x, y), k'))
         | (S, CTX_L (g0, x, CTX_L (g1, y, CTX_L (g2, z, k'))))
           => iterate3 (DEC (RED_S (g2, x, y, z), k'))
         | (Y, CTX_L (g0, x, k'))
           => iterate3 (DEC (RED_Y (g0, x), k'))
         | (a, k)
           => refocus_context3 (k, g))
  | refocus_graph3 (g as ref (A (gl, gr)), k)
    = refocus_graph3 (gl, CTX_L (g, gr, k))

(*  decomposition_or_nf -> graph  *)
and iterate3 (NF g)
    = g
  | iterate3 (DEC (RED_I (r, x), k))
    = refocus_graph3 (rewire (k, x), k)
  | iterate3 (DEC (RED_K (r, x, y), k))
    = (replace (r, ref (C I), x);
       refocus_graph3 (rewire (k, x), k))
  | iterate3 (DEC (RED_S (r, x, y, z), k))
    = refocus_graph3 (replace (r, ref (A (x, z)), ref (A (y, z))), k)
  | iterate3 (DEC (RED_Y (r, x), k))
    = refocus_graph3 (replace (r, x, r), k)

(*  graph -> graph  *)
fun normalize3 g
    = refocus_graph3 (g, CTX_MT)
```

Figure 9.3: Big-step abstract machine obtained by lightweight fusion

```
(*  decomposition_or_nf -> graph  *)
fun iterate2 (NF g)
    = g
  | iterate2 (DEC (RED_I (r, x), k))
    = iterate2 (refocus2 (rewire (k, x), k))
  | iterate2 (DEC (RED_K (r, x, y), k))
    = (replace (r, ref (C I), x);
       iterate2 (refocus2 (rewire (k, x), k)))
  | iterate2 (DEC (RED_S (r, x, y, z), k))
    = iterate2 (refocus2 (replace (r, ref (A (x, z)), ref (A (y, z))), k))
  | iterate2 (DEC (RED_Y (r, x), k))
    = iterate2 (refocus2 (replace (r, x, r), k))
```

```
(*  context * graph -> graph  *)
fun refocus_context4 (CTX_MT, g)
    = g
  | refocus_context4 (CTX_L (g, gr, k), gl)
    = refocus_graph4 (gr, CTX_R (g, gl, k))
  | refocus_context4 (CTX_R (g, gl, k), gr)
    = refocus_context4 (k, g)

(*  graph * context -> graph  *)
and refocus_graph4 (g as ref (C a), k)
    = (case (a, k)
        of (I, CTX_L (g0, x, k'))
           => refocus_graph4 (rewire (k', x), k')
         | (K, CTX_L (g0, x, CTX_L (g1, y, k')))
           => (replace (g1, ref (C I), x);
               refocus_graph4 (rewire (k', x), k'))
         | (S, CTX_L (g0, x, CTX_L (g1, y, CTX_L (g2, z, k'))))
           => refocus_graph4 (replace (g2, ref (A (x, z)),
                                       ref (A (y, z))), k')
         | (Y, CTX_L (g0, x, k'))
           => refocus_graph4 (replace (g0, x, g0), k')
         | (a, k)
           => refocus_context4 (k, g))
  | refocus_graph4 (g as ref (A (gl, gr)), k)
    = refocus_graph4 (gl, CTX_L (g, gr, k))

(*  graph -> graph  *)
fun normalize4 g
    = refocus_graph4 (g, CTX_MT)
```

Figure 9.4: Reduction machine obtained by transition compression

Lightweight fusion. By lightweight fusion [67, 155] of $\texttt{iterate}_2$ and $\texttt{refocus}_2$ (as defined by `decompose_graph` and `decompose_context`) we transform the small-step abstract machine into a big-step abstract machine in the sense that the functions $\texttt{iterate}_3$, $\texttt{refocus\_graph}_3$, and $\texttt{refocus\_context}_3$ have become transition functions, as shown in Figure 9.3. Here $\texttt{refocus\_graph}_3$ is the composition of $\texttt{iterate}_2$ and `decompose_graph`, while $\texttt{refocus\_context}_3$ is the composition of $\texttt{iterate}_2$ and `decompose_context` that directly calls $\texttt{iterate}_3$ instead of returning to the caller.

Compression of corridor transitions. We proceed to compress the corridor transitions, meaning we inline any call with a uniquely known target. For example, the call $\texttt{iterate}_3$`(NF g)` is known to be handled by the first case in $\texttt{iterate}_3$, and we therefore replace it with `g`. Completing this process we obtain the code in Figure 9.4, where the iteration process has been completely inlined.

Notice how similar the machine of Figure 9.4 is to the reduction machine of Section 9.2 in Figure 9.1. The two are in fact the same where:

- $\texttt{refocus\_graph}_4$ coincides with `unwind` where `apply` has been inlined;
- $\texttt{refocus\_context}_4$ coincides with `continue`;
- the type `context` coincides with `stack` where the right graph component in `CTX_L` is obtained indirectly via the graph root; and
- `rewire` coincides with `setptr`, and the definition of `replace` has been inlined.

Thus we have directly derived Turner's reduction machine from an implementation of Barendregt et al.'s graph rewriting system, using a series of simple and mechanical program transformations. This derivation is significant in two ways: (1) it connects Turner's reduction machine and Barendregt et al.'s graph rewriting system, which is new; and (2) it shows that Biernacka and Danvy's syntactic correspondence scales to term-graph rewriting, which is also new.

9.5 Towards graph evaluation

In this section, we briefly investigate whether Reynolds's functional correspondence [5], which further connects abstract machines and evaluators for terms, can supply a counterpart for reduction machines and evaluators for graphs.

The reduction machine of Section 9.2 is not in defunctionalized form [69, 70]. Each of `refocus_context`$_4$, `refocus_graph`$_4$, and our utility procedure `rewire` deconstructs the data type of contexts. In order to refunctionalize the machine, there must be only a single procedure deconstructing the data type of contexts. To obtain a reduction machine that is in defunctionalized form, we reintroduce the ancestor list we eliminated in Section 9.3.2. In particular, we represent each context by a list of ancestors, corresponding to a series of `CTX_L`s, and a stack context, corresponding to the occurrence of a `CTX_R`, to mark a traversal on a right subgraph. The stack context is defined as:

```
datatype stack_context
  = INIT
  | FRAME of graph * graph * (graph * graph) list * stack_context
```

The translation from contexts into pairs of ancestor lists and stack contexts is:

Property 84 (context to ancestor list and stack contexts)**.** For any `k : context`, `split k` = `(gs, sk)` where the translation is defined by:

```
(*  context -> (graph * graph) list * stack_context  *)
fun split (CTX_MT)
    = ([], INIT)
  | split (CTX_L (g, gr, k))
    = let val (gs, sk) = split k
      in ((g, gr) :: gs, sk)
      end
  | split (CTX_R (g, gl, k))
    = let val (gs, sk) = split k
      in ([], FRAME (g, gl, gs, sk))
      end
```

This translation has a uniquely determined inverse and thus forms a bijection between the two context representations. In addition, we must move the rewiring on `CTX_R` into the frame case of `refocus_context`$_5$, since any call to rewire on a right frame will become a call to rewire on an empty stack followed by a transition to `refocus_context`$_5$ from `refocus_graph`$_5$.[3] Using this translation, we replace each occurrence of the context with its translation:

[3] Compared to the previous machine, this placement of rewire can result in an additional but harmless assignment that replaces the contents of a cell by itself, e.g., in the case of `refocus_graph`$_6$`(ref (C S), [], k)`.

```
(*  (graph * graph) list * graph -> graph  *)
fun rewire' ([], g)
    = g
  | rewire' ((r, gr) :: _, gl)
    = (r := A (gl, gr); gl)

(*  (graph * graph) list * stack_context * graph -> graph  *)
fun refocus_context5 ([], INIT, g)
    = g
  | refocus_context5 ((g, gr) :: gs, k, gl)
    = refocus_graph5 (gr, [], FRAME (g, gl, gs, k))
  | refocus_context5 ([], FRAME (g, gl, gs, k), gr)
    = (g := A (gl, gr); refocus_context5 (gs, k, g))

(*  graph * (graph * graph) list * stack_context -> graph  *)
and refocus_graph5 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs')
           => refocus_graph5 (rewire' (gs', x), gs', k)
         | (K, (g0, x) :: (g1, y) :: gs')
           => (replace (g1, ref (C I), x);
               refocus_graph5 (rewire' (gs', x), gs', k))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs')
           => refocus_graph5 (replace (g2, ref (A (x, z)),
                                       ref (A (y, z))), gs', k)
         | (Y, (g0, x) :: gs')
           => refocus_graph5 (replace (g0, x, g0), gs', k)
         | (a, gs)
           => refocus_context5 (gs, k, g))
  | refocus_graph5 (g as ref (A (gl, gr)), gs, k)
    = refocus_graph5 (gl, (g, gr) :: gs, k)

(*  graph -> graph  *)
fun normalize5 g
    = refocus_graph5 (g, [], INIT)
```

This reduction machine is in defunctionalized form and uses frames to manage right branches in the graph while saving the ancestor list for later normalization. The data type `stack_context` together with the function `refocus_context`$_5$ is the first-order implementation of a higher-order function. Refunctionalizing [69] this machine gives an evaluator in continuation-passing style where all continuations, i.e., the refunctionalized stack contexts, are used in a linear fashion as shown in Figure 9.5. The direct-style counterpart of this evaluator is shown in Figure 9.6. This direct-style evaluator handles right branches at return time by recursively normalizing the right subgraphs in the ancestor lists. The ML call stack now implicitly encodes the frames used to keep the state at each right branch.

9.6 Conclusion and perspectives

We have presented the first mechanical inter-derivation of graph rewriting, graph reduction and graph evaluation for cyclic graphs, extending previous work in the context of directed acyclic graphs [222]. Based on the restrictive use of side effects, this derivation adapts Biernacka and Danvy's syntactic correspondence to the setting of graphs as opposed to terms. Furthermore, this work illustrates how recent developments in the construction of a reduction semantics can be applied to graphs as well as to terms [76]. In so doing, we

```
(*  graph * (graph * graph) list * ((graph * graph) list * graph -> 'a)  *)
(*  -> 'a  *)
fun refocus_graph6 (g as ref (C a), gs, k)
    = (case (a, gs)
        of (I, (g0, x) :: gs')
           => refocus_graph6 (rewire' (gs', x), gs', k)
         | (K, (g0, x) :: (g1, y) :: gs')
           => (replace (g1, ref (C I), x);
               refocus_graph6 (rewire' (gs', x), gs', k))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs')
           => refocus_graph6 (replace (g2, ref (A (x, z)),
                                       ref (A (y, z))), gs', k)
         | (Y, (g0, x) :: gs')
           => refocus_graph6 (replace (g0, x, g0), gs', k)
         | (a, gs)
           => k (gs, g))
  | refocus_graph6 (g as ref (A (gl, gr)), gs, k)
    = refocus_graph6 (gl, (g, gr) :: gs, k)

(*  graph -> graph  *)
fun normalize6 g
    = let fun init ([], g) = g
            | init ((g, gr) :: gs, gl)
              = let fun frame (g0, gs0, k) ([], gr')
                        = (g := A (gl, gr'); k (gs0, g0))
                      | frame (g0, gs0, k) ((g, gr) :: gs, gl)
                        = refocus_graph6 (gr, [], frame (g, gs,
                                                         frame (g0, gs0, k)))
                in refocus_graph6 (gr, [], frame (g, gs, init))
                end
      in refocus_graph6 (g, [], init)
      end
```

Figure 9.5: Graph evaluator in continuation-passing style

have connected the graph rewriting systems of Barendregt et al. to the graph reduction machines of Turner.

We have considered a simple setting: Combinatory Logic with just the basic combinators S, K, I and Y. However, in our experience, the derivation techniques scale and have subsequently been successfully applied to both more formal and more involved graph rewriting systems, e.g., equational theories for graph rewriting [73], the call-by-need λ-calculus [77], and the spineless tagless G-machine [167]. Combined, these works shows how different reduction machines can be derived from different graph rewriting systems and it is our experience that the derivation techniques help in understanding their differences. Furthermore, the techniques provide the possibility of incrementally refining either of the semantic artifacts such that the refinements are reflected constructively in the derived semantic counterparts.

Acknowledgments. Thanks are due to the anonymous HOSC reviewers and to Dennis Decker Jensen and the anonymous TFP'09 reviewers for their comments on earlier versions of this article. I am also grateful to Olivier Danvy for his supervision and for his course on functional programming at Aarhus University, where this work originates.

```
(*  graph * (graph * graph) list -> (graph * graph) list * graph  *)
fun refocus_graph7 (g as ref (C a), gs)
    = (case (a, gs)
        of (I, (g0, x) :: gs')
           => refocus_graph7 (rewire' (gs', x), gs')
         | (K, (g0, x) :: (g1, y) :: gs')
           => (replace (g1, ref (C I), x);
               refocus_graph7 (rewire' (gs', x), gs'))
         | (S, (g0, x) :: (g1, y) :: (g2, z) :: gs')
           => refocus_graph7 (replace (g2, ref (A (x, z)),
                                       ref (A (y, z))), gs')
         | (Y, (g0, x) :: gs')
           => refocus_graph7 (replace (g0, x, g0), gs')
         | (a, gs)
           => (gs, g))
  | refocus_graph7 (g as ref (A (gl, gr)), gs)
    = refocus_graph7 (gl, (g, gr) :: gs)

(*  graph -> graph  *)
fun normalize7 g
    = let fun init ([], g) = g
            | init ((g, gr) :: gs, gl)
              = let fun frame (g0, gs0, ([], gr))
                        = (g := A (gl, gr); (gs0, g0))
                      | frame (g0, gs0, ((g, gr) :: gs, gl))
                        = frame (g0, gs0,
                                 frame (g, gs, refocus_graph7 (gr, [])))
                in init (frame (g, gs, refocus_graph7 (gr, [])))
                end
      in init (refocus_graph7 (g, []))
      end
```

Figure 9.6: Graph evaluator in direct style

Chapter 10

Three syntactic theories for combinatory graph reduction

This chapter will appear in [74]: Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. *ACM Transactions on Computational Logic*, 2013.

An earlier version appeared in [73]: Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. In María Alpuente, editor, *Logic Based Program Synthesis and Transformation, 20th International Symposium, LOPSTR 2010, revised selected papers*, number 6564 in Lecture Notes in Computer Science, pages 1–20, Hagenberg, Austria, July 2010. Springer. Invited talk.

Abstract

We present a purely syntactic theory of graph reduction for the canonical combinators S, K, and I, where graph vertices are represented with evaluation contexts and let expressions. We express this first syntactic theory as a storeless reduction semantics of combinatory terms. We then factor out the introduction of let expressions to denote as many graph vertices as possible *upfront* instead of *on demand*. The factored terms can be interpreted as term graphs in the sense of Barendregt et al. We express this second syntactic theory, which we prove equivalent to the first, as a storeless reduction semantics of combinatory term graphs. We then recast let bindings as bindings in a global store, thus shifting, in Strachey's words, from denotable entities to storable entities. The store-based terms can still be interpreted as term graphs. We express this third syntactic theory, which we prove equivalent to the second, as a store-based reduction semantics of combinatory term graphs. We then refocus this store-based reduction semantics into a store-based abstract machine. The architecture of this store-based abstract machine *coincides with that of Turner's original reduction machine.* The three syntactic theories presented here therefore properly account for combinatory graph reduction As We Know It.

These three syntactic theories scale to handling the Y combinator. This article therefore illustrates the scientific consensus of theoreticians and implementors about graph reduction: it is the same combinatory elephant.

10.1 Introduction

In the mid-1990s Ariola, Felleisen, Maraist, Odersky, and Wadler [15] presented a purely syntactic theory for the call-by-need λ-calculus. In retrospect, their key insight was to syntactically represent 'def-use chains' for identifiers with evaluation contexts. For example, here is one of their contraction rules:

$$(\lambda x.E[x])\,v \to (\lambda x.E[v])\,v$$

In the left-hand side, an identifier, x, occurs (i.e., is 'used') in the eye of an evaluation context, E: its denotation is therefore needed.[1] This identifier is declared (i.e., is 'defined') in a λ-abstraction that is applied to a (syntactic) value v. In the right-hand side, v hygienically replaces x in the eye of E. There may be other occurrences of x in E: if another such one is needed later in the reduction sequence, this contraction rule will intervene again—it implements memoization.

In this article, we take a next logical step and present a purely syntactic theory of graph reduction for the canonical combinators S, K and I:

$$\begin{array}{rcl} S\,f\,g\,x & = & f\,x\,(g\,x) \\ K\,x\,y & = & x \\ I\,x & = & x \end{array}$$

Our key technique is to syntactically represent def-use chains for graph vertices using evaluation contexts and let expressions declaring unique references. For example, the traditional specification of K as $K\,t_1\,t_0 = t_1$, for any terms t_0 and t_1, does not account for sharing of subterms before and after contraction. In contrast, our specification does account for sharing, algebraically:

$$\begin{array}{l} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\,t_1\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_0 = x_1\,t_0\ \mathsf{in} \\ \qquad\quad E_0[x_0]]] \end{array} \quad \to \begin{array}{l} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ \boxed{x_3} = t_1\ \mathsf{in} \\ \qquad \mathsf{let}\ x_1 = x_2\ \boxed{x_3}\ \mathsf{in} \\ \qquad E_1[\mathsf{let}\ x_0 = \boxed{x_3}\ \mathsf{in} \\ \qquad\quad E_0[x_0]]] \end{array}$$

where x_3 is fresh

This contraction rule should be read inside-out:

- In the left-hand side, i.e., in the redex, a reference, x_0, occurs in the eye of an evaluation context: its denotation is therefore needed. The definiens of x_0 is the application of a second reference, x_1, to a term t_0: the denotation of x_1 is therefore also needed. The definiens of x_1 is the application of a third reference, x_2, to a term t_1: the denotation of x_2 is therefore also needed. The definiens of x_2 is the K combinator.
- In the right-hand side, i.e., in the contractum, a fresh (and thus unique) reference, x_3, is introduced to denote t_1: it replaces the application of x_1 to t_0. Reducing the K combinator is thus achieved (1) by creating a fresh reference to t_1 to share any subsequent reduction in t_1,[2] and (2) by replacing the reference to the application $x_1\,t_0$ by x_3.

[1] The notation $E[t]$ stands for a term that decomposes into a reduction context, E, and a subterm, t.

[2] References ensure sharing of subterms: they are uniquely defined, but they can have many uses. References can be freely duplicated, but what they refer to, i.e., their denotation, is not duplicated and is thus shared.

There may be other occurrences of x_0 in E_0: if another such one is needed later in the reduction sequence, this contraction rule for K will not intervene again—its result has been memoized.

In Section 10.2, we fully specify our syntactic theory of combinatory graph reduction as a reduction semantics, and we then apply the first author's programme [61, 62] to derive the first storeless abstract machine for combinatory graph reduction, in a way similar to what we recently did for the call-by-need λ-calculus [77].

Our syntactic theory introduces let expressions for applications on demand. In Section 10.3, we preprocess source terms by introducing let expressions upfront for all source applications, and we present the corresponding reduction semantics and storeless abstract machine. We show that the preprocessed terms can be interpreted as Barendregt et al.'s term graphs [24].

In Section 10.4, we map the explicit let bindings for graph vertices to implicit bindings in a global store. Still, terms can be interpreted as term graphs. Again, we present the corresponding store-based reduction semantics and store-based abstract machine. This store-based abstract machine essentially coincides with Turner's original graph-reduction machine [209]. This coincidence provides an independent, objective bridge between the modern theory of combinatory graph reduction with term graphs and its classical computational practice with reduction machines.

Prerequisites and notations We expect an elementary awareness of the S, K and I combinators and how combinatory terms can be reduced to head normal form, either in principle (as a formal property in Combinatory Logic [22]) or in practice (as a stack-based graph-reduction machine [159, 209]). We also assume a basic familiarity with the format of reduction semantics and abstract machines as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [61]; and with the concept of term graphs, as pedagogically presented in Blom's PhD dissertation [33]. In particular, we use the terms 'reduction context' and 'evaluation context' interchangeably.

For a notion of reduction $\mathscr{R}$ defined with a set of contraction rules, we define the contraction of a redex r into a contractum t as $(r, t) \in \mathscr{R}$. We use $\rightarrow_{\mathscr{R}}$ for the compatible closure of $\mathscr{R}$, and $\twoheadrightarrow_{\mathscr{R}}$ for the transitive-reflexive closure of $\rightarrow_{\mathscr{R}}$. A term t is in $\mathscr{R}$-normal form ($\mathscr{R}$-nf) if t does not contain a redex of $\mathscr{R}$.

Pictorial overview

	terms	graphs	store
reduction semantics	Section 10.2.1	Section 10.3.1	Section 10.4.1
abstract machine	Section 10.2.2	Section 10.3.2	Section 10.4.2
compressed abstract machine	Section 10.2.3	Section 10.3.3	Section 10.4.3

10.2 Three inter-derivable semantic artifacts for storeless combinatory graph reduction

Our starting point is the following grammar of combinatory terms:

$$t ::= I \mid K \mid S \mid t\,t$$

So a combinatory term is a combinator or a combination, i.e., the application of a term to another term.

We embed this grammar into a grammar of terms where sub-terms can be referred to through let expressions and where a program p is a term t denoted by a reference whose denotation is needed:

$$\begin{aligned} t &::= I \mid K \mid S \mid t\,t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid x \\ p &::= \mathsf{let}\ x = t\ \mathsf{in}\ x \end{aligned}$$

In this grammar, a term is a combinatory term (i.e., a combinator or a combination), the declaration of a reference to a term in another term, or the occurrence of a declared reference. Initially, a program contains only one let expression: the outer one, whose definiens is a let-less combinatory term.

In our experience, however, there is a better fit for the contraction rules, namely the following sub-grammar where a denotable term is an original combinatory term or a term that generalizes the original program into declarations nested around a declared reference:

$$\begin{aligned} p &::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\ d &::= I \mid K \mid S \mid d\,d \mid t \\ t &::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \end{aligned}$$

This grammar of terms excludes terms with let expressions whose body is a combinator or a combination. Again, initially, a program contains only one let expression.

At this point, it would be tempting to define reduction contexts to reflect how references are needed in the reduction process:

$$\mathsf{Reduction\ Context} \ni E ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\,d\ \mathsf{in}\ E[x]$$

The constructor "let $x = d$ in E" accounts for the recursive search for the innermost reference in a term. The constructor "let $x = E\,d$ in $E[x]$" accounts for the need of intermediate references.

In our experience, however, there is a better grammatical fit for contexts, namely one which separates the search for the innermost reference in a term and the subsequent construction that links needed references to their declaration, i.e., usage to definition. The former gives rise to delimited reduction contexts and the latter to def-use chains:

$$\begin{aligned} \mathsf{Reduction\ Context} \ni E &::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\ \mathsf{Def\text{-}use\ Chain} \ni C &::= [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]] \end{aligned}$$

We are not alone to appreciate this grammatical fit: in their abstract machine for the call-by-need λ-calculus [44], Chang et al. also segment their control stack according to the current λ-binders to connect uses and definitions.

10.2.1 A reduction semantics

A reduction semantics is a small-step operational semantics [171] with an explicit representation of the reduction context. Its components are an abstract syntax, a grammar of reduction contexts, a collection of axioms (contraction rules) to map a redex into a contractum, a decomposition function mapping a value term into itself and a non-value term into a redex and its reduction context, and a recomposition function mapping a contractum and a reduction context into a reduct. With these components, one can define a one-step reduction function enumerating the reduction sequence. This reduction function

(1) decomposes a non-value term into a redex and its reduction context, (2) contracts this redex into a contractum,[3] and (3) recomposes this contractum and this reduction context into the next reduct in the reduction sequence. Let us specify each of these components: abstract syntax and reduction contexts, axioms, recomposition, and decomposition.

Here is the full definition of the syntax:

$$\begin{aligned}
\text{Program} \ni p &::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\
\text{Denotable Term} \ni d &::= I \mid K \mid S \mid d\,d \mid t \\
\text{Term} \ni t &::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \\
\text{Reduction Context} \ni E &::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\
\text{Def-use Chain} \ni C &::= [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]
\end{aligned}$$

Initially, a program contains only one let expression: the outer one. Reduction contexts reflect the recursive search for the innermost reference in a term. While returning from this search, def-use chains are constructed to connect each reference whose denotation is needed with its declaration site. We abbreviate $\mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]$ as $(x, d, E)\cdot C$ and we write $\Pi_0^{i=n}(x_i, d_i, E_i)\cdot C$ as short hand for $(x_n, d_n, E_n)\cdot\ldots\cdot(x_0, d_0, E_0)\cdot C$, and $|C|$ for the length of C, so that $|\Pi_0^{i=n}(x_i, d_i, E_i)\cdot[\,]| = n+1$.

Axioms (i.e., contraction rules) Figure 10.1 displays the axioms. Each of (I), (K) and (S) is much as (K) was described in Section 10.1, with the addition of the inner def-use chain: it carries out a particular rearrangement while preserving sharing through common references.[4]

In the left-hand side of (*comb*), a reference, x_0, occurs in the eye of the current def-use chain, C: its denotation is therefore needed. Its definiens is a combination of two denotable terms, d_0 and d_1. In the right-hand side, a fresh reference, x_1, is introduced to denote d_0. This fresh reference extends the current def-use chain for d_0, thereby ensuring that any subsequent reduction in d_0 is shared.[5] This specific choice of d_0 ensures that any redex found in a subsequent search will be on the left of this combination, thereby enforcing left-most reduction.

The axiom (*assoc*) is used to flatten let expressions,[6] and the axiom (*ref*) to resolve indirect references.

The corresponding notion of reduction is $\mathscr{T}$:

$$\mathscr{T} = (I) \cup (K) \cup (S) \cup (comb) \cup (assoc) \cup (ref)$$

Reduction strategy The reduction strategy is left-most outermost: the def-use chains force us to only consider the outermost combination, and (*comb*) ensures that this outermost combination is the left-most one.

[3] The contraction rules may be *context-sensitive* in that they map a redex and its context into a contractum and a new context. In that case, the contractum is recomposed into the new context.

[4] On the right-hand side of (I), (K) and (S), we have kept $E_0[C[x_0]]$ in order to highlight each particular rearrangement. It would be simple to "optimize" these right-hand sides by taking advantage of [a subsequent use of] (*ref*) and (*comb*), so that, e.g., the right-hand side of (K) contains $E_0[C[x_3]]$ instead.

[5] If d_0 is already a reference, it is already part of a def-use chain and no contraction need take place: $\mathsf{let}\ x_0 = x\,d_1\ \mathsf{in}\ E_0[C[x_0]]$ is not a redex.

[6] Let expressions are themselves denotable terms and must be reassociated in search of a def-use chain. When reassociating, there is no need for the condition "x_1 does not occur free in E_0" since each reference is unique.

$$
\begin{array}{lll}
(I) & \mathsf{let}\ x_1 = I\ \mathsf{in} & \to\ \mathsf{let}\ x_1 = I\ \mathsf{in} \\
& \quad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \quad\ E_1[\mathsf{let}\ x_0 = d_0\ \mathsf{in} \\
& \qquad E_0[C[x_0]]] & \qquad E_0[C[x_0]]] \\
(K) & \mathsf{let}\ x_2 = K\ \mathsf{in} & \to\ \mathsf{let}\ x_2 = K\ \mathsf{in} \\
& \quad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & \quad\ E_2[\mathsf{let}\ x_3 = d_1\ \mathsf{in} \\
& \qquad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \qquad\ \mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in} \\
& \qquad\ E_0[C[x_0]]]] & \qquad\ E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in} \\
& & \qquad\ E_0[C[x_0]]]] \\
& & \text{where } x_3 \text{ is fresh} \\
(S) & \mathsf{let}\ x_3 = S\ \mathsf{in} & \to\ \mathsf{let}\ x_3 = S\ \mathsf{in} \\
& \quad E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} & \quad\ E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\
& \qquad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & \qquad\ \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\
& \qquad\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \qquad\ E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\
& \qquad\ E_0[C[x_0]]]]] & \qquad\ \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\
& & \qquad\ E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\
& & \qquad\ \mathsf{let}\ x_0 = (x_4\, x_6)(x_5\, x_6)\ \mathsf{in} \\
& & \qquad\ E_0[C[x_0]]]]] \\
& & \text{where } x_4,\ x_5 \text{ and } x_6 \text{ are fresh} \\
(comb) & \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in} & \to\ \mathsf{let}\ x_1 = d_0\ \mathsf{in} \\
& \quad E_0[C[x_0]] & \quad\ \mathsf{let}\ x_0 = x_1\, d_1\ \mathsf{in} \\
& & \quad\ E_0[C[x_0]] \\
& & \text{where } d_0 \text{ is not a reference} \\
& & \quad \text{and } x_1 \text{ is fresh} \\
(assoc) & \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in} & \to\ \mathsf{let}\ x_1 = d_1\ \mathsf{in} \\
& \quad E_0[C[x_0]] & \quad\ \mathsf{let}\ x_0 = t_0\ \mathsf{in} \\
& & \quad\ E_0[C[x_0]] \\
(ref) & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \to\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_1]]
\end{array}
$$

Figure 10.1: Reduction semantics for combinatory graph reduction: axioms

Recompositions Figure 10.2 displays the recompositions of a reduction context around a term, and of a def-use chain around a reference:

Definition 85 (inside-out recomposition of contexts around terms)**.** A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E, t\rangle_{io} \Uparrow_{\mathrm{rec}} t'$ holds. (See Figure 10.2 and also Footnote 1 for the notation $E[t]$.)

Definition 86 (outside-in recomposition of contexts around terms)**.** A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E, t\rangle_{oi} \Uparrow_{\mathrm{rec}} t'$ holds. (See Figure 10.2.)

Outside-in recomposition of contexts is used as an auxiliary judgment in the recomposition of def-use chains:

Inside-out recomposition of a reduction context around a term:

$$\frac{}{\langle [\,],\, t\rangle_{io} \Uparrow_{\text{rec}} t} \qquad \frac{\langle E,\, \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{io} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\, t\rangle_{io} \Uparrow_{\text{rec}} t'}$$

Outside-in recomposition of a reduction context around a term:

$$\frac{}{\langle [\,],\, t\rangle_{oi} \Uparrow_{\text{rec}} t} \qquad \frac{\langle E,\, t\rangle_{oi} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\, t\rangle_{oi} \Uparrow_{\text{rec}} \mathsf{let}\ x = d\ \mathsf{in}\ t'}$$

Recomposition of a def-use chain around a reference:

$$\frac{}{\langle [\,],\, x\rangle_{chain} \Uparrow_{\text{rec}} x} \qquad \frac{\langle C,\, x'\rangle_{chain} \Uparrow_{\text{rec}} t \qquad \langle E,\, t\rangle_{oi} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x' = [\,]\, d\ \mathsf{in}\ E[C[x']],\, x\rangle_{chain} \Uparrow_{\text{rec}} \mathsf{let}\ x' = x\ d\ \mathsf{in}\ t'}$$

Figure 10.2: Reduction semantics for combinatory graph reduction: recompositions

Definition 87 (recomposition of def-use chains around references)**.** A def-use chain C is recomposed around a reference x into a term $t = C[x]$ whenever $\langle C,\, x\rangle_{chain} \Uparrow_{\text{rec}} t$ holds.[7] (See Figure 10.2.)

Decomposition Decomposition implements the reduction strategy by searching for a redex and its reduction context in a term. Figure 10.3 displays this search as a state-transition system with three states:

term-**state** $\langle t,\, E\rangle_{term}$

cont-**state** $\langle E,\, t\rangle_{cont}$ and $\langle E,\, (x,\, E,\, C)\rangle_{cont}$

den-**state** $\langle x,\, d,\, E,\, C,\, E\rangle_{den}$

term-**transitions** Given a term, we recursively dive into the bodies of its nested let expressions until its innermost reference x, which is therefore needed.

cont-**transitions over** t Given a context, we dispatch on its top component, if there is one.

cont-**transitions over** $(x,\, E,\, C)$ Having found a reference x that is needed, we backtrack in search of its declaration, incrementally constructing a def-use chain for it.[8] If we do not find any redex, the term is in $\mathscr{T}$-normal form.

den-**transitions** Having found the declaration of the reference that was needed, we check whether we have also found a redex and thus a decomposition. Otherwise, a combinator is not fully applied or a new reference is needed. We then resume a *cont*-transition, either on our way to a $\mathscr{T}$-normal form or extending the current def-use chain for this new reference.

[7] As already pointed out in Footnote 1, the notation $C[x]$ stands for a term that decomposes into a def-use chain, C, and a reference, x.

[8] There is no transition for $\langle [\,],\, (x_0,\, E_0,\, C)\rangle_{cont}$ because all references are declared.

$$\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} \downarrow_{\mathrm{dec}} \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term}$$
$$\langle x,\ E\rangle_{term} \downarrow_{\mathrm{dec}} \langle E,\ (x,\ [\ ],\ [\ ])\rangle_{cont}$$

$$\langle [\ ],\ t\rangle_{cont} \downarrow_{\mathrm{dec}} \langle t\rangle_{nf}$$
$$\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont}$$

$$\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \downarrow_{\mathrm{dec}} \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den}$$
$$\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \downarrow_{\mathrm{dec}} \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \quad \text{where } x \neq x_0$$

$$\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]\rangle_{dec}$$
$$\text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]]\rangle_{dec}$$
$$\text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]]]\rangle_{dec}$$
$$\text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont}$$
$$\text{where } d_0 = I \text{ and } |C| < 1 \quad \text{or } d_0 = K \text{ and } |C| < 2 \quad \text{or } d_0 = S \text{ and } |C| < 3$$
$$\text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_0,\ x_1\, d_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ (x_1,\ [\ ],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont}$$
$$\langle x_0,\ d_0\, d_1,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec}$$
$$\text{where } d_0 \text{ is not a reference}$$
$$\text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec}$$
$$\text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$
$$\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} \langle E,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec}$$
$$\text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \text{ and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]$$

Figure 10.3: Reduction semantics for combinatory graph reduction: decomposition

Definition 88 (decomposition)**.** The decomposition relation, $\downarrow^*_{\text{dec}}$, is the transitive closure of $\downarrow_{\text{dec}}$. A term $t = E[r]$ is decomposed into a redex r and a context E whenever $\langle t, [\,]\rangle_{term} \downarrow^*_{\text{dec}} \langle r, E\rangle_{dec}$ holds. (See Figure 10.3.)

The transition system implementing decomposition can be seen as a big-step abstract machine [67]. As repeatedly pointed out in the first author's lecture notes at AFP 2008 [61], such a big-step abstract machine is often in defunctionalized form—as is the case here. In the present case, it can be refunctionalized into a function over source terms which is compositional. Ergo, it is expressible as a catamorphism over source terms. Further, the parameters of this catamorphism are total functions. Therefore, the decomposition function is total. It yields either the given term if this term is in $\mathscr{T}$-normal form, or its left-most outermost redex and the corresponding reduction context. Formally:

Property 89 (vacuous decomposition of normal forms)**.** For any $t \in \mathscr{T}$-nf,

$$\langle t, [\,]\rangle_{term} \downarrow^*_{\text{dec}} \langle t\rangle_{nf}$$

Property 90 (unique decomposition of non-normal forms)**.** For any $t \notin \mathscr{T}$-nf,

$$\langle t, [\,]\rangle_{term} \downarrow^*_{\text{dec}} \langle E, r\rangle_{dec}$$

where r is the left-most outermost redex of t and E is its corresponding reduction context.

The referee pointed out that termination can also be argued with an ordering on abstract states:

- In any transition from $\langle t, E\rangle_{term}$ to $\langle t', E'\rangle_{term}$, t' is a proper part of t.
- In any transition from $\langle t, E\rangle_{term}$ to $\langle E, t\rangle_{cont}$, E is unchanged.
- In any transition from $\langle E, t\rangle_{cont}$ to $\langle E', t'\rangle_{cont}$, E' is a proper part of E.
- In any transition from $\langle E, (x_0, E_0, C)\rangle_{cont}$ to $\langle x_0, \_, E_0, C, E'\rangle_{den}$, E' is a proper part of E.
- In any transition from $\langle E, (x_0, E_0, C)\rangle_{cont}$ to $\langle E', (x_0, E_0, C)\rangle_{cont}$, E' is a proper part of E.
- In any transition from $\langle x_0, d, E_0, C, E\rangle_{den}$, to $\langle E, t\rangle_{cont}$, E is unchanged.
- In any transition from $\langle x_0, d, E_0, C, E\rangle_{den}$, to $\langle E, (x_1, E_1, C')\rangle_{cont}$, E is unchanged.

In any two consecutive transitions, the term or the context decreases and therefore decomposition terminates.

One-step reduction Performing one contraction in a term that is not in $\mathscr{T}$-normal form proceeds as follows: (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 91 (standard one-step reduction)**.** For any t,

$$t \mapsto_{\mathscr{T}} t'' \quad \text{iff} \quad \begin{cases} \langle t, [\,] \rangle_{term} \downarrow^{*}_{\text{dec}} \langle E, r \rangle_{dec} \\ (r, t') \in \mathscr{T} \\ \langle E, t' \rangle_{io} \Uparrow_{\text{rec}} t'' \end{cases}$$

The standard one-step reduction is not the compatible closure of $\mathscr{T}$. Indeed, the standard one-step reduction is closed over evaluation contexts, whereas the compatible closure of $\mathscr{T}$ is closed over arbitrary contexts (i.e., terms with a hole).

Reduction-based evaluation The standard reduction-based evaluation is defined as the iteration of the standard one-step reduction. It thus enumerates the standard reduction sequence of any given program:

Definition 92 (standard reduction-based evaluation)**.** Standard reduction-based evaluation, $\mapsto^{*}_{\mathscr{T}}$, is the transitive-reflexive closure of standard one-step reduction, $\mapsto_{\mathscr{T}}$.

Most of the time, decomposition and recomposition(s) are kept implicit in published reduction semantics. We however observe that what was kept implicit is then progressively revealed as, e.g., one constructs an abstract machine to implement evaluation [99]. We believe that it is better to completely spell out reduction semantics upfront, because one is then in position to systematically calculate the corresponding abstract machines [28, 61], as illustrated in the next section for syntactic graph reduction. In our experience, this systematic calculation is not only generally applicable but it yields simpler abstract machines than the ones constructed by hand.

10.2.2 A storeless abstract machine

Reduction-based evaluation, as defined in Section 10.2.1, is inefficient because of its repeated decompositions and recompositions that construct each successive term in a reduction sequence. Refocusing [71] deforests these intermediate terms, and is defined very simply as continuing decomposition with the contractum and its reduction context. The reduction semantics of Section 10.2.1 satisfies the formal requirements for refocusing [71] and so its reduction-based evaluation can be simplified into a reduction-free evaluation that does not construct each successive term in a reduction sequence. Reflecting the structure of decomposition of Figure 10.3, the result is an abstract machine displayed in Figure 10.4:

term-**transitions** The *term*-transitions are the same as for decomposition.

cont-**transitions** The *cont*-transitions are the same as for decomposition.

den-**transitions** The *den*-transitions are the same as for decomposition.

dec-**transitions** Having found a redex we contract it and decompose the contractum in the current context.

After an initial decomposition of the input term in the empty context, reduction-free evaluation is thus defined as the iteration of contraction and decomposition:

$$\rightarrow_{\text{step}} \quad \supset \quad \downarrow_{\text{dec}}$$

$$\langle E, \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]],\, E\rangle_{term}$$

$$\langle E, \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_3 = d_1\ \mathsf{in}\ \mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in}\ E_0[C[x_0]]],\, E\rangle_{term}$$
where x_3 is fresh

$$\langle E, \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in}\ E_0[C[x_0]]]]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in}\ \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in}\ E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in}\ \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in}\ E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in}\ \mathsf{let}\ x_0 = (x_4\, x_6)\,(x_5\, x_6)\ \mathsf{in}\ E_0[C[x_0]]]]],\, E\rangle_{term}$$
where x_4, x_5 and x_6 are fresh

$$\langle E, \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_1 = d_0\ \mathsf{in}\ \mathsf{let}\ x_0 = x_1\, d_1\ \mathsf{in}\ E_0[C[x_0]],\, E\rangle_{term}$$
where d_0 is not a reference and x_1 is fresh

$$\langle E, \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ \mathsf{let}\ x_0 = t_0\ \mathsf{in}\ E_0[C[x_0]],\, E\rangle_{term}$$

$$\langle E, \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \rightarrow_{\text{step}} \langle \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_1]],\, E\rangle_{term}$$

Figure 10.4: Storeless abstract machine for combinatory graph reduction

Definition 93 (standard reduction-free evaluation)**.** Standard reduction-free evaluation, $\rightarrow^*_{\text{step}}$, is the transitive closure of $\rightarrow_{\text{step}}$. (See Figure 10.4.)

Proposition 94 (full correctness of the storeless abstract machine)**.** *For any program p,*

$$p \mapsto^*_{\mathscr{T}} t \ \wedge\ t \in \mathscr{T}\text{-}nf \iff \langle p, [\,]\rangle_{term} \rightarrow^*_{\text{step}} \langle t\rangle_{nf}$$

Proof. Correctness of refocusing. □

10.2.3 A storeless abstract machine after transition compression

In the abstract machine of Section 10.2.2, some of the transitions yield a configuration for which there unconditionally exists another transition: all transitions to a *term-*

configuration with a known term, all transitions to a *cont*-configuration with a known context, and all transitions to a *dec*-configuration with a known redex (i.e., all transitions to *dec*). These so-called "corridor transitions" [36] from one configuration to another can be hereditarily compressed so that the first configuration yields the last one in one transition.

Other transition compressions are determined by the structure of terms, evaluation contexts, and def-use chains. They collapse several consecutive steps into one. We state these properties using the following two recompositions of evaluation contexts:

Definition 95 (inside-out recomposition of contexts around contexts)**.** A context that was constructed inside out (resp. outside in) is composed with a context that was constructed outside in (resp. inside out) as follows:

$$[\,] \circ_{io} E = E$$
$$(\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{io} E = E' \circ_{io} \mathsf{let}\ x = d\ \mathsf{in}\ E$$

The resulting context is constructed outside in (resp. inside out).

Definition 96 (outside-in recomposition of contexts around contexts)**.** Two contexts that were constructed outside in (resp. inside out) are composed into an outside-in (resp. inside-out) context as follows:

$$[\,] \circ_{oi} E = E$$
$$(\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{oi} E = \mathsf{let}\ x = d\ \mathsf{in}\ (E' \circ_{oi} E)$$

This composition function is associative.

Property 97 (restoring outside-in evaluation contexts)**.** For any t, E and E_1 such that $\langle E_1, t\rangle_{oi} \Uparrow_{\mathrm{rec}} E_1[t]$,

$$\langle E_1[t], E\rangle_{term} \to^*_{\mathrm{step}} \langle t, E_1 \circ_{io} E\rangle_{term}$$

Property 98 (restoring def-use chains)**.** For any x, C and E such that $\langle C, x\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x]$,

$$\langle C[x], E\rangle_{term} \to^*_{\mathrm{step}} \langle E, (x, [\,], C)\rangle_{cont}$$

Property 99 (continuing search for references)**.** For any x, C, E, E_1 and E_2 where x is not bound in E_1,

$$\langle E_1 \circ_{oi} E, (x, E_2, C)\rangle_{cont} \to^*_{\mathrm{step}} \langle E, (x, E_1 \circ_{io} E_2, C)\rangle_{cont}$$

Corollary 100 (restoring evaluation contexts and def-use chains)**.** *For any x, C, E and* E_1 *such that* $\langle C, x\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x]$ *and* $\langle E_1, C[x]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_1[C[x]]$,

$$\langle E_1[C[x]], E\rangle_{term} \to^*_{\mathrm{step}} \langle E, (x, E_1, C)\rangle_{cont}$$

The resulting machine is displayed in Figure 10.5.

Definition 101 (standard reduction-free evaluation)**.** Standard reduction-free evaluation, $\to^*_{\mathrm{step}}$, is the transitive closure of $\to_{\mathrm{step}}$. (See Figure 10.5.)

Proposition 102 (full correctness of the storeless abstract machine after transition compression)**.** *For any program p,*

$$p \mapsto^*_{\mathscr{T}} t \ \wedge\ t \in \mathscr{T}\text{-}nf \iff \langle p, [\,]\rangle_{term} \to^*_{\mathrm{step}} \langle t\rangle_{nf}$$

Proof. Property 97, Property 98, Property 99 and Corollary 100. □

$$
\begin{array}{rl}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} \rightarrow_{\mathrm{step}} & \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} \rightarrow_{\mathrm{step}} & \langle E,\ (x,\ [\ ],\ [\ ])\rangle_{cont} \\
 & \\
\langle [\ ],\ t\rangle_{cont} \rightarrow_{\mathrm{step}} & \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} \rightarrow_{\mathrm{step}} & \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
 & \\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \rightarrow_{\mathrm{step}} & \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \rightarrow_{\mathrm{step}} & \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
 & \quad \text{where } x \neq x_0 \\
 & \\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle x_0,\ d_0,\ E_0,\ C,\ E_1 \circ_{io} \mathsf{let}\ x_1 = I\ \mathsf{in}\ E\rangle_{den} \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle x_3,\ d_1,\ E',\ C,\ E_2 \circ_{io} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E\rangle_{den} \\
 & \quad \text{where } E' = (\mathsf{let}\ x_1 = x_2\ x_3\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = x_3\ \mathsf{in}\ E_0) \\
 & \quad \text{and } x_3 \text{ is fresh} \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle x_4,\ d_2,\ E',\ C',\ E_3 \circ_{io} \mathsf{let}\ x_3 = S\ \mathsf{in}\ E\rangle_{den} \\
 & \quad \text{where } E' = \begin{pmatrix} \mathsf{let}\ x_2 = x_3\ x_4 \\ \mathsf{in}\ E_2 \end{pmatrix} \circ_{oi} \begin{pmatrix} \mathsf{let}\ x_5 = d_1 \\ \mathsf{in let}\ x_1 = x_2\ x_5 \\ \quad \mathsf{in}\ E_1 \end{pmatrix} \circ_{oi} \begin{pmatrix} \mathsf{let}\ x_6 = d_0 \\ \mathsf{in}\ [\ ] \end{pmatrix} \\
 & \quad \text{and } C' = (x_7,\ x_6,\ [\ ])\cdot(x_0,\ x_5\ x_6,\ E_0)\cdot C \\
 & \quad \text{and } x_4, x_5, x_6 \text{ and } x_7 \text{ are fresh} \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
 & \quad \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \\
 & \quad \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
 & \quad \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\ d_0,\ E_0,\ C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle E,\ (x_1,\ [\ ],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ d_0\ d_1,\ E_0,\ C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle x_1,\ d_0,\ [\ ],\ (x_0,\ d_1,\ E_0)\cdot C,\ E\rangle_{den} \\
 & \quad \text{where } d_0 \text{ is not a reference and } x_1 \text{ is fresh} \\
\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle x_0,\ t_0,\ E_0,\ C,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ E\rangle_{den} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} \rightarrow_{\mathrm{step}} & \langle E,\ (x_1,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\ C)\rangle_{cont}
\end{array}
$$

Figure 10.5: Storeless abstract machine for combinatory graph reduction after transition compression

10.2.4 Summary and conclusion

Starting from a completely spelled-out reduction semantics for combinatory terms with sharing, we have mechanically derived a storeless abstract machine. These two semantic artifacts share the same syntactic representations and precisely follow the same reduction strategy.

10.3 Preprocessing combinatory terms into term graphs

In Section 10.2, references are declared on demand in the reduction sequence. In this section, we factor out all possible such declarations for combinations into a preprocessing phase.

We start by revising the $(comb)$ and (S) axioms:

$$\begin{array}{lll} (comb') & \mathsf{let}\ x = d_0\, d_1\ \mathsf{in}\ E[C[x]] & \rightarrow\ \mathsf{let}\ x_0 = d_0\ \mathsf{in} \\ & & \quad \mathsf{let}\ x_1 = d_1\ \mathsf{in} \\ & & \quad \mathsf{let}\ x = x_0\, x_1\ \mathsf{in}\ E[C[x]] \\ & & \quad \text{where } d_0 \text{ is not a reference} \\ & & \quad \text{and } x_0 \text{ and } x_1 \text{ are fresh} \end{array}$$

In contrast to the $(comb)$ axiom, the revised axiom $(comb')$ declares references for both sides of a combination. Unlike in Section 10.2, there can thus be references to denotable terms whose denotation is not needed. In the same spirit, we revise the (S) axiom so that it declares references to both sides of any combination in the contractum:

$$\begin{array}{lll} (S') & \mathsf{let}\ x_3 = S\ \mathsf{in} & \rightarrow\ \mathsf{let}\ x_3 = S\ \mathsf{in} \\ & \quad E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} & \quad E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\ & \qquad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & \qquad \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\ & \qquad\quad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \qquad E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\ & \qquad\qquad E_0[C[x_0]]]]] & \qquad\quad \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\ & & \qquad\quad E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\ & & \qquad\qquad \mathsf{let}\ x_7 = x_4\, x_6\ \mathsf{in} \\ & & \qquad\qquad \mathsf{let}\ x_8 = x_5\, x_6\ \mathsf{in} \\ & & \qquad\qquad \mathsf{let}\ x_0 = x_7\, x_8\ \mathsf{in} \\ & & \qquad\qquad E_0[C[x_0]]]]] \\ & & \text{where } x_4, x_5, x_6, x_7 \text{ and } x_8 \text{ are fresh} \end{array}$$

The corresponding notion of reduction is $\mathscr{T}'$:

$$\mathscr{T}' = (I) \cup (K) \cup (S') \cup (comb') \cup (assoc) \cup (ref)$$

We split $\mathscr{T}'$ into two: a compile-time notion of reduction $\mathscr{C}$ and a run-time notion of reduction $\mathscr{R}$:

$$\begin{aligned} \mathscr{C} &= (assoc) \cup (comb') \\ \mathscr{R} &= (I) \cup (K) \cup (S') \cup (ref) \end{aligned}$$

$\mathscr{T}'$ (and thus each of $\mathscr{C}$ and $\mathscr{R}$) contains only left-linear axioms and no critical pairs: it is orthogonal and thus confluent [125]. Furthermore, $\mathscr{C}$ is strongly normalizing.

The $\mathscr{C}$-normal forms are contained within the following sub-grammar of terms:

$$\begin{aligned} \mathsf{Denotable\ Term} \ni d &::= I \mid K \mid S \mid x\,x \mid x \\ \mathsf{Term} \ni t &::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \end{aligned}$$

In this grammar, only combinations of references are admitted and furthermore let expressions are completely flattened, in a way reminiscent of monadic normal forms [111, 112].

Proposition 103 (preprocessing)**.** *If* $p \twoheadrightarrow_{\mathscr{T}'} t \ \wedge\ t \in \mathscr{T}'\text{-}nf$ *then* $\exists t' \in \mathscr{C}\text{-}nf\,.\ p \twoheadrightarrow_{\mathscr{C}} t' \twoheadrightarrow_{\mathscr{R}} t$.

Proof. Strong normalization of $\mathscr{C}$ ensures the existence of t'. Confluence of $\mathscr{T}'$ gives $t' \twoheadrightarrow_{\mathscr{T}'} t$. $\mathscr{R}$ is closed over $\mathscr{C}$-nf. Thus, only $\mathscr{R}$ is needed in the reduction $t' \twoheadrightarrow_{\mathscr{R}} t$. □

We observe that a preprocessed term is a syntactic representation of a graph where every denotable term has been declared with a reference. Indeed it is straightforward to interpret preprocessed terms as term graphs:

Definition 104 (term graphs [24, Definition 4.2.6])**.** A *term graph* is a tuple $(N, lab, succ, r)$ over a set of function symbols F where

- N is a set of unique node identifiers;
- $lab : N \to F$ is a labeling function mapping nodes to function symbols;
- $succ : N \to N^n$ is a successor function mapping nodes to an n-tuple of successor nodes for some natural number n; and
- $r \in N$ is the root of the term graph.

Two term graphs are equivalent, $g_1 \equiv g_2$, if they are componentwise equivalent. A term graph is a sub-graph of another term graph, $g_1 \sqsubseteq g_2$, if $N_1 \subseteq N_2$ and

$$\forall n \in N_1\ .\ lab_1(n) = lab_2(n) \ \wedge\ succ_1(n) = succ_2(n)$$

Definition 105 (interpreting preprocessed combinatory terms as term graphs)**.** For any preprocessed term t, its term graph $\gamma(t)$ over the function symbols I, K, S, and A is defined as follows:

- N_t is the set of declared references in t.
- lab_t is defined on the definiens of a reference (i.e., a denotable term): for a combinator, it yields the respective function symbol I, K or S; for a combination, it yields the application symbol A; and for a reference, it yields the result of applying lab_t to this reference, which in effect acts as an alias for a node.[9]
- $succ_t$ is defined on the definiens of a reference: for a combination, it yields the corresponding pair of references, and for everything else, the empty tuple.
- r_t is the innermost reference of the term.

[9] This application is well behaved since terms are acyclic.

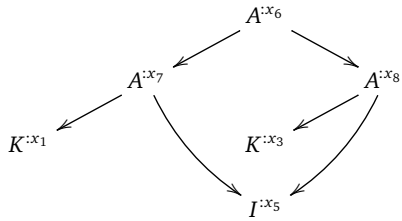
For example, the first reduct of the term $S\,K\,K\,I$ after preprocessing:

$$\begin{array}{l}\mathsf{let}\ x_0 = S\ \mathsf{in}\\ \mathsf{let}\ x_1 = K\ \mathsf{in}\\ \mathsf{let}\ x_2 = x_0\,x_1\ \mathsf{in}\\ \mathsf{let}\ x_3 = K\ \mathsf{in}\\ \mathsf{let}\ x_4 = x_1\,x_3\ \mathsf{in}\\ \mathsf{let}\ x_5 = I\ \mathsf{in}\\ \mathsf{let}\ x_7 = x_1\,x_5\ \mathsf{in}\\ \mathsf{let}\ x_8 = x_3\,x_5\ \mathsf{in}\\ \mathsf{let}\ x_6 = x_7\,x_8\ \mathsf{in}\\ x_6\end{array}$$

translates to the term graph: $N = \{x_0, \cdots, x_8\}$,

$$lab(x) = \begin{cases} S & \text{if } x = x_0\\ K & \text{if } x = x_1\\ K & \text{if } x = x_3\\ I & \text{if } x = x_5\\ A & \text{otherwise}\end{cases} \qquad succ(x) = \begin{cases} \{x_0, x_1\} & \text{if } x = x_2\\ \{x_1, x_3\} & \text{if } x = x_4\\ \{x_7, x_8\} & \text{if } x = x_6\\ \{x_1, x_5\} & \text{if } x = x_7\\ \{x_3, x_5\} & \text{if } x = x_8\\ \emptyset & \text{otherwise}\end{cases}$$

with root $r = x_6$. After garbage collecting unreachable nodes (i.e., x_0, x_2 and x_4) this term graph is depicted as:



Using the interpretation of Definition 105, we can translate the contraction rules over combinatory terms to graph-rewriting rules [24, Section 4.4.4]. The translation of (I), (K) and (S') gives us rewriting rules with the side condition that the redex is *rooted*, meaning that there is a path from the root of the graph to the redex, which is the case here and is manifested by its def-use chain. Terms in our language are therefore a restricted form of term graphs: directed acyclic graphs with an ordering hierarchy imposed on *succ* by the scoping of nested let expressions. (In his PhD thesis [33], Blom refers to this property of term graphs as 'horizontal sharing.')

10.3.1 A reduction semantics

Here is the full definition of the syntax after preprocessing terms into $\mathscr{C}$-nf:

$$\begin{array}{rl}\text{Program} \ni p & ::= \mathsf{let}\ x = d\ \mathsf{in}\ t\\ \text{Denotable Term} \ni d & ::= I \mid K \mid S \mid x\,x \mid x\\ \text{Term} \ni t & ::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x\\ \text{Reduction Context} \ni E & ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E\\ \text{Def-use Chain} \ni C & ::= [\,] \mid \mathsf{let}\ x = [\,]\,x\ \mathsf{in}\ E[C[x]]\end{array}$$

$$
\begin{array}{lll}
(I) & \begin{array}{l}\mathsf{let}\ x_1 = I\ \mathsf{in}\\ E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in}\\ \quad E_0[C[x_0]]]\end{array} & \rightarrow \begin{array}{l}\mathsf{let}\ x_1 = I\ \mathsf{in}\\ E_1[\mathsf{let}\ x_0 = y_0\ \mathsf{in}\\ \quad E_0[C[x_0]]]\end{array}\\[2ex]
(K) & \begin{array}{l}\mathsf{let}\ x_2 = K\ \mathsf{in}\\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\\ \quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in}\\ \quad\quad E_0[C[x_0]]]]\end{array} & \rightarrow \begin{array}{l}\mathsf{let}\ x_2 = K\ \mathsf{in}\\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\\ \quad E_1[\mathsf{let}\ x_0 = y_1\ \mathsf{in}\\ \quad\quad E_0[C[x_0]]]]\end{array}\\[2ex]
(S) & \begin{array}{l}\mathsf{let}\ x_3 = S\ \mathsf{in}\\ E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in}\\ \quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\\ \quad\quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in}\\ \quad\quad\quad E_0[C[x_0]]]]]\end{array} & \rightarrow \begin{array}{l}\mathsf{let}\ x_3 = S\ \mathsf{in}\\ E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in}\\ \quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\\ \quad\quad E_1[\mathsf{let}\ x_4 = y_2\, y_0\ \mathsf{in}\\ \quad\quad\quad \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in}\\ \quad\quad\quad \mathsf{let}\ x_0 = x_4\, x_5\ \mathsf{in}\\ \quad\quad\quad E_0[C[x_0]]]]]\\ \text{where } x_4 \text{ and } x_5 \text{ are fresh}\end{array}\\[2ex]
(ref) & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \rightarrow\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_1]]
\end{array}
$$

Figure 10.6: Reduction semantics for combinatory graph reduction over preprocessed terms: axioms

Axioms Figure 10.6 displays the axioms. Each of (I) and (K) is much as the corresponding axiom in Section 10.2, though we have specialized it with respect to the grammar of preprocessed terms. The (S) axiom is a further specialization of the (S') axiom. Specifically, since the right-hand side of any combination is known to be a reference, there is no need to introduce new let expressions to preserve sharing. As for the (ref) axiom, it is unchanged.

The notion of reduction on preprocessed terms is $\mathscr{G}$:

$$\mathscr{G} = (I) \cup (K) \cup (S) \cup (ref)$$

Recompositions The recompositions of contexts and def-use chains are defined in the same way as in Section 10.2.

Decomposition Decomposition is much as in Section 10.2, though we have specialized it with respect to the grammar of preprocessed terms. Its definition is displayed in Figure 10.7.

One-step reduction Performing one contraction in a term that is not in $\mathscr{G}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

$$
\begin{array}{rl}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t, E\rangle_{term} \downarrow_{\mathrm{dec}} & \langle t, \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x, E\rangle_{term} \downarrow_{\mathrm{dec}} & \langle E, (x, [\,], [\,])\rangle_{cont} \\
\\
\langle [\,], t\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E, t\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E, (x_0, E_0, C)\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle x_0, d, E_0, C, E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E, (x_0, E_0, C)\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle E, (x_0, \mathsf{let}\ x = d\ \mathsf{in}\ E_0, C)\rangle_{cont} \\
& \quad \text{where } x \neq x_0 \\
\\
\langle x_1, I, E_1, \Pi_0^{i=0}(x_i, y_i, E_i)\cdot C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} \\
& \qquad E_0[C[x_0]]]\rangle_{dec} \\
& \quad \text{where } \langle C, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \quad \text{and } \langle E_0, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_2, K, E_2, \Pi_0^{i=1}(x_i, y_i, E_i)\cdot C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
& \qquad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} \\
& \qquad\quad E_0[C[x_0]]]]\rangle_{dec} \\
& \quad \text{where } \langle C, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \quad \text{and } \langle E_0, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_3, S, E_3, \Pi_0^{i=2}(x_i, y_i, E_i)\cdot C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
& \qquad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
& \qquad\quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} \\
& \qquad\qquad E_0[C[x_0]]]]]\rangle_{dec} \\
& \quad \text{where } \langle C, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \quad \text{and } \langle E_0, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0, d_0, E_0, C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
& \quad \text{where } d_0 = I \text{ and } |C| < 1 \\
& \qquad \text{or } d_0 = K \text{ and } |C| < 2 \\
& \qquad \text{or } d_0 = S \text{ and } |C| < 3 \\
& \quad \text{and } \langle C, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \quad \text{and } \langle E_0, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0, x_1\, y_0, E_0, C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, (x_1, [\,], (x_0, y_0, E_0)\cdot C)\rangle_{cont} \\
\langle x_0, x_1, E_0, C, E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E, \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
& \quad \text{where } \langle C, x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \quad \text{and } \langle E_0, C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]
\end{array}
$$

Figure 10.7: Reduction semantics for combinatory graph reduction over preprocessed terms: decomposition

Definition 106 (standard one-step reduction)**.** For any t,

$$
t \mapsto_{\mathscr{G}} t'' \quad \text{iff} \quad \begin{cases} \langle t, [\,]\rangle_{term} \downarrow^{*}_{\mathrm{dec}} \langle E, r\rangle_{dec} \\ (r, t') \in \mathscr{G} \\ \langle E, t'\rangle_{io} \Uparrow_{\mathrm{rec}} t'' \end{cases}
$$

Reduction-based evaluation The standard reduction-based evaluation is defined as the iteration of the standard one-step reduction. It thus enumerates the reduction sequence of any given program:

Definition 107 (standard reduction-based evaluation)**.** Standard reduction-based evaluation, $\mapsto^*_{\mathscr{G}}$, is the transitive-reflexive closure of standard one-step reduction, $\mapsto_{\mathscr{G}}$.

10.3.2 A storeless abstract machine

Using refocusing, the reduction-free abstract machine corresponding to Definition 107 is calculated as in Section 10.2.2. For brevity, we omit this uncompressed abstract machine here.

10.3.3 A storeless abstract machine after transition compression

The abstract machine of Section 10.3.2 can be hereditarily compressed as in Section 10.2.3. We display it in Figure 10.8.

Definition 108 (standard reduction-free evaluation)**.** Standard reduction-free evaluation, $\rightarrow^*_{\text{step}}$, is the transitive closure of $\rightarrow_{\text{step}}$. (See Figure 10.8.)

Proposition 109 (full correctness)**.** *For any program p,*

$$p \mapsto^*_{\mathscr{G}} t \ \wedge \ t \in \mathscr{G}\text{-}nf \iff \langle p, [\,] \rangle_{term} \rightarrow^*_{\text{step}} \langle t \rangle_{nf}$$

Proof. Correctness of refocusing and transition compression. □

10.3.4 Summary and conclusion

Starting from a completely spelled-out reduction semantics for preprocessed combinatory term graphs, we have derived a storeless abstract machine. As in Section 10.2, these two semantic artifacts share the same syntactic representations and proceed in lock step.

10.4 Store-based combinatory graph reduction

In this section, we no longer denote graph vertices with let bindings, but as bindings in a global store:

$$\begin{aligned} \text{Global Store} \ni \sigma \\ \text{Location} \ni x, y \end{aligned}$$

Indeed, in the storeless accounts of Sections 10.2 and 10.3, let expressions declare references to denotable terms, and all these references are distinct. In the store-based account presented in this section, a global store maps locations to storable terms. Given a store σ, a location x and a storable term s, we write $\sigma[x := s]$ for the store σ' such that $\sigma'(x) = s$ and $\sigma'(x') = \sigma(x')$ for $x' \neq x$.

We therefore revise the syntax of Section 10.3 as follows:

$$\begin{aligned} \text{Program} \ni p &::= (x, \sigma) \\ \text{Storable Term} \ni s &::= I \mid K \mid S \mid x\,x \mid x \\ \text{Ancestor Stack} \ni a &::= [\,] \mid (x, x) \cdot a \end{aligned}$$

$$
\begin{aligned}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} &\to_{\mathrm{step}} \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} &\to_{\mathrm{step}} \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
\\
\langle [\,],\ t\rangle_{cont} &\to_{\mathrm{step}} \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} &\to_{\mathrm{step}} \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\to_{\mathrm{step}} \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\to_{\mathrm{step}} \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
&\quad \text{where } x \neq x_0 \\
\\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E_1 \circ_{io} \mathsf{let}\ x_1 = I\ \mathsf{in}\ E,\ (y_0,\ \mathsf{let}\ x_0 = y_0\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E_2 \circ_{io} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E,\ (y_1,\ E',\ C)\rangle_{cont} \\
&\quad \text{where } E' = (\mathsf{let}\ x_1 = x_2\, x_1\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = y_1\ \mathsf{in}\ E_0) \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ y_i,\ E_i)\cdot C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E_3 \circ_{io} \mathsf{let}\ x_3 = S\ \mathsf{in}\ E,\ (y_2,\ E',\ C')\rangle_{cont} \\
&\quad \text{where } E' = (\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in}\ E_1) \\
&\quad \text{and } C' = (x_4,\ y_0,\ \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in}\ [\,]) \cdot (x_0,\ x_5,\ E_0) \cdot C \\
&\quad \text{and } x_4 \text{ and } x_5 \text{ are fresh} \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
&\quad \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \\
&\quad \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
&\quad \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\, y_0,\ E_0,\ C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E,\ (x_1,\ [\,],\ (x_0,\ y_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} &\to_{\mathrm{step}} \langle E,\ (x_1,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\ C)\rangle_{cont}
\end{aligned}
$$

Figure 10.8: Storeless abstract machine for combinatory graph reduction over preprocessed terms after transition compression

A program now pairs the root of a graph in a store with this store. Denotable terms have been replaced by storable terms. Terms and reduction contexts have been replaced by locations in the store. Def-use chains have become what Turner calls *(left) ancestor stacks* [209, page 42]. We write $|a|$ for the height of an ancestor stack a.

Translating the preprocessed term graphs of Section 10.3 to store-based term graphs is straightforward:

Definition 110 (let-based to store-based)**.**

$$[\![\mathsf{let}\ x = d\ \mathsf{in}\ t]\!]_\sigma = [\![t]\!]_{\sigma[x:=[\![d]\!]']} \qquad [\![x]\!]_\sigma = (x,\sigma) \qquad [\![I]\!]' = I \quad [\![K]\!]' = K \quad [\![S]\!]' = S \qquad [\![x_0\,x_1]\!]' = [\![x_0]\!]'\,[\![x_1]\!]' \qquad [\![x]\!]' = x$$

where the auxiliary mapping $[\![\cdot]\!]'$ maps denotables to storables.

This compositional encoding maps the explicit declaration of a reference in a let expression to an implicit declaration of a location in the store. The resulting store-based combinatory terms can therefore also be interpreted as term graphs:

Definition 111 (interpreting store-based combinatory terms as term graphs)**.** For any translated store-based term (x,σ), its term graph $\gamma(x,\sigma)$ over the function symbols I, K, S, and A is defined as follows:

- $N_{(x,\sigma)}$ is the set of declared locations in σ.
- $lab_{(x,\sigma)}$ is defined on the contents of a location (i.e., a storable term): for a combinator, it yields the respective function symbol I, K or S; for a combination, it yields the application symbol A; and for a location, it yields the result of applying $lab_{(x,\sigma)}$ to this location, which in effect acts as an alias for a node.[10]
- $succ_{(x,\sigma)}$ is defined on the contents of a location: for a combination, it yields the corresponding pair of locations, and for everything else, the empty tuple.
- $r_{(x,\sigma)}$ is the root of the store-based term, i.e., x.

Proposition 112 (equivalence of term graphs)**.** *The term graph of any closed let-based combinatory term, t, coincides with the term graph of the corresponding store-based combinatory term, $[\![t]\!]_\sigma$, for any σ, i.e., $\gamma(t) \equiv \gamma([\![t]\!]_\sigma)$.*

Proof. By induction on t, using a sub-induction on denotable terms. □

10.4.1 A reduction semantics

Axioms An axiom is of the form $(x,\sigma) \to (x',\sigma')$ where x and x' are the left and right root respectively. For such an axiom, a redex is a pair (x'',σ'') together with a renaming of locations defined by a structure-preserving function on storable terms, π, such that:

$$\pi(x) = x'' \text{ and } \forall y \in dom(\sigma)\ .\ \pi(\sigma(y)) = \sigma''(\pi(y))$$

[10] Again, this application is well behaved since translated terms are acyclic.

$$
\begin{array}{lll}
(I) & (x_0,\ \sigma[x_1 := I][x_0 := x_1\, y_0]) & \to\ (x_0,\ \sigma[x_1 := I][x_0 := y_0]) \\
(K) & \begin{array}{l}(x_0,\ \sigma[x_2 := K] \\ \quad [x_1 := x_2\, y_1] \\ \quad [x_0 := x_1\, y_0])\end{array} & \to\ \begin{array}{l}(x_0,\ \sigma[x_2 := K] \\ \quad [x_1 := x_2\, y_1] \\ \quad [x_0 := y_1])\end{array} \\
(S) & \begin{array}{l}(x_0,\ \sigma[x_3 := S] \\ \quad [x_2 := x_3\, y_2] \\ \quad [x_1 := x_2\, y_1] \\ \quad [x_0 := x_1\, y_0])\end{array} & \to\ \begin{array}{l}(x_0,\ \sigma[x_3 := S] \\ \quad [x_2 := x_3\, y_2] \\ \quad [x_1 := x_2\, y_1] \\ \quad [x_4 := y_2\, y_0] \\ \quad [x_5 := y_1\, y_0] \\ \quad [x_0 := x_4\, x_5]) \\ \text{where } x_4 \text{ and } x_5 \text{ are fresh}\end{array} \\
(loc_1) & (x_0,\ \sigma[x_0 := x_1]) & \to\ \begin{array}{l}(x_1,\ \sigma[x_0 := x_1]) \\ \text{where } x_0 \text{ is the graph root}\end{array} \\
(loc_2) & (x_0,\ \sigma[x_0 := x_1]) & \to\ \begin{array}{l}(x_1,\ \sigma[x_0 := x_1][x := x_1\, y]) \\ \text{where } x \text{ is reachable from the graph root} \\ \text{and } \sigma(x) = x_0\, y \text{ for some } y\end{array}
\end{array}
$$

Figure 10.9: Reduction semantics for store-based combinatory graph reduction: axioms

In words, the renaming must map the left root to the root of the redex, and any location in the store of the axiom must have a corresponding location in the store of the redex. As before, we write $\sigma[x := s]$ for a store mapping the location x to the storable term s.

The axioms are displayed in Figure 10.9. They assume that there is a path from the graph root to the redex root. This assumption mirrors the decomposition conditions in the axioms of Figure 10.6. Consequently, the location axiom is split in two cases: one if the redex root is the graph root, corresponding to a decomposition into the empty def-use chain, and one if the redex root is not the graph root, corresponding to a decomposition into a non-empty def-use chain.

The notion of reduction on store-based terms is $\mathscr{H}$:

$$\mathscr{H} = (I) \cup (K) \cup (S) \cup (loc_1) \cup (loc_2)$$

Recomposition The recomposition of an ancestor stack with a store-based term relocates the root of the graph:

$$\frac{}{\langle [\,],\, x,\, \sigma\rangle_{stack} \Uparrow_{\mathrm{rec}} (x, \sigma)} \qquad \frac{\langle a,\, x_0,\, \sigma\rangle_{stack} \Uparrow_{\mathrm{rec}} (x', \sigma)}{\langle (x_0, y_0) \cdot a,\, x,\, \sigma\rangle_{stack} \Uparrow_{\mathrm{rec}} (x', \sigma)}$$

Definition 113 (recomposition of ancestor stacks with store-based terms)**.** An ancestor stack a is recomposed with a store-based term (x, σ) into a store-based term (x', σ) whenever $\langle a,\, x,\, \sigma\rangle_{stack} \Uparrow_{\mathrm{rec}} (x', \sigma)$ holds.

$$\langle [\,],\, x,\, \sigma\rangle_{stack} \downarrow_{\mathrm{dec}} \langle (x,\sigma)\rangle_{nf}$$
$$\langle (x_0,y_0)\cdot a,\, x_1,\, \sigma\rangle_{stack} \downarrow_{\mathrm{dec}} \langle a,\, x_0,\, \sigma\rangle_{stack}$$

$$\langle x_1,\, I,\, (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec}$$
$$\langle x_2,\, K,\, (x_1,y_1)\cdot (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec}$$
$$\langle x_3,\, S,\, (x_2,y_2)\cdot (x_1,y_1)\cdot (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec}$$
$$\langle x_0,\, s,\, a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle a,\, x_0,\, \sigma\rangle_{stack} \quad \text{where } s = I \text{ and } |a| < 1 \text{ or } s = K \text{ and } |a| < 2 \text{ or } s = S \text{ and } |a| < 3$$
$$\langle x_0,\, x_1\, y_0,\, a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle x_1,\, \sigma(x_1),\, (x_0,y_0)\cdot a,\, \sigma\rangle_{sto}$$
$$\langle x_0,\, x_1,\, a,\, \sigma\rangle_{sto} \downarrow_{\mathrm{dec}} \langle a,\, (x_0,\ \sigma)\rangle_{dec}$$

Figure 10.10: Reduction semantics for store-based combinatory graph reduction: decomposition

Decomposition Decomposition is much as in Section 10.2 though we have further specialized it with respect to store-based terms. The search previously done at return time is now done at call time. Starting from the root location, x, we recursively search for a redex, incrementally constructing an ancestor stack for x. If we do not find any redex, the term is in $\mathscr{H}$-normal form. Figure 10.10 displays this search as a transition system:

Definition 114 (decomposition)**.** The decomposition relation, $\downarrow^*_{\mathrm{dec}}$, is the transitive closure of $\downarrow_{\mathrm{dec}}$. (See Figure 10.10.)

As in Section 10.2.1, decomposition can be refunctionalized to a total and compositional function. Formally:

Property 115 (vacuous decomposition of normal forms)**.** For any $(x,\sigma) \in \mathscr{H}$-nf,

$$\langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \downarrow^*_{\mathrm{dec}} \langle (x,\sigma)\rangle_{nf}$$

Property 116 (unique decomposition of non-normal forms)**.** For any $(x,\sigma) \notin \mathscr{H}$-nf,

$$\langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \downarrow^*_{\mathrm{dec}} \langle a,\, (x',\sigma)\rangle_{dec}$$

where (x',σ) is the left-most outermost redex of (x,σ) and a is the ancestor stack from x to x'.

One-step reduction Performing one contraction in a term that is not in $\mathscr{H}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 117 (standard one-step reduction)**.** For any (x,σ),

$$(x,\sigma) \mapsto_{\mathscr{H}} (x',\sigma') \quad \text{iff} \quad \begin{cases} \langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \downarrow^*_{\mathrm{dec}} \langle a,\, (x'',\ \sigma)\rangle_{dec} \\ ((x'',\ \sigma),(x''',\ \sigma')) \ \in \ \mathscr{H} \\ \qquad \langle a,\, x''',\, \sigma'\rangle_{stack} \Uparrow_{\mathrm{rec}} (x',\sigma') \end{cases}$$

Reduction-based evaluation The standard reduction-based evaluation is defined as the iteration of the standard one-step reduction. It thus enumerates the reduction sequence of any given program:

Definition 118 (standard reduction-based evaluation)**.** Standard reduction-based evaluation, $\mapsto^*_{\mathscr{H}}$, is the transitive-reflexive closure of standard one-step reduction, $\mapsto_{\mathscr{H}}$.

The standard reduction of let-based combinatory terms (Definition 106) is bisimilar to the standard reduction of store-based combinatory terms (Definition 117):

Proposition 119 (bisimilarity of one-step reduction)**.** *For any let-based term t and store-based term (x, σ) such that $\gamma(t) \equiv \gamma(x, \sigma)$,*

$$t \mapsto_{\mathscr{G}} t' \iff (x, \sigma) \mapsto_{\mathscr{H}} (x', \sigma')$$

and $\gamma(t') \equiv \gamma(x', \sigma')$.

The proof of Proposition 119 makes use of the following lemmas:

Lemma 120 (decomposition lookup)**.** *For any let-based term $t = E[C[x]]$,*

$$\langle E, (x, [\,], C)\rangle_{cont} \downarrow^*_{\text{dec}} \langle x, d, E_0, C, E'\rangle_{den}$$

where

$$\begin{gathered}\langle C, x\rangle_{chain} \Uparrow_{\text{rec}} C[x]\\ \langle E_0, C[x]\rangle_{oi} \Uparrow_{\text{rec}} E_0[C[x]]\\ \langle E', \mathsf{let}\ x = d\ \mathsf{in}\ E_0[C[x]]\rangle_{io} \Uparrow_{\text{rec}} t\end{gathered}$$

Proof. By induction on E □

Lemma 121 (decomposition invariance)**.** *Let $\approx$ be a binary relation between den-states and sto-states defined by:*

$$\langle x, d, E_0, C, E\rangle_{den} \approx \langle x, s, a, \sigma\rangle_{sto} \ \textit{iff}\ \gamma(E[t]) \equiv \gamma(r_t, \sigma)$$

where $t = \mathsf{let}\ x = d\ \mathsf{in}\ E_0[C[x]]$, and

$$\begin{gathered}\langle C, x\rangle_{chain} \Uparrow_{\text{rec}} C[x]\\ \langle E_0, C[x]\rangle_{oi} \Uparrow_{\text{rec}} E_0[C[x]]\\ \langle E, t\rangle_{io} \Uparrow_{\text{rec}} E[t]\\ \langle a, x, \sigma\rangle_{stack} \Uparrow_{\text{rec}} (r_t, \sigma)\end{gathered}$$

If $\langle x, d, E_0, C, E\rangle_{den} \approx \langle x, s, a, \sigma\rangle_{sto}$, then

$$\begin{array}{ccc}\langle x, d, E_0, C, E\rangle_{den} & \downarrow^*_{\text{dec}} & \langle x', d', E'_0, C', E'\rangle_{den}\\ \Updownarrow & & \\ \langle x, s, a, \sigma\rangle_{sto} & \downarrow^*_{\text{dec}} & \langle x', s', a', \sigma\rangle_{sto}\end{array}$$

where $\langle x', d', E'_0, C', E'\rangle_{den} \approx \langle x', s', a', \sigma\rangle_{sto}$.

Proof. By case analysis on *den*-states using sub-induction for recompositions and using Lemma 120 for the combination case. □

Lemma 122 (sub-graph contraction)**.** *If* $\gamma(t) \sqsubseteq \gamma(x,\sigma)$ *and* $\langle a, x, \sigma\rangle_{stack} \Uparrow_{\text{rec}} (r_t, \sigma)$*, then*

$$(t,t') \in \mathscr{G} \iff ((x,\ \sigma),(x',\sigma')) \in \mathscr{H}$$

where $\gamma(t') \sqsubseteq \gamma(x',\sigma')$*,* $\langle a, x', \sigma'\rangle_{stack} \Uparrow_{\text{rec}} (r_{t'}, \sigma')$*, and* $\forall y \in N_{(x,\sigma)} \backslash N_t\ .\ \sigma(y) = \sigma'(y)$*.*

Proof. By case analysis on the axioms using sub-induction for recomposition. □

of Proposition 119. Let $\gamma(t) \equiv \gamma(x,\sigma)$, by induction on t and using Lemma 120:

$$\langle t, [\ ]\rangle_{term} \downarrow^*_{\text{dec}} \langle E, (x, [\ ], [\ ])\rangle_{cont} \downarrow^*_{\text{dec}} \langle x, d, E_0, [\ ], E'\rangle_{den}$$

where $t = E[x] = E'[\mathsf{let}\ x = d\ \mathsf{in}\ E_0[x]]$. Thus,

$$\langle x, d, E_0, [\ ], E'\rangle_{den} \approx \langle x, \sigma(x), [\ ], \sigma\rangle_{sto}$$

By Lemma 121 and assuming a redex is found:

$$\begin{array}{ccc} \langle x, d, E_0, [\ ], E'\rangle_{den} \downarrow^*_{\text{dec}} \langle x', d', E'_0, C, E''\rangle_{den} \downarrow_{\text{dec}} \langle t', E''\rangle_{dec} \\ \langle x, \sigma(x), [\ ], \sigma\rangle_{sto} \downarrow^*_{\text{dec}} \quad \langle x', s, a, \sigma\rangle_{sto} \quad \downarrow_{\text{dec}} \langle (x'', \sigma), a\rangle_{dec} \end{array}$$

where $\gamma(E''[t']) \equiv \gamma(r_{t'}, \sigma)$, and thus $\gamma(t') \sqsubseteq \gamma(x'', \sigma)$ and $\langle a, x'', \sigma\rangle_{stack} \Uparrow_{\text{rec}} (r_{t'}, \sigma)$. By Lemma 122: $(t', t'') \in \mathscr{G}$, $((x'', \sigma),(x''', \sigma')) \in \mathscr{H}$, $\gamma(t'') \sqsubseteq \gamma(x''', \sigma')$, $N_{E''[t'']} = N_{(x''',\sigma')}$, and thus we have that $\gamma(E''[t'']) \equiv \gamma(r_{t''}, \sigma')$ where

$$\begin{array}{c} \langle E'', t''\rangle_{io} \Uparrow_{\text{rec}} E''[t''] \\ \langle a, x''', \sigma'\rangle_{stack} \Uparrow_{\text{rec}} (r_{t''}, \sigma') \end{array}$$

using induction on E''. □

10.4.2 A store-based abstract machine

Using refocusing, the reduction-free abstract machine corresponding to Definition 118 is calculated as in Section 10.2.2. For brevity, we omit this uncompressed abstract machine here.

10.4.3 A store-based abstract machine after transition compression

The abstract machine of Section 10.4.2 can be hereditarily compressed as done in Section 10.2.3. We display it in Figure 10.11. Its architecture is that of Turner's SK-reduction machine [209]: the left-ancestor stack is incrementally constructed at each combination; upon reaching a combinator, its arguments are found on top of the ancestor stack and a graph transformation takes place to rearrange them. In particular, our handling of stored locations coincides with Turner's indirection nodes. The only differences are that our machine accepts the partial application of combinators and that Turner's combinators are unboxed, which is an optimization.

Definition 123 (standard reduction-free evaluation)**.** Standard reduction-free evaluation, $\rightarrow^*_{\text{step}}$, is the transitive closure of $\rightarrow_{\text{step}}$. (See Figure 10.11.)

Proposition 124 (full correctness)**.** *For any program* (x,σ)*,*

$$(x,\sigma) \mapsto^*_{\mathscr{H}} (x',\sigma') \ \wedge\ (x',\sigma') \in \mathscr{H}\text{-}nf \iff \langle x, \sigma(x), [\ ], \sigma\rangle_{sto} \rightarrow^*_{\text{step}} \langle (x',\sigma')\rangle_{nf}$$

Proof. Correctness of refocusing and transition compression. □

$$
\begin{aligned}
\langle [\,], x, \sigma\rangle_{stack} &\to_{\text{step}} \langle (x,\sigma)\rangle_{nf}\\
\langle (x_0,y_0)\cdot a, x_1, \sigma\rangle_{stack} &\to_{\text{step}} \langle a, x_0, \sigma\rangle_{stack}\\
\\
\langle x_1, I, (x_0,y_0)\cdot a, \sigma\rangle_{sto} &\to_{\text{step}} \langle x_0, y_0, a, \sigma[x_0 := y_0]\rangle_{sto}\\
\langle x_2, K, (x_1,y_1)\cdot(x_0,y_0)\cdot a, \sigma\rangle_{sto} &\to_{\text{step}} \langle x_0, y_1, a, \sigma[x_0 := y_1]\rangle_{sto}\\
\langle x_3, S, (x_2,y_2)\cdot(x_1,y_1)\cdot(x_0,y_0)\cdot a, \sigma\rangle_{sto} &\to_{\text{step}} \langle y_2, \sigma'(y_2), (x_4,y_0)\cdot(x_0,y_5)\cdot a, \sigma'\rangle_{sto}\\
&\quad \text{where } \sigma' = \sigma[x_4 := y_2\, y_0]\\
&\qquad\qquad [x_5 := y_1\, y_0]\\
&\qquad\qquad [x_0 := x_4\, x_5]\\
&\quad \text{and } x_4 \text{ and } x_5 \text{ are fresh}\\
\langle x_0, s, a, \sigma\rangle_{sto} &\to_{\text{step}} \langle a, x_0, \sigma\rangle_{stack}\\
&\quad \text{where } s = I \text{ and } |a| < 1,\\
&\qquad \text{or } s = K \text{ and } |a| < 2\\
&\qquad \text{or } s = S \text{ and } |a| < 3\\
\langle x_0, x_1\, y_0, a, \sigma\rangle_{sto} &\to_{\text{step}} \langle x_1, \sigma(x_1), (x_0,y_0)\cdot a, \sigma\rangle_{sto}\\
\langle x_0, x_1, [\,], \sigma\rangle_{sto} &\to_{\text{step}} \langle x_1, \sigma(x_1), [\,], \sigma\rangle_{sto}\\
\langle x_0, x_1, (x,y)\cdot a, \sigma\rangle_{sto} &\to_{\text{step}} \langle x_1, \sigma'(x_1), (x,y)\cdot a, \sigma'\rangle_{sto}\\
&\quad \text{where } \sigma' = \sigma[x := x_1\, y]
\end{aligned}
$$

Figure 10.11: Store-based abstract machine for combinatory graph reduction after transition compression

10.4.4 Summary and conclusion

Starting from a completely spelled-out reduction semantics for combinatory term graphs in a store, we have derived a store-based abstract machine. The structure of this store-based abstract machine coincides with that of Turner's SK-reduction machine.

10.5 The Y Combinator

In this section, we briefly cover extending the semantics for the preprocessed terms of Section 10.3 with cyclic constructs.[11] More precisely, we add a fixed-point combinator Y as traditionally specified by the equation $Y\,t = t(Y\,t)$. This equation exhibits two types of duplication:

1. the duplication of non-recursive subterms, in this case t; and
2. the duplication of locally recursive subterms, in this case $Y\,t$.

Both types of duplication can be avoided using graphs, and these graphs can be represented syntactically. To syntactically capture the sharing of non-recursive subterms, we can use let expressions as shown in the previous sections. To syntactically capture the sharing of locally recursive subterms, we enrich the def-use chains with a letrec construct for local recursive declarations, letrec $x = d$ in t, where the denoted term, d, can refer to

[11] The implementation in ML is available at: `http://www.zerny.dk/syntactic-graph-reduction.html`

its reference, x, recursively. This "tying of the knot," to paraphrase Landin, is realized by the following axiom:

$$
\begin{array}{lll}
(Y) & \mathsf{let}\ x_1 = Y\ \mathsf{in} & \to\ \mathsf{let}\ x_1 = Y\ \mathsf{in} \\
& \quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} & \quad\ E_1[\mathsf{letrec}\ x_0 = y_0\, x_0\ \mathsf{in} \\
& \qquad E_0[C[x_0]]] & \qquad E_0[C[x_0]]]
\end{array}
$$

Instead of containing a new subterm of the form $Y\,t$, the contractum recursively refers to itself through the recursive declaration letrec. The contractum thus shares the recursively defined subterm $Y\,t$.

Cyclic terms raise a new issue: naive contraction can bring references out of the scope of their definitions. A correct treatment is thus to λ-lift [72, 121] the cyclic definition so it can be referred to lexically in the contractum.

This situation arises in the (S) axiom. Consider the following redex and its contraction according to the (S) axiom of Figure 10.6:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\qquad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0\ \mathsf{in} \\
\qquad\quad E_0[C[x_0]]]]]
\end{array}
\quad\to\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\qquad E_1[\mathsf{let}\ x_4 = y_2\, x_0\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x_5 = y_1\, x_0\ \mathsf{in} \\
\qquad\quad \mathsf{letrec}\ x_0 = x_4\, x_5\ \mathsf{in} \\
\qquad\quad E_0[C[x_0]]]]] \\
\text{where } x_4 \text{ and } x_5 \text{ are fresh}
\end{array}
$$

Here the contractum contains two occurrences of x_0 that are out of scope. This scoping issue is solved by λ-lifting the recursive binding:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\qquad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0\ \mathsf{in} \\
\qquad\quad E_0[C[x_0]]]]]
\end{array}
\quad\to\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\qquad E_1[\mathsf{letrec}\ x_6 = x_1\, x_6\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x_4 = y_2\, x_6\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x_5 = y_1\, x_6\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ x_0 = x_4\, x_5\ \mathsf{in} \\
\qquad\quad E_0[C[x_0]]]]] \\
\text{where } x_4,\ x_5 \text{ and } x_6 \text{ are fresh}
\end{array}
$$

This solution does maintain proper scoping, but it still suffers from the same duplication of locally recursive subterms as we started out with. If x_6 is needed, the same contraction will again take place instead of reusing the contractum just obtained. In other words, we would like to tie another knot.

From knot to Gordian knot We further enrich the def-use chains with a letrec construct for local and mutually recursive declarations. The example from before can then

be restated as follows:

$$
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_0 = x_1\, x_0\ \mathsf{in} \\
\quad\quad\quad E_0[C[x_0]]]]]
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{let}\ x_3 = S\ \mathsf{in} \\
E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
\quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
\quad\quad E_1[\mathsf{letrec}\ x_4 = y_2\, x_0 \\
\quad\quad\quad\quad\quad\quad x_5 = y_1\, x_0 \\
\quad\quad\quad\quad\quad\quad x_0 = x_4\, x_5 \\
\quad\quad\quad\quad\quad\quad \mathsf{in}\ E_0[C[x_0]]]]] \\
\quad\quad\quad\quad\quad\quad \text{where } x_4 \text{ and } x_5 \text{ are fresh}
\end{array}
$$

Adding mutually recursive definitions makes it possible to share more subgraphs. Indeed, in his PhD thesis, Blom [33, Section 4.5] describes how increasing the number of mutually recursive definitions strictly increases expressible sharing.

Now that terms are cyclic, special care must be taken when defining their interpretation as term graphs. In contrast to Definition 105, the definition of lab_t must map any reference, $t = x$, that is part of a cyclic chain of references[12] to a term graph of undefined value, e.g., the fixed point of I. For example, consider the reduction of $Y\, I$:

$$
\begin{array}{l}
\mathsf{letrec}\ x_2 = Y \\
\quad\quad x_1 = I \\
\quad\quad x_0 = x_2\, x_1 \\
\mathsf{in}\ x_0
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{letrec}\ x_2 = Y \\
\quad\quad x_1 = I \\
\quad\quad x_0 = x_1\, x_0 \\
\mathsf{in}\ x_0
\end{array}
\quad\rightarrow\quad
\begin{array}{l}
\mathsf{letrec}\ x_2 = Y \\
\quad\quad x_1 = I \\
\quad\quad x_0 = x_0 \\
\mathsf{in}\ x_0
\end{array}
$$

The rightmost reduct in the above reduction sequence does not contain a redex. Indeed, our decomposition function as defined for acyclic terms would diverge if applied to such a cyclic term. Following tradition, we reduce these problematic cyclic terms to a "*black hole*" value in our syntactic theory of cyclic graph reduction using letrec. For a syntactic theory of the cyclic call-by-need λ-calculus using letrec, we refer the reader to Ariola and Felleisen's work [1997] and to Nakata and Hasegawa's work [2009].

In any case, after specifying the syntactic treatment of cyclic programs, the story is once again compellingly simple: as in Sections 10.2.2, 10.3.2 and 10.4.2, an abstract machine can be calculated directly from the reduction semantics. The resulting machine is larger but the derivation applies.

10.6 Related work

It has long been noticed that combinators make it possible to do without variables. For example, in the 1960s, Robinson outlined how this could be done to implement logics [181]. However, it took Turner to realize in the 1970's that (1) combinatory graph reduction could be implemented efficiently, and (2) combinatory graph reduction provides an efficient implementation technique for lazy functional languages [209]. Turner's work ignited a culture of implementation techniques in the 1980's [159], whose goal in retrospect can be characterized as designing efficient big-step graph reducers.

[12] By a cyclic chain of references, we mean any series of references for which one points back to a previous element in the chain and without any intermediate occurrence of a combination anywhere in the chain. Whether a reference is part of a such a cyclic chain is a decidable property.

Due to the increased interest in graph reduction, Barendregt et al. [24] developed term graphs and term-graph rewriting. Their work has since been used to model languages with sharing and to reason about program transformations in the presence of sharing [11, 100, 118, 126, 168]. Later work by Ariola and Klop [13] provides an equational theory for term-graph rewriting with cycles, a topic further developed by Blom [33] in his PhD thesis and since by Nakata and Hasegawa [152].

Over the 2000's, the first author and his students have investigated off-the-shelf program-transformation techniques for inter-deriving semantic artifacts [5, 70, 222]. The present work is an outgrowth of this investigation.

10.7 Conclusion and future work

Methodologically, mathematicians who wrote about their art (Alexandre Grothendriek, Jacques Hadamard, Paul Halmos, Godfrey H. Hardy, Donald E. Knuth, John E. Littlewood, and George Pólya for example) clearly describe how their research is typically structured in two stages: (1) an exploratory stage where they boldly move forward, discovering right and left, and (2) a descriptive stage where they retrace their steps and revisit their foray, verify it, structure it, and put it into narrative shape. As far as abstract machines are concerned, tradition has it to seek new semantic artifacts, which is characteristic of the first stage. Our work stems from this tradition, though by now it subscribes to the second stage as we field-test our derivational tools.

The present article reports our field test of combinatory graph reduction. Our main result is that representing def-use chains using reduction contexts and let expressions, which, in retrospect, is at the heart of Ariola et al.'s syntactic theory of the call-by-need λ-calculus, also makes it possible to account for combinatory graph reduction. We have stated in complete detail three reduction semantics and have derived three storeless abstract machines. Interpreting denotable entities as storable ones in a global store, we have rediscovered David Turner's graph-reduction machine.

Currently, we are inter-deriving natural semantics that correspond to the abstract machines. We are also adding literals and strict arithmetic and logic functions, as well as garbage-collection rules such as the following one:

$$\mathsf{let}\ x = d\ \mathsf{in}\ t\ \rightarrow\ t \quad \text{if } x \text{ does not occur in } t$$

We are also wondering which kind of garbage collector is fostered by the nested let expressions of the syntactic theories and also the extent to which its references are akin to Curry's apparent variables [52].

Acknowledgments Thanks are due to the anonymous reviewers and their eagle eyes. We are also grateful to to Marìa Alpuente for her generous invitation to present a preliminary version of this material at LOPSTR 2010, and to Kenichi Asai, Mayer Goldberg, Steffen Daniel Jensen, and Julia Lawall for their comments on an earlier version of this article.

The present study of a computational reality is a tribute to the first author's *directeur de thèse*, Bernard Robinet (1941–2009), who introduced a generation of students to Combinatory Logic and contexts [180].

Chapter 11

A logical correspondence between abstract machines and natural semantics

Joint work with Robert J. Simmons.

Abstract

We present a logical correspondence between natural semantics and abstract machines. This correspondence enables the mechanical and fully-correct construction of an abstract machine from a natural semantics. Our *logical correspondence* mirrors the Reynolds *functional correspondence* but places it within the domain of a logical framework. Natural semantics and abstract machines are instances of *substructural operational semantics*. As a byproduct, using a *substructural* logical framework, we bring concurrent and stateful models into the domain of the logical correspondence.

11.1 Introduction

The literature contains numerous specifications of formal semantics and therefore many proposals for relating them. These relations are stated using a diversity of methods and methodologies. To the best of the authors' knowledge, the only methodology to have seen repeated use outside the work of its inventors [10, 17, 175, 193] is the Reynolds functional correspondence [5, 176].

Our goal [198, 224] is to develop a logical counterpart to this functional correspondence. We want it to be formal, mechanizable, and widely applicable.

Logic provides adequate means for the specification of programming-language semantics in general [108]. In other words, logical specifications are formal objects that can be stated and proved in one-to-one correspondence with 'informal' semantic objects. Specifically, *substructural logics* provide an expressive specification language for concurrent and effectful systems [165, 218].

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\, e_2 \Downarrow v}$$

Figure 11.1: A natural semantics for CBV evaluation

$$\frac{}{k \triangleright \lambda x.e \mapsto k \triangleleft \lambda x.e} \qquad \frac{}{k \triangleright e_1\, e_2 \mapsto (k; \square\, e_2) \triangleright e_1}$$

$$\frac{}{(k; \square\, e_2) \triangleleft \lambda x.e \mapsto (k; (\lambda x.e)\, \square) \triangleright e_2}$$

$$\frac{}{(k; (\lambda x.e)\, \square) \triangleleft v_2 \mapsto k \triangleright e[v_2/x]}$$

Figure 11.2: An abstract machine semantics for CBV evaluation

In this paper, we present a method for relating natural semantics with abstract machines within a logical framework. Following tradition [45, 123], we interpret Horn-clause specifications of a natural semantics as a logic program. The resulting logic program is then transformed in several steps to make its operational behavior explicit. These steps mirror steps in the functional correspondence, yet the transformations have been developed independently and for other purposes. Indeed, this operationalization is an instance of a more general theorem relating backwards-chaining and forwards-chaining proof search.

Outline Section 11.2 motivates the use of different semantic styles with a discussion of related work and the introduction of a simple call-by-value language which will be our running example throughout. Section 11.3 describes the logical interpretation of natural semantics. Section 11.4 describes the logical interpretation of abstract-machine semantics. Section 11.5 establishes the correspondence between these interpretations in terms of two generally correct transformations on specifications, operationalization (Section 11.5.1) and defunctionalization (Section 11.5.2), as well as the prototype implementation of these transformations (Section 11.5.3). Section 11.6 discusses nondeterminism and its relation to proof-search interpretations. Section 11.7 puts the present work in perspective with discussions on type safety (Section 11.7.1), destination passing style (Section 11.7.2), abstract interpretation (Section 11.7.3), and inverse correspondences (Section 11.7.4).

11.2 Accounting for divergence and failure

We start by illustrating the difference in expressiveness of a natural semantics compared to an abstract machine with the problem of distinguishing divergence from failure. This problem will serve as our running example throughout, being simple and concise. However, the development holds in general even for complicated languages with complex control, such as call by need, as discussed in Section 11.5.3.

Expressions are λ-terms in addition to an extra nonsense term, here denoted by `junk`. As usual, syntactic values are λ-abstractions, and contexts are a list of application frames

terminated by the empty frame halt:

$$
\begin{array}{l}
e ::= x \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{junk} \\
v ::= \lambda x.e \\
k ::= \mathsf{halt} \mid k;\Box\, e_2 \mid k;(\lambda x.e)\,\Box
\end{array}
$$

We give two semantics for call-by-value (CBV) evaluation: Figure 11.1 defines a big-step semantics in the form of a natural semantics [45, 123]; Figure 11.2 defines a small-step semantics in the form of an abstract machine. Common to these specifications is the appearance of being specified by the same logical tool: *inductive definitions*. However, their means of characterization are very different.

Consider the term $\omega = (\lambda x.x\,x)(\lambda x.x\,x)$. The abstract machine can characterize developments of ω. The natural semantics cannot find a v such that $\omega \Downarrow v$ is derivable. As a small-step semantics the abstract machine can characterize how $((\lambda x.x\,x)\,\mathsf{junk})$ goes wrong. As a big-step semantics the natural semantics cannot find a v such that $((\lambda x.x\,x)\,\mathsf{junk}) \Downarrow v$ is derivable. Working with the natural semantics, therefore, we cannot express which programs can go wrong, should we for example wish to show that they are outside a set of well-typed programs that includes non-terminating programs.

One solution to this inexpressibility is to introduce an additional big-step predicate that characterizes how evaluation can go wrong. Another solution is to characterize the set of incomplete derivations that operationally correspond to stuck states. In both cases, this doubles the size of the specification and is fragile. Nothing prohibits writing an incorrect specification. This fragility is a known obstacle to proving type soundness with natural-semantics specifications. Several efforts have been made to ameliorate this situation:

A coinductive representation Based on the work of Cousot and Cousot [50], Leroy and Grall strengthened the expressive power of a big-step semantics [137] by using coinductive definitions. Similarly, Danielsson used the partiality monad to account for divergence [54]. Both solutions require inventing a new semantics in a new semantic notation and require much of the same duplication as is the case for a natural semantics, e.g., inductive rules for converging computations and coinductive rules for diverging computations. In contrast, we use established logics to represent established semantic notations and the result, a substructural specification of an abstract machine, represents an artifact of considerable expressiveness, e.g., as a transition relation for a collecting semantics [49].

A functional representation Based on Reynolds's seminal work on definitional interpreters [176], Danvy et al. have shown how semantic specifications can be inter-derived [5, 7, 8, 27, 62]. Their method is to represent the semantic artifacts as functional programs and to inter-derive these functional representations using off-the-shelf and fully-correct program transformations, thereby subsuming the practice of reinventing new semantics on a case-by-case basis. Subsequent work has shown this correspondence to be widely applicable, including applications to algorithms in general [76], to languages with computational effects [8], to properly tail-recursive stack inspection [27], to lazy evaluation [7, 77], and logic [32, 175]. We present a logical correspondence inspired by this work where the logical interpretations of the semantic artifacts are adequate and formally specified within a logical framework.

A logical representation The phrase "natural semantics" was introduced in the TYPOL compiler which translated natural semantics specifications to logic programs in Prolog [45]. The LF logical framework [110] and the logic-programming interpretation given by Elf [161] follow the same tradition, representing natural semantics as an inductive definition and interpreting it operationally by a proof-search procedure. Both Hannan and Miller [106] and Ager [3] derive abstract machines by representing a natural semantics as a logical specification, in λProlog and L-attributed grammars respectively, and then applying logical transformations. Our work follows this tradition of interpreting logical specifications and assigning operational behavior by means of proof search. Furthermore, our transformations yields substructural specifications within a logical framework that supports stateful and concurrent features [165, 198].

11.3 Natural semantics as logic programs

This section makes precise the on-paper inductive definitions that define a natural semantics. Concretely, we follow the LF methodology and require adequate encodings of the syntactic and deductive apparatus in the logical framework LF [110]. These encodings enable formal reasoning with the support of mechanical verification [109]. The methodology of adequate encoding is generally applicable and well-covered elsewhere. In this section, we present adequate LF encodings of our specific running examples: the expressions of the λ-calculus and the natural semantics from Figure 11.1.

In Section 11.3.1, we present and discuss an LF signature as a logical artifact that adequately encodes the natural semantics in Figure 11.1. In Section 11.3.2, we informally describe how our logical artifact can be given an alternative operational interpretation as proof search.

11.3.1 Encoding

The encoding of a natural semantics into LF consists of two parts: first, we must provide an encoding of the informal syntax and judgments of our object language into representations in LF, and second, we must show these encodings adequate.

The syntax of λ-expressions from the introduction is represented by canonical forms of type exp under the following LF signature.

```
exp: type.
lam: (exp → exp) → exp.
app: exp → exp → exp.
junk: exp.
```

In words, λ-expressions are represented by the LF type exp and this type is given three formation rules:

1. lam takes an LF term of function type exp → exp and forms an expression in exp that represents a λ-abstraction;
2. app takes two LF terms of type exp and forms an expression in exp that represents a λ-application; and
3. junk takes no LF terms and forms an expression in exp that represents the extra nonsense term.

This encoding uses higher-order abstract syntax, i.e., it uses abstraction and substitution of the defining language to represent abstraction and substitution of the defined language. The LF framework is designed to provide this substitution mechanism together with a suitable meta-theory to state and prove that the encoding is *adequate* with respect to our 'informal' BNF presentation of expressions.

Theorem 125 (Adequacy for terms)**.** *Up to α-equivalence, there is a bijection between expressions e (with free variables in the set $\{x_1, \ldots, x_n\}$) as defined in the introduction and canonical (β-normal, η-long) LF terms M such that $x_1 : \mathsf{exp}, \ldots, x_n : \mathsf{exp} \vdash M : \mathsf{exp}$.*

We write this bijection as $\ulcorner e \urcorner = M$. We also need to know that substitution as defined over LF terms coincides with the definition of substitution that we refer to in our various operational semantics.

Theorem 126 (Compositionality for terms)**.** *For all e_1 and e_2, if x is not free in e_2 then $\ulcorner e_1[e_2/x] \urcorner = \ulcorner e_1 \urcorner [\ulcorner e_2 \urcorner / x]$.*

Harper and Licata prove a close analogue of these two theorems in great detail in [109, Section 3.2.2].

Having adequately encoded terms, we next use the dependent types of LF to encode the judgment $e \Downarrow v$ as the dependent type family $\mathsf{ev}\ \ulcorner e \urcorner\ \ulcorner v \urcorner$ that takes two arguments of type exp, following the LF encoding principle of *judgments as types* [110].

evlam: ev (lam $\lambda x.\, \mathsf{E}\, x$) (lam $\lambda x.\, \mathsf{E}\, x$).
evapp: ev (app E1 E2) V
$\quad \leftarrow$ ev E1 (lam $\lambda x.\, \mathsf{E}\, x$)
$\quad \leftarrow$ ev E2 V2
$\quad \leftarrow$ ev (E V2) V.

The specification above uses the syntactic sugar of backwards-facing implication, which makes the rules read more like Prolog programs. More specifically, the LF type $M \leftarrow N_1 \leftarrow \ldots \leftarrow N_n$ is syntactic sugar for $N_n \rightarrow \ldots \rightarrow N_1 \rightarrow M$. Capital letters such as E1 and E2 are variables that are universally quantified implicitly over the entire rule.

Proofs constructed using the rules for natural semantics, as given in Figure 11.1, correspond to LF terms of type $\mathsf{ev}\, N_e\, N_v$ for some N_e and N_v. This correspondence is made precise by the following theorem:

Theorem 127 (Adequacy for the natural semantics)**.** *Up to α-equivalence, there is a bijection between derivations $\mathscr{D}$ of the judgment $e \Downarrow v$ and canonical LF terms M such that $\cdot \vdash M : \mathsf{ev}\ \ulcorner e \urcorner\ \ulcorner v \urcorner$.*

Adequacy theorems of this form proceed by straightforward induction, and are thoroughly covered elsewhere [109, Section 3.4]. There are no analogues to compositionality with natural semantics, and so this completes the LF encoding of our natural semantics from Figure 11.1. This is a standard methodology in the context of the Twelf meta-logical framework, and has been shown to be widely applicable.

11.3.2 Execution

We can operationalize a declarative specification in LF with a logic programming interpretation, as is done in the logical frameworks Elf [161] and Twelf [164]. The logic programming interpretation allows us to take an LF term M and attempt to find a term N such

that the dependent type $\mathsf{ev}\, M\, N$ is inhabited. By adequacy, this procedure is equivalent to taking the expression e (where $M = \ulcorner e \urcorner$) and attempting to produce an expression v (where $N = \ulcorner v \urcorner$) such that $e \Downarrow v$ is derivable. The following describes this search strategy:

- If $e = \lambda x.e'$, derive $\lambda x.e' \Downarrow \lambda x.e'$ with the rule evlam.
- If $e = e_1\, e_2$, attempt to derive $e_1\, e_2 \Downarrow v$ using the rule evapp:
  1. Search for a v_1 such that $e_1 \Downarrow v_1$ is derivable.
  2. Assert that $v_1 = \lambda x.e'$ for some e'; fail if it is not.
  3. Search for a v_2 such that $e_2 \Downarrow v_2$ is derivable.
  4. Let $e'' = e'[v_2/x]$
  5. Search for a v such that $e'' \Downarrow v$ is derivable.
  6. If we succeed, we can derive $e_1\, e_2 \Downarrow v$ using the rule evapp.

This procedural interpretation is not the default logic programming interpretation of Twelf or Prolog, because it does not account for backtracking search.[1] Because CBV evaluation in the λ-calculus is deterministic, backtracking is irrelevant, but we will return to this point in Section 11.6.

In the next section, we will describe an abstract machine that we claim is an implementation of this search procedure, and observe that it adequately encodes the abstract machine for the call-by-value λ-calculus from Figure 11.2. Then, in Section 11.5, we will support our claim by deriving the abstract machine from a sequence of generally applicable and provably correct transformations on logic programs.

11.4 Abstract machines as logic programs

In this section, we will describe a small-step abstract machine that adequately implements the search behavior described in Section 11.3.2. We could develop an on-paper formalism for these abstract machines, but rather than doing that, we will encode these abstract machines as another kind of logical specification, an *ordered logical specification* [165].

Like LF signatures, ordered logical specifications have a logic programming interpretation, but the two interpretations are rather different. The logic programming interpretation from the previous section was an instance of backward-chaining, Prolog-style logic programming, whereas the interpretation of ordered logic specifications is a generalization of forward-chaining, Datalog-style logic programming.

Ordered logical specifications are based on a first-order fragment of ordered linear logic [173], a substructural logic disallowing weakening, contraction, and exchange that dates back to Lambek's categorial grammars [130]. For the fragment we will consider in this section, we only need to think of ordered logical specifications as a peculiar syntax for string rewriting rules. Informally, the proposition $S_1 \bullet S_2$ is a pattern describing concatenation: if the string Δ_1 matches the pattern S_1 and the string Δ_2 matches the pattern S_2, then the string $\Delta_1\Delta_2$ matches the pattern $S_1 \bullet S_2$. Informally, the proposition $S_1 \rightarrowtail \{S_2\}$ is

[1] Twelf can specify this proof strategy by declaring the predicate ev to be deterministic. Prolog, along with most other logic programming languages, can specify this search strategy by using what are called *cuts*.

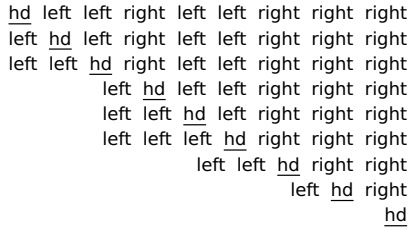


Figure 11.3: Example trace of the ordered logical specification for a PDA

a string rewriting rule[2]: if the string Δ_1 matches the pattern S_1 and the string Δ_2 matches the pattern S_2, then this rule enables the transition step $\Delta_L \Delta_1 \Delta_R \rightsquigarrow \Delta_L \Delta_2 \Delta_R$.

We will illustrate ordered logical specifications first with a very simple example taken from [199]. Consider a push-down automaton (PDA) that reads a string of symbols left-to-right while maintaining and manipulating a separate stack of symbols. We can represent any configuration of the PDA as a string with three regions:

[the stack] [the head] [the string being read]

where the symbols closest to the head are the top of the stack and the symbol waiting to be read from the string. If we represent the head as a token hd, represent the open parenthesis "(" with the token left, and represent the closed parentheses ")" with the token right, then we can describe the behavior of a single-state push-down automaton for checking the proper nesting of parentheses with two rules:

$$\textsf{push}: \ \textsf{hd} \bullet \textsf{left} \rightarrowtail \{\textsf{left} \bullet \textsf{hd}\}$$
$$\textsf{pop}: \ \textsf{left} \bullet \textsf{hd} \bullet \textsf{right} \rightarrowtail \{\textsf{hd}\}$$

The first rule, push, lets us rewrite the string (Δ_L hd left Δ_R) to the string (Δ_L left hd Δ_R). The second rule, pop, lets us rewrite the string (Δ_L left hd right Δ_R) to the string (Δ_L hd Δ_R). The distinguishing feature of these rewriting rules is that they are *local* – they do not mention the entire stack or the entire string, just the relevant fragment of the beginning of the string and the top of the stack.

Execution of the PDA on a particular string of tokens then consists of (1) appending the token hd to the beginning of the string, (2) repeatedly performing rewritings until no more rewrites are possible, and (3) checking to see if only a single token hd remains. Figure 11.3 presents the series of states that arise from executing the PDA on the string "(()(()))". The hd atomic proposition is underlined for emphasis.

Within an appropriately-designed ordered logical framework, it is possible to treat these sequences of rewriting rules as formal artifacts; we say that there is a *step* $\Delta_1 \rightsquigarrow \Delta_2$ if there is a single rule that allows us to rewrite the string Δ_1 to the string Δ_2, and we say that there is a *trace* $\Delta_1 \rightsquigarrow^* \Delta_2$ if it is possible to rewrite the string Δ_1 to the string Δ_2 with a series of zero or more steps. It is possible to assign proof terms to traces analogous to the proof terms we assigned to derivations [198], but we will not consider this here.

[2]We use $\{\cdot\}$ for the lax modality of SLS [198]. For our purpose here, the notation can be regarded as part of the implication syntax.

In general, Δ may have more interesting structure than the simple sequences we consider here. We will call these objects *process states* [80], as they encode the state of some evolving system (such as a push-down automata). Formally, these process states are the hypothetical contexts of ordered linear logic.

11.4.1 Execution

We will now create an ordered logical specification that encodes the search procedure from Section 11.3.2.

As in the PDA example, we will use a stack encoded as a series of propositions. For reasons of presentation, the ordering of the stack will be reversed, and so this stack will be modified by adding and removing propositions from the left-hand-side of the stack – in the PDA example, the top of the stack was its right-hand side.

The general idea is that a trace $(\mathsf{eval}(\ulcorner e \urcorner)\ \Delta \rightsquigarrow^* \mathsf{retn}(\ulcorner v \urcorner)\ \Delta)$ indicates the presence of a derivation $e \Downarrow v$, so the left-most proposition in Δ, if any, represents a continuation that spawned the evaluation of e and needs to receive a v to continue. We import expressions (terms of type exp) from the LF encoding, and define two atomic propositions $\mathsf{eval}(\ulcorner e \urcorner)$ and $\mathsf{retn}(\ulcorner v \urcorner)$, both of which take an argument of type exp.

Describing the execution behavior of the evlam rule is simple: we start in the state $\mathsf{eval}(\ulcorner \lambda x.e \urcorner)\ \Delta$. Because $\lambda x.e \Downarrow \lambda x.e$ is immediately derivable, we can step immediately to $\mathsf{retn}(\ulcorner \lambda x.e \urcorner)\ \Delta$.

evlam: eval (lam $\lambda x.\,\mathsf{E}\,x$) $\rightarrowtail$ {retn (lam $\lambda x.\,\mathsf{E}\,x$)}.

The interpretation of the evapp rule requires some extra machinery. If we are in a state $\mathsf{eval}(\ulcorner e_1\ e_2 \urcorner)\ \Delta$, then the search procedure from Section 11.3.2 indicates that we first should search for a v_1 such that $e_1 \Downarrow v_1$. In terms of our abstract machine, this translates to picking a Δ' and trying to find a trace $\mathsf{eval}(\ulcorner e_1 \urcorner)\ \Delta' \rightsquigarrow^* \mathsf{retn}(\ulcorner v_1 \urcorner)\ \Delta'$. We introduce a new atomic proposition $\mathsf{cont\_app1}(\ulcorner e_2 \urcorner)$ that stores e_2 in on the top of the continuation stack, letting $\Delta' = \mathsf{cont\_app1}(\ulcorner e_2 \urcorner)\ \Delta$, while we attempt to evaluate e_1 to a value.

evapp: eval (app E1 E2) $\rightarrowtail$ {eval E1 • cont_app1 E2)}.

If we ever complete a trace of this form:

$$\mathsf{eval}(\ulcorner e_1 \urcorner)\ \ \mathsf{cont\_app1}(\ulcorner e_2 \urcorner)\ \ \Delta \rightsquigarrow^* \mathsf{retn}(\ulcorner v_1 \urcorner)\ \ \mathsf{cont\_app1}(\ulcorner e_2 \urcorner)\ \ \Delta$$

then we know there is a proof of $e_1 \Downarrow v_1$. Once that happens, we can proceed to check that v_1 has the form $\lambda x.e$ and evaluate e_2 to a value, storing the body of the function $\lambda x.e$ in another new atomic proposition $\mathsf{cont\_app2}(\lambda x.\ulcorner e \urcorner)$. Note that this slight abuse of notation indicates that $\ulcorner e \urcorner$ is the encoding of e where x can occur free; the domain of cont_app2 is exp $\rightarrow$ exp.

evapp1: retn (lam $\lambda x.\,\mathsf{E}\,x$) • cont_app1 E2
$\rightarrowtail$ {eval E2 • cont_app2 ($\lambda x.\,\mathsf{E}\,x$)}.

Finally, once a value v_2 returns to the left of the proposition $\mathsf{cont\_app2}(\lambda x.\ulcorner e \urcorner)$, we know that in order to prove $e_1\ e_2 \Downarrow v$ it suffices to prove $e[v_2/x] \Downarrow v$.

$\mathsf{eval}(\ulcorner(\lambda x.x)(\lambda y.y\ y)\urcorner)$
$\mathsf{eval}(\ulcorner\lambda x.x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{retn}(\ulcorner\lambda x.x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{eval}(\ulcorner\lambda y.y\ y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner x\urcorner)$
$\mathsf{retn}(\ulcorner\lambda y.y\ y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner x\urcorner)$
$\mathsf{eval}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{retn}(\ulcorner\lambda y.y\ y\urcorner)$

Figure 11.4: Terminating evaluation of $(\lambda x.x)(\lambda y.y\ y)$

$\mathsf{eval}(\ulcorner(\lambda x.x\ x)(\lambda y.y\ y)\urcorner)$
$\mathsf{eval}(\ulcorner\lambda x.x\ x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{retn}(\ulcorner\lambda x.x\ x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{eval}(\ulcorner\lambda y.y\ y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner x\ x\urcorner)$
$\mathsf{retn}(\ulcorner\lambda y.y\ y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner x\ x\urcorner)$
$\mathsf{eval}(\ulcorner(\lambda y.y\ y)(\lambda y.y\ y)\urcorner)$
...

Figure 11.5: Nonterminating evaluation of $(\lambda x.x\ x)(\lambda y.y\ y)$

$\mathsf{eval}(\ulcorner(\lambda x.\mathsf{junk}\,x)(\lambda y.y)\urcorner)$
$\mathsf{eval}(\ulcorner\lambda x.\mathsf{junk}\,x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\urcorner)$
$\mathsf{retn}(\ulcorner\lambda x.\mathsf{junk}\,x\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\ y\urcorner)$
$\mathsf{eval}(\ulcorner\lambda y.y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner\mathsf{junk}\,x\urcorner)$
$\mathsf{retn}(\ulcorner\lambda y.y\urcorner)\quad \mathsf{cont\_app2}(\lambda x.\ulcorner\mathsf{junk}\,x\urcorner)$
$\mathsf{eval}(\ulcorner\mathsf{junk}(\lambda y.y)\urcorner)$
$\mathsf{eval}(\ulcorner\mathsf{junk}\urcorner)\quad \mathsf{cont\_app1}(\ulcorner\lambda y.y\urcorner)$

Figure 11.6: Evaluation of $(\lambda x.\mathsf{junk}\,x)(\lambda y.y)$, which goes wrong

evapp2: retn V2 • cont_app2 ($\lambda x.$ E x) $\rightarrowtail$ {eval (E V2)}.

This completes our description of the search procedure from Section 11.3.2 as an ordered logical specification. Examples of this search procedure encoding evaluations that succeed, diverge, and go wrong are shown, respectively, in Figure 11.4, Figure 11.5, and Figure 11.6. In Section 11.5, we see that this specific abstract machine can be derived from the LF encoding of the natural semantics by a general compilation procedure from Horn clause logic programs to ordered logical specifications.

11.4.2 Encoding

Now that we have described an operational semantics for proof search as an ordered logical specification, we can observe that this ordered logical specification adequately encodes the abstract machine semantics in Figure 11.2, following the methodology used by Cervesato et al. [42] and Schack-Nielsen [183].

$$\begin{aligned}
\ulcorner k \triangleright e \urcorner &= \mathsf{eval}(\ulcorner e \urcorner)\ \ulcorner k \urcorner \\
\ulcorner k \triangleleft v \urcorner &= \mathsf{retn}(\ulcorner v \urcorner)\ \ulcorner k \urcorner \\
\ulcorner \mathsf{halt} \urcorner &= \cdot \\
\ulcorner k; \Box\, e_2 \urcorner &= \mathsf{cont_app1}(\ulcorner e_2 \urcorner)\ \ulcorner k \urcorner \\
\ulcorner k; (\lambda x.e)\, \Box \urcorner &= \mathsf{cont_app2}(\lambda x.\ulcorner e \urcorner)\ \ulcorner k \urcorner
\end{aligned}$$

Figure 11.7: Encoding of abstract machine states

We are trying to encode states of our abstract machine as process states in ordered logic. This encoding is defined by the functions $\ulcorner s \urcorner$ and $\ulcorner k \urcorner$ given in Figure 11.7, where s is an abstract machine state, either $k \triangleright e$ or $k \triangleleft v$. (Returning a value to $k = \mathsf{halt}$ represents a successfully completed computation.)

The strategy that we present here does not establish a one-to-one correspondence between abstract machines and process states, as we can talk about process states such as:

$$\mathsf{cont\_app2}(\lambda x.\ulcorner x \urcorner)\ \ \mathsf{eval}(\ulcorner \lambda x.x \urcorner)\ \ \mathsf{cont\_app2}(\lambda x.\ulcorner x \urcorner)$$

that do not correspond correctly to abstract-machine states. As a result, the statement of adequacy must be slightly different.[3]

Theorem 128 (Preservation of adequacy for the abstract machine)**.** *If* $\ulcorner s \urcorner \rightsquigarrow \Delta$*, then there exists a unique* s' *such that* $\ulcorner s' \urcorner = \Delta$*.*

Proof. By case analysis on all possible transitions under the rules in Section 11.4.1. □

Theorem 129 (Adequacy for the abstract machine)**.**
$\ulcorner s \urcorner \rightsquigarrow \ulcorner s' \urcorner$ *iff* $s \mapsto s'$*.*

Proof. Both directions are established by case analysis: the reverse direction by case analysis on the definition in Figure 11.2, and the forward direction by case analysis on all possible transitions under the rules in Section 11.4.1. □

11.4.3 Correctness

Just as the ordered logical specification in Section 11.4.1 can be derived from the natural-semantics specification in Section 11.3.1 by the compilation procedure for logic programs given in Section 11.5, the correctness properties in this section follow from the correctness of that compilation procedure, along with the relevant adequacy theorems.

For any pair of a natural semantics and an abstract machine we need two adequacy theorems: the adequacy theorem that mediates between the natural semantics and the LF specification, and the adequacy theorem that mediates between the ordered logical specification and the abstract machine.

Theorem 130 (Success)**.** $e \Downarrow v$ *if and only if* $\mathsf{halt} \triangleright e \mapsto^* \mathsf{halt} \triangleleft v$*.*

[3] An alternate approach, which allows for a one-to-one correspondence between abstract machine states and process states, is described in [198, Chapter 4]. This is beyond the scope of this paper, however.

Proof. For the forward direction, adequacy (Theorem 127) gives us a term N such that $\cdot \vdash N : \mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner$, and completeness of the compilation procedure (Theorem 134 from Section 11.5.1 below) gives us a trace $\mathsf{eval}(\ulcorner e \urcorner) \rightsquigarrow^* \mathsf{retn}(\ulcorner v \urcorner)$. Then, adequacy (Theorem 129) gives us an abstract-machine trace $\mathsf{halt} \triangleright e \mapsto^* \mathsf{halt} \triangleleft v$.

The backward direction is analogous, but using soundness of the compilation procedure (Theorem 133 from Section 11.5.1 below) instead of completeness. □

From the successful execution of the ordered logical specification, we can therefore conclude that a proof in the natural semantics exists. The converse is not true in general. If a specification is nondeterministic (as discussed later in Section 11.6) it could be possible for the non-backtracking execution of the ordered logical specification to diverge and/or get stuck when nondeterminism is resolved in one way but to succeed when nondeterminism is resolved in a different way.

Given that our particular running example is, in fact, deterministic, we can make some slightly stronger statements about the relationship between our natural semantics and divergence and stuck states in the abstract machine.

Theorem 131 (Divergence)**.** *If* $\mathsf{halt} \triangleright e \mapsto^\infty$ *then there is no* v *such that* $e \Downarrow v$.

Theorem 132 (Going wrong)**.** *If* $\mathsf{halt} \triangleright e \mapsto^* s \not\mapsto$*, then either* $s = \mathsf{halt} \triangleleft v$ *or there is no* v *such that* $e \Downarrow v$.

Proof. In both cases, the proof is by refutation: if there were a derivation of $e \Downarrow v$, then we would have an abstract machine trace $\mathsf{halt} \triangleright e \mapsto^* \mathsf{halt} \triangleright x$ by Theorem 130; this trace must differ from the trace that diverges or goes wrong at some point, which contradicts the determinism of CBV evaluation. □

11.5 The logical correspondence

In this section, we will consider general transformations from Horn-clause logic programs to ordered logical specifications, and we will consider other transformations from ordered logical specifications to other ordered logical specifications. The composition of these transformations allows us to derive our abstract machine specification directly from our natural-semantics specification.

We begin by giving a more formal summary of both of our primary objects of interest: LF signatures on one hand, and ordered logical specifications on the other. We make certain issues, such as universal quantification, more explicit. Nevertheless, this is intended only to be a high-level overview and not a complete introduction.

LF signatures An LF signature Σ_{LF} is defined as in [109], but we will deal with a restricted set of signatures here. First, because LF merrily conflates types that are intended to be interpreted as first-order data and types that are intended to be interpreted as logic programs, we will disambiguate the two by distinguishing the atomic LF types intended to be relevant to logic programming. We will call the types relevant to logic programming the *propositions*, and call the non-proposition types the *pure LF* types; pure LF terms are just terms of pure LF type. Propositions depend on pure LF types, but pure LF types must be independent of propositions.[4]

[4]This restriction is just a use-case of the theory of subordination [212].

Atomic propositions Q have the form $(\mathsf{a}\, N_1\, N_2)$, where N_1 and N_2 must have some particular LF types A_1 and A_2, respectively.

Constructors of propositions are called *rules* and have the form

$$\mathsf{c}\colon\ \forall x_1{:}A_1.\ \ldots\ \forall x_n{:}A_n.\, Q_k \rightarrow \ldots \rightarrow Q_1 \rightarrow \mathsf{a}\, N_0^-\, N_{k+1}^+$$

where the A_i are all pure LF types and where $Q_i = \mathsf{a}_i\, N_i^+\, N_i^-$.[5] The subscripts indicate the order in which the logic programming interpretation considers each term. The superscripts indicate the operational interpretation of atomic propositions as nondeterministic partial functions: plus designates input, and minus designates output. The rule that we would write in Twelf as:

$$\mathsf{rule}\colon\ \mathsf{a}\, X\, Z \leftarrow \mathsf{b}\, X\, Y \leftarrow \mathsf{c}\, Y\, Z$$

is reconstructed by Twelf into an LF signature declaration:

$$\mathsf{rule}\colon\ \forall x{:}A_1.\, \forall y{:}A_2.\, \forall z{:}A_3.\, \mathsf{c}\, y\, z \rightarrow \mathsf{b}\, x\, y \rightarrow \mathsf{a}\, x\, z$$

for some pure LF types A_1, A_2, and A_3.

Rules must be well-moded which allows us to interpret $\mathsf{a}\, N_1\, N_2$ as a partial function from terms N_1 of type A_1 to terms N_2 of type A_2. In a rule, such as c above, each of the variables x must appear in one of the $k+1$ atomic propositions (that is, in one of the Q_i or in the head $(\mathsf{a}\, N_0^-\, N_{k+1}^+)$). Furthermore, whenever a variable x_i occurs in N_i^+ (or occurs in a non-strict[6] position in N_i^-), there must be a N_j^- such that $j < i$ where x_i occurs strictly.

Ordered logical specifications An ordered logical specification Σ_O is similar to an LF signature; all the pure LF types and constants of an LF signature can also be contained in an ordered logical specification. An ordered logical specification also declares ordered propositions $P = \mathsf{p}\, N$, where N has some particular pure LF type A, and *clauses* which are defined in ordered logic according to the following BNF grammar:

$$\begin{array}{lcll} \text{Clauses} & D & ::= & \forall x{:}A.D \mid S \rightarrowtail \{H\} \\ \text{Heads} & H & ::= & P \mid H \bullet H \mid D \\ \text{Goals} & S & ::= & P \mid S \bullet S \end{array}$$

Implicitly quantified variables work the same in ordered logical specifications as they did in LF: they are universally quantified on the outside. When we do write universal quantifiers explicitly in the examples below, we will omit the pure LF type annotation A, writing $\forall x.D$ instead of $\forall x{:}A.D$, as the type will always be clear from the context.

This language of ordered logical specifications has one significant addition relative to the language discussed in Section 11.4 and in [165], justified by the development in [198, Chapter 4]: we allow *nested* specifications where clauses appear in the heads of other clauses. These clauses can then appear in the context: a clause $P \rightarrowtail \{P' \bullet D\}$, when it appears in the context, is a token that cannot satisfy any subgoal. Instead, it can only react to the presence of the ordered proposition P *immediately to the clause's left*. That is, the process state $\Delta_L\ P\ (P \rightarrowtail \{P' \bullet D\})\ \Delta_R$ can rewrite to the process state $\Delta_L\ P'\ D\ \Delta_R$. Figure 11.8 gives another example evolution.

[5] LF has one proposition, $\Pi x{:}A.A'$, that acts as both a universal quantifier $\forall x{:}A.A'$ (when x is free in A') and as implication $A \rightarrow A'$ (when x is not free in A').

[6] Strictness is defined in [163]; it is important in general, but is not relevant to our examples, so we omit a precise definition in this paper.

$$\mathsf{p2(c)}\ \ \mathsf{p1(c)}\ \ (\forall x.\,\mathsf{p1}(x) \rightarrowtail \{\mathsf{p2}(x) \rightarrowtail \{\mathsf{p3}(x)\}\})\ \ (\mathsf{p3(c)} \rightarrowtail \{\mathsf{p4}\})$$
$$\mathsf{p2(c)}\ \ (\mathsf{p2(c)} \rightarrowtail \{\mathsf{p3(c)}\})\ \ (\mathsf{p3(c)} \rightarrowtail \{\mathsf{p4}\})$$
$$\mathsf{p3(c)}\ \ (\mathsf{p3(c)} \rightarrowtail \{\mathsf{p4}\})$$
$$\mathsf{p4}$$

Figure 11.8: Evolution of a process state containing nested rules

11.5.1 Operationalization

We translate a LF signature Σ_{LF} into an ordered logical specification Σ_{O} as follows. The pure LF types and constants remain unchanged. For each defined proposition a that takes an input argument of type A_1 and an output argument of type A_2, we define two ordered propositions in the ordered logical specification: eval_a, which takes an argument of type A_1, and retn_a, which takes an argument of type A_2.

Every rule in the LF signature gets translated into a clause in the ordered logical specification. Given a rule:

$$\mathsf{c}:\ \forall x_1{:}A_1.\ldots\forall x_n{:}A_n.\,Q_k \rightarrow \ldots \rightarrow Q_1 \rightarrow \mathsf{a}\,N_0^-\,N_{k+1}^+$$

where each $Q_i = \mathsf{a}_i\,N_i^+\,N_i^-$, let $\forall\overline{xs_i}.D$ for $0 \leq i \leq k$ be a shorthand for writing a universal quantifier for each variable free in N_i^- that is not free in N_j^- for $j < i$. Then, the corresponding clause in the ordered logical specification is:

$$\mathsf{c}:\ \forall\overline{xs_0}.\,\mathsf{eval\_a}\,N_0^- \rightarrowtail \{[\![Q_1,\ldots,Q_k]\!](\mathsf{a},N_{k+1}^+,\mathsf{id})\}$$

where the function $[\![Args]\!](\mathsf{a},N_{k+1}^+,\sigma)$ is defined as follows:

$$[\![\,]\!](\mathsf{a},N_{k+1}^+,\sigma) = \mathsf{retn\_a}(N_{k+1}\sigma)$$

$$[\![(\mathsf{a}_i\,N_i^+\,N_i^-),Args]\!](\mathsf{a},N_{k+1}^+,\sigma) =$$
$$\mathsf{eval\_a}_i(N_i^+\sigma)\bullet$$
$$(\forall\overline{xs_i}.\,\mathsf{retn\_a}_i(N_i^-\sigma) \rightarrowtail \{[\![Args]\!](\mathsf{a},N_{k+1}^+,\sigma)\})$$

Substitutions are partial, and id is just the empty substitution that leaves terms unchanged. If we were instead describing total substitutions, when we write σ in the recursive call above, we would instead need to write $(\sigma,\overline{xs_i}/\overline{xs_i})$ to emphasize that the substitution accepts all the variables in $\overline{xs_i}$ and returns them unchanged. Substitutions other than the identity substitution only come up as intermediate stages in the correctness proof.

The following is the output of this transformation on our running example, except that we leave outermost universal quantifiers implicit and just write eval and retn instead of eval_ev and retn_ev.

```
evlam: eval (lam λx. E x) ↣ {retn (lam λx. E x)}.
evapp: eval (app E1 E2)
         ↣ {eval E1 •
              (∀e. retn (lam λx. e x)
                ↣ {eval E2 •
                     (∀v2. retn v2
                       ↣ {eval (e v2) •
                            (∀v. retn v ↣ {retn v})})})}.
```

Tail-recursion optimization

The rule evapp under the operationalization transformation contains an innermost clause, $\forall v.\, \mathsf{retn}\, v \rightarrowtail \{\mathsf{retn}\, v\}$, that is essentially just the identity. It corresponds to the last stage in our proof search procedure where we searched for a v such that $e[v_2/x] \Downarrow v$ – upon success, we could successfully return a v such that $e_1\, e_2 \Downarrow v$.

In work on the functional correspondence, it would be possible to observe at this point that this last call is tail-recursive. We can do the same in our logical correspondence, modifying this rule to be explicitly tail-recursive.

$$\begin{array}{l}\mathsf{evapp}\colon \mathsf{eval}\ (\mathsf{app}\ \mathsf{E1}\ \mathsf{E2})\\ \quad \rightarrowtail \{\mathsf{eval}\ \mathsf{E1} \bullet\\ \qquad (\forall e.\ \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e\, x)\\ \qquad\ \rightarrowtail \{\mathsf{eval}\ \mathsf{E2} \bullet\\ \qquad\qquad (\forall v_2.\ \mathsf{retn}\ v_2 \rightarrowtail \{\mathsf{eval}\ (e\, v_2)\})\})\}.\end{array}$$

Such a modification on general ordered logical specifications is complete but is not sound in general, because these identity-like propositions can get in the way of other possible transitions. For instance, the following cannot transition to d:

$$\mathsf{a}\ \ (\mathsf{a} \rightarrowtail \{\mathsf{b} \bullet (\mathsf{c} \rightarrowtail \{\mathsf{c}\})\})\ \ (\mathsf{b} \rightarrowtail \{\mathsf{d}\}) \not\rightsquigarrow^* \mathsf{d}$$

whereas if we "tail-call optimize" the $(\mathsf{c} \rightarrowtail \{\mathsf{c}\})$ proposition, we can transition all the way to d:

$$\mathsf{a}\ \ (\mathsf{a} \rightarrowtail \{\mathsf{b}\})\ \ (\mathsf{b} \rightarrowtail \{\mathsf{d}\}) \rightsquigarrow \mathsf{b}\ \ (\mathsf{b} \rightarrowtail \{\mathsf{d}\}) \rightsquigarrow \mathsf{d}$$

Essentially the same behavior can arise in ordered logical specifications that come from our compilation procedure, though the problem is at the level of terms, not types. The clause that results from compiling the rule $\mathsf{a}\,\mathsf{X}\,\mathsf{false} \leftarrow \mathsf{b}\,\mathsf{X}\,\mathsf{Y} \leftarrow \mathsf{a}\,\mathsf{Y}\,\mathsf{false}$ contains the head $(\mathsf{eval\_a}(y) \bullet (\mathsf{retn\_a}(\mathsf{false}) \rightarrowtail \{\mathsf{retn\_a}(\mathsf{false})\}))$, which must be preserved so that the search fails if the subgoal returns true instead of false. It is sound to require that only heads of the form $(\forall \overline{x}.\mathsf{retn\_a}(x) \rightarrowtail \{\mathsf{retn\_a}(x)\})$ be optimized away, although a more general condition is that $(\forall \overline{xs}_n.\mathsf{retn\_a}(t) \rightarrowtail \{\mathsf{retn\_a}(t)\})$ can be optimized as long as, for any well-typed term t', there exists a substitution σ such that $t' = t\sigma$.

Tail-call optimization can be represented by extending the definition of the operationalization function as follows:

$$\begin{array}{l}[\![(\mathsf{a}_n\, N_n^+\, N_n^-)]\!](\mathsf{a}, N_{k+1}^+, \sigma) = \mathsf{eval\_a}(N_n^+\sigma)\\ \text{If } \mathsf{retn\_a}_n = \mathsf{retn\_a} \text{ and } N_n^- = N_{k+1}^+ = x, \text{ where } x \in \overline{xs_n}.\end{array}$$

Note that this branch of the definition overlaps with the second branch in the previous definition. This indicates that tail-recursion is an optimization that can be applied eagerly or not, as desired.

Parallel evaluation

Ordered logical specifications can be used to represent parallel computations: if there is more than one ordered proposition $\mathsf{eval}(\ulcorner e \urcorner)$ or $\mathsf{retn}(\ulcorner v \urcorner)$ in the process state that can both evolve by rewriting independent portions of the process state, then we can interpret those evolutions as happening in either order or simultaneously in parallel [165, 218].

We can characterize parallel evaluation by further generalizing the operationalization transformation to allow for parallel proof search in cases where the input of a rule's premise (the first argument N_i^+) doesn't depend on the output of the previous premise (the second argument N_{i-1}^-). In conjunction with tail-recursion optimization, parallel operationalization transforms the rule evapp into the following clause:

$$\begin{array}{l}\mathsf{evapp}\colon \mathsf{eval}\ (\mathsf{app}\ \mathsf{E1}\ \mathsf{E2})\\ \quad \rightarrowtail \{\mathsf{eval}\ \mathsf{E1} \bullet \mathsf{eval}\ \mathsf{E2} \bullet\\ \quad\quad (\forall e.\, \forall v_2.\, \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e\, x) \bullet \mathsf{retn}\ v_2\\ \quad\quad\quad \rightarrowtail \{\mathsf{eval}\ (e\, v_2)\})\}.\end{array}$$

This ordered logical specification can now represent any arbitrary interleaving of the steps that evaluate E1 and E2 to values.

We can adapt our translation to allow for parallelism by replacing the inductive case translation with the following:

$$\begin{array}{l}[\![(\mathsf{a}_i\, N_i^+\, N_i^-), \ldots, (\mathsf{a}_j\, N_j^+\, N_j^-), \mathit{Args}]\!](\mathsf{a}, N_{k+1}^+, \sigma) =\\ \mathsf{eval\_a}_i(N_i^+\sigma) \bullet \cdots \bullet \mathsf{eval\_a}_j(N_j^+\sigma) \bullet\\ (\forall \overline{xs_i}. \ldots \forall \overline{xs_j}.\, \mathsf{retn\_a}_i(N_i^-\sigma) \bullet \cdots \bullet \mathsf{retn\_a}_j(N_j^-\sigma)\\ \rightarrowtail \{[\![\mathit{Args}]\!](\mathsf{a}, N_{k+1}^+, \sigma)\})\\ \text{If } \mathit{FV}(N_k^+) \cap (\overline{xs_i} \cup \cdots \cup \overline{xs_j}) = \emptyset \text{ for } i \le k \le j.\end{array}$$

The previous inductive case is simply the special case for $i = j$, in which case the side condition is trivially satisfied.

Correctness

The correctness proofs for the compilation of LF signatures into abstract machines in an ordered logical specification are structured much like proofs of equivalence between a natural semantics and an abstract machine and can be found in the first author's thesis [198, Chapter 6].

Theorem 133 (Soundness)**.** *Given a trace* $\mathsf{eval\_a}(N^+) \rightsquigarrow^* \mathsf{retn\_a}(N^-)$ *in an ordered logical specification compiled from an LF signature, there exists an* M *such that* $\cdot \vdash M : \mathsf{a}\, N^+\, N^-$ *in the LF signature.*

Theorem 134 (Completeness)**.** *If* $\cdot \vdash M : \mathsf{a}\, N^+\, N^-$ *in an LF signature, then* $\mathsf{eval\_a}\, N^+\, \Delta \rightsquigarrow^* \mathsf{retn\_a}\, N^-\, \Delta$ *in the compiled ordered logical specification.*

The soundness and completeness theorems give a very strong connection between logic programs written as LF signatures and the ordered logical specifications that implement non-backtracking proof search.

11.5.2 Defunctionalization

The final modification we need to make in order to get the SSOS specification from Section 11.4 is defunctionalization, which takes ordered logical specifications with deeply nested clauses like evapp and produces flat specifications where every clause has the form $\forall \overline{x}.\, P_1 \bullet \ldots \bullet P_n \rightarrowtail \{P_1' \bullet \ldots \bullet P_n'\}$.

Whenever we have a clause like $(\forall x.\, \mathsf{p1}(x) \rightarrowtail \{\mathsf{p2}(\mathsf{f}(\mathsf{h}\, x)\, \mathsf{d})\})$ in the process state, we can replace it with a fresh atomic proposition newprop as long as we simultaneously

extend the signature with a new clause ($\forall x.\, \mathsf{p1}(x) \bullet \mathsf{newprop} \rightarrowtail \{\mathsf{p2}(\mathsf{f}(\mathsf{h}\, x)\, \mathsf{d})\}$). The key to defunctionalization is to observe that the revised specification can evolve in lock-step with the original one. Whenever the in-context clause can step by consuming a proposition $\mathsf{p1}(N)$ (for some N) to its left, the new clause can take an analogous step by consuming $\mathsf{p1}(N)$ to the left of newprop in the process state. The reverse holds as well.

We can furthermore give the fresh atomic proposition arguments: if for example $\forall x.\mathsf{p1} \bullet \mathsf{newprop}(x) \rightarrowtail \{\mathsf{p2}(x)\}$ appears in an ordered logical specification, then an atomic proposition $\mathsf{newprop}(N)$, for some fresh atomic proposition newprop and term N, will behave the same way as the clause ($\mathsf{p1} \rightarrowtail \{\mathsf{p2}(N)\}$) would behave in the same place.

These observations allow us to defunctionalize the nested evapp from Section 11.5.1 (tail-recursion optimized, but not parallel) in two steps. For example, the nested clause ($\forall e.\, \mathsf{retn}(\mathsf{lam}\, \lambda x.\, e\, x) \rightarrowtail \ldots$) has E2 free, so we introduce a fresh atomic proposition cont_app1 that stores the free variable, here E2 of type exp:

$$\begin{array}{l}\mathsf{evapp}\colon \mathsf{eval}\ (\mathsf{app}\ \mathsf{E1}\ \mathsf{E2}) \rightarrowtail \{\mathsf{eval}\ \mathsf{E1} \bullet \mathsf{cont\_app1}\ \mathsf{E2}\}.\\ \mathsf{evapp1}\colon \mathsf{retn}\ (\lambda x.\, \mathsf{E}\, x) \bullet \mathsf{cont\_app1}\ \mathsf{E1}\\ \qquad\qquad \rightarrowtail \{\mathsf{eval}\ \mathsf{E2} \bullet (\forall v_2.\, \mathsf{retn}\ v_2 \rightarrowtail \{\mathsf{eval}\ (\mathsf{E}\, v_2)\})\}.\end{array}$$

The remaining nested clause has only E free, so we defunctionalize this remaining clause by introducing a fresh atomic proposition cont_app2 that stores the free variable, here E of type $\mathsf{exp} \rightarrow \mathsf{exp}$:

$$\begin{array}{l}\mathsf{evapp}\colon \mathsf{eval}\ (\mathsf{app}\ \mathsf{E1}\ \mathsf{E2}) \rightarrowtail \{\mathsf{eval}\ \mathsf{E1} \bullet \mathsf{cont\_app1}\ \mathsf{E2}\}.\\ \mathsf{evapp1}\colon \mathsf{retn}\ (\lambda x.\, \mathsf{E}\, x) \bullet \mathsf{cont\_app1}\ \mathsf{E1}\\ \qquad\qquad \rightarrowtail \{\mathsf{eval}\ \mathsf{E2} \bullet \mathsf{cont\_app2}\ (\lambda x.\, \mathsf{E}\, x)\}.\\ \mathsf{evapp2}\colon \mathsf{retn}\ \mathsf{V2} \bullet \mathsf{cont\_app2}\ (\lambda x.\, \mathsf{E}\, x) \rightarrowtail \{\mathsf{eval}\ (\mathsf{E}\ \mathsf{V2})\}.\end{array}$$

The result is precisely the abstract-machine specification as defined in Section 11.4. The correctness follows by a straightforward lock-step bisimulation [198, Chapter 6]

11.5.3 Implementation

We have implemented[7] the operationalization and defunctionalization transformations (with tail-recursion optimization but without parallel evaluation, though this would be a minor extension). Rather than a transformation from LF specifications to ordered logical specifications, the transformations are implemented within a framework, SLS [198], that conservatively extends both LF and the ordered logical framework from [165] in the style of CLF [218].

To evaluate the correspondence and the implementation we have encoded and transformed some existing and non-trivial semantics, including temporal logic and storeless call-by-need evaluation.

The natural semantics for storeless call-by-need evaluation is taken from Danvy et al. [77, Figure 8]. It is defined by several mutually inductive definitions and makes use of non-trivial control. The result of the logical correspondence produces a substructural operational semantics that, after a fold/unfold transformation [157], adequately represents

[7] A tarball of the implementation is submitted in the supplemental materials; a repository with the current implementation can be found at
`https://github.com/robsimmons/sls`.

the abstract machine in Figure 7 from the same paper. The storeless call-by-need evaluation implementation can be found in the source distribution under `examples/storeless-cbneed/`.

Operationalization and defunctionalization do not apply only to natural-semantics specifications, but to arbitrary Horn-clause logic programs. In course notes, Pfenning presented both an ordered logical specification and backward-chaining logic programming specification of binary addition [162]. These two specifications can be related by operationalization, defunctionalization, and a fold/unfold transformation similar to the one needed in the storeless call-by-need example; the results can be found in the source distribution under `examples/addition/`.

The natural semantics for Davies's Mini-ML$^{\bigcirc}$ language extends a functional Mini-ML core language with support for staged computation as a Curry-Howard interpretation of temporal logic [78]. There are two judgments in the natural-semantics specification of Mini-ML$^{\bigcirc}$: the usual judgment $e \Downarrow v$ that evaluates expressions at time 0 and a judgment $e \Downarrow^n e'$ that searches through an expression that will be evaluated at time n looking for evaluation that must be done now, at time 0. This example illustrates how operationalization can be applied partially to a specification. In this case, only evaluation at time 0 is operationalized. The implementation can be found in the source distribution under `examples/temporal/`.

11.6 Nondeterminism

A major theme of this paper is that the operationalization transformation allows us to reason formally about what it means for evaluation in a natural-semantics specification to diverge or go wrong. The evaluation of e diverges if for all traces $\mathsf{eval}(\ulcorner e \urcorner) \rightsquigarrow^* \Delta$, there is a Δ' such that $\Delta \rightsquigarrow \Delta'$. The evaluation of e goes wrong if for some trace $\mathsf{eval}(\ulcorner e \urcorner) \rightsquigarrow^* \Delta$, there is no v such that $\Delta = \mathsf{retn}(\ulcorner v \urcorner)$, and there is no Δ' such that $\Delta \rightsquigarrow \Delta'$. In this way, operationalization gives us an accessible notion of what it means to prove progress and preservation for a natural-semantics specification.

While this interpretation is the one we want, the use of this interpretation requires that we change the way we write some natural-semantics specifications. This is the usual natural semantics of Boolean values and if-statements:

$$\frac{}{\mathsf{tt} \Downarrow \mathsf{tt}} \qquad \frac{}{\mathsf{ff} \Downarrow \mathsf{ff}} \qquad \frac{e \Downarrow \mathsf{tt} \quad e_t \Downarrow v}{\mathsf{if}\, e\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f \Downarrow v} \qquad \frac{e \Downarrow \mathsf{ff} \quad e_f \Downarrow v}{\mathsf{if}\, e\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f \Downarrow v}$$

This natural semantics describes a deterministic language, but only when we think about complete proofs: for each e there is at most one v such that $e \Downarrow v$. At the level of proof search, when we consider an expression $\mathsf{if}\, e\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f$ there are two rules that apply. One rule predicts that e evaluates to true, the other predicts that e evaluates to false, and only one rule can be considered first. A backtracking logic-programming interpretation of these rules will recover from a mistaken branch prediction and attempt to apply the other rule; this reevaluates the scrutinee e but is otherwise harmless.

The operationalization transformation reflects a non-backtracking interpretation of proof search. Therefore, if we operationalize these rules directly, it is always possible to find a trace starting from $\mathsf{eval}(\ulcorner \mathsf{if}\, e\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f \urcorner)$ that is not a final return value and cannot be extended, which was our criteria for going wrong. This appears to be a problem; however, for languages that are actually deterministic, it is possible to rewrite the natural-semantics specification so that the result of operationalization is also deterministic. We

can revise the specification above with the introduction of a new judgment $(v', e_t, e_f) \Downarrow^? v$ that picks the correct branch of a case statement:

$$\frac{e \Downarrow v' \quad (v', e_t, e_f) \Downarrow^? v}{\mathsf{if}\, e\, \mathsf{then}\, e_t\, \mathsf{else}\, e_f \Downarrow v} \qquad \frac{e_t \Downarrow v}{(\mathsf{tt}, e_t, e_f) \Downarrow^? v} \qquad \frac{e_f \Downarrow v}{(\mathsf{ff}, e_t, e_f) \Downarrow^? v}$$

This specification is equivalent to the previous one at the level of complete derivations. This revision can be seen as an instance of the *factoring* transformation that has been expressed by Poswolsky and Schürmann as a transformation on functional programs in a variant of the Delphin programming language [174]. Furthermore, this style of natural semantics independently arises through the functional correspondence as illustrated by, e.g., the call-by-need natural semantics implemented in Section 11.5.3.

Fundamentally, the issue is that natural semantics do not fully specify the operational behavior of specifications, and different operational interpretations can change the meaning of natural-semantics specifications. This phenomena was observed in the early work on natural semantics by Clement et al., where they questioned when it is appropriate to assign a relational interpretation (backtracking proof search) or a functional interpretation (non-backtracking proof search) to judgments [45, Section 7]. The choice makes a difference in how we interpret natural-semantics specifications. Take the following nondeterministic specification, for instance:

$$\frac{e_1 \Downarrow v}{e_1 \circledast e_2 \Downarrow v} \qquad \frac{e_2 \Downarrow v}{e_1 \circledast e_2 \Downarrow v}$$

Under the non-backtracking or functional interpretation provided by operationalization, this represents a nondeterministic choice that is resolved at runtime and never reconsidered: if either choice goes wrong, then the entire evaluation might go wrong. Under the relational or backtracking interpretation, this operator represents *angelic choice*, in which nondeterminism must be resolved in a way that avoids stuck computations. Therefore, if we evaluate $\mathsf{junk} \circledast v$, the computation can only evaluate to v. The option of getting stuck can only be represented by introducing an additional getting stuck judgment, it cannot be represented by the absence of a derivation $\mathsf{junk} \circledast v \Downarrow v$, as such a derivation exists.

Incidentally, the coinductive interpretation of Leroy and Grall is closely connected with the backtracking interpretation. Therefore, the safety theorem they prove – if e has type τ then either $e \Downarrow v$ or $e \Downarrow^\infty$ – will hold for well-typed expressions that go wrong, but only under certain resolutions of nondeterministic choice. This suggests that the operationalization transformation, and non-backtracking search in general, is a better interpretation if reasoning about language safety, especially in the presence of nondeterminism.

11.7 Perspectives

In this section, we put into perspective the logical correspondence with a number of potential applications. In many cases these are straightforward applications of preexisting techniques.

11.7.1 Safety theorems for typed languages

The informal statement of safety for a natural semantics in our setting is similar to the statement of safety in coinductive big-step operational semantics: if e has type τ, then the

proof search procedure for a v such that $e \Downarrow v$ will either diverge or succeed. Our methodology allows us to state soundness with respect to the ordered logical specification: if e has type τ and $\mathsf{eval}(\ulcorner e \urcorner) \rightsquigarrow^* \Delta$, then either $\Delta = \mathsf{retn}(\ulcorner v \urcorner)$ for some v or $\Delta \rightsquigarrow \Delta'$. In previous work, the first author details a methodology for describing type safety for substructural operational semantics in ordered logic [198], and this methodology applies to the outputs of our compilation procedure. Therefore, the logical correspondence between natural semantics and abstract machines gives an approach for proving *small-step safety for big-step operational semantics*.

11.7.2 Linear destination passing style

Simmons and Pfenning describe a translation from ordered logical specifications to linear logical specifications that adds parameters, called *destinations*, to all ordered atomic propositions [199]. These destinations act as abstract pointers and replace information which, in ordered logic, is represented implicitly by the structure of the context. These destinations are semantically relevant in substructural operational semantics specifications of first-class continuations.

Just as substructural operational semantics in ordered logic predated the logical correspondence we present here, the destination-passing style predates the transformation that gives rise to destinations [42]; these two transformations can be composed to relate natural semantics directly to substructural operational semantics specifications in a language, like CLF, that implements linear logical specifications. The Celf implementation of CLF, unlike the current SLS prototype, implements proof search and can therefore be used to animate substructural operational semantics specifications [184].

11.7.3 Abstracting (logical) abstract machines

Simmons and Pfenning also discuss an approximation strategy for linear logical specifications [199] that mirrors Might and Van Horn's Abstracting Abstract Machines methodology [211] in a logical setting.

The starting point for that paper's logical derivation of a control flow analysis is a substructural operational semantics much like the one we derive from the natural semantics in this work. One difference – the use of an environment semantics instead of the direct substitution of values into λ-expressions – is discussed in [165] but has not been described as a general, provably-correct program transformation. The other difference between the substructural operational semantics described in [199] and the one given here is that the former semantics did not include the tail-recursion optimization. This change was needed in order to expose more information to the control flow analysis; while the difference was difficult to motivate and explain in [199], given the logical correspondence that we have presented here, it is simple and intuitive.

Therefore, between these two papers we have very nearly described a connection between natural semantics for a call-by-value λ-calculus and a control flow analysis for the call-by-value λ-calculus entirely in terms of generally applicable, provably-correct program transformations. The only missing piece is a general connection between ordered logical specifications with substitution to ordered logical specifications using environment semantics as described in [165].

11.7.4 Partial inverses of compilation

In some respects, the transformation from a more abstract natural semantics to a more concrete abstract machine semantics is the less interesting direction; a partial inverse of our compilation function would derive a natural-semantics specification directly from an abstract machine semantics. This backward translation from ordered logical specifications to LF signatures might even expose additional opportunities for parallelism which could be utilized as discussed above in Section 11.5.1. Such a procedure would presumably be very partial. It is easy to describe the LF signatures which can be compiled to ordered logical specifications – those that are limited to Horn clause logic programs. It is not so obvious how to characterize those ordered logical specifications that can be de-compiled into an LF signature, except by characterizing them as being in the image of our compilation procedure.

It is also interesting to contemplate the natural-semantics specifications that might correspond to a backwards translation of an ordered logical specification that utilizes linear resources. Linear resources in ordered logical specifications are not bound to a particular part of the process state, and can be used to represent stateful features of programming languages like mutable storage. Presumably the natural-semantics specification corresponding to a stateful ordered logical specification would be akin to a natural semantics version of modular structural operational semantics, a formalism which gracefully accounts for stateful features in SOS-style presentations [150].

11.8 Conclusion

We have shown how to mechanically construct an abstract machine semantics from a natural semantics. Our construction is based on backward-chaining proof search and formalized as an operationalization transformation on logical specifications.

It is our thesis that this logical correspondence provides a direct, consistent, and useful account of adequate representations for operational semantics. Direct because the representations are in one-to-one correspondence with their informal counterparts, i.e., what we would typically write by hand; consistent because the representations are internally compatible with each other; and useful because the representations are independently sought after. Furthermore, as illustrated by the present work, the interpretations are applicable to a wide range of uses such as abstract interpretation and type safety.

This logical correspondence provides a means by which we can resolve the tension between different styles of operational semantics. Following the methodology of the functional correspondence, instead of committing to a particular semantic specification and style or inventing several we should rather inter-derive them. Here we have shown a logical justification of this method.

In this work, we have leveraged existing logics and methods for formalizing and manipulating semantics descriptions. Combined, these logics and methods provide a correspondence that is mechanical and of which we have given a prototype implementation. Other systems exist within this area and provide specialized support for formalizing and manipulating particular styles of semantics. Notable examples include the K Semantic Framework [182], LNgen [18], and PLT Redex [89]. To the best of our knowledge, no existing framework provides built-in support for working with several different semantics styles as done here, let alone support for automatically inter-deriving them. Future work would be to further develop SLS as a unified logic enabling across-the-board support

for formalizing, mechanizing and reasoning about semantics specifications by means of inter-derivation. It is our hope to encourage a more systematic investigation of logical correspondences in programming-language semantics, an area which has thus far been given only piecemeal treatment (as in Hannan and Miller's work [106]). One starting point is extending this logical correspondence to the multitude of small-step operational semantics treated by Biernacka and Danvy [27].

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. 48, 62

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409–470, 2000. 7

[3] Mads Sig Ager. From natural semantics to abstract machines. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, revised selected papers*, number 3573 in Lecture Notes in Computer Science, pages 245–261, Verona, Italy, August 2004. Springer. 242

[4] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006. 4

[5] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press. 47, 94, 109, 152, 159, 173, 204, 237, 239, 241

[6] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, Department of Computer Science, Aarhus University, Aarhus, Denmark, March 2003. 179

[7] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3. 47, 126, 149, 159, 162, 173, 178, 241

[8] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28. 47, 241

[9] Konrad Anton and Peter Thiemann. Towards deriving type systems and implementations for coroutines. In Kazunori Ueda, editor, *Programming Languages and Systems – 8th Asian Symposium, APLAS 2010*, number 6461 in Lecture Notes in Computer Science, pages 63–79, Shanghai, China, December 2010. Springer. 48, 62, 73

[10] Konrad Anton and Peter Thiemann. Typing coroutines. In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Trends in Functional Programming*, volume 6546 of *LNCS*, pages 16–30. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22940-4. 48, 62, 239

[11] Zena M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146(1&2):69–108, 1995. 188, 237

[12] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997. 126, 129, 130, 148, 158, 165, 177, 236, 262

[13] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3/4):207–240, 1996. 237

[14] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In Lee [135], pages 233–246. 178

[15] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In Lee [135], pages 233–246. Journal versions available as [12] and [141]. 49, 63, 129, 158, 165, 178, 210

[16] Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. Classical call-by-need and duality. In Luke Ong, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2011*, number 6690 in Lecture Notes in Computer Science, pages 27–44, Novi Sad, Serbia, June 2011. Springer. 179

[17] Zena M. Ariola, Paul Downen, Hugo Herbelin, and Alexis Nakata, Keiko Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming, 11th International Symposium, FLOPS 2012*, number 7294 in Lecture Notes in Computer Science, pages 32–46, Kobe, Japan, May 2012. Springer. 4, 62, 159, 239

[18] Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, University of Pennsylvania Dept. of Computer and Information Science, June 2010. 258

[19] Thibaut Balabonski. A unified approach to fully lazy sharing. In John Field and Michael Hicks, editors, *Proceedings of the Thirty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 469–480, Philadelphia, PA, USA, January 2012. ACM Press. 160

[20] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, January 2004. ACM Press. 148

[21] Anindya Banerjee and David A. Schmidt. Stackability in the typed call-by-value lambda calculus. *Science of Computer Programming*, 31(1):47–73, 1998. 179

[22] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984. 126, 211

[23] Henk P. Barendregt. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, volume B, chapter 7, pages 321–364. Elsevier and The MIT Press, 1990. 190

[24] Henk P. Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In Jaco de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, number 259 in Lecture Notes in Computer Science, pages 141–158, Eindhoven, The Netherlands, June 1987. Springer-Verlag. 4, 159, 177, 187, 188, 200, 211, 223, 224, 237

[25] Michael Beeler, R. William Gosper, and Rich Schroeppel. HAKMEM. AI Memo 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1972. `http://home.pipeline.com/~hbaker1/hakmem/`. 158

[26] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, January 2006. 4

[27] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375 (1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18. 62, 73, 94, 149, 159, 163, 170, 178, 241, 259

[28] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007. Article #6. Extended version available as the research report BRICS RS-06-3. 62, 73, 76, 78, 177, 201, 218

[29] Małgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In Palsberg [156], pages 186–206. 62, 189, 201

[30] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04). 47, 62

[31] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2005. 4

[32] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer. 47, 241

[33] Stefan Blom. *Term Graph Rewriting – Syntax and Semantics*. PhD thesis, Institute for Programming Research and Algorithmics, Vrije Universiteit, Amsterdam, The Netherlands, March 2001. 211, 224, 236, 237

[34] Roel Bloo and Kristoffer Høgsbro Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN-95: Computer Science in the Netherlands*, pages 62–72, 1995. 130

[35] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press. 269, 270

[36] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991. 128, 220

[37] Arie de Bruin and Erik P. de Vink. Continuation semantics for Prolog with cut. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 351 in Lecture Notes in Computer Science, pages 178–192, Barcelona, Spain, March 1989. Springer-Verlag. 47

[38] Torsten Bülck, Achim Held, Werner E. Kluge, Stefan Pantke, Carsten Rathsack, Sven-Bodo Scholz, and Raimund Schröder. Experience with the implementation of a concurrent graph reduction system on an nCUBE/2 platform. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 - VAPP VI*, number 854 in Lecture Notes in Computer Science, pages 497–508. Springer, 1994. 189

[39] Geoffrey Burn, Simon L. Peyton Jones, and J. D. Robson. The spineless G-machine. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, July 1988. ACM Press. 189

[40] F. Warren Burton. A linear space translation of functional programs to Turner combinators. *Information Processing Letters*, 14(5):201–204, 1982. 189

[41] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Proceedings of the 1994 International Symposium on Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 244–272, Sendai, Japan, April 1994. Springer-Verlag. 48, 148

[42] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-102, Dept. of Comp. Sci., CMU, March 2002. Revised May 2003. 247, 257

[43] Stephen Chang and Matthias Felleisen. The call-by-need lambda calculus, revisited. In Helmut Seidl, editor, *Programming Languages and Systems, 21st European Symposium on Programming, ESOP 2012*, Lecture Notes in Computer Science, pages 128–147, Tallinn, Estonia, March 2012. Springer. 158

[44] Stephen Chang, David Van Horn, and Matthias Felleisen. Evaluating call by need on the control stack. In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Trends in Functional Programming, Volume 11*, number 6546 in Lecture Notes in Computer Science, pages 1–15, Norman, Oklahoma, May 2011. Springer. 179, 212

[45] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report 416, INRIA, June 1985. 240, 241, 242, 256

[46] John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004. A preliminary version was presented at the 12th European Symposium on Programming (ESOP 2003). 68

[47] William D. Clinger. Proper tail recursion and space efficiency. In Keith 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, 62

[48] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987. 47

[49] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *TCS*, 277(1–2):47–103, 2002. 241

[50] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In Andrew W. Appel, editor, *Proc. of POPL'92*, pages 83–94. ACM Press, January 1992. 241

[51] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007. A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming. 4, 47, 49, 63, 162, 177

[52] Haskell B. Curry. Apparent variables from the standpoint of Combinatory Logic. *Annals of Mathematics*, 34:381–404, 1933. 237

[53] Haskell B. Curry, J. Roger Hindley, and Robert Feys. *Combinatory Logic: Volume II*. North Holland, 1972. 190

[54] Nils Anders Danielsson. Operational semantics using the partiality monad. In Robby Findler and Peter Thiemann, editors, *Proc. of ICFP'12*, pages 127–138, September 2012. 241

[55] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991. 39

[56] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992). 46, 103, 110, 145, 152

[57] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk. 47, 101

[58] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer. Recipient of the 2004 Peter Landin prize. Extended version available as the research report BRICS RS-03-33. 45

[59] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk. 61, 62

[60] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006. 39

[61] Olivier Danvy. From reduction-based to reduction-free normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises. 94, 109, 127, 138, 148, 159, 173, 179, 190, 192, 201, 211, 217, 218

[62] Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, September 2008. ACM Press. Invited talk. 47, 61, 126, 152, 189, 190, 192, 201, 211, 241

[63] Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines. In Palsberg [156], pages 162–185. 48, 189

[64] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press. 19, 47, 121

[65] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76:302–323, 2010. 48, 62, 123, 189

[66] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press. 145

[67] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008. 61, 73, 94, 103, 105, 122, 144, 151, 179, 203, 217

[68] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *Logical Methods in Computer Science*, 4(4:12):1–67, November 2008. 47, 188, 189, 190

[69] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Extended version available as the research report BRICS RS-08-04. 46, 103, 126, 144, 145, 204, 205

[70] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23; most influential paper at PPDP 2001. 45, 69, 103, 109, 126, 144, 204, 237

[71] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4. 57, 58, 61, 62, 68, 76, 103, 104, 126, 139, 140, 164, 201, 218

[72] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 2004(1), July 2004. Available online at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/`. A preliminary version was presented at the Sixth International Symposium on Functional and Logic Programming (FLOPS 2002). 235

[73] Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. In María Alpuente, editor, *Logic Based Program Synthesis and Transformation, 20th International Symposium, LOPSTR 2010, revised selected papers*, number 6564 in Lecture Notes in Computer Science, pages 1–20, Hagenberg, Austria, July 2010. Springer. Invited talk. Extended version to appear in ACM Transactions on Computational Logic and appears in Chapter 10. 6, 73, 149, 159, 177, 206, 209, 267

[74] Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. *ACM Transactions on Computational Logic*, 2013. To appear. A preliminary version appears in [73]. This version appears in Chapter 10. 6, 209

[75] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS*

2010, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer. An extended version appears in [77] and in Chapter 7. 6, 125, 268

[76] Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011)*, pages 1–12, Austin, Texas, January 2011. ACM Press. Invited talk. An extended version appears in Chapter 6. 6, 52, 73, 93, 122, 189, 190, 193, 201, 205, 241

[77] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science*, 435:21–42, 2012. A preliminary version appears in [75]. This version appears in Chapter 7. 6, 73, 125, 158, 159, 170, 174, 178, 189, 206, 211, 241, 254, 268

[78] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proc. of LICS'96*, pages 184–195, 1996. 255

[79] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004. 87, 91

[80] Yuxin Deng, Robert J. Simmons, and Iliano Cervesato. Relating reasoning methodologies in linear logic and process algebra. Technical Report CMU-CS-11-145, Dept. of Comp. Sci., CMU, December 2011. 246

[81] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000. 158

[82] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems*, 20(2): 344–387, 1998. 158, 159

[83] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. *Journal of Functional Programming*, 17(6):687–730, 2007. 153

[84] Jon Fairbairn and Stuart Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, Oregon, September 1987. Springer-Verlag. 162, 179

[85] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987. 29, 62, 67, 69, 102

[86] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes available at `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html` and last accessed in April 2008, 1989-2001. 139

[87] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986. 47, 62, 78

[88] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. 62, 149

[89] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 3, 20, 67, 68, 69, 78, 92, 102, 159, 193, 258

[90] Andrzej Filinski. Representing monads. In Boehm [35], pages 446–457. 16, 121

[91] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press. 16

[92] Michael J. Fischer. Lambda-calculus schemata. In Talcott [204], pages 259–288. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972. 42

[93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press. 41

[94] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. The MIT Press, third edition, 2008. 69

[95] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, Scotland, July 1976. 160

[96] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Lynn Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20 (3):271–293, 2007. 126, 158, 177

[97] Stefan Fünfrocken. Transparent migration of Java-based mobile agents. In Kurt Rothermel and Fritz Hohl, editors, *Mobile Agents, Second International Workshop, MA'98, Proceedings*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, September 1998. Springer. 148

[98] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, September 1999. ACM Press. 58, 105

[99] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. *Logical Methods in Computer Science*, 6(3:1):1–39, July 2010. A preliminary version was presented at the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages (POPL 2009). 68, 126, 131, 132, 148, 158, 170, 178, 218

[100] John R. W. Glauert, Richard Kennaway, and M. Ronan Sleep. Dactl: An experimental graph rewriting language. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Proceedings*, volume 532 of *Lecture Notes in Computer Science*, pages 378–395, Bremen, Germany, March 1990. Springer. 188, 237

[101] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979. 9

[102] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 211–222, Coronado Island, San Diego, California, USA, November 2001. IEEE Computer Society. ISBN 0-7695-1426-X. 48

[103] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001. 148

[104] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming, 24th European Conference*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150, Maribor, Slovenia, June 2010. Springer-Verlag. 87, 88, 90, 91

[105] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992. 9

[106] John Hannan and Dale Miller. From operational semantics to abstract machines. In Mitchell Wand, editor, *Special issue on the 1990 ACM Conference on Lisp and Functional Programming*, Mathematical Structures in Computer Science, Vol. 2, No. 4, pages 415–459. December 1992. 47, 242, 259

[107] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998. 159

[108] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2013. 20, 239

[109] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *JFP*, 17(4-5):613–673, 2007. 32, 242, 243, 249

[110] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in the proceedings of the First IEEE Symposium on Logic in Computer Science, pages 194–204, June 1987. 32, 242, 243

[111] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [35], pages 458–471. 41, 223

[112] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997. 223

[113] Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, January 1976. 126, 160

[114] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 7

[115] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. 136, 199

[116] John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, August 1982. ACM Press. 189

[117] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5. Intellect, February 2006. 189

[118] Alan Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 82–91, Paris, France, July 1994. IEEE Computer Society Press. 188, 237

[119] Jacob Johannsen. An investigation of Abadi and Cardelli's untyped calculus of objects. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, June 2008. BRICS research report RS-08-6. 4, 48, 62

[120] Thomas Johnsson. Efficient compilation of lazy evaluation. In Susan 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. 189

[121] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag. 235

[122] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989. 126

[123] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, February 1987. Springer-Verlag. 20, 240, 241

[124] Gabriel Kerneis. *Continuation-Passing C: Program Transformations for Compiling Concurrency in an Imperative Language*. PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, 2012. 48

[125] Jan W. Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980. 223

[126] Pieter W. M. Koopman. *Functional Programs as Executable Specifications*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1990. 188, 189, 237

[127] Jean-Louis Krivine. Un interprète du λ-calcul. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`, 1985. 47, 62

[128] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007. 47, 62, 177

[129] George Kuan. A rewriting semantics for type inference. Master's thesis, Department of Computer Science, University of Chicago, March 2007. Technical report TR-2007-03. 48, 62

[130] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:363–386, 1958. 244

[131] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. 3, 9, 20, 32, 45, 47, 148, 188

[132] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation, Parts 1 and 2. *Communications of the ACM*, 8:89–101 and 158–165, 1965. 3, 20, 32

[133] Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, Aarhus University, pages 1:1–9, Paris, France, January 1997. 16, 39

[134] John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press. 126, 138, 158, 175, 177, 178

[135] Peter Lee, editor. *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press. 262

[136] Xavier Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, February 1990. 62

[137] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284–304, 2009. 241

[138] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. Thèse d'état, Université de Paris VII, Paris, France, 1978. 160

[139] Hans-Wolfgang Loidl, Fernando Rubio, Norman Scaife, Kevin Hammond, Susumu Horiguchi, Ulrike Klusik, Rita Loogen, Greg Michaelson, Ricardo Pena, Steffen Priebe, Álvaro J. Rebón Portillo, and Philip W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003. 189

[140] Florian Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice, Nice, France, March 2009. 148

[141] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998. 126, 129, 130, 132, 158, 165, 175, 262

[142] John McCarthy. An algebraic language for the manipulation of symbolic expressions. Technical Report AI Memo No. 1, MIT AI Lab, Cambridge, Massachusetts, September 1958. 3, 32

[143] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, June 2007. 4

[144] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, May 2007. 4

[145] Robin Milner. *Communication and Concurrency*. Prentice Hall International, 1989. 164

[146] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990. 4, 31

[147] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 190

[148] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1993. 7, 20

[149] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991. 7, 10, 41

[150] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004. 258

[151] Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2007. BRICS research report RS-08-3. 4, 62, 149

[152] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009. 158, 174, 178, 236, 237

[153] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, July 2001. BRICS DS-01-7. 4

[154] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992. 7, 20, 112

[155] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, January 2007. ACM Press. 58, 105, 122, 144, 179, 203

[156] Jens Palsberg, editor. *Semantics and Algebraic Specification: Essays dedicated to Peter D. Mosses on the occasion of his 60th birthday*, number 5700 in Lecture Notes in Computer Science, 2009. Springer. 263, 266

[157] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *JLP*, 19:261–320, 1994. 254

[158] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, SIGPLAN Notices, Vol. 40, No. 9, pages 216–227, Tallinn, Estonia, September 2005. ACM Press. 148

[159] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987. 189, 190, 192, 211, 236

[160] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992. 48, 138, 158, 189

[161] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991. 242, 243

[162] Frank Pfenning. Lecture notes for 15-816: Linear logic, February 2012. 255

[163] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In *Types for Proofs and Programs*, pages 179–193, 1998. 250

[164] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of CADE'16*, pages 202–206. Springer LNAI 1632, 1999. 243

[165] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proc. of LICS'09*, pages 101–110, 2009. 239, 242, 244, 250, 252, 254, 257

[166] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. 20, 48, 62, 69

[167] Maciej Pirog and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. In Jeremy Gibbons, editor, *Haskell '10: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, pages 25–36, Baltimore, Maryland, September 2010. ACM Press. 48, 138, 149, 159, 206

[168] Marinus J. Plasmeijer and Marko C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993. 188, 190, 237

[169] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. 42, 122, 180, 188

[170] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977. 4, 33

[171] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, Aarhus University, Aarhus, Denmark, September 1981. Reprinted in the Journal of Logic and Algebraic Programming 60-61:17-139, 2004, with a foreword [172]. 67, 71, 105, 212

[172] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. 275

[173] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proc. of MFPS'99*, volume 20 of *ENTCS*, pages 449–466, April 1999. 244

[174] Adam Poswolsky and Carsten Schürmann. Factoring report. Technical Report YALEU/DCS/TR-1256, Dept. of Comp. Sci., Yale University, November 2003. 256

[175] Matthias Puech. *Certificates for Incremental Type Checking*. PhD thesis, Alma Mater Studiorum, Università di Bologna and Université Denis Diderot (Paris VII), April 2013. 48, 239, 241

[176] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [179]. 3, 18, 32, 41, 44, 45, 46, 47, 69, 78, 94, 144, 152, 159, 173, 239, 241

[177] John C. Reynolds. The discoveries of continuations. In Talcott [204], pages 233–247. 16, 39

[178] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. 7, 9, 33

[179] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998. 275

[180] Bernard Robinet. *Contribution à l'étude de réalités informatiques*. Thèse d'état, Université Pierre et Marie Curie (Paris VI), Paris, France, May 1974. 237

[181] John A. Robinson. A note on mechanizing higher order logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 5, pages 123–133. Edinburgh University Press, 1969. 236

[182] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79:397–434, 2010. 258

[183] Anders Schack-Nielsen. Induction on concurrent terms. In *Proc. of LFMTP'07*, pages 37–51, 2007. 247

[184] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *Proc. of IJCAR'08*, pages 320–326. Springer LNCS 5195, 2008. 257

[185] David A. Schmidt. State transition machines for lambda calculus expressions. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 415–440, Aarhus, Denmark, 1980. Springer-Verlag. 276

[186] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986. 7, 9, 32

[187] David A. Schmidt. State-transition machines for lambda-calculus expressions. *Higher-Order and Symbolic Computation*, 20(3):319–332, 2007. Journal version of [185], with a foreword [188]. 47

[188] David A. Schmidt. State-transition machines, revisited. *Higher-Order and Symbolic Computation*, 20(3):319–332, 2007. 276

[189] Dana Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. 32

[190] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, 1971. 9

[191] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In Paolo Ciancarini and Alexander L. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION '99, Proceedings*, volume 1594 of *Lecture Notes in Computer Science*, pages 211–226, Amsterdam, The Netherlands, April 1999. Springer. 148

[192] Ilya Sergey. *Operational Aspects of Type Systems: Inter-Derivable Semantics of Type Checking and Gradual Types for Object Ownership*. PhD thesis, Department of Computer Science, KU Leuven, Leuven, Belgium, October 2012. 48, 62

[193] Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Information Processing Letters*, 112(13-20): 13–20, 2011. 239

[194] Ilya Sergey and Dave Clarke. From type checking by recursive descent to type checking with an abstract machine. In Claus Brabrand and Eric Van Wyk, editors, *11th International Workshop on Language Descriptions, Tools, and Applications*, Saarbrücken, Germany, March 2011. ETAPS. 48, 62, 73

[195] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997. 126, 138, 158, 175, 178

[196] Filip Sieczkowski. Automated derivation of abstract machines from reduction semantics: A formalisation of the refocusing transformation in the coq proof system. Master's thesis, University of Wroclaw, Wroclaw Poland, July 2010. 62

[197] Filip Sieczkowski, Małgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages, – 22nd International Symposium, IFL*

2010, number 6647 in Lecture Notes in Computer Science, pages 72–88, Alphen aan den Rijn, The Netherlands, September 2010. Springer. Revised Selected Papers. 58, 62, 68, 76, 103, 140, 164

[198] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, School of Computer Science, Computer Science Department, Carnegie Mellon University, November 2012. 4, 239, 242, 245, 248, 250, 253, 254, 257

[199] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 2011. 245, 257

[200] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474. 69

[201] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977. 7, 9, 45

[202] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [216]. 10

[203] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001. 148

[204] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, 1993. 269, 275

[205] Wei Tao. *A portable mechanism for thread persistence and migration*. PhD thesis, University of Utah, Salt Lake City, Utah, 2001. 148

[206] Robert D. Tennent. *Principles of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981. 7

[207] Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991. 9

[208] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, 2002. 19

[209] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979. 3, 159, 187, 189, 192, 211, 229, 233, 236

[210] Marko van Eekelen, Sjaak Smetsers, and Rinus Plasmeijer. Graph rewriting semantics for functional programming languages. In *Proceedings of the Fifth Annual Conference of the European Association for Computer Science Logic*, pages 106–128. Springer-Verlag, 1996. 188

[211] David Van Horn and Matthew Might. Abstracting abstract machines. In Paul Hudak and Stephanie Weirich, editors, *Proceedings of the 2010 ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, SIGPLAN Notices, Vol. 45, No. 10, pages 51–62, Baltimore, Maryland, September 2010. ACM Press. 73, 257

[212] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999. 249

[213] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974. 148

[214] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21-24, 1988. 104

[215] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Computing Laboratory, Oxford University, Oxford, UK, 1971. 148, 160, 187

[216] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000. 277

[217] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980. 122

[218] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-2002-101, Dept. of Comp. Sci., CMU, March 2002. Revised May 2003. 239, 252, 254

[219] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993. 4, 7, 9, 20

[220] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. 20

[221] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001. 30, 92

[222] Ian Zerny. On graph rewriting, reduction and evaluation. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Trends in Functional Programming, Volume 10*, pages 81–112, Komárno, Slovakia, June 2009. Intellect Books. Best student-paper award of TFP 2009. Extended version to appear in [223] and appears in Chapter 9. 6, 73, 149, 187, 188, 205, 237, 278

[223] Ian Zerny. On graph rewriting, reduction, and evaluation in the presence of cycles. *Higher-Order and Symbolic Computation*, 2013. To appear. A preliminary version appears in [222]. This version appears in Chapter 9. 6, 187, 278

[224] Ian Zerny. *The interpretation and inter-derivation of small-step and big-step specifications*. PhD thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2013. 239