AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction

Department of Computer Science

Jacob Johannsen

PhD Dissertation

ISBN: 978-8775073160

DOI: 10.7146/aul.8.8

On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction

Jacob Johannsen

Revised version of April 30, 2015
Submitted January 31, 2015

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction

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

by
Jacob Johannsen
Revised version of April 30, 2015
Submitted January 31, 2015

Abstract

We study the relationship between small-step semantics, big-step semantics and abstract machines, for programming languages that employ an outermost reduction strategy, i.e., languages where reductions near the root of the abstract syntax tree are performed before reductions near the leaves. In particular, we investigate how Biernacka and Danvy's syntactic correspondence and Reynolds's functional correspondence can be applied to inter-derive semantic specifications for such languages.

The main contribution of this dissertation is three-fold: First, we identify that backward overlapping reduction rules in the small-step semantics cause the refocusing step of the syntactic correspondence to be inapplicable. Second, we propose two solutions to overcome this in-applicability: *backtracking* and *rule generalization*. Third, we show how these solutions affect the other transformations of the two correspondences.

Other contributions include the application of the syntactic and functional correspondences to Boolean normalization. In particular, we show how to systematically derive a spectrum of normalization functions for negational and conjunctive normalization.

Resumé

Vi studerer sammenhængen mellem small-step semantikker, big-step semantikker og abstrakte maskiner, for programmeringssprog, der bruger yderste reduktionsstrategier, dvs. sprog, hvor reduktioner tæt på roden af det abstrakte syntaks-træ bliver udført før reduktioner nær bladene. Specielt undersøger vi, hvordan Biernacka og Danvys syntaktiske korrespondence og Reynolds' funktionelle korrespondence kan anvendes til at udlede semantiske specifikationer fra hinanden for sådanne sprog.

Det primære resultat i denne afhandling består af tre dele: Først observerer vi, at reduktionsregler med bagudrettede overlap i small-step semantikker gør, at refokuserings-skridtet i den syntaktiske korrespondence ikke kan anvendes. Derefter fremsætter vi to løsninger til at overvinde denne manglende anvendelighed: *tilbagevenden* og *regel-generalisering*. Til sidst viser vi, hvordan disse løsninger påvirker de øvrige transformationer i de to korrespondencer.

De øvrige resultater inkluderer anvendelsen af de syntaktiske og funktionelle korrespondencer til Boolesk normalisering. Specielt viser vi, hvordan man systematisk kan udlede et spektrum af normaliseringsfunktioner for negationel og konjunktiv normalisering.

Acknowledgments

This dissertation, along with its subsequent thesis defense, concludes my PhD studies at Aarhus University. My work would not have been possible had it not been for the help and support of a number of people.

First and foremost, I am grateful to my thesis advisor, Olivier Danvy. His enthusiasm, his pedagogical insights, and his scientific rigor and wisdom have been an inspiration throughout the work on this dissertation, but most important of all, he has been a good friend.

I would also like to thank Zena Ariola and Małgorzata Biernacka for serving on my PhD committee, and Christian Storm Pedersen for chairing it.

Additional thanks are due to Zena Ariola, as well as to Paul Downen, Anna Gladstone, Philip Johnson-Freyd, and Luke Maurer for hosting an inspiring 6-month visit to the University of Oregon.

Throughout, I have had fruitful and stimulating discussions with many fellow researchers and students. Thanks are due to all of them, but especially so to Lasse R. Nielsen and Ian Zerny.

I would also like to extend my grateful thanks to the technical, administrative, systems, and library staff at Aarhus University, in particular Ann Eg Mølhave, Nanna Maria Elgaard Pedersen, Michael Glad, and Ellen Kjemtrup.

Finally, thanks are due to my friends and family. I would like in particular to thank Tatiana Barfod, Iben Maria Behrmann Kristensen, Thomas Jensen, Erik Søe Sørensen, my parents, and my brother Claus for their constant support and patience. I would also like to thank Bedour Alshaigy for her incessant and contagious good mood, and Ninni Schaldemose for translation help and for a particularly well-timed piece of encouragement.

Jacob Johannsen,
Aarhus, April 30, 2015.

Contents

Part I

Background and overview

Chapter 1

Introduction

We investigate how program transformations can be used to inter-derive semantic descriptions for programming languages that employ an outermost reduction strategy.

The traditional way to obtain different semantic descriptions (e.g., small-step semantics, big-step semantics, and abstract machines) of the same programming language is to invent them, and then manually prove the equivalence [54]. This approach allows some flexibility in the design of the individual semantic descriptions, so that the various features of the language can be studied (e.g., compile-time aspects such as type safety, and run-time aspects such as environments and stores) using the semantic description best suited for that type of reasoning. However, this approach also requires the invention of new semantic artifacts, and furthermore, the task of proving their equivalence requires even more inventiveness. Also, any subsequent changes to the intended semantics of the language need to be reflected in all the invented semantic artifacts, as well as their equivalence proofs.

To overcome these problems, various approaches to mechanically inter-deriving semantic descriptions have been suggested. Of particular interest here are two derivational techniques: Biernacka and Danvy's syntactic correspondence [13] between small-step semantics and abstract machines, and Reynolds's functional correspondence [2, 102] between big-step semantics and abstract machines. Both of these techniques consist of a sequence of meaning-preserving program transformations that manipulate the *functional representation* of the semantic artifacts. The use of meaning-preserving transformations also eliminates the need for ad hoc proofs of equivalence between the semantic artifacts: The semanticists can use their brains for things that matter, instead of for leg work.

Most programming-language semantics employ an innermost reduction strategy, preferring reductions near the leaves of the program syntax tree rather than reductions near the root. Accordingly, the syntactic and functional correspondences have mostly been applied to programming languages that employ innermost reduction. However, for programming languages that employ an outermost reduction strategy, the syntactic and functional correspondences do not apply as readily.

It is our thesis that
the syntactic and functional correspondences can be made applicable to
semantic descriptions of programming languages that employ an outermost reduction strategy.

The main challenge is the existence of *backward overlapping reduction rules*. The application of a backward overlapping rule can create a new redex at a more outermost position than

the position of the contractum. When this happens, the refocusing transformation of the syntactic correspondence is no longer applicable.

In this work, we have not reverted to designing ad hoc derivations. Instead, we have designed methods that address backward overlapping reduction rules, and which let us reuse the rest of the correspondences as is.

1.1 Motivation

Outermost reduction strategies are uncommon in programming language semantics, though they are not unheard of. Languages based on weak-head normalization of the call-by-name or call-by-need λ-calculus (such as Haskell) are examples, but the syntactic and functional correspondences are applicable to these languages [3, 45, 46, 96]. For the call-by-name λ-calculus, the two correspondences give rise to the Krivine machine [2, 13, 107].

However, when moving to the problem of full normalization of the λ-calculus, the β-rule exhibits a backward overlap which causes refocusing to be inapplicable. Full normalization of the λ-calculus is used for optimization techniques in compilers for functional languages. Additionally, automated proof assistants must be able to perform cut elimination, and for proof assistants based on λ-terms (such as Coq), full cut elimination is equivalent to full normalization of the proof term in question.

Finally, normalization problems outside the realm of programming languages can sometimes be made more efficient when using an outermost reduction strategy. This is the case, for instance, when the reduction rules can eliminate entire subterms; it is more efficient to eliminate subterms that have not yet been normalized than to first normalize them and then eliminate the result.

1.2 Methodology

The goal of the present work is to discover techniques to inter-derive semantic descriptions for programming languages that use outermost reduction strategies.

We are interested in derivations that have the following characteristics:

- Methodical derivations: The derivations should be reusable for a large number of semantics, with little or no change in the individual transformations or the sequence in which they are performed. In other words, we are specifically not interested in ad hoc derivations that will only work for one specific semantic description.

- Minimalistic transformations: The individual transformations of the derivations should be as simple as possible. Minimalistic transformations make it easier to establish the soundness of the transformation, and make it easier to actually perform the transformation. Furthermore, the simpler the transformation, the easier it is to reuse in other settings, e.g., to establish new derivations for new classes of problems. Any new transformations that need to be developed should therefore either be reusable in a different setting, or be based on transformations that are already known.

We choose to specifically focus on adapting Biernacka and Danvy's syntactic correspondence and Reynolds's functional correspondence. In Section 7.6 we list a number of alternative

derivational techniques in the literature. Common to these alternative techniques is that they have never been reused, and involve new transformations rather than pre-existing ones.

In contrast, the syntactic and functional correspondences have both historically been methodical and reusable derivations consisting of minimalistic and well-known transformations. The syntactic correspondence has been applied to a large number of small-step semantics [5, 7, 8, 11, 14, 30, 59, 74, 89, 90, 112, 128, 135], in some cases to derive known abstract machines, and in some cases to derive new ones. Similarly, the functional correspondence has been applied to a large number of big-step semantics [1, 15, 30, 33, 72, 74, 83, 84, 89, 96, 101, 135], in some cases to derive known abstract machines, and in some cases to derive new ones.

Terminology We use the word "methodical" to refer to a derivation (i.e., a sequence of transformations) that can be re-used on many different semantic descriptions with little or no change. "Methodical" derivations contrast "ad hoc" derivations, but should not be confused with "identical" derivations, which would indicate that the derivations are always applicable directly and without change.

We use the word "mechanical" to refer to individual transformations that require little or no human thought or insight to perform. "Mechanical" transformations contrast "manual" transformations, which require a large amount of human thought or insight, but should not be confused with "automatic", which would indicate that the transformation could always be performed without human intervention at all.

For an explanation of the technical terms used in this dissertation, we refer the reader to the technical glossary in Appendix B.

1.3 Contributions

The contributions of this work are as follows:

- We develop the new concept of *backtracking* in the syntactic correspondence, which mechanically enables refocusing for outermost reduction in the presence of backward overlaps (Chapter 6).
- We show how the syntactic and functional correspondences can be applied to the De Morgan laws and the law of double negation, to systematically derive a spectrum of negational normalizers (Section 7.3).
- We develop the new concept of *rule generalization*, which enables refocusing for outermost reduction in the presence of backward overlaps, by constructing context-sensitive reduction rules out of the backward overlapping rules (Section 7.4.0).
- We show how the syntactic and functional correspondences can be applied to the distributive laws for disjunctions over conjunctions, to systematically derive a spectrum of negational normalizers (Section 7.4).
- We show how the exponential blowup of conjunctive normal forms can be reduced to a linear blowup by sharing delimited continuations, and how the corresponding normalization process can be inverted (Section 7.5).

- We present a number of open questions in relation to outermost reduction strategies, as well as partial answers to some of these questions (Chapter 8).

1.4 Overview

This dissertation is structured into four parts:

- Part I presents the background material for the thesis and an overview of the results: Chapter 2 presents the styles of semantics that the thesis is concerned with, and how to represent them. Chapter 3 presents a useful type isomorphism. Chapter 4 presents the syntactic correspondence between small-step semantics and abstract machines. Chapter 5 presents the functional correspondence between big-step semantics and abstract machines.
- Part II contains the publications that have come out of the present work. Chapter 6 presents the notion of backtracking, and shows how it can be applied to enable refocusing, using the problem of addition of Peano numbers as a case study. Chapter 7 presents a spectrum of Boolean normalization functions, linked together by the syntactic and functional correspondences. In particular, the chapter presents the notion of rule generalization, and shows how it can be applied to enable refocusing.
- Part III contains open questions related to the present work: Chapter 8 presents a series of questions that have arisen during the work on this thesis. The chapter also attempts to provide partial answers to some of the questions.
- Part IV contains two appendices: Appendix A presents examples of how various styles of semantics can be represented in Standard ML, and gives an overview of the program transformations that link the representations together. Appendix B contains a technical glossary.

Chapter 2

Semantics and representations

This chapter outlines the various forms of semantics used in the present work, and how those semantics are represented.

We start by reviewing the notion of term rewriting as it relates to semantics (Section 2.1). We then review the notions of reduction semantics (Section 2.2) and abstract machines (Section 2.3). We summarize and conclude in Section 2.4.

2.1 Term rewriting

All definitions in this section are taken from TERESE [126], except where explicitly mentioned.

A *term rewriting system* consists of a grammar of terms and a set of *reduction rules*. A reduction rule consists of a left-hand side and a right-hand side, both of which are *meta-terms*, i.e., terms that may contain meta-variables ranging over terms. The left-hand side of a reduction rule is called a *redex*. The right-hand side of a reduction rule is called a *contractum*. A contractum can only contain meta-variables that occur in the corresponding redex.

The grammar of terms induces a notion of *contexts* as "a term with a hole", and the left-hand sides of the reduction rules induce a grammar of redexes. In the grammar of redexes, meta-variables are represented as non-terminals of the grammar of terms. The grammars of terms and redexes induce a notion of *normal form* as a term that contains no redexes.

In the present work, we are especially interested in sets of reduction rules that exhibit *backward overlaps* [50, 66]. The definition of of backward overlaps as well as the dual definition of forward overlaps is as follows:

Definition 1 (Backward and forward overlaps)**.** *Let* $\mathrm{pr}_1 \mapsto c_1$ *and* $\mathrm{pr}_2 \mapsto c_2$ *be reduction rules. If there exist a non-empty context* C *and a meta-term* t *containing at least one term constructor such that*

- $\mathrm{pr}_1 = \mathsf{C}[\mathsf{t}]$
- t *unifies with* c_2 *with the most general unifier* σ*,*

then pr_1 *and* c_2 *are said to exhibit a* backward overlap*, and the two rules are said to be* backward overlapping*.*

Dually, if there exist a non-empty context C *and a meta-term* t *containing at least one term constructor such that*

- $c_1 = C[t]$
- t *unifies with* pr_2 *with the most general unifier* σ,

then pr_2 *and* c_1 *are said to exhibit a* forward overlap, *and the two rules are said to be* forward overlapping.

Section 6.3 explores the applicability of refocusing when the reduction rules contain backward and forward overlaps. The conclusion is that refocusing is inapplicable only when the reduction rules contain backward overlaps. We therefore only focus on backward overlaps for the remainder of this work.

For each rule and each instantiation of meta-variables of that rule, the term corresponding to the instantiated redex of that rule is said to *reduce* to the term corresponding to the instantiated contractum of that rule. Together, the reduction rules define a *one-step reduction relation* between terms, and that reduction relation is closed under contexts. Operationally, this means that reduction can take place in any context. The set of terms that a term can be reduced to is known as the *reducts* of the term.

The transitive closure of the one-step reduction relation defines a *normalization relation* between terms and values. Operationally, this means that normalization is not deterministic; a term can have several different normal forms. The enumeration of reducts obtained by applying the one-step relation successively on the way to a normal form is known as a *reduction sequence*.

In Section 7.4.0, we use a construction similar to *backward chaining* [50, 66]. We give the definition of backward chains here:

Definition 2 (Backward chain)**.** *Given a set of rewrite rules* R*, a* backward chain *of* R *is a reduction sequence* $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n$ *such that all reductions in the sequence (except possibly for the last one) give rise to a backward overlap.*

The construction in Section 7.4.0 amounts to introducing a new reduction rule for each backward chain in the reduction system while respecting the reduction strategy. We introduce reduction strategies next.

2.1.1 Term rewriting vs. programming-language semantics

A term rewriting system defines a bare-bones version of a small-step operational semantics; through the iteration of the one-step reduction relation, a term is related to a set of normal forms which define the possible meanings of the term.

However, in programming languages, we are typically interested in introducing various restrictions to the rewrite system. For instance, in order to obtain a deterministic programming language it is necessary to restrict the one-step reduction relation to be a *reduction function*. Another example is the need to introduce static guarantees that normalization of a term does not get stuck, e.g., by introducing a type system. We therefore introduce the following additional notions that we will use to impose useful restrictions on our term rewriting systems:

Reduction strategy A reduction strategy restricts the one-step reduction relation to only consider some of the redexes in a term.

Typically, the restriction is imposed by considering the relative positions of the redexes in the term. We therefore define the notion of *position* within a term:

Definition 3 (Position)**.** *The position of a subterm* s *within a given term* t *is a vector of natural numbers defined inductively as follows:*

- *If* $s = t$*, then* s *has position* $\langle\rangle$*.*
- *If* s *is the* n*th child of a subterm with position* p*, then* s *has position* $p \cdot \langle n \rangle$*, where* $\cdot$ *is the concatenation operator for vectors.*

For example, consider the following Standard ML data type and example values:

```
datatype term = T0 of string
              | T1 of term
              | T2 of term × term

val t0 = T0 "a"
val t1 = T2 (T0 "b", T0 "a")
val t2 = T2 (T2 (T0 "a", T1 (T0 "b")), T2 (T1 (T1 (T0 "c")), T0 "d"))

val sa = T0 "a"
val sb = T0 "b"
val sc = T0 "c"
```

The term `sa` occurs at position $\langle\rangle$ in `t0`, position $\langle 1 \rangle$ in `t1`, and $\langle 0, 0 \rangle$ in `t2`. The term `sb` occurs at position $\langle 0 \rangle$ in `t1`, and $\langle 0, 1, 0 \rangle$ in `t2`, but does not occur as a subterm in `t0`, and therefore does not have a well-defined position in that term. The term `sc` occurs at position $\langle 1, 0, 0, 0 \rangle$ in `t2`, but does not occur as a subterm in `t0` or `t1`, and therefore does not have a well-defined position in those terms.

If a position vector p is the concatenation of two other vectors p_1 and p_2 (i.e., $p = p_1 \cdot p_2$), we say that p_1 is a *prefix* of p. If $p_2 \neq \langle\rangle$, we say that p_1 is a *proper prefix* of p. If a subterm s occurs at position p, then any term node that is more outermost than s (i.e., on the path from s to the root of the term) has a position which is a proper prefix of p.

We are now in a position to define various classes of reduction strategies based on the relative positions of the redexes in a term:

Definition 4 (Reduction strategies)**.** *Let* pr_1 *and* pr_2 *be redexes in* t *with positions* p_1 *and* p_2*, respectively. The following list contains examples of reduction strategies:*

- *Outermost reduction: If* p_1 *is a proper prefix of* p_2*, then* pr_2 *is not considered for reduction. If* p_2 *is a proper prefix of* p_1*, then* pr_1 *is not considered for reduction. Otherwise, both* pr_1 *and* pr_2 *are considered for reduction.*
- *Innermost reduction: If* p_1 *is a proper prefix of* p_2*, then* pr_1 *is not considered for reduction. If* p_2 *is a proper prefix of* p_1*, then* pr_2 *is not considered for reduction. Otherwise, both* pr_1 *and* pr_2 *are considered for reduction.*
- *Leftmost reduction: If* p_1 *is a prefix of* p_2*, or* p_2 *is a prefix of* p_1*, then both* pr_1 *and* pr_2 *considered for reduction. Otherwise, if* p_1 *is lexicographically less than* p_2*, then* pr_2 *is not considered for reduction. Otherwise (i.e.,* p_2 *is lexicographically less than* p_1*),* pr_1 *is not considered for reduction.*

- *Rightmost reduction: If* $\mathtt{p}_1$ *is a prefix of* $\mathtt{p}_2$*, or* $\mathtt{p}_2$ *is a prefix of* $\mathtt{p}_1$*, then both* $\mathtt{pr}_1$ *and* $\mathtt{pr}_2$ *considered for reduction. Otherwise, if* $\mathtt{p}_1$ *is lexicographically less than* $\mathtt{p}_2$*, then* $\mathtt{pr}_1$ *is not considered for reduction. Otherwise (i.e.,* $\mathtt{p}_2$ *is lexicographically less than* $\mathtt{p}_1$*),* $\mathtt{pr}_2$ *is not considered for reduction.*

- *Leftmost-outermost reduction: The combination of leftmost and outermost reduction, i.e., lexicographical ordering.*

- *Leftmost-innermost reduction: The combination of leftmost and innermost reduction, i.e., lexicographical ordering using reverse prefix ordering.*

- *Rightmost-outermost reduction: The combination of rightmost and outermost reduction, i.e., reverse lexicographical ordering but with standard prefix ordering.*

- *Rightmost-innermost reduction: The combination of rightmost and innermost reduction, i.e., reverse lexicographical ordering.*

Each reduction strategy defines an ordering of redexes in a term, and, more generally, an ordering of the nodes in a term. Continuing the example above, consider the following terms:

```
val t3 = T2 (T2 (T0 "a", T1 (T0 "b")), T2 (T1 (T1 (T0 "a")), T0 "b"))

val sa  = T0 "a"
val sab = T2 (T0 "a", T1 (T0 "b"))
```

The term `sa` occurs at positions $\langle 0,0\rangle$ and $\langle 1,0,0,0\rangle$ in `t3`. The occurrence at position $\langle 0,0\rangle$ is more leftmost than the occurrence at position $\langle 1,0,0,0\rangle$, because $\langle 0,0\rangle$ comes before $\langle 1,0,0,0\rangle$ in a lexicographical ordering. The term `sab` occurs at position $\langle 0\rangle$ in `t3`. This occurrence is more outermost than the occurrence of `sa` at position $\langle 0,0\rangle$, because $\langle 0\rangle$ is a proper prefix of $\langle 0,0\rangle$.

The reduction strategy restricts the one-step reduction relation to only consider the least redexes in that ordering. If the ordering is total, the reduction strategy is deterministic (in the list above, leftmost-outermost, leftmost-innermost, rightmost-outermost and rightmost-innermost are deterministic). However, a deterministic reduction strategy does not guarantee that the one-step reduction relation is a function, because the least redex might be reducible according to multiple reduction rules (or, equivalently, a reduction rule might itself be non-deterministic). If every redex is only reducible by one reduction rule, then the one-step reduction relation is a function, and normalization is deterministic.

Section 6.3 explores the applicability of refocusing when the reduction strategy is innermost and outermost. The conclusion is that refocusing is inapplicable only when the reduction strategy is outermost, and the reduction rules contain backward overlaps. Additionally, refocusing is known to only be applicable when the reduction strategy is deterministic [43]. We therefore only focus on deterministic outermost strategies for reduction systems with backward overlaps for the remainder of this work.

Since we are only concerned with deterministic reduction strategies, we can always assume that the subterms must be searched in a left-to-right fashion. If that is not the case, we can simply re-order the subterms of each term constructor such that left-to-right order is obtained [43].

2.1.2 Representation

In the present work, we do not represent term rewriting system explicitly. Rather, we use term rewriting as a stepping stone towards enabling the syntactic correspondence for outermost reduction.

2.2 Reduction semantics

A *reduction semantics* is a small-step operational semantics over a term language. In addition to an underlying term rewriting system, a reduction semantics consists of an explicit notion of *values* and a linear grammar of *reduction contexts*.

The choice of values and reduction contexts affect the behavior of the underlying term rewriting system. In particular, if the chosen set of values differs from the induced set of normal forms, reduction can now get stuck in a normal form that is not a value. Also, the choice of values and reduction contexts affect the reduction strategy.

Let us now briefly introduce these notions, and explain how we can implement them to suit our purposes.

Values The values of a reduction semantics is a set of terms that the semanticist deems to be final, i.e., terms that should not be reduced any further.

The set of values often differs from the set of normal forms induced by the underlying term rewriting system [97]. For example, the values of a weak-head normalization of the λ-calculus are:

$$v ::= \lambda x.t$$

Since a λ-abstraction can contain an arbitrary, un-normalized term, a value can contain a redex, and is therefore not necessarily a normal form. In this case, we adjust the reduction strategy to not consider redexes in subterms that are values.

Similarly, not all normal forms are values; for instance, in a language where the values are Booleans and integers, and the term language includes an if-then-else construct (with the standard reduction rules), the term $\mathtt{ifthenelse}(0, 1, 2)$ does not contain a redex, and is therefore technically a normal form. However, the term cannot be said to be meaningful; rather, it indicates an error, and we refer to such an erroneous term as *stuck*. In this case, we make the distinction between *potential redexes* and *actual redexes*. A potential redex is a term for which a reduction rule might apply, but which might also be a stuck term, e.g., because of incorrect types. An actual redex is a potential redex for which a reduction rule does apply.

In a reduction semantics, the grammar of potential redexes can use non-terminals from the grammar of values as well as non-terminals from the grammar of terms. A potential redex that uses a value non-terminal instead of a term non-terminal as one of its subterms indicates that that subterm must be fully normalized before this potential redex can be considered for reduction. Potential redexes with value non-terminals therefore indicate that the reduction strategy is innermost. Conversely, potential redexes with only term non-terminals indicate that the reduction strategy is outermost. These indications are not hard rules, though: For instance, the β-rule for weak-head normalization of the λ-calculus contains a term non-terminal inside a λ-abstraction, but because the λ-abstraction is a value, the reduction strategy could be either innermost or outermost.

Reduction contexts A reduction context is a context in which reduction is allowed to take place. The set of reduction contexts is specified explicitly with a linear grammar. This grammar can refer to non-terminals of the grammar of terms (to represent as of yet unevaluated terms), as well as to non-terminals of the grammar of values (to represent terms that have already been fully normalized).

The purpose of having an explicit representation of the contexts in the semantics is twofold:

First, it provides another way to specify the intended reduction strategy, in that such a grammar can be used to disregard potential redexes in some parts of the term, or postpone the contraction of redexes in a subterm until another subterm has been normalized to a value. For this reason, not all contexts of the underlying term rewriting system are reduction contexts in the reduction semantics. We will often refer a context that is a reduction context as a "legal reduction context".

Second, it provides a simple way to specify the behavior of control operators such as `callcc`, `shift` and `reset`, because it allows the reduction rules to be specified using all or parts of the current reduction context. In the latter case, the grammar of terms uses non-terminals from the grammar of contexts to represent reified reduction contexts.

2.2.1 Unique decomposition

The combination of a potential redex and its reduction context is referred to as a *decomposition*. When designing a reduction semantics it is often useful to establish the *unique decomposition* property [134]:

Definition 5 (Unique decomposition)**.** *A reduction semantics has the unique decomposition property iff every term is either a value, or contains only one redex with a legal reduction context.*

In a reduction semantics with the unique decomposition property only one potential redex is ever considered for reduction, namely the one that has a legal reduction context. Consequently, in a reduction semantics with the unique decomposition property, the grammar of contexts essentially encodes a deterministic reduction strategy.

Although unique decomposition is commonly regarded as a fundamental property of reduction semantics, in the present work we will not assume that a reduction semantics automatically has the unique decomposition property. Dropping this assumption is justified for two reasons; first, the combination of backward overlaps and outermost reduction makes refocusing inapplicable even if the reduction semantics exhibits unique decomposition, because the backward overlap constructs a new redex at a more outermost position than that of the contractum. Second, it is not obvious that it is even possible to establish unique decomposition for a reduction semantics that have backward overlaps and uses an outermost reduction strategy (see Section 8.3 for a discussion).

2.2.2 Representation

We represent reduction semantics using data types and functions of Standard ML.

We represent the grammars of terms, values, potential redexes and reduction contexts using inductive data types. The grammars of terms, values and potential redexes are designed and encoded by hand. The grammar of reduction contexts is derived using Danvy and Zerny's prelude to reduction semantics described in Section 4.1.

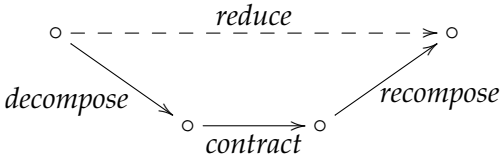
We represent the reduction strategy using a recursive traversal over the term being normalized[1]. The traversal is performed by two mutually tail-recursive functions in combination, *decompose_term* and *decompose_cont*. *decompose_term* takes the current subterm and its context as parameters, and dispatches on the term constructor at the root of the subterm. *decompose_cont* takes the current subterm (as an instance of the value data type) and its context as parameters, and dispatches on the immediate context constructor. The traversal is initiated through a function *decompose*, which takes a term as a parameter, and calls *decompose_term* with the term and the empty context.

If the input term is a value, *decompose* returns the same term as a result of the value data type. If the term is not a value, then the function returns a decomposition consisting of the next potential redex to be contracted, and its reduction context. Since every non-value term contains a potential redex, *decompose* is a total function.

We represent the reduction rules using a function from potential redexes to contractums (i.e., terms), called *contract*. Since a potential redex might not be an actual redex, this function not necessarily total[2].

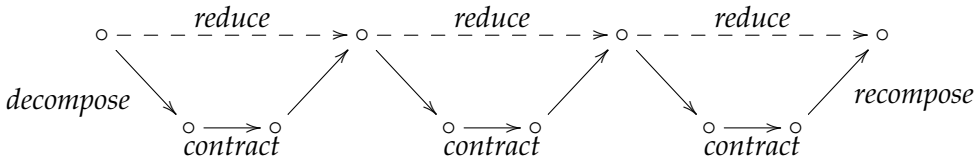
In order to construct the next reduct in the reduction sequence, we also need a function that builds a reduct (of the type of terms) out of a contractum and a context. This function is called *recompose*, and is a left inverse of *decompose*.

We represent one-step reduction as the composition of *decompose*, *contract* and *recompose*:



One-step reduction is a partial function, because the input term might be a value (which is irreducible), or because the next potential redex is not an actual redex (the term is stuck).

We represent normalization using an iterator that continually calls the one-step reduction function, until a value is reached (or until reduction is stuck):



The resulting function is a *small-step normalizer*, because it enumerates the reduction sequence. The normalizer can get stuck (because *contract* is a partial function), but can also diverge if no value is reachable from the initial term through the reduction strategy.

2.2.3 A concrete example of a reduction semantics

As an example, let us show the representation of the reduction semantics for subtraction of natural numbers:

[1] Note that we do not assume that the reduction strategy is enforced by the grammar of contexts. The reduction strategy is enforced through the traversal function alone.

[2] In some cases, a reduction rule can be non-deterministic, or, equivalently, an actual redex can be contracted using two or more distinct reduction rules. In such cases, *contract* is implemented in a non-deterministic fashion, because it represents a relation rather than a function.

The grammar of terms An arithmetic expression is either a literal or a subtraction. We represent the abstract syntax of arithmetic expressions with the following ML data type:

```
datatype expr = LIT of int (* assumed to represent a natural number *)
              | SUB of expr × expr
```

Literals are assumed to represent natural numbers, i.e., to be non-negative integers. An ML value represents an abstract-syntax tree when it is inductively constructed using the constructors `LIT` and `SUB`. It has then the type `expr`.

So for example, `SUB (LIT 3, LIT 2)` represents $3-2$ and $(4-1)-(7-5)$ is represented by `SUB (SUB (LIT 4, LIT 1), SUB (LIT 7, LIT 5))`. (If one wanted to be pedantic, one would write that `SUB (LIT 3, LIT 2)` evaluates to an ML value that represents the abstract-syntax tree corresponding to $3-2$.)

The notion of value A value is an expression without any subtractions. Since ML does not support subtyping, we represent normal forms with the following specialized data type:

```
datatype expr_nf = NAT of int

(*  expr_nf → expr  *)
fun embed_nf (NAT n)
    = LIT n
```

Our notion of value is this normal form:

```
type value = expr_nf
```

As for the meaning of an expression as a natural number or an error, we represent it with the following data type:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string
```

Potential redexes There is only one:

```
datatype potential_redex = PR_SUB of value × value
```

The result of contraction is either an expression or an error message:

```
datatype contractum_or_error = CONTRACTUM of expr
                             | ERROR      of string
```

The contraction function Given an actual redex, the contraction function yields an expression; otherwise it yields an error message:

```
(*  potential_redex → contractum_or_error  *)
fun contract (PR_SUB (NAT n1, NAT n2))
    = if n1 < n2
      then ERROR ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else CONTRACTUM (LIT (n1 - n2))
```

The reduction strategy It determines the following grammar of reduction contexts:

```
datatype cont = C0
              | C1 of expr × cont
              | C2 of cont × value
```

Thus equipped, we define the notion of value or decomposition with the following data type:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex × cont
```

We are then in position to define a decomposition function and a recomposition function. The decomposition function maps an expression in normal form to this normal form and an expression not in normal form to a potential redex and its reduction context:

```
(* decompose : expr → value_or_decomposition  *)
```

The recomposition function recomposes the reduction context over an expression; as such it is a left inverse of the decomposition function:

```
(* recompose : cont × expr → expr *)
```

The result of reduction is either an expression or an error message:

```
datatype reduct_or_stuck = REDUCT of expr
                         | STUCK  of string
```

We are then in position to define a decompose-contract-recompose function representing one-step reduction:

```
(*  expr → reduct_or_stuck  *)
fun reduce e
    = (case decompose e
        of (VAL v)
           ⇒ STUCK "irreducible expression"
         | (DEC (pr, k))
           ⇒ (case contract pr
                of (CONTRACTUM e') ⇒ REDUCT (recompose (k, e'))
                 | (ERROR s)       ⇒ STUCK s))
```

Accordingly, we define a reduction-based evaluation function that iterates one-step reduction, using the result of `decompose` to detect whether we have reached a normal form:

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM e') ⇒ iterate (decompose (recompose (k, e')))
         | (ERROR s)       ⇒ WRONG s)

(*  expr → result_or_wrong  *)
fun normalize e = iterate (decompose e)
```

This evaluation function enumerates the following reduction sequence, for example: the first redex in the term $(5-7)-(4-1)$ is $4-1$, which contracts to 3 and gives rise to the first reduct $(5-7)-3$; the first redex in this reduct is $5-7$, which is not an actual redex and thus yields the error message "underflow: $5-7$."

2.3 Abstract machines

An *abstract machine* is a state-transition system over source terms, where each transition requires no subgoal. The machine is equipped with one or more *stacks*, and possibly also with

other components such as an *environment* or a *store*. In addition to the grammar of source terms, the abstract machine comes with an explicit notion of values, a grammar of stack values, and grammars for any additional components. Just like for reduction semantics, the grammars of terms, values, stack values and other state components can depend on non-terminals from each other.

The initial state of the machine is a term with all empty stacks. The final state of the machine is a value with all empty stacks. The value part of the final state is considered the final value of the normalization process. If the initial term does not have a corresponding value, the abstract machine will either be stuck in some state from which there is no transition, or will diverge.

The purpose of using abstract machines as a semantic description of a language is to simulate the implementation of the language on a real machine.

2.3.1 Representation

An abstract machine is implemented as a set of individual functions in Standard ML. Each state corresponds to one function, and the components of each state are implemented as the parameters to the corresponding function. A pattern match on the components of the state allows the representation to determine which transition to take, and therefore also which state will be next. All calls made by the machine state functions are tail calls.

2.3.2 Two concrete examples of abstract machines

An abstract machine can be represented either in a small-step fashion or a big-step fashion. As an example, we show the small-step and big-step representations of an abstract machine that implements a deterministic finite automaton recognizing whether a given list of Booleans contains an even number of occurrences of `true`.

A small-step abstract machine A small-step representation contains an external driver loop or "trampoline" function [57] which drives the normalization process. When the functions representing the machine states return a state, the driver loop checks whether the state is a final or stuck state. If it is neither, the driver loop calls the machine state functions again. The representation looks as follows:

```
datatype intermediate_state = EVEN of bool list
                            | ODD  of bool list

datatype state = INTERMEDIATE of intermediate_state
               | FINAL        of bool

fun single_step (EVEN nil)           = FINAL        true
  | single_step (EVEN (true  :: bs)) = INTERMEDIATE (ODD bs)
  | single_step (EVEN (false :: bs)) = INTERMEDIATE (EVEN bs)
  | single_step (ODD  nil)           = FINAL        false
  | single_step (ODD  (true  :: bs)) = INTERMEDIATE (EVEN bs)
  | single_step (ODD  (false :: bs)) = INTERMEDIATE (ODD bs)

fun driver_loop (FINAL b)         = b
  | driver_loop (INTERMEDIATE is) = driver_loop (single_step is)

fun main bs = driver_loop (INTERMEDIATE (EVEN bs))
```

A big-step abstract machine A big-step representation contains no driver loop. Rather, the machine states are mutually tail-recursive, and continues the normalization process until a final or stuck state is reached. The representation looks as follows:

```
fun even nil            = true
  | even (true  :: bs) = odd bs
  | even (false :: bs) = even bs
and odd  nil            = false
  | odd  (true :: bs)  = even bs
  | odd  (false :: bs) = odd bs

fun main bs = even bs
```

2.4 Summary and conclusion

This chapter has introduced the notion of term rewriting as a bare-bones version of an operational semantics. Additionally, we have introduced the two styles of operational semantics that form the foundations of this work, namely reduction semantics and abstract machines. Since we are interested in manipulating functional representations of these styles of semantics, and we have therefore also described the nature of these representations. Term rewriting will serve as a stepping stone towards obtaining the representation of a reduction semantics.

The topic of the present work is how to make refocusing applicable for outermost reduction strategies, and to this end, this chapter has introduced a formal definition of outermost reduction strategies. The challenge of making refocusing applicable in this setting is the existence of backward-overlapping reduction rules, and we have therefore introduced a formal definition of backward overlaps, and of the dual notion of forward overlaps. One of the results of the present work is based on the notion of backward chains, of which we have also given a formal definition.

In the following chapters we present the program transformations that we use to link the representations of semantics that have been presented in this chapter.

Chapter 3

A Useful Type Isomorphism

This chapter is dedicated to the type isomorphism $A + B \to C \cong (A \to C) \times (B \to C)$. This isomorphism will be useful in the prelude to reduction semantics (Section 4.1) and in the functional correspondence (Chapter 5).

We are specifically interested in the isomorphism when a direct-style function returns a sum. In this case, the continuation in its CPS counterpart can be split into two – one for each summand [29]. If one of the summands is used for an exceptional situation, then the domain of answers can be instantiated and the exceptional continuation can be lambda-dropped and inlined. We can therefore lambda-drop and inline the exceptional continuation. The resulting program uses the remaining continuation to drive the computation. Should the exceptional case arise, the remaining continuation is not applied and the computation stops.

Unlike the CPS-transformation, which is fully formalized, the type isomorphism presented in this chapter is only intended as a guideline. The proof obligation still exists at the term level on a case-by-case basis. However, in our specific use of the isomorphism, the correctness of the individual transformations that manifest the isomorphism is obvious, so we will not give a detailed correctness proof.

3.1 An example from programming

For example, consider the archetypal program mapping a tree of integers

```
datatype tree = LEAF of int
              | NODE of tree × tree
```

into the product of these integers. If one of these integers is zero (the exceptional situation), we know there and then that the result of the program is zero. In anticipation, we can use the option type and define an auxiliary function

```
(*  prod0_aux : tree → int option  *)
fun prod0_aux (LEAF 0)
    = NONE
  | prod0_aux (LEAF n)
    = SOME n
  | prod0_aux (NODE (t1, t2))
    = (case prod0_aux t1
         of NONE
            ⇒ NONE
          | (SOME p1)
```

```
            ⇒ (case prod0_aux t2
                  of NONE
                     ⇒ NONE
                   | (SOME p2)
                     ⇒ SOME (p1 × p2)))
```

and use it to carry out the multiplications:

```
fun prod0 t (* : tree → int *)
    = (case prod0_aux t
         of NONE   ⇒ 0
          | SOME p ⇒ p)
```

After CPS-transformation (which is described in detail in Section 5.1), the auxiliary function looks like this:

```
(*  prod1_aux : tree × (int option → α) → α  *)
fun prod1_aux (LEAF 0, k)
    = k NONE
  | prod1_aux (LEAF n, k)
    = k (SOME n)
  | prod1_aux (NODE (t1, t2), k)
    = prod1_aux (t1,
        fn NONE
           ⇒ k NONE
         | (SOME p1)
           ⇒ prod1_aux (t2,
                fn NONE
                   ⇒ k NONE
                 | (SOME p2)
                   ⇒ k (SOME (p1 × p2))))
```

Note how the co-domain of answers is generic.

We can use this function to carry out the multiplications by supplying an initial continuation:

```
fun prod1 t (* : tree → int *)
    = prod1_aux (t, fn NONE   ⇒ 0
                    | SOME p ⇒ p)
```

We now split the sum-expecting continuation into two continuations, one for each summand, using the type isomorphism:

```
(*  tree × (unit → α) × (int → α) → α  *)
fun prod2_aux (LEAF 0, k0, kp)
    = k0 ()
  | prod2_aux (LEAF n, k0, kp)
    = kp n
  | prod2_aux (NODE (t1, t2), k0, kp)
    = prod2_aux (t1, k0, fn p1 ⇒ prod2_aux (t2, k0, fn p2 ⇒ kp (p1 × p2)))
```

We then supply two initial continuations to carry out the multiplications:

```
fun prod2 t (* : tree → int *)
    = prod2_aux (t, fn () ⇒ 0, fn p ⇒ p)
```

Finally, we can lambda-drop and inline the first continuation, thereby instantiating the domain of answers from being generic to being the type of the result of the multiplication:

```
(*  tree × (int → int) → int  *)
fun prod3_aux (LEAF 0, k)
```

```
    = 0
  | prod3_aux (LEAF n, k)
    = k n
  | prod3_aux (NODE (t1, t2), k)
    = prod3_aux (t1, fn p1 ⇒ prod3_aux (t2, fn p2 ⇒ k (p1 × p2)))
```

We can then supply one initial continuation to carry out the multiplications:

```
fun prod3 t (* : tree → int *)
    = prod3_aux (t, fn p ⇒ p)
```

The final version of program uses the continuation to drive the traversal of the input tree. If a zero-leaf occurs, the continuation is not applied and the traversal stops. Otherwise, the continuation is applied and the traversal is carried on.

3.2 An example from semantics

As another example, consider the following data type representing subtraction expressions over natural numbers:

```
datatype expr = LIT of int (* assumed to represent a natural number *)
              | SUB of expr × expr
```

An evaluator (corresponding to `prod0` in the previous section) for this expression language looks like this:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string

(*  expr → result_or_wrong  *)
fun evaluate_expr (LIT n)
    = RESULT (NAT n)
  | evaluate_expr (SUB (e1, e2))
    = (case evaluate_expr e2
         of (RESULT (NAT n2))
            ⇒ (case evaluate_expr e1
                 of (RESULT (NAT n1))
                    ⇒ if n1 < n2
                      then WRONG ("underflow: " ^
                                  (toString n1) ^ " - " ^ (toString n2))
                      else RESULT (NAT (n1 - n2))
                  | (WRONG s)
                    ⇒ WRONG s)
          | (WRONG s)
            ⇒ WRONG s)

(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr e
```

Since the subtraction of two natural numbers is not necessarily a natural number, evaluating these expressions can go wrong. Whenever evaluation goes wrong, we are in an exceptional situation where the evaluator must return immediately.

After CPS transforming the evaluator, we can split the `RESULT` and `WRONG` continuations, and then lambda-drop and inline the `WRONG` continuation. We then obtain the following evaluator (corresponding to `prod3` in the previous section):

```
(*  expr × (value → result_or_wrong) → result_or_wrong  *)
fun evaluate_expr (LIT n, k)
    = k (NAT n)
  | evaluate_expr (SUB (e1, e2), k)
    = evaluate_expr (e2, fn (NAT n2) ⇒
        evaluate_expr (e1, fn (NAT n1) ⇒
          if n1 < n2
          then WRONG ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
          else k (NAT (n1 - n2))))

(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr (e, fn v ⇒ RESULT v)
```

Note how the evaluator returns errors directly, without applying a continuation.

3.3 Summary and conclusion

This chapter has introduced a type isomorphism that allows us to split the continuation of a function returning a sum type into a function with two continuations, one for each type summand.

This type isomorphism has several applications outside of the scope of the present work. First, it provides a general utility to shortcut evaluation in case an exceptional situation arises, as illustrated in the example. Second, it provides the link between a representation of structural operational semantics [99] and a representation of reduction semantics, as illustrated in Appendix A.

In the present work, we use the type isomorphism in two places:

- In Section 4.1, we use it in the prelude to reduction semantics, which transforms a simple search function to a representation of a reduction semantics. By using the type isomorphism, we ensure that if a search function finds a potential redex, that potential redex is returned directly rather than through the continuation.
- In Chapter 5, we use it in the functional correspondence, which transforms between representations of abstract machines and big-step normalizers. By using the type isomorphism, we ensure that whenever normalization goes wrong, an error value is returned directly rather than through the continuation.

Chapter 4

The Syntactic Correspondence

Biernacka and Danvy's syntactic correspondence [13] is a sequence of program transformations that converts the representation of a small-step semantics into a representation of an abstract machine. Hence, the syntactic correspondence links small-step semantics to abstract machine by means of program transformations. The syntactic correspondence specifically applies to representations of a reduction semantics, which is a form of small-step semantics where reduction contexts have a syntactic specification.

Of particular significance is that the sequence of transformations is the same for every reduction semantics it is applied to. The syntactic correspondence can therefore be said to capture a structural connection between reduction semantics and abstract machines, rather than simply being an ad hoc way to derive a representation of a particular abstract machine from the representation of a particular reduction semantics.

We start this chapter by reviewing Danvy and Zerny's prelude to reduction semantics, which is a way to derive the representation of a reduction semantics from a grammar of terms and a reduction strategy (Section 4.1). We then describe each of the transformations that constitute the syntactic correspondence: refocusing (Section 4.2), lightweight fusion and inlining (Section 4.3), transition compression (Section 4.4), and context specialization (Section 4.5). In each section we describe how the transformation is usually applied in the syntactic correspondence, and how the transformation must be adjusted to apply for outermost reduction. We summarize and conclude in Section 4.6.

4.1 Prelude to reduction semantics

The *prelude to reduction semantics* developed by Danvy and Zerny [47] is a sequence of program transformations that can be used to derive the representation of a reduction semantics. We use the prelude because grammars of reduction contexts are non-trivial to specify by hand. The prelude allows us to specify just the grammar of terms, the reduction rules and a compositional *search function* that implements the desired reduction strategy. From these three components we are able to derive the remaining components of the representation, as described in Section 2.2.

The reason for using the prelude is two-fold: Firstly, writing down the search function forces the semanticist to spell out the reduction strategy completely. Secondly, once the search function is written down, the grammar of reduction contexts can be methodically derived, rather than invented by hand and then proven correct. The reason for this is that the

(implicit) evaluation context of the search function corresponds to the reduction contexts of the reduction semantics. The prelude derivation makes these contexts explicit, and this explicit representation corresponds to a specification of the reduction contexts of the reduction semantics.

4.1.1 The prelude for innermost reduction

The search function implements the reduction strategy. It traverses a term according the reduction strategy, in search of a potential redex. When the search function finds a potential redex, the redex is returned immediately. If no potential redex is found, the term is a value, and that value is returned. Thus, the search function induces the data type of values as the type of objects returned by the search function if no potential redex is found.

In order to introduce the remaining parts of the representation (the grammar of reduction contexts and the functions *decompose_term* and *decompose_cont*), we now proceed in 5 steps:

1. We CPS transform the search function (CPS transformation is described in detail in Section 5.1).

2. We observe that the continuations introduced by the CPS transformations are applied to either potential redexes or values. We therefore exploit the type isomorphism described in Chapter 3 to split each continuation into two; one for values and one for potential redexes.

3. We observe that the continuations that are applied to potential redexes are always the identity function. We therefore eliminate these continuations from the function, and return the potential redexes directly.

4. We defunctionalize the value continuations (defunctionalization is described in detail in Section 5.2).

5. We make both the apply function for value continuations (which was introduced by defunctionalization) and the search function return not just the potential redex that was found, but also the first-order representation of the value continuation (which was also introduced by defunctionalization).

The resulting program has three components: The data type of defunctionalized value continuations represents to the grammar of reduction contexts. The apply function implements the *decompose_cont* function. The search function implements the *decompose_term* function. Thus, we have obtained the remaining components of a representation of a reduction semantics.

The five steps of the prelude are mechanical and minimalistic, and the derived grammar of contexts has the unique decomposition property.

4.1.2 The prelude for outermost reduction

For outermost reduction strategies the prelude does not always work. The reason for this discrepancy is not quite clear, so we defer the discussion to Section 8.3.

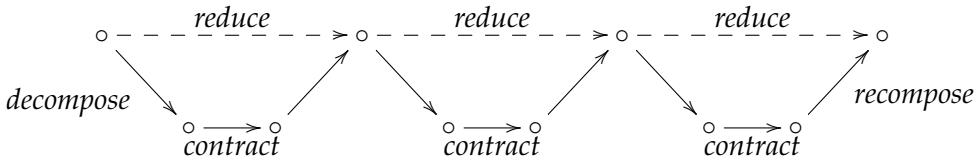
In order to use the prelude for outermost reduction strategies, we therefore sometimes use adjusted versions:

- In Section 6.4 we simply use the grammar of contexts derived from an innermost reduction strategy. The grammar of reduction contexts derived like this does not exhibit unique decomposition. However, the grammar is sufficiently accurate to allow for a reduction semantics to be represented.
- In Section 7.3 the unadjusted prelude suffices, even though the reduction rules exhibit backward overlaps. (See Section 8.1 for a discussion about which backward overlaps are problematic).
- In Section 7.4 and Section 7.5 we first construct an adjusted set of reduction rules using backward chaining, and then use a two-stage version of the prelude. The grammar of reduction contexts derived like this does exhibit unique decomposition, but cannot be performed without some prior knowledge of the grammar of contexts. This knowledge is partially obtained from the backward chaining construction from Section 2.1.

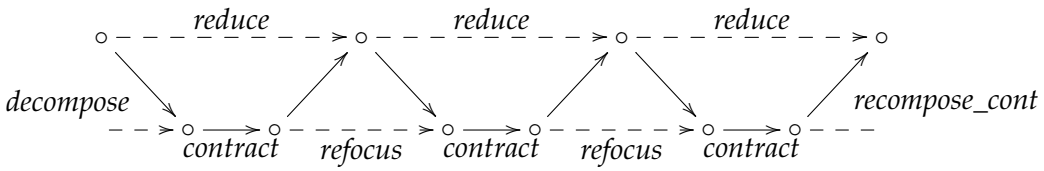
4.2 Refocusing

Danvy and Nielsen's *refocusing* is a transformation that is applied to a representation of a reduction semantics [43]. The purpose of refocusing is to eliminate the recomposition and subsequent decomposition of each reduct in the reduction sequence.

Recall the diagram that describes the functions of our representation of a reduction semantics:



The normalizer constructs each reduct in the reduction sequence by recomposing each contractum into its context. Refocusing eliminates the recomposition and part of the subsequent decomposition of each reduct, which makes the normalizer go directly from redex site to redex site instead of constructing each reduct:



4.2.1 Refocusing for innermost reduction

Refocusing is obtained by replacing every application of *recompose* and the subsequent application of *decompose* by an application of *decompose_term*. The initial application of *decompose* can be composed with a vacuous call to *recompose* (recomposing the initial term into the empty context), and therefore refocusing eliminates all calls to *recompose* and *decompose*.

Refocusing is meaning-preserving only if the *decompose* function always returns to the position of the last contraction. In other words, if *decompose* returns the decomposition consisting of the potential redex pr and the reduction context C, and pr reduces to the contractum c by a reduction rule, then refocusing is meaning-preserving if

$$decompose(recompose(\mathrm{C}, c)) = decompose\_term(c, \mathrm{C})$$

This condition is known to hold for reduction semantics that have the unique decomposition property, and which use an innermost reduction strategy [43].[1]

4.2.2 Refocusing for outermost reduction

For outermost reduction strategies, the existence of backward overlaps can invalidate the refocusing condition. If contraction occurs at position p using a backward overlapping rule, then a new potential redex might be created at position p_0, where p_0 is a prefix of p. This new potential redex is more outermost than any potential redex in the contractum (which still resides at position p). Decomposing the contractum in its context might find a potential redex somewhere in the contractum, whereas decomposing the reduct from its root will find the potential redex at position p_0. Therefore, the refocusing condition does not hold for outermost reduction strategies in the presence of backward overlaps, and hence, refocusing is not in general meaning-preserving in these circumstances.

We are aware of two case studies in the literature where refocusing has been applied successfully without any adjustment to a reduction semantics with backward overlaps using an outermost reduction strategy:

- Weak-head normalization of the λ-calculus using call by name [13, 43]. Call by name is an outermost reduction strategy, and the β-rule backward overlaps with itself:

  $$(\lambda x_1.t)\ t_1 \mapsto t[t_1/x_1]$$

  If t has the form $\lambda x_2.t'$ and the β-redex occurs as the left subterm of an application, contraction gives rise to a backward overlap. However, since the contractum $\lambda x_2.t'$ is a value, the reduction strategy does not allow the contraction of any potential redex that occurs as a subterm of the contractum. Therefore, decomposing the contractum results in a value, and the *decompose* function moves upwards in the term to find the newly constructed redex.[2] Therefore, refocusing can be applied in this case.

- Outermost tree flattening using the associativity rule [31, Sections 10-11]. The associativity rule backward overlaps with itself:

  $$(t_1 \cdot t_2) \cdot t_3 \mapsto t_1 \cdot (t_2 \cdot t_3)$$

[1] Danvy and Nielsen do not mention innermost reduction as an assumption of their proof [43]. However, once it has been checked that a subterm is not a potential redex, their proof assumes that that subterm is a value. This assumption is equivalent to assuming an innermost reduction strategy.

[2] Effectively, this phenomenon allows call by name to be implemented using an innermost reduction strategy. In a similar fashion, the Glasgow Haskell Compiler implements Haskell (i.e., call-by-need λ-calculus) using an innermost reduction strategy.

If a redex occurs as the left subterm of a tree node, contraction gives rise to a backward overlap. However, if this is the case, then the term before contraction is of the form

$$((t_1 \cdot t_2) \cdot t_3) \cdot t_4$$

This outermost redex of this term is rooted at the node with t_4 as its right subterm, so the backward overlap only occurs when contracting a redex that is not outermost. Therefore, refocusing can be applied in this case. For the same reason, refocusing can be applied without problems in Section 7.3, even though that case study (negational normalization of Boolean terms) also exhibits backward overlaps.

In the present work we propose two techniques for making refocusing applicable for outermost reduction in general:

- Backtracking (Chapter 6). Whenever a backward overlapping rule is applied, rather than recomposing the contractum into the entire context, only recompose the contractum into a part of the context. The number of steps to recompose can be determined by analyzing the reduction rules that exhibit backward overlaps.
- Rule generalization based on backward chaining (Chapter 7). Backward overlapping rules are generalized to capture not just the potential redex but also the part of the context that may give rise to backward overlaps. The position of the potential redex of a generalized rule is defined as the top of this part of the context. Therefore, outermost reduction by the generalized rules captures the maximal context that can give rise to a backward overlap, and hence, backward overlaps no longer prevent refocusing.

In addition to the solutions proposed in the present work, we are aware of two cases where refocusing is applied to a reduction semantics with outermost reduction and backward overlaps. In both cases, the language being studied is the full normalization of the $\lambda\hat{\rho}$-calculus using normal-order reduction:

- Danvy, Millikin and Munk [48, 84, 89] make one of the reduction rules applicable only in certain contexts. Therefore, if a potential redex of that rule is found, the normalizer must first check that it is in a legal context for that rule. Effectively, this is another way to perform backtracking. However, it is not obvious if or how this solution scales to other reduction semantics, since their form of backtracking occurs before contraction rather than after. Also, the reduction rule that is inapplicable in certain contexts is not the one that exhibits the backward overlap. Hence, it is not clear how to identify the problematic reduction rules.
- García-Pérez and Nogueira [59, 60] develop a notion of hybrid reduction strategy, where the reduction strategy is specified using a "hybrid" strategy, and a "subsidiary" strategy which the hybrid strategy relies on. As with Danvy, Millikin and Munk's solution, one of the reduction rules is inapplicable in the subsidiary strategy. Additionally, the hybridization means that their refocus function must dispatch between the two decompose functions, which is done by pattern-matching on the immediate context. Furthermore, their refocus function silently performs a backtracking step if necessary.

Since the hybrid approach requires a redefinition of the refocus function, it is not obvious that the hybrid approach to refocusing captures the same structural connection between reduction semantics and abstract machines as the one captured by standard refocusing.

Although the refocus step itself is different, the hybrid approach to refocusing has some similarities to the backward chaining approach proposed in Chapter 7. We discuss these similarities in Section 8.4.

Neither of these case studies identify backward overlaps as the underlying issue.

4.3 Lightweight fusion and inlining

We now inline the *contract* function. The resulting normalizer consists of the functions *decompose_term* and *decompose_cont* (both of which perform only tail calls), and the iterator function which acts as a driver loop for the normalization process. The normalizer is therefore a small-step representation of an abstract machine.

To obtain a big-step representation, we apply Ohori and Sasano's *lightweight fusion by fixed-point promotion* [93], which is an optimization technique for fusing recursive functions. Given the definitions of recursive functions f and g, lightweight fusion by fixed-point promotion calculates a function h that is equivalent to $g \circ f$, but where the fixed-point has been moved outside of f and g. In other words, h does not compute the intermediate data structure produced by f before applying g, and lightweight fusion can therefore be seen as a form of inlining of recursive functions.

4.3.1 Lightweight fusion and inlining for innermost reduction

We use lightweight fusion to fuse the iterator with the decompose functions. The resulting function traverses the input term until it finds a decomposition, then contracts the potential redex in the decomposition (if it is an actual redex), and finally *decompose_term* on the contractum and the reduction context of the decomposition. Hence, the effect of the iterator is now modeled by direct recursion in the decompose functions.

The result is that the call to *decompose_term* that used to appear in non-tail positions in the main loop has now been pushed to tail positions in *decompose_term* or *decompose_cont*. Therefore, all calls are now tail calls, and the iterator is no longer used. The normalizer is therefore a big-step representation of an abstract machine.

4.3.2 Lightweight fusion and inlining for outermost reduction

The choice of reduction strategy does not affect how we apply lightweight fusion and inlining. However, in both the proposed solutions in the present work, after inlining *contract* and fusing the iterator and the decompose functions, the resulting normalizer contains non-tail calls to *recompose*. We now describe the causes of these non-tail calls, and how they are eliminated:

- Backtracking: After each contraction, *decompose_term* is applied to the result of an application of the *backtrack* function. When fusing the iterator, the non-tail call to *backtrack* and the tail call to *decompose_term* are collectively pushed to tail positions

in *decompose_term* and *decompose_cont*. The call to *backtrack* is therefore still in non-tail position, and since the *backtrack* function is the *recompose* function parameterized with the number of steps to *recompose*, the normalizer has a non-tail call to *recompose*.

This non-tail call can be eliminated by inlining the call to *recompose*. Since the number of backtracking steps is known in advance, any recursive calls to *recompose* can be unfolded. Thus, the non-tail call to *recompose* is eliminated.

- Rule generalization by backward chaining: After each contraction, the contractum consists of a term and a delimited context (the part of the context that can cause backward overlaps). To construct a term out of these two components, *contract* must make a call to *recompose*. This call becomes a non-tail call when *contract* is inlined.

  The result of this non-tail call will immediately be decomposed again. We therefore have another case where a call to *recompose* is followed by a call to *decompose*, so we can apply refocusing once more and resume decomposition inside the captured delimited context. This second application of refocusing is sound for the same reason that the first application was sound; the captured delimited context is the result of an application of *decompose*, and since backward overlaps have been eliminated, we can conclude that the term corresponding to this context still does not contain a redex.

  Thus, the non-tail call to *recompose* is eliminated.

4.4 Transition compression

At each transition, pattern matching on the state components determines which transition the abstract machine takes next. But sometimes, a transition moves to a state S where the next transition (to state S′) can be determined without further pattern matching of S. We can therefore shortcut the transition system, and move directly to S′, thereby making the abstract machine more efficient. The resulting abstract machine shortcuts some steps, and is therefore more efficient.

4.4.1 Transition compression for innermost reduction

Typically, transition compression can be applied for the transitions that involve contraction, and in some cases transitions that arise as the result of light-weight fusion of the iterator can also be compressed. Determining which transitions can be compressed is an entirely mechanical process.

4.4.2 Transition compression for outermost reduction

When using backtracking, the transitions involving the *backtrack* function are usually also candidates for transition compression. Other than that, the choice of reduction strategy does not affect how we apply transition compression, nor does it affect the result of transition compression.

4.5 Context specialization

In some cases, the normalizer introduces context constructors for which parts of the components are known. In these cases, we can introduce new context constructors that specialize the existing ones with respect to these known components. The purpose of specializing contexts is that the terms in question will typically need to be decomposed at a later stage.

The resulting abstract machine shortcuts some steps, and is therefore more efficient.

4.5.1 Context specialization for innermost reduction

The typical candidates for context specialization are context constructors where a component is a term for which we know the root constructor. Rather than constructing this new term and then later decomposing it (including the already known term constructor), we can shortcut this decomposition step by introducing the specialized context constructor, and having *decompose_cont* perform the shortcut when it encounters the new context constructor.

4.5.2 Context specialization for outermost reduction

Typically, the choice of reduction strategy has no effect on how context specialization is applied. However, when using rule generalization by backward chaining, we can end up in a situation where a component of a context constructor is the result of a non-tail call to *recompose*. Such a situation arises in Section 7.4 and Section 7.5, because the backward overlapping reduction rule duplicates a term constructor for disjunctions.

The result of this non-tail call to *recompose* is not decomposed immediately, so we cannot apply refocusing to eliminate the non-tail call. However, the result will be decomposed eventually, so we can introduce a specialized context constructor that captures the arguments of *recompose*. Once the result is due for decomposition (i.e., when *decompose_cont* encounters the specialized context constructor), we implicitly apply refocusing, and continue decomposition in the captured context.

When this last case of context specialization has been performed, all calls are finally tail calls, and the normalizer represents a big-step abstract machine. For this reason, we consider context specialization part of the syntactic correspondence, rather than simply an optimization technique for abstract machines.

4.6 Summary and conclusion

This chapter has introduced Biernacka and Danvy's syntactic correspondence between reduction semantics and abstract machines. The correspondence takes the form of transformations that are applied to a representation of a reduction semantics, and we have described each individual transformation: Refocusing, lightweight fusion, inlining, transition compression and context specialization. The result is a big-step representation of an abstract machine.

Additionally, we have introduced Danvy and Zerny's prelude to reduction semantics. The prelude takes the form of transformations that are applied to a search function: CPS transformation, the type isomorphism from Chapter 3, and defunctionalization. The transformations are the same as the ones of the functional correspondence, and so we have deferred the description of CPS transformation and defunctionalization to the following chap-

ter. The result of applying the prelude to a search function is the representation of a reduction semantics.

For innermost reduction strategies, the transformations of the prelude and the syntactic correspondence are entirely methodical. For outermost reduction strategies, however, the existence of backward overlaps require some adjustments to the transformations of the prelude and the correspondence:

- The prelude: In general, the existence of backward overlaps makes the prelude inapplicable. To make the prelude applicable, one must either derive and use the grammar of contexts from the corresponding innermost reduction semantics, or use a two-stage version of the prelude. The former solution is entirely methodical, but in general gives rise to a reduction semantics that does not exhibit unique decomposition. The latter solution does give rise to a reduction semantics exhibiting unique decomposition, but requires some prior knowledge of the structure of the grammar of contexts. This knowledge can be partially obtained using generalized reduction rules based on backward chaining, but this rule generalization is only partially mechanical, and therefore requires some human insight. Starting from the generalized rules, however, the two-stage version of the prelude itself is methodical.

- Refocusing: In general, the existence of backward overlaps makes refocusing inapplicable. To make refocusing applicable, one must either use backtracking, or use the backward chaining construction to generalize the backward overlapping reduction rules. The former solution is entirely mechanical, but in general gives rise to abstract machines that are sub-optimal, and that are not in defunctionalized form (so the functional correspondence does not apply). The latter solution requires a two applications of refocusing (one as the first transformation, and one after transition compression), which is mechanical, but also requires generalized reduction rules, and as mentioned above, rule generalization is only partially mechanical.

- Lightweight fusion: Lightweight fusion is applicable without any adjustments, and the transformation is therefore entirely mechanical.

- Inlining: Inlining is applicable without any adjustments, and the transformation itself is therefore entirely mechanical. However, when using backtracking it is necessary to inline not only the *contract* function but also the *backtrack* function.

- Context specialization: Context specialization is applicable without any adjustments, and the transformation itself is therefore entirely mechanical. However, when using rule generalization, context specialization needs to be generalized to also specialize according to calls to *recompose*. The generalization of context specialization is standard in partial evaluation [22], and is therefore entirely mechanical.

Identifying backward overlaps in the reduction rules is entirely mechanical, and the semanticist can therefore choose which of the proposed solutions to use.

When using backtracking, one must perform an additional analysis of the backward overlaps in order to obtain the number of steps to backtrack. This analysis is also entirely mechanical, and allows for the mechanical introduction of the *backtrack* function in the refocusing step. The only other adjustment needed to the syntactic correspondence is the inlining of the call to *backtrack*, which is also entirely mechanical. The derivation therefore

consists only of mechanical transformations in a predictable sequence, and the derivation is therefore methodical.

When using rule generalization, the actual rule generalization is only partially mechanical. However, once the rules have been generalized, the derivation continues using existing transformations in the usual sequence (with an additional use of refocusing at the end). The derivation itself is therefore methodical, but does rely on a non-mechanical stepping stone.

Chapter 5

The Functional Correspondence

Reynolds's functional correspondence [2, 102] is a sequence of program transformations that converts between representations of big-step semantics and representations of abstract machines. Hence, the functional correspondence links big-step semantics to abstract machines by means of program transformations. The functional correspondence specifically applies to representations of big-step semantics in the form of evaluation functions.

Of particular significance is that the sequence of transformations is the same for every evaluation function it is applied to. The functional correspondence can therefore be said to capture a structural connection between evaluation functions and abstract machines, rather than simply being an ad hoc way to derive a representation of a particular abstract machine from the representation of a particular big-step semantics.

An additional advantage of the functional correspondence is that every transformation in the functional correspondence has a left inverse [25, 41]. This reversibility is of particular interest in the present work, because our starting point is the representation of an abstract machine derived from a small-step semantics using the syntactic correspondence of Chapter 4.

We now describe each of the transformations that constitute the functional correspondence: CPS transformation, along with its left inverse direct-style transformation (Section 5.1), and defunctionalization, along with its left inverse refunctionalization (Section 5.2). In each section we describe how the transformation is usually applied in the functional correspondence, and how the transformation must be adjusted to apply for outermost reduction. We summarize and conclude in Section 5.3.

5.1 CPS transformation and its left inverse

A *continuation* is a functional representation of "the rest of the computation". A *continuation-passing style transformation* or *CPS transformation* is a transformation that makes the current continuation explicitly available to the program at every program point. The current continuation is represented as an anonymous function, which is passed as an extra parameter to the original functions of the program. (For details about the multiple discoveries of continuations and the CPS transformation, we refer to Reynolds's historical account [103] and its addition [40, Section 8]).

If the original function returns a value (i.e., the function causes no computation to take place), the continuation is applied to that value; this causes the rest of the computation to

take place. If the original function makes one or more function calls, these calls are transformed into a single call to the first one of the original calls. The continuation that is passed to this new call is the same as the one passed to the original function, but extended with the rest of the calls that the original function makes. If the continuation is extended with multiple calls, then the extension is CPS transformed as well. Once the first call reaches a value, the remaining calls are performed by the extended continuation before the original continuation is applied; this ensures that the rest of the computation takes place.

As an example of the CPS transformation, consider a Standard ML implementation of the Ackermann function:

```
(* int × int → int *)
fun A(m, n) =
    if m = 0
    then n + 1
    else if n = 0
         then A(m - 1, 1)
         else A(m - 1, A(m, n - 1))
```

For simplicity, we regard applications of the functions +, - and = as simple terms, so we do not transform them. The first consequent is the only branch in which a simple value is returned. In the first alternative branch, both conditional branches contain tail calls. In the second alternative, there is also a recursive call which is not in tail position. The non-tail call is performed first (because of call by value), and the result is used as an actual parameter in tail call.

CPS transformation yields the following function:

```
(* int × int × (int → α) → α *)
fun A(m, n, k) =
    if m = 0
    then k (n + 1)
    else if n = 0
         then A(m - 1, 1, k)
         else A(m, n - 1, fn v' ⇒ A(m - 1, v', k))
```

In the first consequent, the received continuation `k` is applied to the result of the original function. Applying the continuation ensures that "the rest of the computation" is performed when a result is reached. The tail call of second consequent of the original program is passed the received continuation. Since the call was originally a tail call, no further computation occurs in that branch, and so "the rest of the computation" is unchanged for that recursive calls. The second alternative contains the same two calls as before, but they have 'switched places' syntactically. However, they will be performed in the same order as before: The former non-tail call is now a tail call, and the continuation that is passed to this call is the original continuation extended with the original tail call, thereby performing "the rest of the computation" of that branch.

Since it is necessary to identify which original call will be performed first, the exact definition of the CPS transformation depends on the evaluation order of the meta-language (i.e., the language that the un-transformed program is written in) [102]. Our representations are all written in Standard ML, we will use the left-to-right call-by-value CPS transformation [97].

The functional correspondence exploits two inter-related properties of normalizers in continuation-passing style. First, all calls are tail calls, i.e., the value of a function application is either a simple value (requiring no further computation), or is given by the value of exactly

one other function application (requiring only trivial intermediate computation). Second, the CPS transformation makes the evaluation context of the normalizer explicitly available to the normalizer as the current continuation.

5.1.1 The left inverse

The left inverse of the CPS transformation is the *direct-style transformation* [25]. As with CPS transformations, the exact definition of the direct-style transformation depends on the meta-language, and we will use the direct-style transformation specific to left-to-right call-by-value evaluation.

A direct-style transformation is only applicable to programs in continuation-passing style. However, some CPS programs are not in the image of the CPS transformation:

- Programs that sometimes ignore their continuations: These programs ignore the rest of the computation, and return their results directly (e.g., in the case of an error). In these situations one can perform the reverse transformation as the one described in Chapter 3: Introduce an identity function and lambda-lift it (thereby introducing an exceptional continuation), and then apply the type isomorphism from right to left. This puts the program in the image of the CPS transformation, and it can then be direct-style transformed.

- Programs that apply another continuation than the current one: These programs ignore the "current" rest of the computation, and instead apply a rest of the computation that was captured at a previous program point. Programs of this type can be direct-style transformed using the control operator `callcc` [38].

- Programs that apply continuations in non-tail positions: These programs perform the rest of the computation, but then perform additional computations on the result. In effect, such continuations capture only a part of the rest of the computation. For this reason, we refer to such continuations as *delimited continuations*. Programs of this type can be direct-style transformed using the control operators `shift` and `reset` [34].

In the specific case studies where we use the direct-style transformation (Section 7.4 and Section 7.5), the program in question is in *continuation-composing style*. This means that there is one un-delimited continuation, which represents the rest of the computation, and one delimited continuation. The direct-style transformation of such a program works in two stages: First, the un-delimited continuation is transformed. Second, the delimited continuation is transformed. The resulting program is in direct style, and uses `shift` and `reset`.

5.2 Defunctionalization and its left inverse

Reynolds's *defunctionalization* [102] is a transformation that turns higher-order programs into equivalent first-order programs.

In the transformed program, each anonymous function is represented by a constructor of a new data type, which means that the functions become "firstified" by the transformation. Each constructor holds the values that occur free in the transformed function. To model the application of such a firstified function, a special apply function is also introduced. The

apply function dispatches on the constructor of the data type, and performs the computation of the function that the constructor represents.

As an example, consider the following higher-order implementation of an environment:

```
exception UNBOUND of string

(* string → α *)
val empty = fn x ⇒ raise UNBOUND x

(* α × (α → β) → β *)
fun lookup (x, env)
    = env x

(* 'α × β × ('α → β) → 'α → β *)
fun extend (x, v, env)
    = fn x' ⇒ if x = x'
              then v
              else env x'
```

The program contains two anonymous abstractions, one in the definition of `empty` (containing no free variables), and another in the definition of `extend` (containing the free variables `x`, `v` and `env`).

To defunctionalize the program, we introduce a new data type containing two constructors, one for each of the abstractions. The first abstraction is replaced by the constructor `EMPTY`, and the second abstraction is replaced by the constructor `NON_EMPTY`, which itself contains the three values `x`, `v` and `env`. We also introduce a function `apply`, which dispatches on the introduced data type and performs the duties of each of the anonymous abstractions in the original program.

The result of defunctionalization is as follows:

```
datatype α env = EMPTY
               | NON_EMPTY of string × α × α env

exception UNBOUND of string

(* α env × string → α *)
fun apply (EMPTY, x')
    = raise UNBOUND x'
  | apply (NON_EMPTY (x, v, env), x')
    = if x = x'
      then v
      else apply (env, x')

(* α env *)
val empty = EMPTY

(* string × α env → α *)
fun lookup (x, env)
    = apply (env, x)

(* string × α × α env → α env *)
fun extend (x, v, env)
    = NON_EMPTY (x, v, env)
```

Notice how the introduced data type is isomorphic to the type `(string * 'a) list` with `EMPTY` and `NON_EMPTY` taking the places of `nil` and `::`, respectively. In other words, the result

of defunctionalizing an environment implemented using anonymous abstractions (i.e., higher-order) is isomorphic to an environment implemented using association lists (i.e., first-order).

The functional correspondence exploits that defunctionalization can be used to transform the higher-order continuations of a CPS transformed normalizer into a first-order representation. This means that the defunctionalized normalizer has access to a first-order representation of its evaluation context. A first-order representation of the context enables the interpreter to inspect the context (rather than just to apply it to a value).

The result of defunctionalizing the continuations of a CPS transformed functional interpreter is a small-step representation of an abstract machine.

5.2.1 The left inverse

The left inverse of defunctionalization is *refunctionalization* [41].

We are particularly interested in applying refunctionalization to big-step representations of abstract machines. In these representations, the data type for stack values represents defunctionalized continuations. The apply function (if it exists) is a function of a state where the transitions depend on the top element of the stack.

Refunctionalization is only applicable to programs in defunctionalized form. However, not all small-step representations of abstract machines are in defunctionalized form. This is the case for instance for the abstract machine derived for outermost addition of Peano numbers derived in Section 6.4.

In the specific cases where we use refunctionalization (Section 7.4 and Section 7.5), the abstract machine in question contains two data types and two apply functions, in which case it is necessary to defunctionalize the abstract machine twice. The resulting normalizer is in continuation-composing style.

5.3 Summary and conclusion

This chapter has introduced Reynolds's functional correspondence between big-step semantics and abstract machines. The correspondence takes the form of transformations that are applied to a representation of a big-step semantics, and we have described each individual transformation: CPS transformation, the type isomorphism from Chapter 3, and defunctionalization. The result is a big-step representation of an abstract machine.

Since all the transformations have left inverses, the functional correspondence can be applied in reverse order, starting from a big-step representation of an abstract machine and resulting in the representation of a big-step semantics. Since the starting point in the present work is an abstract machine derived using the syntactic correspondence, we are primarily interested in this reverse order. For this reverse transformation to be applicable, the big-step abstract machine must be in defunctionalized form, i.e., in the image of defunctionalization.

For innermost reduction strategies, the transformations of the functional correspondence are entirely methodical. For outermost reduction strategies, the existence of backward overlaps in the corresponding reduction semantics causes the syntactic correspondence to produce either an abstract machine which is not in defunctionalized form, or which operates using two stacks. For the functional correspondence to be applicable to an abstract machine with two stacks, we must apply each transformation in two stages:

- Refunctionalization: The abstract machine contains two stacks, and in the representation each stack has an associated data type of stack values and a function that dispatches on the contents of that stack. Each of the data types represent a defunctionalized function space, with the associated dispatch function as the associated *apply* function. Each of these data types can be refunctionalized separately, which results in a two-stage version of refunctionalization. Each stage is entirely mechanical, provided that the data type and *apply* function can be identified.

- The type isomorphism from Chapter 3: The refunctionalized abstract machine is in continuation-passing style, but may contain branches where no continuation is applied. As mentioned above, we can introduce a new identity continuation in these branches, lambda-lift it, and use the type isomorphism from right to left to obtain a program that always applies a continuation to its result.

- Direct-style transformation: The representation is now in continuation-composing style, which means that the representation uses both un-delimited and delimited continuations. The representation can be direct-style transformed in two stages by first transforming the un-delimited continuation, and then transforming the delimited continuation. Each transformation is entirely mechanical, since it is simple to distinguish between the delimited continuation and the un-delimited continuation: The co-domain of the un-delimited continuation is polymorphic.

If the abstract machine is in defunctionalized form, the two-stage version of the functional correspondence is a simple extension of the traditional, one-stage correspondence. The transformation is therefore still entirely methodical.

Part II

Publications

Chapter 6

From Outermost Reduction Semantics to Abstract Machine

Presented at the 23rd International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR), 2013 ([37])

with Olivier Danvy

Abstract

Reduction semantics is a popular format for small-step operational semantics of deterministic programming languages with computational effects. Each reduction semantics gives rise to a reduction-based normalization function where the reduction sequence is enumerated. Refocusing is a practical way to transform a reduction-based normalization function into a reduction-free one where the reduction sequence is not enumerated. This reduction-free normalization function takes the form of an abstract machine that navigates from one redex site to the next without systematically detouring via the root of the term to enumerate the reduction sequence, in contrast to the reduction-based normalization function.

We have discovered that refocusing does not apply as readily for reduction semantics that use an outermost reduction strategy and have overlapping rules where a contractum can be a proper subpart of a redex. In this chapter, we consider such an outermost reduction semantics with backward-overlapping rules, and we investigate how to apply refocusing to still obtain a reduction-free normalization function in the form of an abstract machine.

6.1 Introduction

A Structural Operational Semantics [99] is a small-step semantics where reduction steps are specified with a relation. For a deterministic programming language, this relation is a function, and evaluation is defined as iterating this one-step reduction function until a normal form is found, if there is one. This way of evaluating a term is said to be "reduction-based" because it enumerates each reduct in the reduction sequence, reduction step by reduction step. A reduction step from a term t_i to the reduct t_{i+1} is carried out by locating a redex r_i in t_i, contracting r_i into a contractum c_i, and then constructing t_{i+1} as an instance of t_i where c_i replaces r_i. In a Structural Operational Semantics, the context of every redex is represented logically as a proof tree.

A Reduction Semantics [53] is a small-step semantics where the context of every redex is represented syntactically as a term with a hole. To reduce the term t_i to the reduct t_{i+1}, t_i is decomposed into a redex r_i and a reduction context $C_i[\,]$, r_i is contracted into a contractum c_i, and $C_i[\,]$ is recomposed with c_i to form t_{i+1}. Graphically:

$$t_i = C_i[r_i] \rightarrow C_i[c_i] = t_{i+1}$$

A reduction step is therefore carried out by rewriting a redex into a contractum according to the reduction rules, with a rewriting strategy that matches the reduction order and is often reflected in the structure of the reduction context. If the reduction strategy is deterministic, it can be implemented with a function. Applying this decomposition function to a term which is not in normal form gives a reduction context and a potential redex.

Reduction is stuck for terms that are in normal form (i.e., where no potential redex occurs according to the reduction strategy), or if a potential redex is found which is not an actual one (e.g., if an operand has a type that the semantics deems incorrect).

For a deterministic programming language, the reduction strategy is deterministic, and so it yields a unique next potential redex to be contracted, if there is one. Furthermore, for any actual redex, only one reduction rule can apply. Therefore, there are no critical pairs and rewriting is confluent.

The format of reduction semantics lends itself well to ensure properties such as type safety [133], thanks to the subject reduction property from type theory. It also makes it possible to account for control operators and first-class continuations by making the reduction context part of the reduction rules [12, 53]. Today reduction semantics are in common use in the area of programming languages [54, 95].

6.1.1 Reduction-based vs. reduction-free evaluation

Evaluating a term is carried out by enumerating its reduction sequence, reduction step after reduction step:

$$\ldots \rightarrow \overbrace{C_{i-1}[c_{i-1}] = C_i[r_i]}^{t_i} \rightarrow \overbrace{C_i[c_i] = C_{i+1}[r_{i+1}]}^{t_{i+1}} \rightarrow \overbrace{C_{i+1}[c_{i+1}] = C_{i+2}[r_{i+2}]}^{t_{i+2}} \rightarrow \ldots$$

This reduction-based enumeration requires all of the successive reducts to be constructed, which is inefficient. So in practice, alternative, reduction-free evaluation functions are sought, often in the form of an abstract machine, and many such abstract machines are described in the literature.

Over the last decade, Danvy and his students have been putting forward a methodology for systematically constructing such abstract machines [31, 32, 43]: instead of recomposing the reduction context with the contractum to obtain the next reduct in the reduction sequence and then decomposing this reduct into the next potential redex and its reduction context, we simply continue the decomposition of the contractum in its reduction context, as depicted with a squiggly arrow:

$$\ldots \rightarrow C_{i-1}[c_{i-1}] \rightsquigarrow C_i[r_i] \rightarrow C_i[c_i] \rightsquigarrow C_{i+1}[r_{i+1}] \rightarrow C_{i+1}[c_{i+1}] \rightsquigarrow C_{i+2}[r_{i+2}] \rightarrow \ldots$$

This shortcut works for deterministic reduction strategies where after recomposition, decomposition always comes back to the contractum and its reduction context before continuing [43]. In particular, it always works for innermost reduction, and has given rise to a 'syntactic correspondence' between reduction semantics and abstract machines [12, 13].

This syntactic correspondence has proved successful to reconstruct many pre-existing abstract machines as well as to construct new ones [5, 49, 60, 113], even in the presence of control operators [12, 40]. For a class of examples, it applies to all the reduction semantics of Felleisen et al.'s latest textbook [54]. More generally, it concretizes Plotkin's connection between calculi and programming languages [97] in that it mechanizes the connection between reduction order (in the small-step world) and evaluation order (in the big-step world), and between not getting stuck (in the small-step world) and not going wrong (in the big-step world).

That said, we have discovered that for reduction semantics that use an outermost strategy and have backward-overlapping rules [50, 63, 66], refocusing does not apply as readily: indeed after recomposition, decomposition does not always come back to the contractum and its reduction context – it might stop before, having found a potential redex that was in part constructed by the previous contraction. The goal of our work here is to study reduction semantics that use an outermost strategy ("outermost reduction semantics") and that have backward-overlapping rules, and to investigate how to apply refocusing to still obtain an abstract machine implementing a reduction-free normalization function.

6.1.2 Overview

We first illustrate reduction semantics for arithmetic expressions with an innermost reduction strategy (Section 6.2), where all the elements of our domain of discourse are touched upon: BNF of terms; reduction rules and contraction function; reduction strategy and BNF of reduction contexts; recomposition of a context with a term; decomposition of a term either into a normal form or into a potential redex and a reduction context; left inverseness of recomposition with respect to decomposition; one-step reduction as decomposition, contraction, and recomposition; reduction-based evaluation as the iteration of one-step reduction; refocusing; and reduction-free evaluation. We then turn to the issue of overlapping rules (Section 6.3). With respect to refocusing, the only problematic combination of overlaps and strategies is backward-overlapping rules and outermost strategy (Section 6.4). To solve the problem, we suggest to backtrack after contracting a redex, which enables refocusing (Section 6.5). For symmetry, we also consider foretracking (Section 6.6). We then review related work (Section 6.7).

6.2 A simple example with an innermost strategy

We consider a simple language of arithmetic expressions with a zero-ary constructor 0, a unary constructor S, and a binary constructor A. The goal is to normalize a given term into a normal form using only the constructors 0 and S.

Terms: The BNF of terms reads as follows:

$$t ::= 0 \mid S(t) \mid A(t, t)$$

Terms in normal form: The BNF of terms in normal form reads as follows:

$$t^{nf} ::= 0 \mid S(t^{nf})$$

Reduction rules: The BNF of potential redexes reads as follows:

$$\mathrm{pr} ::= A(0, \mathrm{t}_2) \mid A(\mathrm{S}(\mathrm{t}_1^{\mathrm{nf}}), \mathrm{t}_2)$$

The reduction rules read as follows:

$$A(0, \mathrm{t}_2) \mapsto \mathrm{t}_2$$
$$A(\mathrm{S}(\mathrm{t}_1^{\mathrm{nf}}), \mathrm{t}_2) \mapsto \mathrm{S}(A(\mathrm{t}_1^{\mathrm{nf}}, \mathrm{t}_2))$$

Note the occurrence of $\mathrm{t}_1^{\mathrm{nf}}$, which is in normal form, in the left-hand side of the second reduction rule: it is characteristic of innermost reduction.

All potential redexes are actual ones, i.e., no terms are stuck. We can thus implement contraction as a total function:

$$\frac{\mathrm{pr} \mapsto c}{\mathit{contract}(\mathrm{pr}) = c}$$

Reduction strategy: We are looking for the leftmost-innermost redex. This reduction strategy is materialized with the following grammar of reduction contexts:

$$\mathrm{C}[\,] ::= \Box[\,] \mid \mathrm{C}[\mathrm{S}[\,]] \mid \mathrm{C}[A([\,], \mathrm{t})]$$

We obtained this grammar by CPS-transforming a search function implementing the innermost reduction strategy and then defunctionalizing its continuation [47].

Lemma 1 (Unique decomposition)**.** *Any term not in normal form can be decomposed into exactly one reduction context and one potential redex.*

Recomposition: As usual, a reduction context is iteratively recomposed with a term using a left fold, as specified by the following abstract-machine transitions:

$$\langle \Box[\mathrm{t}] \rangle^{\mathrm{rec}} \uparrow \mathrm{t}$$
$$\langle \mathrm{C}[\mathrm{S}[\mathrm{t}]] \rangle^{\mathrm{rec}} \uparrow \langle \mathrm{C}[\mathrm{S}(\mathrm{t})] \rangle^{\mathrm{rec}}$$
$$\langle \mathrm{C}[A([\mathrm{t}_1], \mathrm{t}_2)] \rangle^{\mathrm{rec}} \uparrow \langle \mathrm{C}[A(\mathrm{t}_1, \mathrm{t}_2)] \rangle^{\mathrm{rec}}$$

This abstract machine is a deterministic finite automaton with two states: an intermediate state pairing a context and a term, and a final state holding a term. Each transition corresponds to a context constructor. There is therefore no ambiguity and no incompleteness. Recomposition is defined as the iteration of these transitions:

$$\frac{\langle \mathrm{C}[\mathrm{t}] \rangle^{\mathrm{rec}} \uparrow^* \mathrm{t}'}{\mathit{recompose}(\mathrm{C}, \mathrm{t}) = \mathrm{t}'}$$

Since a context constructor is peeled off at each iteration, making the size of the context decrease, the recomposition function is total.

Decomposition: Likewise, a term is iteratively decomposed in an innermost fashion into a potential redex and its reduction context, as specified by the following abstract-machine transitions:

$$\langle \mathrm{C}[\mathbf{[}0\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[\mathbf{[}0\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{cont}}$$
$$\langle \mathrm{C}[\mathbf{[}S(\mathrm{t})\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[S[\mathbf{[}\mathrm{t}\mathbf{]}]] \rangle^{\mathrm{dec}}_{\mathrm{term}}$$
$$\langle \mathrm{C}[\mathbf{[}A(\mathrm{t}_1, \mathrm{t}_2)\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[A([\mathbf{[}\mathrm{t}_1\mathbf{]}], \mathrm{t}_2)] \rangle^{\mathrm{dec}}_{\mathrm{term}}$$

$$\langle \Box[\mathbf{[}\mathrm{t}^{\mathrm{nf}}\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \mathrm{t}^{\mathrm{nf}}$$
$$\langle \mathrm{C}[S[\mathbf{[}\mathrm{t}^{\mathrm{nf}}\mathbf{]}]] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \langle \mathrm{C}[\mathbf{[}S(\mathrm{t}^{\mathrm{nf}})\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{cont}}$$
$$\langle \mathrm{C}[A([\mathbf{[}0\mathbf{]}], \mathrm{t}_2)] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \mathrm{C}[A(0, \mathrm{t}_2)]$$
$$\langle \mathrm{C}[A([\mathbf{[}S(\mathrm{t}^{\mathrm{nf}})\mathbf{]}], \mathrm{t}_2)] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \mathrm{C}[A(S(\mathrm{t}^{\mathrm{nf}}), \mathrm{t}_2)]$$

This abstract machine is a deterministic pushdown automaton with four states where the context is the stack: two intermediate states pairing a context and a term, and two final states, one for the case where the given term is in normal form, and one for the case where it decomposes into a context and a potential redex. Each transition from the first intermediate state corresponds to a term constructor, and each transition rule from the second intermediate state corresponds to a context constructor. Each transition from the first intermediate state peels off a term constructor, and each transition from the second intermediate state peels off a context constructor. There is therefore no ambiguity and no incompleteness.

Furthermore, each transition preserves an invariant: recomposing the current context with the current term yields the original term.

Given a term to decompose, the initial machine state pairs this term with the empty context. There are two final states: one for terms in normal form (and therefore containing no redex), and one for potential redexes and their reduction context. Decomposition, which is defined as the iteration of these machine transitions, is therefore a total function:

$$\frac{\langle \Box[\mathbf{[}\mathrm{t}\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow^* \mathrm{t}^{\mathrm{nf}}}{\mathit{decompose}(\mathrm{t}) = \mathrm{t}^{\mathrm{nf}}} \qquad \frac{\langle \Box[\mathbf{[}\mathrm{t}\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow^* \mathrm{C}[\mathrm{pr}]}{\mathit{decompose}(\mathrm{t}) = \mathrm{C}[\mathrm{pr}]}$$

A notable property: Due to the invariant of the abstract machine implementing decomposition, the recomposition function is a left inverse of the decomposition function.

One-step reduction: One-step reduction is implemented as, successively, the decomposition of a given term into a potential redex and its reduction context; the contraction of this redex into a contractum; and the recomposition of the reduction context with the contractum:

$$\frac{\langle \Box[\mathbf{[}\mathrm{t}\mathbf{]}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow^* \mathrm{C}[\mathrm{pr}] \qquad \mathrm{pr} \mapsto c \qquad \langle \mathrm{C}[\mathbf{[}c\mathbf{]}] \rangle^{\mathrm{rec}} \uparrow^* \mathrm{t}'}{\mathrm{t} \to \mathrm{t}'}$$

Reduction-based evaluation: A term is evaluated into a normal form by iterating one-step reduction:

$$\frac{\mathrm{t} \to^* \mathrm{t}^{\mathrm{nf}}}{\mathrm{t} \Rightarrow_{\mathrm{rb}} \mathrm{t}^{\mathrm{nf}}}$$

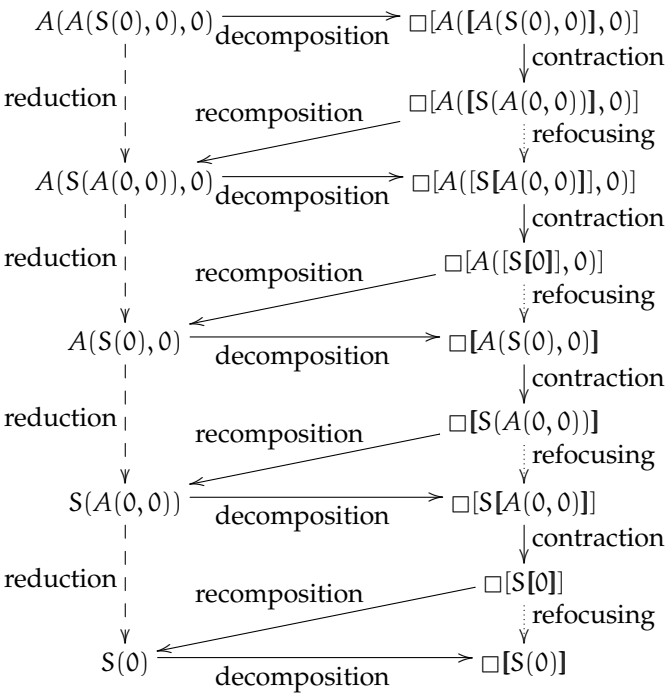


Figure 6.1: Innermost reduction sequence for $A(A(S(0),0),0)$

Towards reduction-free evaluation: Between one contraction and the next, we recompose the reduction context with the contractum until the next reduct, which we decompose into the next potential redex and its reduction context. But since the reduction strategy is innermost (and deterministic), the decomposition of the next reduct will come back to the site of this contractum and this context before continuing. This offers us the opportunity to shortcut the recomposition and decomposition to this contractum and this context and thus to refocus by just continuing the decomposition *in situ*. 5 More formally, we have

$$\frac{t \downarrow^* C[pr] \qquad C[pr]\ ([\mapsto];\downarrow^*)^*\ t^{nf}}{t \Rightarrow_{rf} t^{nf}}$$

where $([\mapsto];\downarrow^*)$ denotes contraction in context followed by decomposition (and was noted $\rightsquigarrow$ in Section 6.1.1).

An example: See Figure 6.1.

Reduction-free evaluation: After applying refocusing, we follow the steps of the syntactic correspondence [12, 13, 31], fusing the iteration and refocus functions, inlining the contract function, and compressing corridor transitions. The resulting normalizer implements a tran-

sition system described by the following abstract machine:

$$\begin{aligned}
\mathrm{t} &\rightarrowtail \langle \Box[\mathrm{t}] \rangle_{\mathrm{term}} \\
\langle \mathrm{C}[0] \rangle_{\mathrm{term}} &\rightarrowtail \langle \mathrm{C}[0] \rangle_{\mathrm{cont}} \\
\langle \mathrm{C}[\mathrm{S}(\mathrm{t})] \rangle_{\mathrm{term}} &\rightarrowtail \langle \mathrm{C}[\mathrm{S}[\mathrm{t}]] \rangle_{\mathrm{term}} \\
\langle \mathrm{C}[A(\mathrm{t}_1, \mathrm{t}_2)] \rangle_{\mathrm{term}} &\rightarrowtail \langle \mathrm{C}[A([\mathrm{t}_1], \mathrm{t}_2)] \rangle_{\mathrm{term}} \\
\langle \Box[\mathrm{t}^{\mathrm{nf}}] \rangle_{\mathrm{cont}} &\rightarrowtail \mathrm{t}^{\mathrm{nf}} \\
\langle \mathrm{C}[\mathrm{S}[\mathrm{t}^{\mathrm{nf}}]] \rangle_{\mathrm{cont}} &\rightarrowtail \langle \mathrm{C}[\mathrm{S}(\mathrm{t}^{\mathrm{nf}})] \rangle_{\mathrm{cont}} \\
\langle \mathrm{C}[A([0], \mathrm{t}_2)] \rangle_{\mathrm{cont}} &\rightarrowtail \langle \mathrm{C}[\mathrm{t}_2] \rangle_{\mathrm{term}} \\
\langle \mathrm{C}[A([\mathrm{S}(\mathrm{t}^{\mathrm{nf}})], \mathrm{t}_2)] \rangle_{\mathrm{cont}} &\rightarrowtail \langle \mathrm{C}[\mathrm{S}[A([\mathrm{t}^{\mathrm{nf}}], \mathrm{t}_2)]] \rangle_{\mathrm{cont}}
\end{aligned}$$

6.3 Backward-overlapping rules

Refocusing (i.e., the short-cutting of recomposition and decomposition after contraction) is possible when, after recomposing a reduction context with a contractum into a reduct, the subsequent decomposition of this reduct comes back to this contractum and context before continuing.

However, there are cases where decomposition of the reduct does not come back to the contractum. For example, this is the case when the reduction strategy is outermost and the contractum is a proper subpart of a potential redex: then after recomposing a reduction context with a contractum into a reduct, the subsequent decomposition of this reduct would *not* come back to this contractum and context—it would stop at the newly created potential redex, above the contractum. So when the reduction strategy is outermost and a contractum can be a subpart of a potential redex, refocusing is not possible.

A contractum can be a subpart of a potential redex when the reduction rules contain *backward overlaps*:

Definition 6 (Backward-overlapping rules)**.** *Let* $\mathrm{l}_1 \rightarrow \mathrm{r}_1$ *and* $\mathrm{l}_2 \rightarrow \mathrm{r}_2$ *be two reduction rules. If* l_1 *decomposes into a non-empty context* C *and a term* t *that contains at least one term constructor and that unifies with* r_2*, then the two rules are said to be* backward-overlapping *[50, 63, 66].*

Symmetrically, if the left-hand side of one reduction rule can form a proper subpart of the right-hand side of another rule, the reduction rules are said to be *forward-overlapping*.

The combination of backward-overlapping rules and outermost reduction does not occur often in programming languages. However, it does occur in the full normalization of λ-terms using normal-order reduction, which has applications for comparing normal forms in proof assistants. Other occurrences can more readily be found outside the field of programming-language semantics, in the area of term rewriting.

We distinguish four cases of reduction strategy in combination with rule overlaps, and treat each of them in the following sections:

	innermost strategy	outermost strategy
forward-overlapping rules	Section 6.3.1	Section 6.3.2
backward-overlapping rules	Section 6.3.3	Section 6.3.4

6.3.1 Forward overlaps and innermost strategy

In this case, a contractum may contain a potential redex. This redex will be found in due course when the contractum is decomposed. The detour via an intermediate reduct can therefore be avoided.

6.3.2 Forward overlaps and outermost strategy

In this case, a contractum may contain a potential redex. This redex will also be found in due course when the contractum is decomposed. The detour via an intermediate reduct can therefore be avoided.

6.3.3 Backward overlaps and innermost strategy

In this case, a contractum may be a proper subpart of a potential redex. However, it should be considered *after* the contractum has been decomposed in search for an innermost redex, which will happen in due course. The detour via an intermediate reduct can therefore be avoided.

6.3.4 Backward overlaps and outermost strategy

In this case, a contractum may be a proper subpart of a potential redex. This potential redex should be considered *before* decomposing the contractum since it occurs further out in the term (i.e., towards its root). Avoiding the detour via an intermediate reduct would in general miss this potential redex and therefore not maintain the reduction order. Does it mean that we need to detour via every intermediate reduct to normalize a term outside-in in the presence of backward overlaps? In this worst-case scenario, reduction-free outside-in normalization would be impossible in the presence of backward overlaps.

It is our observation that this worst-case scenario can be averted: most of the detour via an intermediate reduct can be avoided if we can identify the position of the correct potential redex without detouring all the way to the root.

In the next section, we show how to systematically determine the position of the next potential redex relative to the contractum in the presence of backward overlaps. This extra piece of information makes it possible to move upwards in the term to the position of the potential redex. Most of the detour via the intermediate reduct can therefore be avoided.

6.4 The simple example with an outermost strategy

We now consider the same simple language of arithmetic expressions again, but this time using an outermost reduction strategy.

Terms: The BNF of terms is unchanged:

$$t ::= 0 \mid S(t) \mid A(t, t)$$

Terms in normal form: The BNF of terms in normal form is also unchanged:

$$t^{nf} ::= 0 \mid S(t^{nf})$$

Reduction rules: The BNF of potential redexes now reads as follows:

$$\mathrm{pr} ::= A(0, \mathrm{t}_2) \mid A(\mathrm{S}(\mathrm{t}_1), \mathrm{t}_2)$$

The reduction rules now read as follows:

$$\begin{array}{c} A(0, \mathrm{t}_2) \mapsto \mathrm{t}_2 \\ A(\mathrm{S}(\mathrm{t}_1), \mathrm{t}_2) \mapsto \mathrm{S}(A(\mathrm{t}_1, \mathrm{t}_2)) \end{array}$$

Note the occurrence of t_1, which is not necessarily in normal form, in the left-hand side of the second reduction rule: it is characteristic of outermost reduction.

All potential redexes are actual ones, i.e., no terms are stuck. We can thus implement contraction as a total function:

$$\frac{\mathrm{pr} \mapsto c}{\mathit{contract}(\mathrm{pr}) = c}$$

Reduction strategy: We are looking for the leftmost-outermost redex. We materialize this reduction strategy with the same grammar of reduction contexts as in the innermost case:

$$\mathrm{C}[\,] ::= \Box[\,] \mid \mathrm{C}[\mathrm{S}[\,]] \mid \mathrm{C}[A([\,], \mathrm{t})]$$

As in Section 6.2, we obtained this grammar by CPS-transforming a search function implementing the outermost reduction strategy and then defunctionalizing its continuation.[1]

In contrast to Section 6.2, a term not in normal form can be decomposed into more than one reduction context and one potential redex. For example, the term $A(\mathrm{S}(A(\mathrm{S}(\mathrm{t}_0), \mathrm{t}_1)), \mathrm{t}_2)$ can be decomposed into $\Box[\![A(\mathrm{S}(A(\mathrm{S}(\mathrm{t}_0), \mathrm{t}_1)), \mathrm{t}_2)]\!]$ and $\Box[A([\![\mathrm{S}[\![A(\mathrm{S}(\mathrm{t}_0), \mathrm{t}_1)]\!]], \mathrm{t}_2)]$. The first of these decompositions corresponds to the leftmost-outermost redex of the term, so that decomposition is the one returned by *decompose*. However, this non-unique decomposition puts us outside the validity conditions of refocusing [43], so we are on our own here.

Recomposition: It is defined as in Section 6.2.

Decomposition: A term is decomposed in an outermost fashion into a potential redex and its reduction context with the following abstract-machine transitions:

$$\begin{array}{c} \langle \mathrm{C}[\![0]\!] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[\![0]\!] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \\ \langle \mathrm{C}[\![\mathrm{S}(\mathrm{t})]\!] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[\mathrm{S}[\![\mathrm{t}]\!]] \rangle^{\mathrm{dec}}_{\mathrm{term}} \\ \langle \mathrm{C}[\![A(\mathrm{t}_1, \mathrm{t}_2)]\!] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow \langle \mathrm{C}[\![A(\mathrm{t}_1, \mathrm{t}_2)]\!] \rangle^{\mathrm{dec}}_{\mathrm{add}} \end{array}$$

$$\begin{array}{c} \langle \mathrm{C}[\![A(0, \mathrm{t}_2)]\!] \rangle^{\mathrm{dec}}_{\mathrm{add}} \downarrow \mathrm{C}[A(0, \mathrm{t}_2)] \\ \langle \mathrm{C}[\![A(\mathrm{S}(\mathrm{t}_1), \mathrm{t}_2)]\!] \rangle^{\mathrm{dec}}_{\mathrm{add}} \downarrow \mathrm{C}[A(\mathrm{S}(\mathrm{t}_1), \mathrm{t}_2)] \\ \langle \mathrm{C}[\![A(A(\mathrm{t}_{11}, \mathrm{t}_{12}), \mathrm{t}_2)]\!] \rangle^{\mathrm{dec}}_{\mathrm{add}} \downarrow \langle \mathrm{C}[A([\![A(\mathrm{t}_{11}, \mathrm{t}_{12})]\!], \mathrm{t}_2)] \rangle^{\mathrm{dec}}_{\mathrm{add}} \end{array}$$

$$\begin{array}{c} \langle \Box[\![\mathrm{t}^{\mathrm{nf}}]\!] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \mathrm{t}^{\mathrm{nf}} \\ \langle \mathrm{C}[\mathrm{S}[\![\mathrm{t}^{\mathrm{nf}}]\!]] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \downarrow \langle \mathrm{C}[\![\mathrm{S}(\mathrm{t}^{\mathrm{nf}})]\!] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \end{array}$$

[1] A more precise grammar for contexts exists in the outermost case. It presents the same problems for refocusing as the one used here, and the solution we present also applies to it. Being unaware of any mechanical way to derive a precise grammar for outermost reduction, we therefore present our solution using this less precise but mechanically derivable grammar.

As in Section 6.2, this abstract machine is a pushdown automaton where the context is the stack. This time, the machine has five states: two intermediate states pairing a context and a term, one intermediate state with two terms and a context (this state handles A terms – the A is shown in the transitions above, but can be left implicit in an implementation), and two final states, one for the case where the given term is in normal form, and one for the case where the term decomposes into a context and a potential redex.

Each transition rule from the first intermediate state corresponds to a term constructor. Each transition rule from the second intermediate state corresponds to a term constructor on the left-hand side of an addition. Each transition rule from the third intermediate state corresponds to a context constructor. There is no transition rule to handle A context constructors in the third state, because the machine will move to the second state if it sees a A term constructor, after which the machine is guaranteed to find a potential redex. There is therefore no ambiguity and no incompleteness.

Furthermore, each transition preserves an invariant: recomposing the current context with the current term yields the original term. Given a term to decompose, the initial machine state pairs this term with the empty context. There are two final states: one for terms in normal form (and therefore containing no redex at all), and one for potential redexes and their reduction context. Decomposition, which is defined as the iteration of these machine transitions, is therefore a total function:

$$\frac{\langle \Box[\mathrm{t}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow^* \mathrm{t}^{\mathrm{nf}}}{\mathit{decompose}(\mathrm{t}) = \mathrm{t}^{\mathrm{nf}}} \qquad \frac{\langle \Box[\mathrm{t}] \rangle^{\mathrm{dec}}_{\mathrm{term}} \downarrow^* \mathrm{C}[\mathrm{pr}]}{\mathit{decompose}(\mathrm{t}) = \mathrm{C}[\mathrm{pr}]}$$

A notable property: Due to the invariant of the abstract machine implementing decomposition, as in Section 6.2, the recomposition function is still a left inverse of the decomposition function.

One-step reduction: It is defined as in Section 6.2.

Reduction-based evaluation: It is defined as in Section 6.2.

A backward overlap: The reduction rules contain a backward overlap:

$$A(0, \mathrm{t}_2) \mapsto \mathrm{t}_2$$
$$A(\mathrm{S}(\mathrm{t}_1), \mathrm{t}_2) \mapsto \mathrm{S}(A(\mathrm{t}_1, \mathrm{t}_2))$$

On the right-hand side of both reduction rules, the contractum may occur as the first subterm in the left-hand side of the second rule. Additionally, the contractum of the first rule may occur as the first subterm of the left-hand side of the first rule.

Towards reduction-free evaluation: Between one contraction and the next, we recompose the reduction context with the contractum until the next reduct, which we decompose into the next potential redex and its reduction context.

Contrary to the innermost case, we now cannot be sure that decomposition of the next reduct will come back to this contractum and context before continuing, because a contractum in the context of an addition may be a new redex.

However, we can see from the reduction rules that any new redex constructed in this way cannot be positioned any higher than one step above the contractum, so decomposition will always return at least to the site of this new redex. Hence, if we backtrack/recompose one step after each contraction, we can short-cut the recomposition and decomposition to this point, and just continue the decomposition *in situ*.

More formally, we have

$$\frac{t \downarrow^* C[pr] \qquad C[pr]\ ([\mapsto];\uparrow;\downarrow^*)^*\ t^{nf}}{t \Rightarrow_{rf} t^{nf}}$$

where $([\mapsto];\uparrow;\downarrow^*)$ denotes contraction under context followed by one step of backtracking/recomposition and then decomposition.

The need for backtracking is caused by the existence of backward-overlapping rules. In the present example, we only need to backtrack one step, but in general, multiple steps are needed (Section 6.5 explains how to determine the number of necessary backtracking steps). Our contribution here is that backtracking is sufficient to enable refocusing and therefore reduction-free evaluation.

An example: See Figure 6.2.

Reduction-free evaluation: After applying refocusing, we fuse the iteration and refocus functions, we inline the contract function and the backtracking function, and we compress corridor transitions. The resulting normalizer implements a transition system described by the following abstract machine:

$$\begin{aligned}
t &\rightarrowtail \langle \Box[t] \rangle_{term} \\
\langle C[0] \rangle_{term} &\rightarrowtail \langle C[0] \rangle_{cont} \\
\langle C[S(t)] \rangle_{term} &\rightarrowtail \langle C[S[t]] \rangle_{term} \\
\langle C[A(t_1, t_2)] \rangle_{term} &\rightarrowtail \langle C[A(t_1, t_2)] \rangle_{add} \\
\langle \Box[A(0, t_2)] \rangle_{add} &\rightarrowtail \langle \Box[t_2] \rangle_{term} \\
\langle C[S[A(0, t_2)]] \rangle_{add} &\rightarrowtail \langle C[S[t_2]] \rangle_{term} \\
\langle C[A([A(0, t_2)], t_2')] \rangle_{add} &\rightarrowtail \langle C[A(t_2, t_2')] \rangle_{add} \\
\langle \Box[A(S(t_1), t_2)] \rangle_{add} &\rightarrowtail \langle \Box[S[A(t_1, t_2)]] \rangle_{add} \\
\langle C[S[A(S(t_1), t_2)]] \rangle_{add} &\rightarrowtail \langle C[S[S[A(t_1, t_2)]]] \rangle_{add} \\
\langle C[A([A(S(t_1), t_2)], t_2')] \rangle_{add} &\rightarrowtail \langle C[A(S(A(t_1, t_2)), t_2')] \rangle_{add} \\
\langle C[A(A(t_{11}, t_{12}), t_2)] \rangle_{add} &\rightarrowtail \langle C[A([A(t_{11}, t_{12})], t_2)] \rangle_{add} \\
\langle \Box[t^{nf}] \rangle_{cont} &\rightarrowtail t^{nf} \\
\langle C[S[t^{nf}]] \rangle_{cont} &\rightarrowtail \langle C[S(t^{nf})] \rangle_{cont}
\end{aligned}$$

The effect of backtracking can be seen in the third and sixth transitions of the second intermediate state, where contraction in an addition context gives rise to a new redex above the position of the contractum. In these cases, the machine peels off a context constructor until it reaches the position of the new redex.

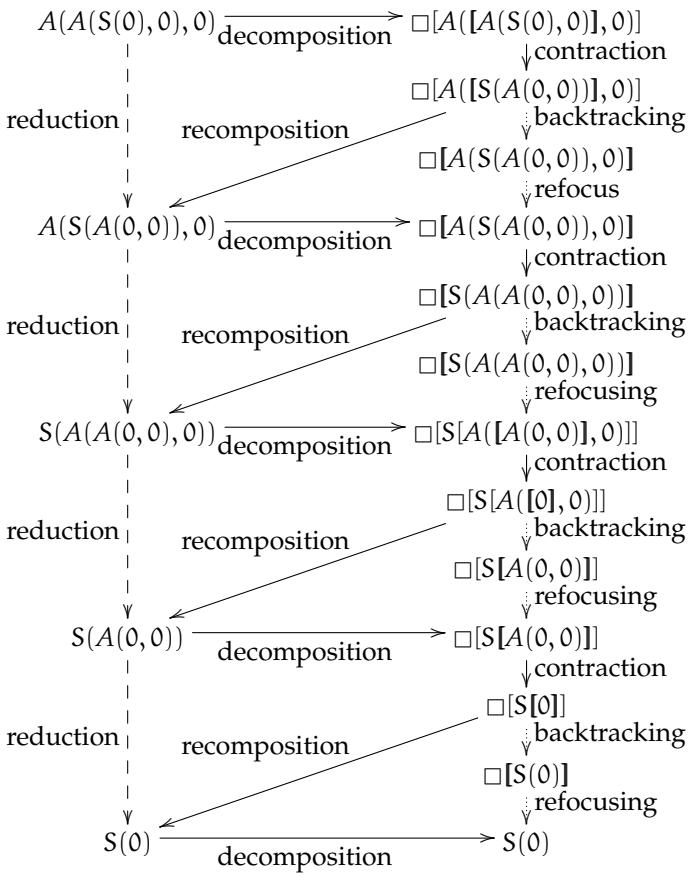


Figure 6.2: Outermost reduction sequence for $A(A(S(0),0),0)$

6.5 Backtracking

6.5.1 Identifying the number of backtracking steps

In our example, it is sufficient to backtrack one step after each contraction. In general, it may be necessary to backtrack further in order to discover a new potential redex and enable refocusing.

For each contractum, the number of steps to backtrack can be determined by analyzing the reduction rules for backward overlaps, i.e., by identifying which subterms of left-hand sides the contractum unifies with. The number of steps to backtrack is the depth of the unifying subterm, i.e., the depth of the hole in the context C of Definition 6. This analysis can be performed statically because the depth of the hole is a property of the reduction rules, not of the reduction strategy. In other words, the analysis is neither performed over the constitutive elements of the normalization function (so no case-by-case semantic manipulation is required) nor during the normalization process (so no extra overhead is introduced).

Determining the existence of backward overlaps is local and mechanical, and hence, so is determining the necessary number of backtracking steps for each contractum. However, rather than determining the number of backtracking steps *for each reduction rule*, we can obtain a conservative estimate of the necessary number of backtracking steps *for all reduction*

rules by

- always using the maximum depth of the left-hand sides of the reduction rules of the system; or by
- always using the maximum depth of the unifying subterms in the backward overlap analysis.

6.5.2 The effect of backtracking on the abstract machine

In practice, the choice of analysis (one precise number of backtracking steps for each reduction rule or one conservative number of backtracking steps for all reduction rules) has little impact on the resulting abstract machine. The reason is that any superfluous backtracking steps introduced in the abstract machine by an overly conservative analysis can be removed by the subsequent transition compressions. The contract function pattern-matches on terms, so after it is inlined, the abstract machine knows a number of term constructors of the contractum. The backtrack function pattern-matches on contexts, so after it is inlined, the abstract machine knows a number of context constructors of the immediate context. Within the window between the top-most known context constructor and the bottom-most known term constructor, transition compression makes the abstract machine move directly to the earliest position (according to the reduction strategy) at which the next redex can be found. Hence, if the context does not give rise to a redex, all the backtracking steps into that context are removed by transition compression.

Still, avoiding superfluous backtracking has two beneficial consequences; first, it simplifies transition compression because lowering the number of backtracking steps reduces the number of cases that need to be considered in the abstract machine. Second, it ensures that backtracking is only performed one new redex pattern at a time, thereby limiting the depth of pattern matching on the context.

6.5.3 Backward overlaps without the need for backtracking

In some cases, the combination of backward overlaps and outermost reduction can be dealt with without backtracking. Two examples in the literature illustrate cases where backtracking is not needed:

- The call-by-name λ-calculus [13, 43]. In this case, the contractum that gives rise to a backward overlap is in normal form: it is a λ-abstraction that occurs on the left of an application node; this application node forms a new β-redex. Decomposing the contractum therefore does not yield a potential redex inside the contractum, and thus the decomposition process moves outwards in the term and finds the newly formed potential redex.
- Outermost tree flattening [31]. In this case, the backward overlap only occurs when contracting a redex which is not outermost, so backtracking is not needed.

6.6 Foretracking

Symmetrically to backtracking, one could envision foretracking as a symmetric solution to innermost reduction in reduction systems with forward overlaps, i.e., where the contraction

may construct a new redex at a lower position than the contractum. However, refocusing is defined as resuming decomposition in the context of the contractum, so the newly constructed redex will be found in due time without using a separate foretracking function.

Additionally, one might envision that foretracking would result in a more efficient abstract machine, because unnecessary decomposition steps could be eliminated by a forward overlap analysis. However, the same superfluous steps are eliminated by transition compression of the abstract machine derived without foretracking.

So all in all, foretracking is not needed to go from an innermost reduction semantics to an abstract machine.

6.7 Related work

Refocusing has mainly been applied for weak reduction in the λ-calculus, for normal order, applicative order, etc. For full reduction in the λ-calculus, a backward overlap exists. As analyzed in Section 6.3, this overlap is only problematic for outermost reduction, e.g., normal order. We are aware of two previous applications of refocusing to full normal-order reduction of the λ-calculus: one by Danvy, Millikin and Munk [48, 84, 89], in the mid-2000's, and a recent one by García-Pérez and Nogueira [59, 60]:

- Danvy, Millikin and Munk make one of the reduction rules applicable only in certain contexts. Therefore, if a potential redex of that rule is found, the normalizer must first check that it is in a legal context for that rule. Effectively, this is another way to perform backtracking. However, it is not obvious if or how this solution scales to other reduction semantics, since their form of backtracking occurs before contraction rather than after. Also, the reduction rule that is inapplicable in certain contexts is not the one that exhibits the backward overlap. Hence, it is not clear how to identify the problematic reduction rules.

- García-Pérez and Nogueira overcome the backward overlap (without identifying it as such) by developing a notion of hybrid strategy and by integrating backtracking in the refocus function. Our solution is more minimalistic and remains mechanical: we simply analyze the reduction rules to detect backward overlaps when the reduction strategy is outermost, and in that case, we backtrack accordingly after contraction and before refocusing.

Backward and forward overlaps have been considered for some 30 years in relation to termination and confluence properties of term rewriting systems [50, 51, 63, 66, 70], and more recently in Jiresch's thesis [73]. Whereas term rewriting studies normalization *relations*, where any potential redex in the term may be contracted, we consider normalization operationally as *functions*, where a deterministic reduction strategy determines which potential redex to contract next.

As mentioned in Section 6.5, refocusing can in some cases be applied without backtracking, even if the reduction semantics contains backward overlaps. A formal definition of backward overlaps for which backtracking is needed would be similar to the definition of *narrowable terms* [126], which is a concept used in term rewriting [78, 118]. However, narrowing is used to solve equations [82], and hence it is unrelated to our goal here.

6.8 Conclusion

We have considered refocusing for reduction semantics with an outermost reduction strategy, and we have discovered that in that case, the original conditions for refocusing [43] are not satisfied. We have then singled out backward-overlapping rules as the only stumbling block towards reduction-free normalization, and we have outlined how to overcome this stumbling block in a systematic way, by analyzing the backward overlaps in the reduction rules. In particular, we have shown how to implement the backtracking function, how to incorporate the backtracking function into the derivation, and how to statically determine the minimal number of backtracking steps, be it relative to each reduction rule or to all of them. We have also shown how to determine whether backtracking is actually necessary.

We have also analyzed all the other combinations (innermost / outermost reduction strategy and forward / backward overlaps in the reduction rules) and demonstrated how refocusing is a simple and effective way to go from reduction-based normalization in the form of a reduction semantics to reduction-free normalization in the form of an abstract machine.

Acknowledgments: We are grateful to the anonymous reviewers for their comments. Thanks are also due to Ian Zerny and Lasse R. Nielsen for their early feedback and kind encouragement.

Chapter 7

A Spectrum of Boolean Normalization Functions

Preliminary version presented as an invited talk at the 20th ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), 2011 ([49])

with Olivier Danvy and Ian Zerny

Abstract

This chapter stems from the 20th anniversary of PEPM. It illustrates semantics-based program manipulation by inter-deriving reduction-based, small-step normalization functions and reduction-free, big-step normalization functions for Boolean terms. We successively consider negational normal forms and conjunctive normal forms. At the beginning of the spectrum, the reduction-based normalization functions proceed in many reduction steps: they explicitly implement double negation and De Morgan's laws (resp. the distributivity of disjunctions over conjunctions) by enumerating the reduction sequence according to a given reduction strategy. The reduction-free normalization functions operate in one pass: they internalize the reduction strategy into an evaluation order and they carry out double negation and De Morgan's laws (resp. the distributivity of disjunctions over conjunctions) implicitly, in passing. At the end of the spectrum, the reduction-free negational normalization functions are compositional and in direct style, and the reduction-free conjunctive normalization functions are compositional, use delimited continuations, and can be expressed in direct style with the delimited-control operators shift and reset. The exponential blowup of conjunctive normal forms can be reduced to linear by sharing delimited continuations in normal forms, yielding a normalization function that is invertible. 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 inter-derived methodically.

7.1 Introduction

The normalization of Boolean terms into negational normal form and the subsequent normalization into conjunctive normal form can be equivalently viewed as a *reduction-based, small-step process*, where the reduction rules are repeatedly applied until a normal form is obtained, and as a *reduction-free, big-step process*, where a given Boolean term is traversed in one pass, using structural recursion. 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 [39].

The goal of this chapter is to inter-derive these normalization processes methodically, using the program transformations in the functional correspondence between evaluators and big-step abstract machines [2, 102] and in the syntactic correspondence between calculi and small-step abstract machines [12], to which we have added a prelude [47].

Overview In Section 7.2, we specify the abstract syntax of Boolean terms, of negational normal forms, and of conjunctive normal forms. In Section 7.3, we inter-derive the normalization of Boolean terms into negational normal form, listing each intermediate step in full detail (a preliminary version of Section 7.3 was presented as an invited talk at the 20th anniversary of PEPM [49]). In Section 7.4, we inter-derive the normalization of negational normal forms into conjunctive normal forms, succinctly reusing the same presentation as in Section 7.3. For emphasis, the presentations of Sections 7.3.1 to 7.3.4 and 7.4.1 to 7.4.4 are deliberately parallel, so that the reader can easily identify what is generic to the methodology and what is specific to each example. In Section 7.5, we show how sharing delimited continuations in the conjunctive normal form conflates it into a size that is linear with respect to the size of the original Boolean term. In Section 7.6, we review related work and we conclude in Section 7.7. Throughout, we use pure ML as a functional meta-language and additionally we make use of the delimited control operators shift and reset for the direct-style normalization function in Section 7.4.5.

7.2 Domain of discourse

Our Boolean terms consist of variables, negations, conjunctions, and disjunctions:

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

Double negation and De Morgan's laws provide *conversion rules* between Boolean terms, where double negations are introduced or eliminated and 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 eliminate double negations and 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 to 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:

$$l ::= x \mid \neg x$$
$$t_{dnf} ::= t_{dnf} \vee t_{dnf} \mid l$$
$$t_{cnf} ::= t_{cnf} \wedge t_{cnf} \mid t_{dnf}$$

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

In the rest of this section, we implement terms and normal forms in ML.

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
withtype ide  = string
```

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

```
(*  (ide → α) × (α → α) × (α × α → α) × (α × α → α) → term → α  *)
fun foldr_term (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:

Proposition 1 (Adequacy of Boolean terms)**.** *There is a bijection between derivable terms in the grammar of Boolean terms and constructible values in the data type* `term`*.*

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

It is out of the scope of this work to formally establish the adequacy of our implementation in ML. (In that, we follow the standard practice of textbooks on formal semantics [71, 91].)

Negational normal forms A Boolean term is in negational normal form when the only subterms that are negated are variables. 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 → α) × (ide → α) × (α → β) × (β × β → β) × (β × β → β)  *)
(*  → term_nnf → β  *)
fun foldr_term_nnf (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
    = foldr_term_nnf (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
    = foldr_term_nnf (VAR, fn x ⇒ NEG (VAR x), fn t ⇒ t, CONJ, DISJ) t
```

The ML encoding of negational normal forms is an adequate representation:

Proposition 2 (Adequacy of negational normal forms)**.** *There is a bijection between derivable terms in the grammar of negational normal forms and constructible 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:

```
(* (literal → α) × (α × α → α) × (α → β) × (β × β → β) *)
(* → conj_cnf → β *)
fun foldr_term_cnf (dleaf, dnode, cleaf, cnode) t
    = let fun visit_d (DISJ_leaf l)         = dleaf l
            | visit_d (DISJ_node (d1, d2)) = dnode (visit_d d1, visit_d d2)
          fun visit_c (CONJ_leaf d)         = cleaf (visit_d d)
            | visit_c (CONJ_node (c1, c2)) = cnode (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
    = foldr_term_cnf (LIT_nnf, DISJ_nnf, fn t ⇒ t, CONJ_nnf) t
```

The ML encoding of conjunctive normal forms is an adequate representation:

Proposition 3 (Adequacy of conjunctive normal forms)**.** *There is a bijection between derivable terms in the grammar of conjunctive normal forms and constructible values in the data type* `term_cnf`*.*

7.3 Leftmost-outermost negational normalization

In this section, we consider negational normal forms. We go from a leftmost-outermost (small-step) *reduction* strategy to the corresponding leftmost-outermost (big-step) *evaluation* strategy. We first implement the reduction strategy (Section 7.3.1) as a prelude to implementing the corresponding reduction semantics (Section 7.3.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 7.3.3) and to the functional correspondence between abstract machines and normalization functions (Section 7.3.4). The resulting normalization function is compositional, and can be variously expressed with the fold functional associated to source terms (Section 7.3.5).

7.3.1 Prelude to a reduction semantics

The reduction strategy along with the notions of value and of potential redex (i.e., an actual redex or one that is stuck) puts us in position to state a compositional search function that implements the reduction strategy and that maps a given term either to the corresponding value, if it is in normal form, or to a potential redex (Section 7.3.1.1). 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 7.3.1.2). As a corollary, we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 7.3.1.3).

7.3.1.1 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

(*  potential_redex → term  *)
fun pr2t (PR_NEG t)          = NEG (NEG t)
  | pr2t (PR_CONJ (t1, t2)) = NEG (CONJ (t1, t2))
  | pr2t (PR_DISJ (t1, t2)) = NEG (DISJ (t1, t2))
```

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 upon to decide whether this negation is a value or a potential redex.

By adequacy of our representation, the search function implements the leftmost-outermost reduction strategy:

Property 1 (soundness)**.** *Let* `t` *represent the term* `t`*, let* `pr` *represent the potential redex pr, and let* `t_nnf` *represent a negational normal form:*

- `search t` *evaluates to* `POTRED pr` *if and only if* `pr` *is the leftmost-outermost redex in* `t`*; and*
- `search t` *evaluates to* `VAL t_nnf` *and* `embed_nnf t_nnf` *evaluates to* `t` *if and only if* `t` *does not contain a redex.*

7.3.1.2 From searching to decomposing

Let us transform the search function of Section 7.3.1.1 into a decomposition function for the reduction semantics of Section 7.3.2. 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 [34, 119] (and then simplifying it one bit) and (2) defunctionalizing its continuation [42, 102].

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 → α) → α  *)
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 → α) → α  *)
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 constructors 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. Transliterating it from ML, the grammar of reduction contexts reads as follows:

$$\mathrm{C} ::= [\,] \mid v \wedge \mathrm{C} \mid \mathrm{C} \wedge \mathrm{t} \mid v \vee \mathrm{C} \mid \mathrm{C} \vee \mathrm{t}$$

NB. This grammar of contexts has been mechanically derived, not invented.

We have defined `apply3_cont` in curried form to emphasize that it maps each constructor to a continuation. In the following, we consider its uncurried definition to manifest that each function is always fully applied. With this uncurried definition, the program above implements a pushdown automaton where the defunctionalized continuation plays the role of the pushdown stack. This automaton has four states: `search3` (the initial state), `search3_term`, `search3_term_neg` (a final state), and `apply3_cont` (another final state). Each transition is carried out with a tail call: this pushdown automaton is implemented as a big-step abstract machine [39].

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 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))
```

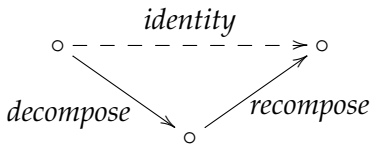
```
  | decompose_term (NEG t,          k) = decompose_term_neg (t, k)
  | decompose_term (CONJ (t1, t2), k) = decompose_term (t1, C2 (k, t2))
  | decompose_term (DISJ (t1, t2), k) = decompose_term (t1, C4 (k, t2))

(*  term → value_or_decomposition  *)
fun decompose t = decompose_term (t, C0)
```

The decompose function implements the decomposition of a term as a leftmost-outermost redex in its reduction context:

Property 2 (soundness)**.** *Let* `t` *represent the term* C[*pr*]*, let* `pr` *represent the potential redex pr, and let* `k` *represent the reduction context* C*:* `decompose t` *evaluates to* `DEC (pr, k)`*.*

7.3.1.3 Recomposing

The recompose function is the left inverse of the decompose function:



We implement it with a left fold over the given reduction context to recompose it around the given term:

```
(*  cont × term → term  *)
fun recompose_cont (C0,          t ) = t
  | recompose_cont (C1 (v1, k), t2) = recompose_cont (k, CONJ (embed_nnf v1,
    t2))
  | recompose_cont (C2 (k, t2), t1) = recompose_cont (k, CONJ (t1, t2))
  | recompose_cont (C3 (v1, k), t2) = recompose_cont (k, DISJ (embed_nnf v1,
    t2))
  | recompose_cont (C4 (k, t2), t1) = recompose_cont (k, DISJ (t1, t2))

(*  value_or_decomposition → term  *)
fun recompose (VAL v)       = embed_nnf v
  | recompose (DEC (pr, k)) = recompose_cont (k, pr2t pr)
```

The recompose function implements the recomposition of a context around a term:

Property 3 (soundness)**.** *Let* `k` *represent the context* C*, let* `t` *represent the term* `t`*, and let* `t'` *represent the term* C[t]*:* `recompose_cont (k, t)` *evaluates to* `t'`*.*

7.3.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for negational normalization.

7.3.2.1 Notion of reduction

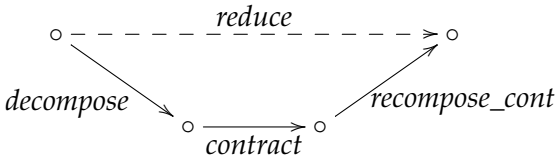
The reduction rules implement double negation and De Morgan's laws:

```
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.

7.3.2.2 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 around the contractum:



If the potential redex is not an actual one, reduction is stuck. Given a value term, reduction is also impossible:

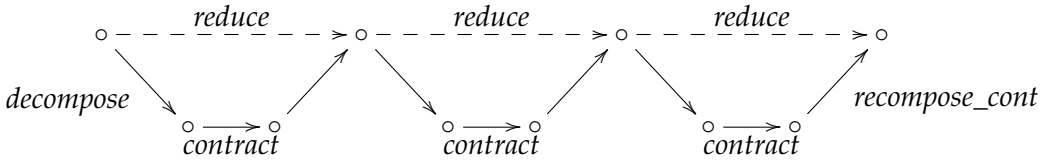
```
datatype reduct_or_stuck = REDUCT of term
                         | STUCK  of string

(*  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_cont (k, t'))
                 | (ERROR s)       ⇒ STUCK s))
```

This one-step reduction function is the hallmark of a reduction semantics [53, 54]:

Property 4 (soundness)**.** *Let* `t` *represent* `t` *and* `t'` *represent* `t'`*:* `reduce t` *evaluates to* `REDUCT t'` *if and only if* `t` *reduces to* `t'` *in one step by leftmost-outermost reduction.*

7.3.2.3 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 normalization, and if it becomes stuck, normalization 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_cont (k, t')))
         | (ERROR s)       ⇒ WRONG s)

(*  term → result_or_wrong  *)
fun normalize t = iterate (decompose t)
```

Property 5 (soundness)**.** *Let* `t` *represent* `t` *and* `v` *represent* ν*:* `normalize t` *evaluates to* `RESULT v` *if and only if* `t` *reduces to* ν *by leftmost-outermost reduction.*

7.3.3 From reduction-based to reduction-free normalization

In this section, we transform the reduction-based normalization function of Section 7.3.2.3 into a spectrum of reduction-free normalization functions. Whereas the reduction-based normalization function enumerates the successive reducts in the reduction sequence, the reduction-free normalization functions do not construct any intermediate reduct.

We first refocus the reduction-based normalization function to deforest the intermediate reducts [43, 116]. The result is a small-step abstract machine implementing the iteration of the refocus function (Section 7.3.3.1). We then inline the contraction function (Section 7.3.3.2), and we apply lightweight fusion [93] to transform this small-step abstract machine into a big-step one [39] (Section 7.3.3.3). This machine exhibits a number of corridor transitions, and we compress them (Section 7.3.3.4). As a postlude to transition compression, we specialize the contexts of the machine (Section 7.3.3.5). The resulting abstract machine is in defunctionalized form [42], and we refunctionalize it [41] (Section 7.3.4.1). The result is in continuation-passing style and we re-express it in direct style [25] (Section 7.3.4.2). The resulting direct-style function is a traditional conversion function for Boolean formulas; in particular, it is compositional.

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.

7.3.3.1 Refocusing

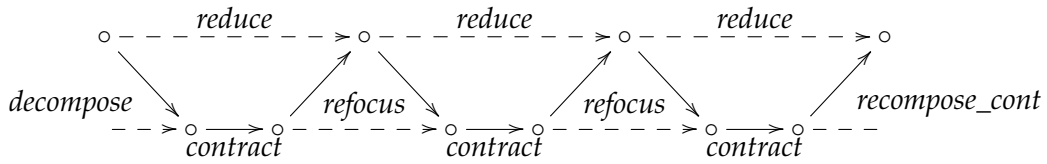
The normalization function of Section 7.3.2.3 is reduction-based because it constructs every intermediate reduct in the reduction sequence. In its definition, `decompose` is always applied to the result of `recompose_cont` after the first decomposition. In fact, a vacuous initial call to `recompose_cont` ensures that `decompose` is always applied to the result of `recompose_cont`:

```
fun normalize t = iterate (decompose (recompose_cont (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_cont (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))
```

Extensionally, the refocus function goes from a redex site to the next redex site, if there is one:



Intensionally, it first recomposes the reduction context around the contractum into the next reduct, and then it decomposes this reduct into the next redex site, if there is one: that is what `refocus0` does, just above. But since the reduction strategy is deterministic, the subsequent decomposition comes back to the current redex site and then, depending on the contractum, either keeps heading toward the next redex site by decomposing the contractum if it is not a value or keeps exploring the context if the contractum is a value – which is what `decompose_term` does. Therefore the next reduct need not be constructed: it can be deforested away by making the refocus function simply *continue the decomposition of the contractum in the current context*, i.e., by defining it as `decompose_term` [43]:

```
(*  term × cont → value_or_decomposition  *)
fun refocus1 (t, k) = decompose_term (t, k)
```

This deforestation technique applies under common conditions (e.g., the grammar of reduction contexts should be context-free) that are satisfied here.

The refocused normalization 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.

7.3.3.2 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))
```

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

The refocused normalization function is a small-step abstract machine [39] implementing a pushdown automaton in which the context is represented using the stack: `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' [57]) keeps activating the inner transition function until a value is obtained. Using Ohori and Sasano's 'lightweight fusion by fixed-point promotion' [93], 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 implementing a pushdown automaton and 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` in the way `decompose_cont` did;

- `iterate3` is a clone of `iterate2` that calls the fused function `normalize3_term`; and

- `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)
```

7.3.3.4 Hereditary transition compression

In the abstract machine of Section 7.3.3.3, 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. We consider each of their clauses in turn:

Clause `normalize3_cont (C0, v)`:

```
normalize3_cont (C0, v)
= (* by inlining normalize3_cont *)
iterate3 (VAL v)
= (* by inlining iterate3 *)
RESULT v
```

Clause `normalize3_term_neg (NEG t, k)`:

```
normalize3_term_neg (NEG t, k)
= (* by inlining normalize3_term_neg *)
iterate3 (DEC (PR_NEG t, k))
= (* by inlining iterate3 *)
normalize3_term (t, k)
```

Clause `normalize3_term_neg (CONJ (t1, t2), k)`:

```
normalize3_term_neg (CONJ (t1, t2), k)
= (* by inlining normalize3_term_neg *)
iterate3 (DEC (PR_CONJ (t1, t2), k))
= (* by inlining iterate3 *)
normalize3_term (DISJ (NEG t1, NEG t2), k)
= (* by inlining normalize3_term *)
normalize3_term (NEG t1, C4 (k, NEG t2))
= (* by inlining normalize3_term *)
normalize3_term_neg (t1, C4 (k, NEG t2))
```

Clause `normalize3_term_neg (DISJ (t1, t2), k)`:

```
normalize3_term_neg (DISJ (t1, t2), k)
= (* by inlining normalize3_term_neg *)
iterate3 (DEC (PR_DISJ (t1, t2), k))
= (* by inlining iterate3 *)
normalize3_term (CONJ (NEG t1, NEG t2), k)
= (* by inlining normalize3_term *)
normalize3_term (NEG t1, C2 (k, NEG t2))
= (* by inlining normalize3_term *)
normalize3_term_neg (t1, C2 (k, NEG t2))
```

As a corollary of the compressions, the definition of `iterate3` is now unused and can be elided. Renaming the indices from `3` to `4`, 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)
```

7.3.3.5 Context specialization

As a postlude to transition compression, we observe that the normalizer sometimes uses context constructors `C2` and `C4` where the second argument is a negation. We therefore introduce specialized context constructors `C2NEG` and `C4NEG` for those cases, and subsequently compress corridor transitions in `normalize5_cont` to directly call `normalize5_term_neg` for the new contexts:

```
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)
```

A consequence of this context specialization is that the definitions of `normalize5_term` and `normalize4_term_neg` are now symmetric.

7.3.4 From abstract machines to normalization functions

In this section, we transform the abstract machine of Section 7.3.3.5 into two compositional normalization functions, one in continuation-passing style (Section 7.3.4.1) and one in direct style (Section 7.3.4.2).

7.3.4.1 Refunctionalization

Like many other big-step abstract machines [2, 31], the abstract machine of Section 7.3.3.5 is in defunctionalized form [42]: 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 → α) → α  *)
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 → α) → α  *)
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 corresponding left-hand side. It can therefore be expressed with the fold functional for terms.

7.3.4.2 Back to direct style

The refunctionalized definition of Section 7.3.4.1 is in continuation-passing style since it has a functional accumulator and all of its calls are tail calls [25]. 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.

7.3.5 Catamorphic normalization functions

As a postlude to the functional correspondence, we observe that the compositional normalization function of Section 7.3.4.2 features two mutually recursive functions from terms to

values. These two functions can be expressed as one, using the following type isomorphism:

$$(A \to B) \times (A \to B) \;\simeq\; A \to B^2$$

Computationally, this isomorphism can be exploited in two ways: by representing B^2 as $2 \to B$, which in ML gives rise to a lazy pair, and by representing B^2 as $B \times B$, which in ML gives rise to an eager pair. Both of these representations are extensionally equal because the normalization functions are total. Let us review each of them.

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 normalization function 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 normalization function is (1) compositional and (2) singly recursive, let us express it as a catamorphism, i.e., as an instance of `foldr_term`:

```
(*  term → (polarity → value)  *)
val normalize9_term
    = foldr_term
        (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 normalization function 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 `foldr_term`:

```
(*  term → value × value  *)
val normalize11_term
    = foldr_term
        (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
```

7.3.6 Implementation and testing

All the elements of the spectrum, from `normalize` and `normalize0` to `normalize11`, are implemented in Standard ML. We have run each of them on an array of tests and verified that they all yield the same results.

7.3.7 Summary and conclusion

We have refocused the reduction-based normalization function of Section 7.3.2.3 into a reduction-free, small-step abstract machine, and we have exhibited a spectrum 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. However, they have not been invented here, but inter-derived in a mechanical way. Proving from scratch 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* [91]. In contrast, each of the derivation steps used here is elementary and methodical.

An additional advantage of the present calculational method is that it is reusable. For example, it is now a simple exercise to consider a rightmost-outermost reduction strategy, or a leftmost-innermost reduction strategy, or a rightmost-innermost reduction strategy, and to calculate the corresponding spectrum of negational normalization functions. In the rest of this chapter, we reuse the method for other reduction semantics: to obtain conjunctive normal forms (Section 7.4), and to obtain conjunctive normal forms with sharing (Section 7.5).

7.4 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 rules for negational normalization, however, the rules for conjunctive normalization do not allow for a straightforward transformation. We therefore begin this section by adjusting the reduction rules for conjunctive normalization (Section 7.4.0). Following the same steps as in Section 7.3, we then implement the reduction strategy for the new rules (Section 7.4.1) as a prelude to implementing the corresponding reduction semantics (Section 7.4.2). We then turn to the syntactic correspondence between reduction semantics and abstract machines (Section 7.4.3) and to the functional correspondence between abstract machines and normalization functions (Section 7.4.4).

7.4.0 Generalized reduction

Consider the two reduction rules provided by the distributivity laws presented in Section 7.2:

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

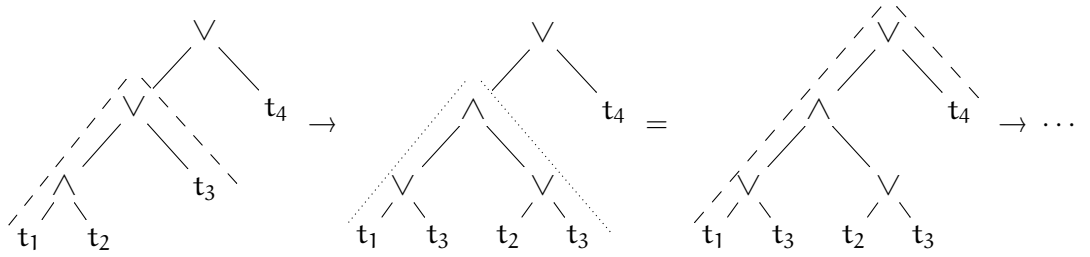
Unlike the reduction rules for negational normalization, these rules overlap in two ways:

1. Redexes overlap because a term can be an instance of several redexes, which is characteristic of *critical pairs*. For example, the term $(t_{nnf1} \wedge t_{nnf2}) \vee (t_{nnf3} \wedge t_{nnf4})$ can be reduced by either rule. For conjunctive normalization, 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.

   Fortunately, we are considering a fixed reduction strategy – leftmost outermost – and therefore there is no ambiguity as to which is the next redex to contract in a non-value term: the reduction strategy can be implemented not just with a relation, but with a function.

2. Redexes and contractums overlap because when a contractum is a conjunction and occurs as an immediate subterm of a disjunction, this disjunction is the root of a new redex, which is characteristic of *backwards overlaps* [50, 66].

   For example, given four terms in negational normal form t_1, t_2, t_3, and t_4, the leftmost-outermost redex of the term $((t_1 \wedge t_2) \vee t_3) \vee t_4$ is the left subterm of the topmost disjunction: it is the dashed subtree below on the left. This term reduces in one step to $((t_1 \vee t_3) \wedge (t_2 \vee t_3)) \vee t_4$. The contractum is a conjunction: it is the dotted subtree below in the middle. The contractum is the left subtree of the top-most disjunction which now forms the next redex to be contracted: it is the dashed tree below on the right.



   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. Hence, it is not meaning-preserving to continue decomposition in the context of the contractum, and refocusing is therefore inapplicable in this situation, unlike in Section 7.3.

   This problem occurs because of 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.

   Leftmost-outermost conjunctive normalization is the first case study we have encountered where refocusing does not directly apply. Subsequently, Danvy and Johannsen have conducted a taxonomic investigation of refocusing for a reduction semantics whose reduction order may not just be innermost, as usual, but also outermost, like here, and where reduction rules may overlap forward or backward [37]. We found out

that the only combination for which refocusing does not apply directly is outermost reduction and backward overlapping rules, which is the situation here. We also showed that it is then sufficient to backtrack in the reduction context on a case-by-case basis to re-enable refocusing.

The issue of backward overlaps also arises when fully normalizing λ-terms using normal order. Danvy, Millikin and Munk [48, 84, 89] overcame this issue (without identifying it as such) by backtracking *after* applying the refocus function. García-Pérez and Nogueira [59, 60] overcame it (without identifying it as such either) by developing a notion of hybrid strategy and by integrating backtracking in the refocus function.

Instead of backtracking, here is a more global solution: eliminating rule overlaps altogether.

1. To eliminate what looks like critical pairs, i.e., overlapping redexes, we restrict the left-hand side of the first reduction rule to $t_{dnf1} \vee (t_{nnf2} \wedge t_{nnf3})$, exploiting the fact that the reduction strategy goes from left to right.

2. To eliminate backward overlaps, i.e., 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:

$$D[t_{nnf1} \wedge t_{nnf2}] \to D[t_{nnf1}] \wedge D[t_{nnf2}] \quad \text{where } D \neq [\,]$$

This generalized rule describes a form of context-sensitive contraction [12] where only a delimited part of the context is used in the contraction. In effect, the generalized contraction hereditarily pulls out a conjunction in one single step, thereby avoiding the construction of intermediate redexes.

The position of a redex of the generalized rule is defined by the position of the top-most constructor of D. Leftmost-outermost reduction by the generalized rule therefore implies that the redex does not reside in the immediate context of a disjunction (if it did, D could be extended upwards, creating a new redex in a more outermost position).

Let us 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 6 (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, meaning that* $t \rightarrowtail C[D[t_{nnf1}] \wedge D[t_{nnf2}]]$*. Then* $t \mapsto^* C[D[t_{nnf1}] \wedge D[t_{nnf2}]]$*.*

Proof. By induction on D:

Case $D = [\,]$ By reflexivity of $\mapsto^*$.

Case $\mathsf{D} = \mathsf{D}_0[[\,] \vee \mathsf{t}_{nnf}]$

$$\begin{array}{ll} & \mathsf{t} \\ = & \mathsf{C}[\mathsf{D}_0[(\mathsf{t}_{nnf1} \wedge \mathsf{t}_{nnf2}) \vee \mathsf{t}_{nnf}]] \\ \mapsto & \mathsf{C}[\mathsf{D}_0[(\mathsf{t}_{nnf1} \vee \mathsf{t}_{nnf}) \wedge (\mathsf{t}_{nnf2} \vee \mathsf{t}_{nnf})]] \\ \mapsto^* & \mathsf{C}[\mathsf{D}_0[\mathsf{t}_{nnf1} \vee \mathsf{t}_{nnf}] \wedge \mathsf{D}_0[\mathsf{t}_{nnf2} \vee \mathsf{t}_{nnf}]] \\ = & \mathsf{C}[\mathsf{D}[\mathsf{t}_{nnf1}] \wedge \mathsf{D}[\mathsf{t}_{nnf2}]] \end{array}$$

Case $\mathsf{D} = \mathsf{D}_0[\mathsf{t}_{dnf} \vee [\,]]$

$$\begin{array}{ll} & \mathsf{t} \\ = & \mathsf{C}[\mathsf{D}_0[\mathsf{t}_{dnf} \vee (\mathsf{t}_{nnf1} \wedge \mathsf{t}_{nnf2})]] \\ \mapsto & \mathsf{C}[\mathsf{D}_0[(\mathsf{t}_{dnf} \vee \mathsf{t}_{nnf1}) \wedge (\mathsf{t}_{dnf} \vee \mathsf{t}_{nnf2})]] \\ \mapsto^* & \mathsf{C}[\mathsf{D}_0[\mathsf{t}_{dnf} \vee \mathsf{t}_{nnf1}] \wedge \mathsf{D}_0[\mathsf{t}_{dnf} \vee \mathsf{t}_{nnf2}]] \\ = & \mathsf{C}[\mathsf{D}[\mathsf{t}_{nnf1}] \wedge \mathsf{D}[\mathsf{t}_{nnf2}]] \end{array}$$

Note that in the proof steps that apply the induction hypothesis (the steps using $\mapsto^*$), the induction hypothesis is applied to the leftmost-outermost redex of the term, as required. □

This generalization of reduction rules is similar to the construction of *backward chains* [50, 66]. However, contrary to the construction of backward chains, it is necessary that our rule generalization preserves the leftmost-outermost reduction strategy of the original reduction semantics. Although rule generalization and the associated proof of soundness is not a mechanical step, in this case study it provides a simple extension which enables refocusing and lets us reuse the sequence of mechanical derivation steps of the syntactic correspondence. With this adjusted reduction rule, we can therefore proceed as in Section 7.3.

7.4.1 Prelude to a reduction semantics

The reduction strategy along with the notions of value and of potential redex puts us 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 grammar of delimited contexts used to specify the generalized reduction rule, 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 7.4.1.0). Second, we search for the leftmost-outermost disjunction containing a conjunction (Section 7.4.1.1). 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 7.4.1.2). As a corollary, we can then state the associated recomposition function that maps a reduction context and a contractum to the corresponding reduct (Section 7.4.1.3).

7.4.1.0 Prelude 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 7.3.1.2, we transform the search function into a decomposition function. We do so by (1) CPS-transforming the search function (and then simplifying it one bit), (2) defunctionalizing its continuation, which gives the implementation of D in Section 7.4.0,

```
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. The result is a pushdown automaton with 3 states (1 initial state and 2 final states) where the defunctionalized continuation plays the role of the stack:

```
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 (D, t2), d1)
    = decompose_term_d (t2, D2 (d1, D))
  | decompose_cont_d (D2 (d1, D), d2)
    = decompose_cont_d (D, DISJ_node (d1, d2))

(*  term_nnf × cont_d → value_or_decomposition_d  *)
and decompose_term_d (LIT_nnf x, D)
    = decompose_cont_d (D, DISJ_leaf x)
  | decompose_term_d (CONJ_nnf (t1, t2), D)
    = DEC_d (t1, t2, D)
  | decompose_term_d (DISJ_nnf (t1, t2), D)
    = decompose_term_d (t1, D1 (D, t2))
```

```
(*  term_nnf → value_or_decomposition_d  *)
fun decompose_d t = decompose_term_d (t, D0)
```

This automaton is implemented as a big-step abstract machine: each transition is carried out by a tail call.

7.4.1.1 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 search0_term_c (LIT_nnf x)
    = VAL_c (CONJ_leaf (DISJ_leaf x))
  | search0_term_c (CONJ_nnf (t1, t2))
    = (case search0_term_c t1
        of (VAL_c c1)
           ⇒ (case search0_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)
  | search0_term_c (DISJ_nnf (t1, t2))
    = (case decompose0_term_d (DISJ_nnf (t1, t2), D0)
        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 search0_c t = search0_term_c t
```

By adequacy of our representation, the search function implements the leftmost-outermost reduction strategy:

Property 7 (soundness)**.** *Let* `t` *represent the term* `t`*, let* `pr` *represent the potential redex pr, and let* `t_cnf` *represent a conjunctive normal form:*

- `search0_c t` *evaluates to* `POTRED pr` *if and only if pr is the leftmost-outermost redex in* `t`*; and*
- `search0_c t` *evaluates to* `VAL t_cnf` *and* `embed_cnf t_cnf` *evaluates to* `t` *if and only if* `t` *does not contain a redex.*

7.4.1.2 From searching to decomposing

As in Section 7.3.1.2, we transform the search function into a decomposition function. We do so by (1) CPS-transforming the search function (and then simplifying it one bit), (2) defunctionalizing its continuation, which gives the implementation of C in Section 7.4.0,

```
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. The result is a pushdown automaton with 5 states (1 initial state and 2 final states) and 2 stacks (each of which is a defunctionalized continuation) implemented as a big-step abstract machine:

```
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)
```

The decompose function implements the decomposition of a term as a leftmost-outermost redex in its reduction context:

Property 8 (soundness)**.** *Let* `t` *represent the term* C[*pr*]*, let* `pr` *represent the potential redex pr, and let* `k` *represent the reduction context* C*:* `decompose t` *evaluates to* `DEC (pr, k)`*.*

7.4.1.3 Recomposing

We implement recomposition with a left fold over the given reduction context to recompose it around the given term:

```
(*  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))

(*  value_or_decomposition → term_nnf  *)
fun recompose (VAL v) = embed_cnf v
  | recompose (DEC (pr, C)) = recompose_c (C, pr2t pr)

(* potential_redex → term_nnf *)
and pr2t (PR_DISTR_DISJ (t1, t2, D)) = recompose_d (D, CONJ_nnf (t1, t2))
```

The recompose function implements the recomposition of a context around a term:

Property 9 (soundness)**.** *Let* `k` *represent the context* C*, let* `t` *represent the term* `t`*, and let* `t'` *represent the term* C[t]*:* `recompose_c (k, t)` *evaluates to* `t'`*.*

7.4.2 A reduction semantics

We are now fully equipped to implement a reduction semantics for conjunctive normalization.

7.4.2.1 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.

7.4.2.2 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 around the contractum. If the potential

redex is not an actual one, reduction is stuck. Given a value term, reduction is also impossible:

```
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))
```

Property 10 (soundness)**.** *Let* `t` *represent* t *and* `t'` *represent* t′*:* `reduce t` *evaluates to* `REDUCT t'` *if and only if* t *reduces to* t′ *in one step by leftmost-outermost reduction.*

7.4.2.3 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 normalization, and if it becomes stuck, normalization 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)
```

Property 11 (soundness)**.** *Let* `t` *represent* t *and* `v` *represent* ν*:* `normalize t` *evaluates to* `RESULT v` *if and only if* t *reduces to* ν *by leftmost-outermost reduction.*

7.4.3 From reduction-based to reduction-free normalization

This section follows the steps of Section 7.3.3: we refocus the reduction-based normalization function of Section 7.4.2.3, inline the contraction function, fuse the resulting small-step abstract machine into a big-step one, and compress its corridor transitions hereditarily.

As a postlude to transition compression, we observe that the normalizer contains a non-tail call to `recompose_d` which stems from the inlining of `contract`, and whose result will

subsequently be decomposed again. We therefore introduce a specialized context constructor `C1'` for this case, to short-cut the recomposition and subsequent decomposition of this subterm.

The result – a big-step abstract machine with two stacks – 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' 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' (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' (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)
```

7.4.4 From abstract machines to normalization functions

This section follows the steps of Section 7.3.4: we refunctionalize the abstract machine of Section 7.4.3, and we transform the resulting normalization function into direct style.

7.4.4.1 Refunctionalization

As in Section 7.3.3, the big-step abstract machine of Section 7.4.3 is in defunctionalized form: the reduction contexts, together with `normalize4_cont_d` and `normalize4_cont_c`, are the first-order counterparts of two functions. The higher-order counterpart of this abstract machine reads as follows:

```
(* term_nnf × (disj_cnf × (value → α) → α) × (value → α) → α *)
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, mk1) ⇒
        normalize5_term_d (t2, fn (d2, mk2) ⇒
           k (DISJ_node (d1, d2), mk2), mk1), mk)

(* term_nnf × (value → α) → α *)
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 (t1, fn (d1, mk1) ⇒
        normalize5_term_d (t2, fn (d2, mk2) ⇒
          mk2 (CONJ_leaf (DISJ_node (d1, d2))), mk1), mk)

(* term_nnf → result_or_wrong *)
fun normalize5 t = normalize5_term_c (t, RESULT)
```

This normalization function is compositional and can be expressed with the fold functional for terms in negational normal form.

7.4.4.2 Back to direct style

The refunctionalized definition of Section 7.4.4.1 is in continuation-passing style with two layered continuations. 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 (t1, fn d1 ⇒
        normalize6_term_d (t2, fn d2 ⇒
          CONJ_leaf (DISJ_node (d1, d2))))

(*  term_nnf → result_or_wrong  *)
fun normalize6 t = RESULT (normalize6_term_c t)
```

This normalization function is still compositional. Its component `normalize6_term_d` is expressed in the "continuation-composing style" characteristic of functional backtracking. This function 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, `normalize6_term_d` is in ordinary continuation-passing style, where all calls are tail calls. In the clause for conjunctions, however, there are two non-tail calls to `normalize6_term_d`. In direct style, this programming idiom is captured with the delimited control operators shift and reset [34].

7.4.5 Delimited continuations in direct style

In this section, we use Filinski's encoding of shift and reset in ML [55]:

```
val shift : ((α → value) → value) → α
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 7.4.4 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/α → value/α  *)
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 (DISJ_node (normalize7_term_d t1,
                                           normalize7_term_d 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.

This normalization function can be expressed with the fold functional for terms in negational normal form.

7.4.6 Implementation and testing

All the elements of the spectrum, from `normalize` to `normalize7` and including the versions with the fold functional, are implemented in Standard ML. We have run each of them on an array of tests and verified that they all yield the same results.

7.4.7 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 the reduction rules. Then, taking the same methodical steps as in Section 7.3, we have refocused the reduction-based normalization function of Section 7.4.2.3 into a reduction-free, small-step abstract machine, and we have exhibited a spectrum of corresponding reduction-free normalization functions that all are inter-derivable.

Ever since Wand's foundational article on continuation-based program-transformation strategies [131], converting a formula into conjunctive normal form is a classic among continuations aficionados. We are adding the following stones to this foundation:

1. our big-step, reduction-free normalization function is not designed from scratch; it is systematically calculated from a small-step, reduction-based normalization function;
2. it is compositional and proceeds in one pass: no intermediate results are constructed and subsequently traversed for further transformation;
3. it shows that delimited continuations form a natural expressive medium to carry out the distributivity laws in the big-step normalization function; and
4. as developed next, it suggests a representation of conjunctive normal forms whose size is linear, instead of exponential, with respect to the size of source Boolean terms.

7.5 Conjunctive normal forms in linear space

In this section, we present a linear-sized representation of conjunctive normal forms. We first identify a source of duplication in conjunctive normal forms (Section 7.5.1). This duplication is due to the duplication of contexts, and therefore we mitigate it by naming continuations in normal forms (Section 7.5.2). In so doing, we discover that the (nested) duplication of contexts is the sole reason for the exponential blowup. We then reflect the naming of continuations through the spectrum of normalization functions, using the syntactic correspondence and the functional correspondence (Section 7.5.3).

Our linear-sized conjunctive normal forms are unrelated to Tseitin's [127]. Tseitin constructs another Boolean expression (with new Boolean variables) which is in conjunctive normal form and which is equisatisfiable to the original expression. In contrast, we construct

a shared conjunctive normal form induced by the distributive laws, and therefore our normal form is semantically equivalent to the original expression, not just equisatisfiable. Also, the variables we introduce are not Boolean variables – they are program variables denoting continuations.

7.5.1 Duplications in conjunctive normal forms

The size of conjunctive normal forms is known to be exponential with respect to the size of the source Boolean term. A source of this blowup can be readily identified in the definition of `normalize6_term_d` in Section 7.4.4.2. In the clause for conjunctions, the continuation is duplicated:

```
  | normalize6_term_d (CONJ_nnf (t1, t2), k)
    = CONJ_node (normalize6_term_d (t1, k), normalize6_term_d (t2, k))
```

Consider, for example, the following Boolean term:

```
DISJ_nnf (CONJ_nnf (LIT_nnf (POSVAR "a"),
                    LIT_nnf (NEGVAR "b")),
          LIT_nnf (POSVAR "c"))
```

When `normalize6_term_d` is called on the left disjunct,

```
          CONJ_nnf (LIT_nnf (POSVAR "a"),
                    LIT_nnf (NEGVAR "b"))
```

its continuation (essentially) reads as follows:

```
fn d1 ⇒ normalize6_term_d (LIT_nnf (POSVAR "c"),
                           fn d2 ⇒ k (DISJ_node (d1, d2)))
```

for a given `k`. Unfolding the call to `normalize6_term_d`, it reads more concisely as follows:

```
fn d1 => k (DISJ_node (d1, DISJ_leaf (POSVAR "c")))
```

This continuation is applied twice, giving rise to the two occurrences of the disjunction node with the same right disjunct, `DISJ_leaf (POSVAR "c")`, in the normal form:

```
CONJ_node (CONJ_leaf (DISJ_node (DISJ_leaf (POSVAR "a"), DISJ_leaf (POSVAR "c"))),
           CONJ_leaf (DISJ_node (DISJ_leaf (NEGVAR "b"), DISJ_leaf (POSVAR "c"))))
```

Nesting conjunctions in disjunctions gives rise to an exponential blowup.

As it happens, this duplicative phenomenon is the same as in the CPS transformation [119]. In the next section, we propose a similar solution [27]: naming (delimited) continuations in the normal form and sharing their names.

To be concrete, the normal form above would read as follows in ML:

```
fn (a, b, c) => (a orelse c) andalso (not b orelse c)
```

With sharing, it would read:

```
fn (a, b, c) => let fun k j = j orelse c
                in k a andalso k (not b)
                end
```

7.5.2 Sharing in conjunctive normal forms

Let us revise the BNF of normal forms with two new productions, one to define a join point (namely the continuation) and one to use this join point:

$$\begin{aligned} l &::= x \mid \neg x \\ t_{dnf} &::= l \mid t_{dnf} \vee t_{dnf} \mid j \\ t_{cnf} &::= t_{cnf} \wedge t_{cnf} \mid \text{defjoin } \lambda j.t_{cnf} \text{ in } t_{cnf} \wedge t_{cnf} \mid \text{usejoin } t_{dnf} \\ t_{root} &::= \text{defjoin } \lambda j.j \text{ in } t_{cnf} \end{aligned}$$

For uniformity, a vacuous join point is added at the root of the normal form. Definitions of join points are nested, but there is no need to name them: due to the tree structure of normal forms, each use of a join point refers to the lexically enclosing definition – in CPS jargon, "one continuation identifier is enough" [26]. (If one was to add associativity as a conversion rule, and orient this conversion rule into a reduction rule, the normal forms would be flat rather than tree-structured: "conjunctions of disjunctions of literals" would be represented as "lists of lists of literals." When sharing contexts, the lexical scope of join points would be shared, and they would require distinct names.)

This revised BNF of normal forms is implemented with the following ML data types, where we implement join points with de Bruijn levels [18]:

```
datatype disj_cnfws = DISJ_leaf_cnfws of literal
                    | DISJ_node_cnfws of disj_cnfws × disj_cnfws
                    | DISJ_var_cnfws of level

datatype conj_cnfws = CONJ_node_cnfws of conj_cnfws × conj_cnfws
                    | DEFJOIN_cnfws of level × conj_cnfws × conj_cnfws ×
                        conj_cnfws
                    | USEJOIN_cnfws of disj_cnfws

datatype term_cnfws = DEFJOIN_init_cnfws of conj_cnfws
```

The clause for conjunctions is revised to not duplicate the current continuation but name it as the current join point, so that it is applied only once:

```
  | normalize7_term_d (CONJ_nnf (t1, t2), l, k)
    = DEFJOIN_cnfws (l,
                     k (DISJ_var_cnfws l, l+1),
                     normalize7_join_c (t1, l),
                     normalize7_join_c (t2, l))
```

(The complete normalization function is described in Section 7.5.3).

Going back to the Boolean term from Section 7.5.1,

```
DISJ_nnf (CONJ_nnf (LIT_nnf (POSVAR "a"),
                    LIT_nnf (NEGVAR "b")),
          LIT_nnf (POSVAR "c"))
```

normalization with sharing yields a normal form where `DISJ_leafws (POSVAR "c")` is not duplicated:

```
DEFJOIN_init_cnfws
  (DEFJOIN_cnfws
     (0,
      USEJOIN_cnfws
        (DISJ_node_cnfws (DISJ_var_cnfws 0, DISJ_leaf_cnfws (POSVAR "c"))),
```

```
      USEJOIN_cnfws
        (DISJ_leaf_cnfws (POSVAR "a")),
      USEJOIN_cnfws
        (DISJ_leaf_cnfws (NEGVAR "b"))))
```

In fact, looking at the definition of the normalization function, we can see that nothing (except for the names of the join points) is duplicated. Therefore, the size of the normal forms is linearly proportional to the size of the corresponding source terms. Furthermore, the duplication of continuations is the only cause of the exponential blowup. In fact, unfolding each definition of join point and their two uses gives back the normal forms of Section 7.4. Finally, it is now simple to invert the normalization function and get back the source term corresponding to a (shared) normal form, compositionally and in one pass, using an environment [25]:

```
(* term_cnfws → term_nnf *)
fun un_normalize (DEFJOIN_init_cnfws c)
    = let fun visit_disj (DISJ_leaf_cnfws x, e)
              = LIT_nnf x
            | visit_disj (DISJ_node_cnfws (d1, d2), e)
              = DISJ_nnf (visit_disj (d1, e), visit_disj (d2, e))
            | visit_disj (DISJ_var_cnfws l, e)
              = lookup (l, e)
          fun visit_conj (CONJ_node_cnfws (c1, c2), e)
              = CONJ_nnf (visit_conj (c1, e), visit_conj (c2, e))
            | visit_conj (DEFJOIN_cnfws (l, b, c1, c2), e)
              = visit_conj (b, extend (l,
                                       CONJ_nnf (visit_conj (c1, e),
                                                 visit_conj (c2, e)),
                                       e))
            | visit_conj (USEJOIN_cnfws d, e)
              = visit_disj (d, e)
      in visit_conj (c, nil)
      end
```

7.5.3 The spectrum of normalization functions with sharing

Here is, in full, the ML implementation of the normalization function described in Section 7.5.2. Essentially, it is a clone of the implementation of Section 7.4.4 where a de Bruijn level is threaded through the recursive calls and where, in the clause for conjunctions, a join point is inserted and the current (delimited) continuation is implicitly reset to yield a join use:

```
(*  term_nnf × level × (disj_cnfws × level → conj_cnfws) → conj_cnfws  *)
fun normalize7_term_d (LIT_nnf x, l, k)
    = k (DISJ_leaf_cnfws x, l)
  | normalize7_term_d (CONJ_nnf (t1, t2), l, k)
    = DEFJOIN_cnfws (l,
                     k (DISJ_var_cnfws l, l+1),
                     normalize7_join_c (t1, l),
                     normalize7_join_c (t2, l))
  | normalize7_term_d (DISJ_nnf (t1, t2), l, k)
    = normalize7_term_d (t1, l, fn (d1, l1) ⇒
        normalize7_term_d (t2, l1, fn (d2, l2) ⇒
          k (DISJ_node_cnfws (d1, d2), l2)))
```

```
(*  term_nnf × level → conj_cnfws  *)
and normalize7_join_c (LIT_nnf x, l)
    = USEJOIN_cnfws (DISJ_leaf_cnfws x)
  | normalize7_join_c (CONJ_nnf (t1, t2), l)
    = CONJ_node_cnfws (normalize7_join_c (t1, l),
                       normalize7_join_c (t2, l))
  | normalize7_join_c (DISJ_nnf (t1, t2), l)
    = normalize7_term_d (t1, l, fn (d1, l1) ⇒
        normalize7_term_d (t2, l1, fn (d2, l2) ⇒
          USEJOIN_cnfws (DISJ_node_cnfws (d1, d2))))

(* term_nnf → result_or_wrong *)
fun normalize t
    = RESULT (DEFJOIN_init_cnfws (normalize7_join_c (t, 0)))
```

The vigilant reader will have observed that this normalization function constructs no empty join points.

In the code files associated with this paper, we have posited a reduction semantics for conjunctive normalization with sharing. The entire spectrum of normalization functions of Section 7.4 – reduction-free normalizer, defunctionalized abstract machine with two stacks, CPS-transformed version with two continuations, version in continuation-composing style, and direct-style version using shift and reset – can be methodically inter-derived using the same steps as in Section 7.4. After refunctionalization, all of these normalization functions are compositional and can therefore be expressed with the fold functional for terms in negational normal form.

7.5.4 Implementation and testing

All the elements of the spectrum, starting from the reduction-based normalization function corresponding to the reduction semantics we posited and including the normalization function displayed in Section 7.5.3 and the versions with the fold functional, are implemented in Standard ML. We have run each of them on an array of tests and verified that they all yield the same results. We have also verified that for each test, unfolding each definition of a join point and its two uses gives the corresponding normal form obtained in Section 7.4.

7.5.5 Summary and conclusion

We have identified that the duplication of evaluation contexts, i.e., continuations, is a cause of the exponential blowup of conjunctive normal forms. We have also identified that this duplication is the only cause of this exponential blowup. We have proposed the same solution as in the CPS transformation to avoid this blowup: naming contexts as join points, and duplicating these names. The result is a representation of conjunctive normal forms that is linear in size with respect to the corresponding source Boolean terms, that can be obtained in one pass, and whose normalization can be inverted in one pass. As to whether this representation is useful in practice, we leave it to a future work, but we note that the dual story holds mutatis mutandis for disjunctive normal forms.

7.6 Related work

This section starts with a very brief history of the dynamic semantics of programming languages.

The notion of semantics arose from the wish to model computation in order to reason about it (Church, Curry, Turing, etc.). As Alan Perlis put it, programming languages are a notation to express computation, and the notion of formal semantics arose from the wish to model this notation in a formal model [56, 77, 81, 123]. This formal model embodies the notion of computation captured by this notation in order to reason about it. At first, models were selected for their reasoning comfort: the λ-calculus initially for denotational semantics [121], and subsequently the underlying domain theory [106] and eventually category theory [86]; natural deduction for natural semantics [76]; etc. Rapidly, though, these formal semantics were refined for their expressivity, to specify particular notions of computation with a modicum of comfort – e.g., jumps in denotational semantics with continuations and control effects in reduction semantics with context-sensitive reduction rules. As a result, multiple semantic specifications are now routinely written to capture one aspect or another of a particular notion of computation. The ones closest to actual implementations are abstract machines. These varieties of formal semantics have given rise to proof obligations of adequacy and to a culture of deriving semantics from one another [58, 67, 114, 115, 132], often revealing foundational connections in passing [23, 24, 92, 125].

We observe, however, that while many derivational frameworks were presented as instrumental to derive new semantic artifacts, virtually none of them has been subsequently reused to derive other semantic artifacts. And indeed each one is tailored to relate distinct semantic models.

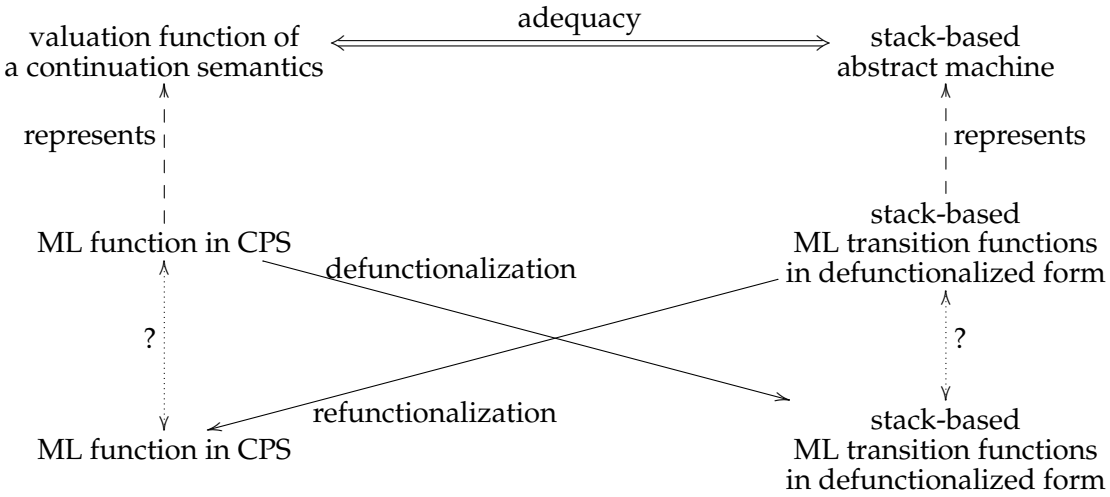
We are therefore taking a structural approach: rather than studying each semantic artifact in its own setting, we study their common representation in the same pure functional language. The resulting functional programs are archetypal: they are first-order or higher-order, they are in direct style or they use continuations, they only use tail calls (i.e., they are tail-recursive, i.e., iterative) or they use non-tail calls (i.e., they are recursive), they use stacks, they construct intermediate data structures, etc. Since many correctness-preserving program transformations exist, it is thus natural to apply these transformations to pre-existing archetypal programs and to compare the result to other pre-existing archetypal programs.

For one example, consider an ML function in CPS and the ML transition functions of a stack-based abstract machine:

- In CPS, all calls are tail calls, all sub-terms are values, and all functions are passed a functional accumulator representing the rest of the computation, the continuation.
- The transition functions of a stack-based abstract machine call each other tail-recursively, all their arguments are values, and they thread a stack.

Since defunctionalizing a continuation yields a last-in, first out data structure, it is natural to consider a continuation semantics and a stack-based abstract machine that specify the same programming language (top part of the diagram just below), a continuation-passing function that represents the valuation function of this continuation semantics (middle part of the diagram on the left), and stack-based transition functions in defunctionalized form that represent this abstract machine (middle part of the diagram on the right). Defunctionalizing the ML function in CPS (from the middle part on the left to the bottom part of the diagram

on the right) gives transition functions in defunctionalized form and that use a stack, and refunctionalizing the stack-based ML transition functions (from the middle part on the right to the bottom part of the diagram on the left) gives a function that is in CPS:



The obvious two questions (the question marks in the diagram above) now are to compare

- in the lower part of the diagram on the left, the ML function in CPS that represents the valuation function of the continuation semantics and the ML function in CPS obtained by refunctionalization, and

- in the lower part of the diagram on the right, the ML transition functions in defunctionalized form that represent the abstract machine and the ML transition functions obtained by defunctionalization.

There are no data bases of continuation semantics and abstract machines: all continuation semantics and abstract machines we are aware of are scattered in many separate publications. Having perused the literature, here is the result of the comparison:

- First of all, we are not aware of continuation semantics whose ML representation cannot be defunctionalized.

- In numerous cases, defunctionalization yields the ML transition functions representing pre-existing abstract machines, and when the abstract machines are in defunctionalized form, refunctionalization yields the ML function in CPS representing pre-existing continuation semantics. These numerous cases range from the varieties of λ-calculus [2, 4] to the Algorithmic Language Scheme [33].

- In some cases, defunctionalization and refunctionalization suggest a tighter continuation semantics and a tighter abstract machine. For example, in the case of Propositional Prolog with Cut [19], the continuation semantics does not need a variable for the cut continuation (like the abstract machine), and the abstract machine can be made properly tail recursive (like the continuation semantics) [16].

- In some cases, the transition functions of some abstract machines are not in defunctionalized form, e.g., Landin's SECD machine [77]. We do not know how significant it is that an abstract machine is in defunctionalized form or not: for example, in Felleisen's study of Landin's J operator [52], the SECD machine is not the original one but a variant that is in defunctionalized form. In any case, heuristics now exist to put abstract machines in defunctionalized form [41].

- In all the other cases we have studied, given a canonical continuation semantics, defunctionalization yields a new abstract machine, and given a canonical abstract machine in defunctionalized form, refunctionalization yields a new continuation semantics. The results are canonical, i.e., they correspond to what an expert would have written by hand.

So continuation semantics and stack-based abstract machines can be related by defunctionalization and refunctionalization of their ML representation.

The same actually holds for the other styles of semantics:

- Closure conversion relates the representation of denotational semantics in direct style and the representation of natural semantics. In numerous cases, these semantics preexisted and were known to be equivalent. In all the other cases we have studied, they were canonical.

- CPS transformation relates the representation of denotational semantics in direct style and the representation of continuation semantics. In numerous cases, these semantics pre-existed and were known to be equivalent. In all the other cases we have studied, they were canonical.

- CPS transformation and defunctionalization relate the representation of structural operational semantics and the constitutive elements of a representation of reduction semantics (i.e., decomposition, contraction, and recomposition). In numerous cases, this semantics and these constitutive elements pre-existed and were known to be equivalent. In all the other cases we have studied, they were canonical.

- The syntactic correspondence (i.e., refocusing, lightweight fusion, transition compression, and context specialization) relates reduction semantics and abstract machines. In numerous cases, these semantic artifacts pre-existed and had been shown to be equivalent. In all the other cases we have studied, they were canonical.

And so one can inter-derive semantic artifacts methodically and uniformly.

By now the syntactic and functional correspondences have been independently used for discovery (when the result of the derivation is new), for rediscovery (when the result of the derivation turns out to be known already), or as a springboard (when the result of the derivation improves on something known or leads somewhere else): Anton and Thiemann used them to derive type systems and implementations for coroutines [5]; Danvy and Johannsen used them to derive the same abstract machine out of the small-step and big-step semantics of Abadi and Cardelli's calculus of objects [36]; Pirog and Biernacki used the functional correspondence to derive the STG machine out of Launchbury's natural semantics for lazy evaluation [96]; Sergey and Clarke used them to inter-derive type checking by reduction and

type checking by evaluation [113]; Van Horn and Might use them to obtain the abstract machines they abstract to do static analysis [128]; Ariola et al. used them to assess the unity of semantic artifacts for call-by-need sequent calculi [7]; Puech used the functional correspondence to connect natural deduction and the sequent calculus [101]; Jedynak et al. used them to provide an operational foundation for the tactic language of Coq [72]; Simmons and Zerny presented a logical analogue of the functional correspondence to connect natural semantics and abstract machines [117]; Danvy and Zerny used them to connect syntactic theories of call by need and the Krivine machine with memo-thunks [45], and to connect Barendregt et al.'s term graph rewriting and Turner's abstract machine for combinatory graph reduction [46]; García-Pérez et al. used them to derive interpretations of the gradually-typed λ-calculus [62]; and Bach Poulsen and Mosses used them to derive pretty-big-step semantics from small-step semantics [8].

In this chapter, we have used these programming-language tools outside the realm of programming languages, to inter-derive Boolean normalization functions.

7.7 Conclusion and perspectives

The inter-derivations illustrated here witness a striking unity of computation across small-step semantics, abstract machines, and big-step semantics. The structural coincidence between reduction contexts and evaluation contexts as defunctionalized continuations, in particular, plays a key role to connect reduction strategies and evaluation strategies, a connection that was first established by Plotkin [97] and that scales to delimited continuations. As for Ohori and Sasano's lightweight fusion [93], it provides the linchpin between the functional representations of small-step and big-step operational semantics [39].

The resulting spectrum of Boolean normalization functions shows that the only cause of the exponential blowup in the representation of conjunctive normal forms is the duplication of continuations. Naming these contexts and continuations as join points is therefore sufficient to obtain a linear representation of conjunctive normal forms. As a consequence, conjunctive normalization operates in linear time and in one pass, and it can be inverted.

Conjunctive normalization also made us realize a limitation of refocusing for outside-in reduction strategies – namely in the presence of backward overlaps in the reduction rules – and how to overcome this limitation.

Overall, the inter-derivations illustrate the conceptual value of semantics-based program manipulation, as promoted at PEPM ever since its inception. For example, we still marvel at how reduction-based, small-step normalization functions keep giving rise to reduction-free, big-step normalization functions that are compositional.

7.8 Acknowledgments

We are grateful to Jeremy Siek and Siau-Cheng Khoo for their invitation to present a preliminary version of Section 7.3 at the 20th anniversary of PEPM [49]. The example of negational normalization originates in a joint work of Danvy and Johannsen [36]. The two preludes to a reduction semantics originate in a joint work of Danvy and Zerny [47]. Thanks are due to the reviewers for their insightful comments.

7.9 Code files

The complete ML code listed in this chapter is available at the following URLs:

```
http://users-cs.au.dk/cnn/PhD-files/spectrum-in-sml.tgz
http://users-cs.au.dk/cnn/PhD-files/spectrum-in-sml.zip
```

It contains the spectrum of Boolean normalization functions for

- leftmost-outermost negational normalization (Section 7.3),
- leftmost-innermost negational normalization,
- leftmost-outermost conjunctive normalization (Section 7.4),
- leftmost-outermost conjunctive normalization with sharing (Section 7.5), and
- leftmost-outermost disjunctive normalization

together with an array of unit tests.

Part III

Open Questions

Chapter 8

Open Questions

This chapter outlines a number of open questions that have arisen in the course our work.

Note: Each open question is discussed in its own section, but it is quite possible that the questions are related to each other.

For reference, we restate the definitions of backward overlaps and backward chains from Chapter 2:

Definition 7 (Backward overlaps)**.** *Let* $\mathtt{pr}_1 \mapsto c_1$ *and* $\mathtt{pr}_2 \mapsto c_2$ *be reduction rules. If there exist a non-empty context* $\mathtt{C}$ *and a meta-term* $\mathtt{t}$ *containing at least one term constructor such that*

- $\mathtt{pr}_1 = \mathtt{C}[\mathtt{t}]$
- $\mathtt{t}$ *unifies with* c_2 *with the most general unifier* σ*,*

then $\mathtt{pr}_1$ *and* c_2 *are said to exhibit a* backward overlap*, and the two rules are said to be* backward overlapping*.*

Definition 8 (Backward chain)**.** *Given a set of rewrite rules* $\mathtt{R}$*, a* backward chain *of* $\mathtt{R}$ *is a reduction sequence* $\mathtt{t}_1 \to \mathtt{t}_2 \to \ldots \to \mathtt{t}_n$ *such that all reductions in the sequence (except possibly for the last one) give rise to a backward overlap.*

In the remainder of this chapter, we shall refer to the meta-term $\mathtt{C}[\sigma(c_2)]$ as the *witness* of the backward overlap. Also, we shall refer to the meta-term $\mathtt{C}[\sigma(\mathtt{pr}_2)]$ as the *pre-witness* of the backward overlap.

8.1 Non-problematic backward overlaps

As mentioned in Section 4.2.2, not all backward overlaps are problematic:

- *If the contractum* c_2 *cannot contain a potential redex*: We illustrate this condition using weak-head normalization of the λ-calculus using call by name [13, 43], where the β-rule backward-overlaps with itself, but where the contractum that gives rise to the backward overlap is a λ-abstraction, and therefore a value.

  If the contractum of the backward overlap cannot contain a potential redex, then applying *decompose_term* to the contractum will return a value, and the decomposition process continues in the context, where the redex created by the backward overlap can

be found. In effect, this phenomenon allows call by name to be implemented using an innermost reduction strategy, because *decompose_term* can in this case safely be applied to a subterm of the current node *before* checking whether the current node is the root of a redex.

Since backward overlaps are unproblematic in combination with innermost reduction, refocusing can be applied without modification in this scenario.

- *If $\sigma(\mathrm{pr}_2)$ is not the first potential redex to be contracted in the pre-witness*: We illustrate this condition using outermost tree flattening using the associativity rule [31, Sections 10-11], where the associativity rule backward-overlaps with itself:

$$(t_1 \cdot t_2) \cdot t_3 \mapsto t_1 \cdot (t_2 \cdot t_3)$$

The pre-witness of this backward overlap is

$$((t_1 \cdot t_2) \cdot t_3) \cdot t_4$$

The outermost redex of the pre-witness is rooted at the node with t_4 as its right subterm, so $(t_1 \cdot t_2) \cdot t_3$ is not the first potential redex to be contracted in the pre-witness.

If the first potential redex of the pre-witness is not the redex that gives rise to the backward overlap, then contraction cannot construct a new redex above the position of the contractum. Therefore, the reduct will not contain a redex at a more outermost position than the contractum, and refocusing can be applied without modification in this scenario.

A similar observation can be made for negational normalization of Boolean terms (Section 7.3), and for conjunctive normalization of Boolean terms using generalized reduction rules (Section 7.4 and Section 7.5), where refocusing can be applied without modification as well.

In the three case studies where refocusing is not applicable without modification (full normalization of the $\lambda\hat{\rho}$-calculus using normal-order [48, 59, 60, 84, 89], addition of Peano numbers (Section 6.4), conjunctive normalization without generalized rules (Section 7.4 and Section 7.5)), neither of these two conditions are satisfied. This observation leads us to state the following two conjectures:

Conjecture 1 (Soundness of refocusing)**.** *For a reduction semantics with a deterministic reduction strategy, refocusing is unsound if all of the following criteria are satisfied:*

- *The reduction strategy is implemented as outermost.*
- *The reduction rules contains one or more backward overlaps for which reduction of the pre-witness results in that backward overlap.*

Conjecture 2 (Completeness of refocusing)**.** *For a reduction semantics with a deterministic reduction strategy, refocusing is unsound only if all of the following criteria are satisfied:*

- *The reduction strategy is implemented as outermost.*

- *The reduction rules contain one or more backward overlaps for which reduction of the prewitness results in that backward overlap.*

Our work makes a case for the soundness conjecture. The completeness conjecture less obviously holds, and it is quite possible that more assumptions need to be spelled out for that conjecture to be true.

Since the prelude to reduction semantics described in Section 4.1 applies in exactly the same cases as unmodified refocusing, it is possible that similar soundness and completeness criteria hold for the prelude.

8.2 Soundness of backward chaining

The rule generalization based on backward chaining which we apply in Section 7.4.0 is proved correct by Proposition 6. However, the soundness proof is specific to the normalization problem, and does not imply that the construction is sound in general.

In order to prove the construction sound in general, it is necessary to formalize the construction, and such a formalization is still work in progress. Here, we outline the steps that are needed towards such a formalization.

Once a backward overlap has been identified, the generic form of the rule generalization works as follows:

1. For each backward chain that respects the reduction strategy, generate a reduction rule that simulates the chain. The set of rules generated in this way is in general unbounded. However, the rules will exhibit a certain regularity, because they will be derived from repetitions of the same backward overlap.

2. Identify the regularity. The regularity will in part be suggested by the context C that gives rise to the backward overlap, but will in general also require multiple applications of the unifier σ.

3. Express the regularity in terms of a restricted grammar of contexts, so that the rules can be collapsed into one context-sensitive rule.

The first step is easy to formalize, as it requires only that the definition of backward chains has to respect the reduction strategy. The second and third steps are possible to formalize for simple reduction systems, but for systems that contain multiple overlaps involving the same rules, the formalization is more challenging. To see the difficulty, we invite the reader to construct the generalized rules corresponding to the addition of Peano numbers described in Section 6.4.

However, once this generic formalization is established, it should be a simple exercise to prove the construction sound in general.

8.3 Unique decomposition for outermost reduction

In the paper that introduced refocusing [43], Danvy and Nielsen showed that refocusing is applicable to any reduction semantics using an innermost reduction strategy, if that reduction semantics satisfies unique decomposition. For outermost reduction, Section 8.1 lists a number of cases where refocusing is also applicable without modification. Common to these

cases is that the reduction semantics can be defined with a grammar of reduction contexts that satisfies the unique decomposition property:

- Weak-head normalization of the λ-calculus using call by name [13, 43]: The grammar of reduction contexts can be expressed with unique decomposition as follows:

$$\mathrm{C} ::= [\,] \mid \mathrm{C}[[\,]\,\mathrm{t}]$$

- Outermost tree flattening using the associativity rule [31, Sections 10-11]: The grammar of reduction contexts can be expressed with unique decomposition as follows:

$$\mathrm{C} ::= [\,] \mid \mathrm{C}[\mathrm{l} \cdot [\,]]$$

  where l is a leaf.

- Conjunctive normalization using generalized reduction rules (Section 7.4): The grammar of reduction contexts can be expressed with unique decomposition as follows:

$$\mathrm{C} ::= [] \mid \mathrm{C}[[\,] \wedge \mathrm{t}_{nnf}] \mid \mathrm{C}[\mathrm{t}_{cnf} \wedge [\,]]$$

  (A similar grammar can be defined for conjunctive normalization with sharing (Section 7.5)).

In all cases, the grammar of reduction contexts can exhibit unique decomposition using only one context non-terminal.

For the cases where refocusing is not applicable without modification, it appears that a grammar exhibiting unique decomposition requires more complicated grammars:

- Outermost addition of Peano numbers: As mentioned in the footnote in Section 6.4, a more precise grammar of reduction contexts exists than the one presented in that section:

$$\begin{aligned} \mathrm{C}[\,] &::= \Box[\,] \mid \mathrm{C}[S[\,]] \\ \mathrm{D}[\,] &::= \mathrm{C}[\,] \mid \mathrm{D}[A([\,], \mathrm{t})] \end{aligned}$$

  This grammar exhibits unique decomposition. However, the grammar does require two "layers" to capture the fact that once an addition node has been encountered, only further addition nodes allow the reduction context to be extended.

  Unique decomposition for conjunctive normalization without generalized rules (Section 7.4) can also be obtained using a two-layered grammar of contexts:

$$\begin{aligned} \mathrm{C} &::= \quad [] \mid \mathrm{C}[[\,] \wedge \mathrm{t}_{nnf}] \mid \mathrm{C}[\mathrm{t}_{cnf} \wedge [\,]] \\ \mathrm{D} &::= \mathrm{C}[\,] \mid \mathrm{D}[[\,] \vee \mathrm{t}_{nnf}] \mid \mathrm{D}[\mathrm{t}_{dnf} \vee [\,]] \end{aligned}$$

  The grammar is identical to the one obtained by rule generalization and the two-stage prelude, except for the base case for D.

- Full normalization of the $\lambda\hat{\rho}$-calculus using normal order: Unique decomposition cannot be obtained under common assumptions about reduction rules. To see this, consider the following term (where we use $\{e\}$ as the notation for an explicit substitution, and @ as the infix application operator):

  $$((\lambda x.t)\{e\} @ c) @ c_1$$

  The parenthesized application is a β-redex and is the first redex to be contracted, which means that the context $[\,] @ c_1$ must be a legal reduction context.

  Since the semantics requires full normalization of all terms, there is also a reduction rule which propagates substitutions inside λ-abstractions. Hence, $(\lambda x.t)\{e\}$ is a redex.

  If that redex occurs inside the context above, we obtain the term

  $$(\lambda x.t)\{e\} @ c_1$$

  The first redex to be contracted in this term is the β-redex at the root. The decompositions consisting of that β-redex and the empty context must therefore be a legal decomposition.

  However, since $(\lambda x.t)\{e\}$ is a redex and $[\,] @ c_1$ is a legal reduction context, that pair also constitutes a legal decomposition. Hence, this non-value term does not decompose uniquely, and unique decomposition therefore fails.

  Both Danvy, Millikin and Munk [48, 84, 89], and García-Pérez and Nogueira [59–61] overcome this problem by disregarding redexes of the form $(\lambda x.t)\{e\}$ on the left-hand side of applications. This means that the set of reduction rules changes depending on the reduction context that the redex appears in, and hence so does the grammar of redexes. Therefore, reduction is not closed under arbitrary legal reduction contexts, which is a non-standard assumption for reduction semantics. Also, in order to distinguish between which redexes are allowed in which contexts, it is necessary to express the grammar of contexts using multiple layers.

It therefore appears that unique decomposition for outermost reduction in the presence of backward overlaps can only be obtained by using a multi-layered grammar of reduction contexts.

The syntactic correspondence maps multi-layered grammars of contexts to multi-stack abstract machines, and if that abstract machine is in defunctionalized form, the functional correspondence maps it to a big-step normalizer in direct style which uses delimited control operators [30, 45]. In Chapter 6 we show how the syntactic correspondence can be applied even though the grammar of contexts is singly-layered. However, the resulting abstract machine is not in defunctionalized form, so the functional correspondence cannot be applied directly in that scenario. The resulting abstract machine has only one stack, but could be optimized into a two-stack machine (one stack to keep track of successor nodes, one to keep track of addition nodes). Such an optimization would put the abstract machine into defunctionalized form, and the functional correspondence could be applied. Unique decomposition therefore seems to have a connection with abstract machines in defunctionalized form.

As mentioned above, the two-stage version of the prelude results in a two-layered grammar of contexts. Since the one-stage version of the prelude to reduction semantics gives

rise to only one data type of contexts, it is likely that the need for multi-layered contexts is the reason why the prelude needs to be adjusted for outermost reduction in the presence of backward overlaps.

All of these observations suggest that outermost reduction and backward overlaps are in some way connected to multi-layered grammars of contexts in reduction semantics, multiple stacks in abstract machines, and delimited control operators in big-step normalizers. It is still an open question whether this connection can be established formally.

8.4 Backward chaining vs. hybrid reduction strategies

In order to make refocusing applicable to the $\lambda\hat{\rho}$-calculus, García-Pérez and Nogueira have developed a notion of *hybrid reduction strategy* [59–61]. Through personal communication with García-Pérez and Nogueira, the present author has been made aware that there might exist a connection between the hybrid reduction strategies and reduction semantics that arise when using the backward chaining construction of Section 7.4.0.

A reduction strategy is defined as hybrid when "it does rely on a subsidiary uniform substrategy" [60]. The possible connection between hybrid strategies and backward chaining is based on the observation that the backward chaining construction can be said to create a subsidiary strategy, and that the overall strategy is therefore hybrid.

It is an open question whether this connection exists. Furthermore, it seems impossible to formally prove such a connection until the definition of hybrid reduction strategy can be made more formal than the one given by García-Pérez and Nogueira.

Intuitively, however, there is some justification for the claim. The backward chaining construction gives rise to a two-layered grammar of contexts, and each layer can be viewed as a separate reduction strategy. Additionally, the backward chaining construction for conjunctive normalization gives rise to a two-layered grammar of values. The two-layered grammar of values also exists in the fully normalizing $\lambda\hat{\rho}$-calculus (full normal forms for the hybrid strategy, weak-head normal forms for the subsidiary strategy), and in addition the fact that one reduction rule does not apply in one of the context layers means that one could think of the grammar of potential redexes as being layered as well.

It therefore appears that in general all the grammars of a reduction semantics that uses a hybrid reduction strategy exhibit this two-layered structure. If this is the case, then the observations made in the present work about backward overlaps and outermost reduction constitutes the criteria under which a reduction semantics needs to use a hybrid strategy, and the backward chaining construction would be a useful tool to construct such a reduction semantics.

That said, we observe that the actual reduction strategy itself (normal-order, or leftmost-outermost) does not in fact change between the hybrid "strategy" and the subsidiary "strategy". It might therefore be more appropriate to refer to this type of reduction semantics as "hybrid reduction *semantics*".

8.5 Summary and conclusion

This chapter has introduced a number of open questions that have arisen in the course of our work. The questions can be summarized as follows:

- Some backward overlaps cause refocusing and the prelude to reduction semantics to be inapplicable, and some do not. What is the exact definition of a problematic backward overlap, and are the same backward overlaps problematic both with respect to refocusing and with respect to the prelude?
- Rule generalization based on backward chaining (as performed in Section 7.4.0) provides a useful stepping stone towards making refocusing applicable in the presence of backward overlaps. Can such rule generalizations be made mechanical, and can it be formalized such that rule generalization can be proved correct in general?
- In the presence of backward overlaps, a reduction semantics exhibiting unique decomposition appears to require a multi-layered grammar of reduction contexts. Can the connection between backward overlaps and multiple layers in the grammar of contexts be established formally? Is this connection related to the need for a two-stage version of the prelude to reduction semantics?
- If the grammar of reduction contexts in a reduction semantics does not exhibit unique decomposition, the syntactic correspondence appears to map the representation of that reduction semantics to a representation of an abstract machine which is not in defunctionalized form. Is there a formal connection between reduction semantics exhibiting unique decomposition and abstract machines in defunctionalized form?
- Rule generalization based on backward chaining appears to have a connection with García-Pérez and Nogueira's notion of "hybrid reduction strategy". Can such a connection be established formally?

Over the last 15 years, the syntactic and functional correspondences have been used extensively in programming languages that use innermost reduction strategies, both to verify the relative soundness between existing small-step semantics, abstract machines and big-step normalizers, and to derive new such semantic artifacts where none were known before. Given the large number of case studies, it must now be taken for granted that the correspondences capture a fundamental connection between these semantic descriptions.

The present work is the first to investigate whether the same fundamental connections exist for languages with an outermost reduction strategy. As can be seen from this chapter, this area is largely uncharted territory. The author hopes that the present work can provide an initial foray.

Part IV

Appendix

Appendix A

Example Representations

with Olivier Danvy and Ian Zerny

This chapter is dedicated to simple arithmetic expressions with natural numbers and subtractions, which we first present informally (Appendices A.1 to A.3). Its goal is to present a natural semantics (Appendix A.4), a denotational semantics in direct style (Appendix A.5), a continuation semantics (Appendix A.6), an abstract machine (Appendix A.7), a reduction semantics (Appendix A.8), and a structural operational semantics (Appendix A.9) – all expressed in the same pure subset of Standard ML as in the body of this work – that each specifies the meaning of these arithmetic expressions. This language of arithmetic expressions is simple enough so that each semantic artifact is concise; it is also complex enough to illustrate the salient traits of each semantics. We then show how to mechanically inter-derive these semantic artifacts (Appendix A.10).

NB. The previous paragraph is unavoidably technical, but each of its technical terms is explained in the glossary, in Appendix B.

A.1 Abstract syntax

An arithmetic expression is either a literal or a subtraction. We implement the abstract syntax of arithmetic expressions with the following ML data type:

```
datatype expr = LIT of int (* assumed to represent a natural number *)
              | SUB of expr × expr
```

Literals are assumed to represent natural numbers, i.e.,, to be non-negative integers. An ML value represents an abstract-syntax tree when it is inductively constructed using the constructors `LIT` and `SUB`. It has then the type `expr`.

So for example, `SUB (LIT 3, LIT 2)` represents $3 - 2$ and $(4 - 1) - (7 - 5)$ is represented by `SUB (SUB (LIT 4, LIT 1), SUB (LIT 7, LIT 5))`. (If one wanted to be pedantic, one would write that `SUB (LIT 3, LIT 2)` evaluates to an ML value that represents the abstract-syntax tree corresponding to $3 - 2$.)

A.2 Informal semantics

Any given arithmetic expression is computed by recursively carrying out all of its subtractions, unless one of them "underflows." An arithmetic expression is constructed inductively and thus it has a finite number of subtractions; therefore computation always terminates.

Overall, the meaning of a given expression is either a natural number (represented as a non-negative integer) or an error (or more precisely, an error message represented as a string). So for example, the meaning of $3 - 2$ is 1 and the meaning of $2 - 3$ is "underflow: $2 - 3$." Likewise for $(4 - 1) - (7 - 5)$ and $(7 - 5) - (4 - 1)$.

A.3 Towards formal semantics

An expression without any subtraction is said to be in *normal form*. Since ML does not support subtyping, we implement normal forms with the following specialized data type:

```
datatype expr_nf = NAT of int

(*  expr_nf → expr  *)
fun embed_nf (NAT n)
    = LIT n
```

Our notion of value is this normal form:

```
type value = expr_nf
```

As for the meaning of an expression as a natural number or an error, we implement it with the following data type:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string
```

A.4 Representation of a natural semantics

The relation between source expression and resulting value is traditionally implemented as a function [71, 91]. Here, the main function, `evaluate`, calls an auxiliary function, `evaluate_expr`, which is structurally recursive and traverses the given expression depth-first and from right to left. Each sub-expression evaluates to a natural number or an error message. For each subtraction, each of its operands is evaluated, and then if they both yield a natural number, the subtraction is performed, yielding a new natural number, or an error message is produced, where `toString` maps an integer to a string and `^` is an infix operator for concatenating strings. Error messages are blindly propagated until the initial call to the main evaluation function:

```
(*  expr → result_or_wrong  *)
fun evaluate_expr (LIT n)
    = RESULT (NAT n)
  | evaluate_expr (SUB (e1, e2))
    = (case evaluate_expr e2
         of (RESULT (NAT n2))
            ⇒ (case evaluate_expr e1
                 of (RESULT (NAT n1))
                    ⇒ if n1 < n2
                      then WRONG ("underflow: " ^
                                  (toString n1) ^ " - " ^ (toString n2))
                      else RESULT (NAT (n1 - n2))
                  | (WRONG s)
                    ⇒ WRONG s)
          | (WRONG s)
            ⇒ WRONG s)
```

```
(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr e
```

Note how `evaluate_expr` is compositional: each of its recursive calls is over a strictly smaller part of the expression, on the left side of the equal sign.

A.5 Representation of a denotational semantics in direct style

The original idea of denotational semantics is that the meaning of a source term is obtained by syntax-directed translation into a meta-language, the λ-calculus [121]. The result of this translation (the 'denotation' of the source term) is then simplified according to the rules of the meta-language [87]. Expressive meta-languages, however, have a way of morphing into practical programming languages, and the λ-calculus is no exception: nowadays, we do not write λ-terms, we write functional programs, and our functional programs are not simplified, they are executed in the spirit of weak-head normalization by evaluation. The valuation function of the denotational semantics of arithmetic expressions is therefore transliterated in (pure) ML to give the following evaluation function:

```
(*  expr → result_or_wrong  *)
fun evaluate_expr (LIT n)
    = RESULT (NAT n)
  | evaluate_expr (SUB (e1, e2))
    = (case evaluate_expr e2
         of (RESULT v2)
            ⇒ (case evaluate_expr e1
                  of (RESULT v1)
                     ⇒ apply_sub (v1, v2)
                   | (WRONG s)
                     ⇒ WRONG s)
          | (WRONG s)
            ⇒ WRONG s)
and apply_sub (NAT n1, NAT n2)
    = if n1 < n2
      then WRONG ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else RESULT (NAT (n1 - n2))

(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr e
```

This evaluation function is expressed as an eval/apply interpreter: `evaluate_expr` dispatches on the syntax of source expressions and `apply_sub` performs the subtraction of two values.

In the original spirit of denotational semantics, the denotation of `SUB (LIT 3, LIT 2)`, for example, reads as follows:

```
case RESULT (NAT 2)
  of (RESULT (NAT n2))
     ⇒ (case RESULT (NAT 3)
           of (RESULT (NAT n1))
              ⇒ if n1 < n2
                then WRONG ("underflow: " ^
                            (toString n1) ^ " - " ^ (toString n2))
                else RESULT (NAT (n1 - n2))
            | (WRONG s)
              ⇒ WRONG s)
```

```
    | (WRONG s)
      ⇒ WRONG s
```

where all the calls to `evaluate` and `evaluate_expr` have been unfolded. This denotation evaluates to `RESULT (NAT 1)`.

A.6 Representation of a continuation semantics

A continuation semantics is a denotational semantics where the valuation function is written using continuations. The main evaluation function, `evaluate`, calls an auxiliary function, `evaluate_expr`, with the source expression and an initial continuation. The auxiliary function is compositional, and all of its calls are tail calls. Its second argument is a functional accumulator that represents how to continue the computation once (and if) its first argument has evaluated to a natural number. In case of underflow, the continuation is unused and the result is an error message:

```
(*  expr × (value → result_or_wrong) → result_or_wrong  *)
fun evaluate_expr (LIT n, k)
    = k (NAT n)
  | evaluate_expr (SUB (e1, e2), k)
    = evaluate_expr (e2, fn v2 ⇒
        evaluate_expr (e1, fn v1 ⇒
          apply_sub (v1, v2, k)))
and apply_sub (NAT n1, NAT n2, k)
    = if n1 < n2
      then WRONG ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else k (NAT (n1 - n2))

(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr (e, fn v ⇒ RESULT v)
```

So for example, the denotation of `SUB (LIT 3, LIT 2)` reads as follows:

```
(fn (NAT n2) ⇒
   (fn (NAT n1) ⇒
      if n1 < n2
      then WRONG ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else (fn v ⇒ RESULT v) (NAT (n1 - n2)))
   (NAT 3))
(NAT 2)
```

where all the calls to `evaluate` and `evaluate_expr` have been unfolded. This denotation evaluates to `RESULT (NAT 1)`.

A.7 Representation of an abstract machine

The abstract machine has two states, `evaluate_expr` and `continue`, each of which is implemented with a function. The transitions are implemented with tail calls to either function. The evaluation context is a pushdown stack that is either empty (`C0`) or that contains an expression (pushed with `C1`) or a natural number (pushed with `C2`):

```
datatype cont = C0
              | C1 of expr × cont
              | C2 of cont × value
```

Each transition function maps a pair (expression and evaluation context for `evaluate_expr`, and evaluation context and value for `continue`) to the final result: a natural number or an error message. One function dispatches on the syntax of source expressions and the other dispatches on the structure of the context:

```
(*  expr × cont → result_or_wrong  *)
fun evaluate_expr (LIT n, k)
    = continue (k, NAT n)
  | evaluate_expr (SUB (e1, e2), k)
    = evaluate_expr (e2, C1 (e1, k))

(*  cont × value → result_or_wrong  *)
and continue (C0, v)
   = RESULT v
  | continue (C1 (e1, k), v2)
    = evaluate_expr (e1, C2 (k, v2))
  | continue (C2 (k, NAT n2), NAT n1)
    = if n1 < n2
      then WRONG ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else continue (k, NAT (n1 - n2))
```

The main evaluation function, `evaluate`, initializes the abstract machine by calling `evaluate_expr` with the input expression and the empty context:

```
(*  expr → result_or_wrong  *)
fun evaluate e = evaluate_expr (e, C0)
```

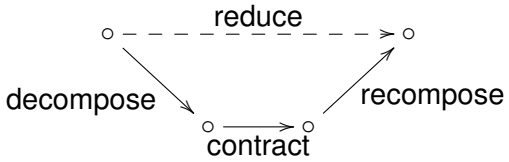
Note how this program is not compositional: the recursive call to `evaluate_expr`, in the second clause of `continue`, is not over a strictly smaller part of the expression, on the left side of the equal sign. It is therefore not obvious that this program always terminates, which it actually does.

For example, the meaning of `SUB (LIT 3, LIT 2)` is computed as follows, from the initial state to the final state:

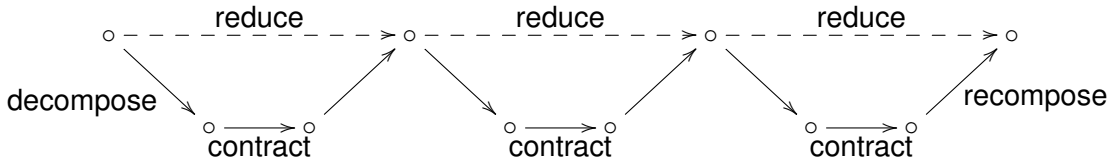
```
evaluate (SUB (LIT 3, LIT 2))
= {definition of evaluate }
evaluate_expr (SUB (LIT 3, LIT 2), C0)
= {definition of evaluate_expr }
evaluate_expr (LIT 2, C1 (LIT 3, C0))
= {definition of evaluate_expr }
continue (C1 (LIT 3, C0), NAT 2)
= {definition of continue }
evaluate_expr (LIT 3, C2 (C0, NAT 2))
= {definition of evaluate_expr }
continue (C2 (C0, NAT 2), NAT 3)
= {definition of continue and simplification of if 3 < 2 then WRONG ... else continue ... }
continue (C0, NAT 1)
= {definition of continue }
RESULT (NAT 1)
```

A.8 Representation of a reduction semantics

A reduction semantics is defined with a grammar of terms, a notion of normal form, a collection of potential redexes, a partial contraction function implementing the reduction rules (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 reduction context around the contractum, yielding a reduct:



Evaluation is defined as the iteration of one-step reduction:



Evaluation becomes stuck or yields a term in normal form. It is reduction-based because it enumerates the reduction sequence.

The grammar of terms It is defined in Appendix A.1.

The notion of normal form It is defined in Appendix A.3.

Potential redexes There is only one:

```
datatype potential_redex = PR_SUB of value × value
```

The result of contraction is either an expression or an error message:

```
datatype contractum_or_error = CONTRACTUM of expr
                             | ERROR      of string
```

The contraction function Given an actual redex, the contraction function yields an expression; otherwise it yields an error message:

```
(* potential_redex → contractum_or_error *)
fun contract (PR_SUB (NAT n1, NAT n2))
    = if n1 < n2
      then ERROR ("underflow: " ^ (toString n1) ^ " - " ^ (toString n2))
      else CONTRACTUM (LIT (n1 - n2))
```

The reduction strategy It determines the following grammar of reduction contexts:

```
datatype cont = C0
              | C1 of expr × cont
              | C2 of cont × value
```

Thus equipped, we define the notion of value or decomposition with the following data type:

```
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex × cont
```

We are then in position to define a decomposition function and a recomposition function. The decomposition function maps an expression in normal form to this normal form and an expression not in normal form to a potential redex and its reduction context:

```
(* decompose : expr → value_or_decomposition  *)
```

The recomposition function recomposes the reduction context over an expression; as such it is a left inverse of the decomposition function:

```
(* recompose : cont × expr → expr *)
```

The result of reduction is either an expression or an error message:

```
datatype reduct_or_stuck = REDUCT of expr
                         | STUCK  of string
```

We are then in position to define a decompose-contract-recompose function implementing one-step reduction:

```
(*  expr → reduct_or_stuck  *)
fun reduce e
    = (case decompose e
        of (VAL v)
           ⇒ STUCK "irreducible expression"
         | (DEC (pr, k))
           ⇒ (case contract pr
                of (CONTRACTUM e') ⇒ REDUCT (recompose (k, e'))
                 | (ERROR s)       ⇒ STUCK s))
```

Accordingly, we define a reduction-based evaluation function that iterates one-step reduction, using the result of `decompose` to detect whether we have reached a normal form:

```
(*  value_or_decomposition → result_or_wrong  *)
fun iterate (VAL v)
    = RESULT v
  | iterate (DEC (pr, k))
    = (case contract pr
        of (CONTRACTUM e') ⇒ iterate (decompose (recompose (k, e')))
         | (ERROR s)       ⇒ WRONG s)

(*  expr → result_or_wrong  *)
fun normalize e = iterate (decompose e)
```

This evaluation function enumerates the following reduction sequence, for example: the first redex in the term $(5-7)-(4-1)$ is $4-1$, which contracts to 3 and gives rise to the first reduct $(5-7)-3$; the first redex in this reduct is $5-7$, which is not an actual redex and thus yields the error message "underflow: $5-7$."

A.9 Representation of a structural operational semantics

Much like a reduction semantics, a structural operational semantics is defined with a grammar of terms (defined in Appendix A.1), a notion of normal form (defined in Appendix A.3), a collection of potential redexes (defined in Appendix A.8), a partial contraction function implementing the reduction rules (also defined in Appendix A.8), and a reduction strategy. Unlike a reduction semantics, however, a structural operational semantics does not have an explicit, syntactic representation of the reduction context as a term with a hole; it represents it implicitly as a proof tree.

An expression either contains no potential redex that is reachable via the reduction strategy, or it contains one. If no potential redex is reachable, the expression is stuck. If a potential redex is reachable, either it is an actual one, in which case this redex can be contracted and give rise to the next reduct in the reduction sequence, or it is not, and the expression is stuck. The result of one-step reduction is thus either the next expression in the reduction sequence or an error message: the given term is irreducible because it is a value or its first potential redex according to the reduction strategy is not an actual one.

We thus implement one-step reduction as a function mapping an expression to the following co-domain:

```
datatype reduct_or_stuck = REDUCT of expr
                         | STUCK  of string
```

This main function, `reduce`, calls an auxiliary function, `reduce_expr`, which implements the reduction strategy. Given a sub-expression, this auxiliary function yields either a value (the sub-expression contained no potential redex), or a new sub-expression containing a contractum (the sub-expression contained a potential redex that was an actual one), or an error message (the sub-expression contained a potential redex that was not an actual one):

```
datatype intermediate_result = VALUE     of value
                             | REDUCED   of expr
                             | ERRONEOUS of string
```

The one-step reduction function maps such an intermediate result to the final result:

```
(*  expr → reduct_or_stuck  *)
fun reduce e
    = (case reduce_expr e
         of VALUE v     ⇒ STUCK "irreducible expression"
          | REDUCED e'  ⇒ REDUCT e'
          | ERRONEOUS s ⇒ STUCK s)
```

The auxiliary function is structurally recursive and traverses the given expression depth-first and from right to left, constructing the next reduct at return time. Error messages are blindly propagated until the initial call to the main reduction function:

```
(*  expr → intermediate_result  *)
fun reduce_expr (LIT n)
    = VALUE (NAT n)
  | reduce_expr (SUB (e1, e2))
    = (case reduce_expr e2
         of (VALUE v2)
            ⇒ (case reduce_expr e1
                 of (VALUE v1)
                    ⇒ (case contract (PR_SUB (v1, v2))
                         of (CONTRACTUM e')
```

```
                                  ⇒ REDUCED e'
                              | (ERROR s)
                                  ⇒ ERRONEOUS s)
                    | (REDUCED e1')
                      ⇒ REDUCED (SUB (e1', embed_nf v2))
                    | (ERRONEOUS s)
                      ⇒ ERRONEOUS s)
          | (REDUCED e2')
            ⇒ REDUCED (SUB (e1, e2'))
          | (ERRONEOUS s)
            ⇒ ERRONEOUS s)
```

Evaluation is implemented by iterating one-step reduction and yields either a value or an error message:

```
datatype result_or_wrong = RESULT of value
                         | WRONG  of string

(*  expr → result_or_wrong  *)
fun reduce_star e
    = (case reduce_expr e
         of VALUE v     ⇒ RESULT v
          | REDUCED e'  ⇒ reduce_star e'
          | ERRONEOUS s ⇒ WRONG s)

(*  expr → result_or_wrong  *)
fun evaluate e
    = reduce_star e
```

As in Appendix A.8, this evaluation function enumerates the following reduction sequence, for example: the first redex in the term $(5-7)-(4-1)$ is $4-1$, which contracts to 3 and gives rise to the first reduct $(5-7)-3$; the first redex in this reduct is $5-7$, which is not an actual redex and thus yields the error message "underflow: $5-7$."

A.10 Inter-deriving representations

All the semantic artifacts from Appendix A.4 to Appendix A.9 are different means to a same end: to formally define the meaning of arithmetic expressions. And as it happens, these representations can be mechanically inter-derived using a toolbox of program transformations, as detailed in the accompanying file:

- Inlining the `apply` function in the denotational semantics of Appendix A.5 yields the natural semantics of Appendix A.4.
- CPS-transforming the denotational semantics of Appendix A.5, using the type isomorphism from Chapter 3 to split its continuation, and lambda-dropping the resulting error continuation yields the continuation semantics of Appendix A.6.
- Defunctionalizing the continuation in the continuation semantics of Appendix A.6 yields the abstract machine of Appendix A.7.
- The reduction semantics of Appendix A.8 syntactically corresponds to the abstract machine of Appendix A.7.

- CPS-transforming the reduction function of the structural operational semantics of Appendix A.9, using the type isomorphism from Chapter 3 to split its continuation, lambda-dropping the resulting error continuation, and defunctionalizing the remaining continuation yields the constitutive elements of the reduction semantics of Appendix A.8 (decomposition, contraction, and recomposition).

Appendix B

Technical Glossary

with Olivier Danvy and Ian Zerny

Each entry is indented, and each reference to an entry is italicized.

abstract machine State-transition system where each transition requires no subgoal. A push-down automaton, for example, is implemented with an abstract machine. See *big-step abstract machine* and *small-step abstract machine* for two concrete examples.

actual redex *Potential redex* for which a *reduction rule* applies. See *reduction rules* and *unique decomposition* for two concrete examples.

artifact Man-made construct.

'big-step' abstract machine *Abstract machine* where transitions happen in one big step rather than step by step (see *small-step abstract machine*). For example, each state of the state-transition system is implemented with an *ML* function and each transition is implemented as a conditional clause and a *tail call* to the function that implements the next state. Each transition function maps the current state to the final result, if there is one. For example, the following big-step abstract machine implements a deterministic finite automaton that recognizes whether a given list of Booleans contains an even number of occurrences of `true`:

```
fun even nil            = true
  | even (true  :: bs) = odd bs
  | even (false :: bs) = even bs
and odd  nil            = false
  | odd  (true :: bs)  = even bs
  | odd  (false :: bs) = odd bs

fun main bs = even bs
```

A big-step abstract machine forms an ideal candidate for *lightweight fission by fixed-point demotion* to obtain a small-step abstract machine.

call Short for "function call."

call unfolding Action of consecutively *inlining* the name of a function in an application and performing the subsequent substitutions and simplifications. For example, let us unfold the call to `foo` in `foo (bar 10, 11)`, where `foo` denotes `fn (x, y)⇒ (x × x)+ (y × y)`:

```
foo (bar 10, 11)
```
= { `foo` is inlined }

```
(fn (x, y)⇒ (x × x)+ (y × y))(bar 10, 11)
```
= { the formal parameters are bound to the actual parameters }
```
let val x = bar 10 val y = 11 in (x × x)+ (y × y)end
```
= { `y` denotes a value: it is inlined and the corresponding multiplication is simplified }
```
let val x = bar 10 in (x × x)+ 121 end
```

Similarly, let us unfold the call to `foo` in the left-to-right call-by-value CPS counterpart of the same example (see *continuation-passing style* and *CPS transformation*):

```
fn k ⇒bar (10, fn v1 ⇒foo (v1, 11, k))
```
= { `foo` is inlined }
```
fn k ⇒bar (10, fn v1 ⇒(fn (x, y, k)⇒k ((x × x)+ (y × y)))(v1, 11, k))
```
= { the formal parameters are bound to the actual parameters }
```
fn k ⇒bar (10, fn v1 ⇒ let val x = v1 val y = 11 val k = k
                        in k ((x × x)+ (y × y))end)
```
= { `x`, `y`, and `k` denote values: they are inlined and the multiplication is simplified }
```
fn k ⇒bar (10, fn v1 ⇒k ((v1 × v1)+ 121))
```

closure A λ-term together with the bindings of its free variables, i.e., its lexical *environment*. As Biernacka and Danvy put it [13]: "This alternative to substitution is due to Hasenjaeger in logic [109, § 54] and to Landin in computer science [77]. In logic, it makes it possible to reason by structural induction over λ-terms (since they do not change across β-reduction), and in computer science, it makes it possible to compile λ-terms (since they do not change at run time). [...] The term is frequently overloaded (as in: a reflexive and transitive closure, a compatible closure, and also a closed term)."

closure conversion Particular case of *defunctionalization* where the function space is represented with one unary summand and the dispatch function is *inlined*. For example, closure-converting the *ML* implementation of a *denotational semantics* in direct style for a higher-order language gives the ML implementation of a *natural semantics* for this higher-order language.

closure unconversion Left inverse of *closure conversion*.

compiler Program for translating other programs from one language to another.

compositionality Property of a function that is recursive and defined by cases over an inductive data type: a function is compositional when all its recursive calls are over a proper subpart of what was on the left-hand side of the equal sign. A structurally recursive function over an inductive data type is always compositional: its control-flow graph is isomorphic to this data type. In general, *defunctionalizing* a compositional function that is *higher-order* (e.g., the evaluation function in Section A.6) yields a *first-order* function that is not compositional (e.g., the abstract machine in Section A.7).

context Literally: what surrounds something. Here: a term with a hole [9].

context-insensitive contraction Application of a *reduction rule* to an *actual redex*, independently of the *reduction context*, and resulting in a *contractum*. For one example, here is a *reduction semantics* for the pure λ-calculus with the left-to-right applicative-order *reduction*

strategy:

$$\begin{aligned}\langle\text{expression}\rangle &::= x \mid \lambda x.\langle\text{expression}\rangle \mid \langle\text{expression}\rangle\ \langle\text{expression}\rangle\\ \langle\text{value}\rangle &::= \lambda x.\langle\text{expression}\rangle\\ \langle\text{context}\rangle &::= [\,] \mid \langle\text{context}\rangle\ \langle\text{expression}\rangle \mid \langle\text{value}\rangle\ \langle\text{context}\rangle\\ \langle\text{potential redex}\rangle &::= \langle\text{value}\rangle\ \langle\text{value}\rangle\end{aligned}$$

The (β-)reduction rule reads as follows:

$$(\lambda x.e)\ v \to e[v/x]$$

Note how β-reduction is performed independently of the current reduction context. See *reduction rules* and *unique decomposition* for two other examples.
This reduction semantics is not random: Biernacka and Danvy have shown that the *syntactic correspondence* maps it into a canonical substitution-based *abstract machine* [13] – a machine that itself is mapped by the *functional correspondence* to the canonical substitution-based *evaluation functions* that are also agreed upon to specify the left-to-right call-by-value evaluation of λ-terms.

context-sensitive contraction Application of a *reduction rule* to an *actual redex* together with its *reduction context*, and resulting in a *contractum* and its reduction context. This sensitivity makes it possible to capture the current reduction context and to reinstate a captured context when specifying a control operator or coroutines. For one example, here is a *reduction semantics* for the pure λ-calculus with the left-to-right applicative-order *reduction strategy* and the control operator callcc [21], which induces quoted contexts in the syntax:

$$\begin{aligned}\langle\text{expression}\rangle &::= x \mid \lambda x.\langle\text{expression}\rangle \mid \langle\text{expression}\rangle\ \langle\text{expression}\rangle\\ &\quad \mid \text{callcc}\ \langle\text{expression}\rangle \mid \ulcorner\langle\text{context}\rangle\urcorner\\ \langle\text{value}\rangle &::= \lambda x.\langle\text{expression}\rangle \mid \ulcorner\langle\text{context}\rangle\urcorner\\ \langle\text{context}\rangle &::= [\,] \mid \langle\text{context}\rangle\ \langle\text{expression}\rangle \mid \langle\text{value}\rangle\ \langle\text{context}\rangle\\ \langle\text{potential redex}\rangle &::= \langle\text{value}\rangle\ \langle\text{value}\rangle \mid \text{callcc}\ \langle\text{value}\rangle\end{aligned}$$

The reduction rules read as follows:

$$\begin{aligned}((\lambda x.e)\ v,\ C) &\to (e[v/x],\ C)\\ (\ulcorner C' \urcorner\ v,\ C) &\to (v,\ C')\\ (\text{callcc}\ v,\ C) &\to (v\ \ulcorner C\urcorner,\ C)\end{aligned}$$

See how β-reduction is performed independently of the current reduction context, and how the other reduction rules act on it: applying a captured context reinstates it, and capturing the current context copies it into a quotation.
This reduction semantics is not random: Biernacka and Danvy have shown that the *syntactic correspondence* maps it into the canonical substitution-based *abstract machine* that is agreed upon to specify callcc [12] – a machine that itself is mapped by the *functional correspondence* to the canonical substitution-based continuation-passing *evaluation function* that is also agreed upon to specify callcc.

continuation Functional representation of the rest of the computation [103]. Applying a continuation to a value has the effect of completing the computation. Got its name to represent the meaning of labels in the range of gotos in denotational semantics [124].

continuation-passing style Programming style where all intermediate results are named, their computation is sequentialized, all *calls* are *tail calls*, and all functions are passed a functional accumulator representing the rest of the computation, the *continuation*. Such a program is indifferent to the evaluation order (e.g., call by name or call by value) of its *meta-language* [97].

contraction Mapping of an *actual redex* to a *contractum* according to a *reduction rule*.

contraction function Partial function implementing *contraction* that maps a *potential redex* to a *contractum* if the potential redex is an *actual redex*. See also *reduction rule*.

contractum Result of *contraction*. See also *reduction rule*.

corridor transitions In a transition system, transition to a state for which the next transition is unconditional.

CPS Acronym of *continuation-passing style*. Due to Guy Steele [119].

CPS transformation Transformation of a program from ordinary direct style to the eponymous *continuation-passing style*. Each CPS transformation encodes an evaluation order [69]. Pure and total (i.e., effect-free) functions need not be CPS-transformed for Plotkin's indifference property to hold [35].
For example, let us transform the following term into CPS, using right-to-left call by value, given three functions `foo`, `bar`, and `baz` and two variables `x` and `y`:

```
val e1 = foo (x, y + bar 10, 11, baz 12)
(* naming and sequentialization: *)
val e2 = let val v1 = baz 12
             val v2 = bar 10
         in foo (x, y + v2, 11, v1) end
(* introducing the continuation: *)
val e3 = fn k ⇒ baz (12, fn v1 ⇒
                    bar (10, fn v2 ⇒
                      foo (x, y + v2, 11, v1, k)))
```

The addition function (the infix `+` just above) is kept in direct style because it is pure and total.
The mindful reader is welcome to verify that CPS-transforming the initial (resp. final) term of the first example in the entry on *call unfolding* yields the initial (resp. final) term of the second example in the same entry: call unfolding and CPS transformation commute with each other.

decomposition For any given *non-value term*, pair that contains a *potential redex* and its *reduction context* such that *recomposing* the reduction context around the potential redex yields the given non-value term.

decomposition function For a given *reduction strategy* that is deterministic, function that maps a *value term* to itself and a *non-value term* to the *decomposition* determined by the *reduction strategy*. See also *unique decomposition*.

decomposition relation Relation between a *non-value term* and its *decompositions*.

deforestation Program transformation that prevents intermediate data structures (typically trees) from being constructed when composing functions. The term is due to Philip Wadler [129].

defunctionalization Generalization of Landin's *closure conversion* due to John Reynolds [102]

that 'firstifies' a *higher-order program* into a *first-order program* where the inhabitants of each higher-order function space are partitioned into first-order summands. A function is defunctionalized into a collection of summands grouped in a data type, and a first-order apply function dispatching on the constructors of this data type, i.e., these summands. A functional value is introduced by injecting a first-order value in the sum, and it is eliminated by calling the apply function.
For example, consider the following higher-order program:

```
(*  (int → int) → int *)
fun aux f = f 10

(*  int → int × int *)
fun main c = (aux (fn a ⇒ a + 1), aux (fn b ⇒ b + c))
```

The main function pairs the result of applying the auxiliary function to two functions. Let us defunctionalize these two arguments. Two classes of functional values inhabit this function space: the one corresponding to `fn a ⇒a + 1`, which has no free variables, and the one corresponding to `fn b ⇒b + c`, which has one free variable, `c`. We therefore defunctionalize this function space into a data type with two constructors (one zero-ary, and one unary) that are interpreted by an apply function dispatching on them:

```
datatype funct = F1
               | F2 of int

(*  funct → int → int  *)
fun apply_funct F1     = (fn a ⇒ a + 1)
  | apply_funct (F2 c) = (fn b ⇒ b + c)
```

In practice, `apply_funct` is always fully applied and therefore it is uncurried so that its type reads $funct \times int \to int$.
At any rate, we defunctionalize the functional values by introducing them with one of the data-type constructors, and eliminating them with a call to `apply_funct`:

```
(*  funct → int *)
fun aux f = apply_funct f 10

(*  int → int × int *)
fun main c = (aux F1, aux (F2 c))
```

In this example, there is one partition with two summands. In general, the number of partitions and their granularity may vary: there can be one partition and one dispatch function, as in Reynolds's original work; there can be as many partitions as there are function types [10]; and there can be even more of them if one uses control-flow information.

denotational semantics Historically the first *formal semantics* for programming languages. The meaning of a given term is obtained by mapping it (compositionally) to a λ-term and then by normalizing this denotation. Due to Christopher Strachey [122] and Dana Scott [111].

direct-style transformation Left inverse of the *CPS transformation* [25, 38].

'eval/apply' interpreter Style of *interpreter* typically for a functional language with an "eval" function dispatching on the syntax of source expressions and with an "apply" function that applies the denotation of functions to the denotation of their arguments. The notion is due to John McCarthy for his meta-circular Lisp interpreter [80].

'eval/continue' abstract machine Style of *abstract machine* for a language of expressions with an "eval" transition dispatching on the syntax of source expressions and with a "continue" transition dispatching on the structure of the context. Typically obtained as the result of *defunctionalizing* an *evaluation function* in *continuation-passing style* for this language of expressions, as done in the *functional correspondence*.

'eval/apply/continue' abstract machine Style of *abstract machine* typically for a functional language with an "eval" transition dispatching on the syntax of source expressions, an "apply" transition to apply functions to their arguments, and a "continue" transition dispatching on the structure of the context. Typically obtained as the result of *defunctionalizing* an *eval/apply interpreter* in *continuation-passing style* for this language of expressions, as done in the *functional correspondence*.

evaluation Interpretation for a language of expressions.

evaluation context First-order representation of the rest of the evaluation. Its grammar is linear and can be mechanically obtained as the data-type part of the *defunctionalized continuation* of an *evaluation function*.

evaluation function *Interpreter* for a language of expressions. Typically specified by a big-step semantics such as *denotational semantics* or *natural semantics*.

evaluator Nickname of *evaluation function*.

filling a context with a term See *recomposition function*.

first-order function Function whose argument is not and does not contain a function. For example, the factorial function is first-order: it maps a number to another number.

first-order program Program containing only *first-order functions*.

fission Inverse of *fusion*.

formal semantics Specification of a meaning using a formal system, e.g., the λ-calculus, domain theory, or mathematical logic.

functional correspondence Series of program transformations connecting *evaluation functions* and *abstract machines*, and consisting in *lambda-lifting* (not used in this work since none of the evaluation functions use block structure), *closure conversion* (not used in this work since all the expressions are first order), *CPS transformation*, and *defunctionalization*. Laid out in John Reynolds's foundational article "Definitional Interpreters" [102], but only identified as such 30 years later [2]. Very versatile: is also used, e.g., for web programming [64].

fusion Act of merging the construction of an intermediate result with its subsequent deconstruction when two functions are composed. Due to William Burge [20] and Peter Landin. *Deforestation* is a form of fusion.

higher-order function Function taking another function as an argument. Any *second-order*, *third-order*, etc. function is higher order.

higher-order program Program containing some *higher-order functions*.

inlining Replacing a name by its denotation.

interpreter Program for executing other programs.

lambda-dropping Left inverse of *lambda-lifting* [44].

lambda-interpreter Nickname of an interpreter for λ-terms. Due to Guy Steele and Gerry Sussman [120].

lambda-lifting Transformation of a scope-sensitive block-structured functional program into scope-insensitive recursive equations. Due to Thomas Johnsson [75].

lightweight fission by fixed-point demotion Left inverse of *lightweight fusion by fixed-point promotion*.

lightweight fusion by fixed-point promotion Form of *fusion* due to Atsushi Ohori and Isao Sasano [93] between a stepping function and a driver loop. Has been found to fittingly connect *small-step abstract machines* and *big-step abstract machines* [39].

literal Compiler jargon for a constant (Boolean, character, integer, etc.) in a program.

meta-language Language used to describe another language. When the description is an implementation, e.g., an interpreter, then the meta-language is called an 'implementation language' and the language is called a 'source language.'

ML Family of functional programming languages that are call by value and polymorphically typed. Originally the name of the *meta-language* of LCF in Edinburgh [98]. Due to Robin Milner.

mutatis mutandis Latin for "changing only what needs to be changed."

natural semantics Big-step *operational semantics* with an implicit, logical representation of the *evaluation context* as a proof tree. Inspired by natural deduction and due to Gilles Kahn [76].

non-value term Term that is not in *normal form*. See also *stuck term*.

normal form Term containing no *potential redex* that can be reached through a given *reduction strategy*.

normalization Process of reducing a term into the corresponding *normal form* according to some *reduction strategy*.

normalization by evaluation See *reduction-free normalization*.

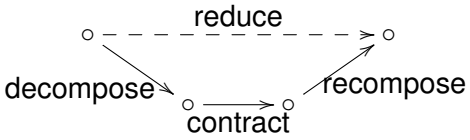
normalization by reduction See *reduction-based normalization*.

normalization function For a given *reduction strategy* that is deterministic, partial function that maps a term to its normal form, if this normal form exists. Typically specified by iterating the *one-step reduction function* of a small-step *operational semantics* or by a big-step semantics such as *denotational semantics* or *natural semantics*.

normalization relation For a given *reduction strategy*, the relation between terms and their *normal form*. Typically specified as the transitive closure of a *one-step reduction relation*.

one-step reduction function For a given *reduction strategy* that is deterministic, partial function that maps a term to the next term in the *reduction sequence*, if this next term exists. Typically specified by a small-step *operational semantics* such as *structural operational semantics* or *reduction semantics*. Operates by (1) finding a *potential redex* in a *reduction context* in a given term using the *reduction strategy*, (2) contracting this potential redex, if it is an *actual one*, and (3) *recomposing* the *reduction context* around the resulting *contrac-*

tum, either explicitly in a reduction semantics or implicitly in a structural operational semantics:



one-step reduction relation For a given *reduction strategy*, relation between a term containing an *actual redex* that can be reached using the *reduction strategy* and the term where this actual redex is replaced by the corresponding *contractum*.

operational semantics *Formal semantics* for programming languages where the execution of a program is described directly, e.g., by rewriting this program or by executing it on an *abstract machine*.

PLT Redex Graphical tool of choice for engineering *reduction semantics* [54].

plugging a term in a context See *recomposition function*.

potential redex Term for which a *reduction rule* might apply. See *reduction rule* and *unique decomposition* for two concrete examples.

prelude to a reduction semantics Derivation of the *reduction contexts* of a *reduction semantics* out of the *search function* implementing a deterministic *reduction strategy* [47]. The derivation consists of *CPS transformation* and *defunctionalization*. The data type of the defunctionalized continuation is that of the reduction contexts, which are notoriously difficult to state correctly, even with experience. (For example (Matthew Flatt, personal communication to Danvy, 2011), try to write the reduction contexts for the λ-calculus with applicative order and exceptions.) All the reduction contexts presented in this work have been obtained using this prelude.

'push/enter' abstract machine Style of *abstract machine* for a language of expressions. Typically obtained as the result of inlining the "continue" transition of an *eval/continue abstract machine* when it is not recursive. This term is due to Simon Marlow and Simon Peyton Jones [79]. (NB. For terminological consistency, we do not overload the term *eval/apply* in favor of *eval/continue*.)

rational deconstruction Left inverse of the *functional correspondence*. Initially its stepping stone [28].

recomposition function Left inverse of the *decomposition function*.

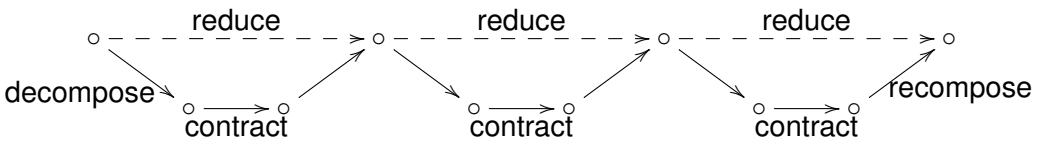
redex Originally, short for "reducible expression." See *actual redex*.

redexes Plural form of *redex*.

redices Alternative plural form of *redex*, in reference to the plural form of 'index.'

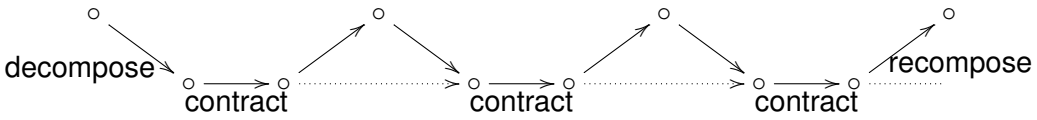
reduct Element of a *reduction sequence*.

reduction-based normalization function *Normalization function* that enumerates the *reduction sequence* to compute its last element (i.e., the normal form), if there is one:



reduction context *Context* in which a *potential redex* can be found according to the *reduction strategy* (see also *decomposition*). In a *structural operational semantics*, is implicitly represented as a proof tree. In a *reduction semantics*, is explicitly specified with a linear grammar. This grammar can be mechanically obtained as the data-type part of the *defunctionalized continuation* of the *search function* implementing the *reduction strategy*, if this strategy is deterministic. See *prelude to a reduction semantics*.

reduction-free normalization function *Normalization function* that does not enumerate the *reduction sequence* to compute its last element (i.e., the normal form), if there is one:



For example, a compositional *evaluation function* and an *abstract machine* are reduction free.

reduction order See *reduction strategy*.

reduction rule Pair consisting of an *actual redex* and a *contractum* that may contain meta-variables ranging over the grammar of terms. For example, consider the following toy language of expressions:

$$\begin{aligned}
\langle \text{boolean} \rangle &::= \text{true} \mid \text{false} \\
\langle \text{integer} \rangle &::= 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid 3 \mid -3 \\
\langle \text{value} \rangle &::= \langle \text{boolean} \rangle \\
&\quad \mid \langle \text{integer} \rangle \\
\langle \text{expression} \rangle &::= \textsf{if } \langle \text{expression} \rangle \textsf{ then } \langle \text{expression} \rangle \textsf{ else } \langle \text{expression} \rangle \\
&\quad \mid \langle \text{expression} \rangle / \langle \text{expression} \rangle
\end{aligned}$$

If the reduction rules are the expected ones, i.e., the following three pairs, noted with an infix right arrow and containing the meta-variables e_1, e_2, n_1, n_2, and n_3,

$$\begin{aligned}
\textsf{if true then } e_1 \textsf{ else } e_2 &\to e_1 && \text{for all expressions } e_1 \text{ and } e_2 \\
\textsf{if false then } e_1 \textsf{ else } e_2 &\to e_2 && \text{for all expressions } e_1 \text{ and } e_2 \\
n_1 / n_2 &\to n_3 && \text{for all integers } n_1, n_2, \text{ and } n_3 \\
& && \text{such that } n_2 \neq 0 \text{ and } n_3 \text{ is the quotient of } n_1 \text{ over } n_2
\end{aligned}$$

then the grammar of *potential redexes* reads as follows:

$$\begin{aligned}
\langle \text{potential redex} \rangle &::= \textsf{if } \langle \text{value} \rangle \textsf{ then } \langle \text{expression} \rangle \textsf{ else } \langle \text{expression} \rangle \\
&\quad \mid \langle \text{value} \rangle / \langle \text{value} \rangle
\end{aligned}$$

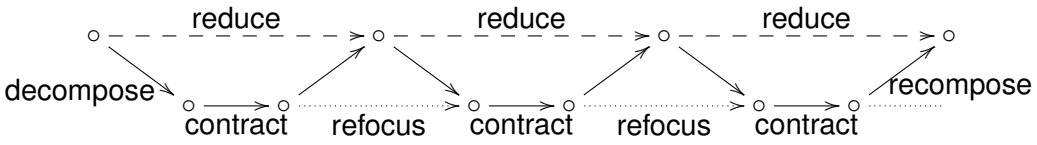
The three terms 8/4, 5/true, and 5/0 all are potential redexes. Among these, only the first one is an *actual redex*, i.e., a potential redex for which a reduction rule applies; its *contractum* is 2. The second one contains a type error, and the third one contains a division by zero; these two terms are *stuck*.

reduction semantics Small-step *operational semantics* with an explicit, syntactic representation of the *reduction context*. Due to Matthias Felleisen [53].

reduction sequence Series of intermediate terms (the *reducts*) obtained by successive applications of a *one-step reduction function*, starting with a given term.

reduction strategy Strategy to find a *potential redex* in a given term. Examples include leftmost-outermost, rightmost-innermost, etc. Specified by 'congruence rules' or again 'compatibility rules' in a *structural operational semantics* and by the grammar of *reduction contexts* in a *reduction semantics*.

refocus function Composition of the *recomposition function* and the subsequent *decomposition function* in a *reduction-based normalization function*:



refocusing *Deforestation* of the *reducts* in a reduction sequence [43]. Is achieved by writing a deforested version of the *refocus function*.

refunctionalization Left inverse of *defunctionalization* [41].

search function Function implementing a deterministic *reduction strategy* that maps a *non-value* term to the first *potential redex* according to this *reduction strategy*.

second-order function Function taking a *first-order function* as argument. For example, in the example of the entry about *defunctionalization*, `aux` is second order.

'small-step' abstract machine *Abstract machine* where transitions happen step by step rather than in one big step (see *big-step abstract machine*). For example, each state of this state-transition system is implemented with an *ML* data-type constructor and each transition is implemented with a conditional clause in an ML function that maps the current state to the next state. An outer driver loop makes the machine progress. For example, the following small-step abstract machine implements a deterministic finite automaton that recognizes whether a given list of Booleans contains an even number of occurrences of `true`:

```
datatype intermediate_state = EVEN of bool list
                            | ODD  of bool list

datatype state = INTERMEDIATE of intermediate_state
               | FINAL        of bool

fun single_step (EVEN nil)           = FINAL        true
  | single_step (EVEN (true  :: bs)) = INTERMEDIATE (ODD bs)
  | single_step (EVEN (false :: bs)) = INTERMEDIATE (EVEN bs)
  | single_step (ODD  nil)           = FINAL        false
  | single_step (ODD  (true  :: bs)) = INTERMEDIATE (EVEN bs)
  | single_step (ODD  (false :: bs)) = INTERMEDIATE (ODD bs)

fun driver_loop (FINAL b)         = b
  | driver_loop (INTERMEDIATE is) = driver_loop (single_step is)

fun main bs = driver_loop (INTERMEDIATE (EVEN bs))
```

A small-step abstract machine forms an ideal candidate for *lightweight fusion by fixed-point promotion* to obtain a big-step abstract machine.

Standard ML (SML) Standardized version of *ML* [85].

Standard ML of New Jersey Conservative extension of *Standard ML* originally developed by Andrew Appel and Dave MacQueen [6]. Programming language of discourse in the present work. Features the control operator callcc [68]. See *context-sensitive contraction*.

structural operational semantics Small-step *operational semantics* with an implicit, logical representation of the *reduction context* as a proof tree. Due to Gordon Plotkin [99].

stuck term For a given *reduction strategy*, term for which the *potential redex* that can be reached is not an *actual redex*. See also *reduction rule*.

syntactic correspondence Series of program transformations connecting the reduction-based evaluation function of a reduction semantics and an abstract machine, and consisting of *refocusing*, inlining the *contraction function*, *lightweight fusion by fixed-point promotion*, *transition compression*, and context specialization. Laid out by Biernacka and Danvy [12, 13].

tail call Function call that is in *tail position*.

tail position In the body of a λ-abstraction, an expression is in tail position if it is the last expression that needs to be evaluated to complete the evaluation of this body. For example, in $\lambda f.\lambda x.f\ (f\ x)$, two sub-expressions are in tail position: the λ-abstraction $\lambda x.f(fx)$ and the outer application $f(fx)$. The inner application fx is not in tail position.

third-order function Function taking a *second-order function* as argument.

transition compression Symbolic composition of transitions to eliminate *corridor transitions*. For example, given a corridor transition from state A to state B and an unconditional transition from state B to state C, the transition from A to B is replaced by a transition from A to C. Each transition compression can give rise to another compression opportunity; for that reason, transition compression is also referred to as 'hereditary.' After transition compression, unreachable corridor transitions are removed.

unique decomposition For any given *potential redexes* and *reduction contexts*, the non-trivial property [134] that if a term can be *decomposed* into a potential redex and its reduction context, this decomposition is unique. For example, consider the following toy language of expressions:

$$\begin{aligned} \langle \text{integer} \rangle &::= 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid 3 \mid -3 \\ \langle \text{value} \rangle &::= \langle \text{integer} \rangle \\ \langle \text{expression} \rangle &::= \langle \text{expression} \rangle + \langle \text{expression} \rangle \end{aligned}$$

If the *reduction rule* is the expected one,

$$n_1 + n_2 \to n_3 \quad \text{where } n_3 \text{ is the sum of } n_1 \text{ and } n_2$$

then the grammar of potential redexes reads as follows:

$$\langle \text{potential redex} \rangle ::= \langle \text{value} \rangle + \langle \text{value} \rangle$$

If the grammar of reduction contexts reads

$$\langle \text{context} \rangle ::= [\,] \mid \langle \text{context} \rangle + \langle \text{expression} \rangle \mid \langle \text{value} \rangle + \langle \text{context} \rangle$$

then the unique decomposition property holds. However, if it reads

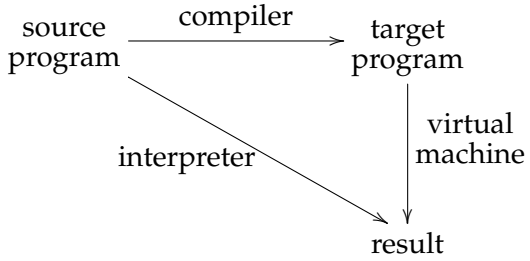
$$\langle \text{context} \rangle ::= [\,] \mid \langle \text{context} \rangle + \langle \text{expression} \rangle \mid \langle \text{expression} \rangle + \langle \text{context} \rangle$$

then the unique decomposition property does not hold, since a term such as $(1+2)+(3+4)$ can be decomposed both into $1+2$ and $[\,]+(3+4)$ and into $3+4$ and $(1+2)+[\,]$.

value Result of *evaluation* and/or nickname of *value term*.

value term Term in *normal form*.

virtual machine Run-time component of the factorized version of a source-language *interpreter*, as depicted in the following traditional commuting diagram:



The virtual machine is a target-language interpreter.

Bibliography

[1] 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. 5

[2] 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. 3, 4, 33, 58, 74, 96, 128

[3] 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. 4

[4] 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. 96

[5] 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. 5, 43, 97

[6] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Małuszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag. 133

[7] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis 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. 5, 98

[8] Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In Zhong Shao, editor, *Programming Languages and Systems, 23rd European Symposium on Programming, ESOP 2014*, number 8410 in Lecture Notes in Computer Science, Grenoble, France, April 2014. Springer. 5, 98

[9] 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. 124

[10] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 32, No. 8, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press. 127

[11] 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. 5

[12] 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. 42, 43, 46, 58, 80, 125, 133

[13] 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. 3, 4, 23, 26, 42, 46, 53, 103, 106, 124, 125, 133

[14] 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 [94], pages 186–206. 5

[15] 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. 5

[16] 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. 96

[17] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press. 139, 141

[18] Nicholas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972. 92

[19] 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. 96

[20] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975. 128

[21] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985. 125

[22] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press. 31

[23] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202, 1987. 95

[24] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991. 95

[25] 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). 33, 35, 68, 75, 93, 127

[26] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer. 92

[27] Olivier Danvy. A new one-pass transformation into monadic normal form. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 77–89, Warsaw, Poland, April 2003. Springer. 91

[28] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Grelck et al. [65], pages 52–71. 130

[29] Olivier Danvy. Sur un exemple de Patrick Greussay. Research Report BRICS RS-04-41, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2004. 19

[30] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006. 5, 107

[31] 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. 26, 42, 46, 53, 74, 104, 106

[32] 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. 42

[33] Olivier Danvy. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines. In Palsberg [94], pages 162–185. 5, 96

[34] 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. 35, 63, 89

[35] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag. 126

[36] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *Journal of Computer and System Sciences*, 76:302–323, 2010. 97, 98

[37] Olivier Danvy and Jacob Johannsen. From outermost reduction semantics to abstract machine. In Gopal Gupta and Ricardo Peña, editors, *Logic Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, revised selected papers*, number 8901 in Lecture Notes in Computer Science, pages 91–108, Madrid, Spain, September 2013. Springer. 41, 79

[38] 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. 35, 127

[39] 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. 57, 65, 68, 70, 98, 129

[40] 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. 33, 43

[41] 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. 33, 37, 68, 97, 132

[42] 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. 63, 68, 74

[43] 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. 10, 25, 26, 42, 49, 53, 55, 68, 69, 103, 105, 106, 132

[44] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000. 129

[45] Olivier Danvy and Ian Zerny. A synthetic operational account of call-by-need evaluation. In Schrijvers and Peña [110], pages 97–108. 4, 98, 107

[46] Olivier Danvy and Ian Zerny. Three syntactic theories for combinatory graph reduction. *ACM Transactions on Computational Logic*, 14(4):1–29, 2013. 4, 98

[47] Olivier Danvy and Ian Zerny. A prequel to reduction semantics. Chapter 5 of [135], June 2013. 23, 44, 58, 98, 130

[48] Olivier Danvy, Kevin Millikin, and Johan Munk. A correspondence between full normalization by reduction and full normalization by evaluation. Manuscript, July 2007. 27, 54, 80, 104, 107

[49] 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. 43, 57, 58, 98

[50] Nachum Dershowitz. Termination of linear rewriting systems (preliminary version). In Shimon Even and Oded Kariv, editors, *Automata, Languages, and Programming, 8th Colloquium*, number 115 in Lecture Notes in Computer Science, pages 448–458, Acre (Akko), Israel, July 1981. Springer-Verlag. 7, 8, 43, 47, 54, 79, 81

[51] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3 (1/2):69–116, 1987. 54

[52] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987. 97

[53] 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. 42, 67, 131

[54] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 3, 42, 43, 67, 130

[55] Andrzej Filinski. Representing monads. In Boehm [17], pages 446–457. 89

[56] Robert W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science: Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967. 95

[57] 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. 16, 70

[58] 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). 95

[59] Álvaro García-Pérez. *Operational Aspects of Full Reduction in Lambda Calculi*. PhD thesis, Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software, Universidad Politecnica de Madrid, Madrid, Spain, 2014. 5, 27, 54, 80, 104, 107, 108

[60] Álvaro García-Pérez and Pablo Nogueira. A syntactic and functional correspondence between reduction semantics and reduction-free full normalisers. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2013)*, pages 107–116, Rome, Italy, January 2013. ACM Press. 27, 43, 54, 80, 104, 108

[61] Álvaro García-Pérez, Pablo Nogueira, and Juan José Moreno-Navarro. Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In Schrijvers and Peña [110], pages 85–96. 107, 108

[62] Álvaro García-Pérez, Pablo Nogueira, and Ilya Sergey. Deriving interpretations of the gradually-typed lambda calculus. In Wei-Ngan Chin and Jurriaan Hage, editors, *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2014)*, pages 157–168, San Diego, California, January 2014. ACM Press. 98

[63] Oliver Geupel. Overlap closures and termination of term rewriting systems. Technical Report MIP-8922, Fakultät für Mathematik und Informatik, Universität Passau, Passau, Germany, July 1989. 43, 47, 54

[64] 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. 128

[65] 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, Lübeck, Germany, September 2004. Springer. 137

[66] John V. Guttag, Deepak Kapur, and David R. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM Journal on Computing*, 12(1):189–214, 1983. 7, 8, 43, 47, 54, 79, 81

[67] John Hannan. Operational semantics directed machine architecture. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994. 95

[68] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991). 133

[69] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [17], pages 458–471. 126

[70] Miki Hermann and Igor Privera. On nontermination of Knuth-Bendix algorithm. In Laurent Kott, editor, *Automata, Languages, and Programming, 13th International Colloquium, ICALP86*, number 226 in Lecture Notes in Computer Science, pages 146–156, Rennes, France, July 1986. Springer-Verlag. 54

[71] Hans Hüttel. *Transitions and Trees – An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. 59, 114

[72] Wojciech Jedynak, Małgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of Coq. In Schrijvers and Peña [110], pages 25–36. 5, 98

[73] Eugen Jiresch. *A Term Rewriting Laboratory with Systematic and Random Generation and Heuristic Test Facilities*. PhD thesis, Vienna University of Technology, Vienna, Austria, June 2008. 54

[74] 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. 5

[75] 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. 129

[76] 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. 95, 129

[77] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4): 308–320, 1964. 95, 97, 124

[78] Dallas S. Lankford. Canonical inference. Technical Report ATP-32, Department of Mathematics, Southwestern University, Georgetown, Texas, December 1975. 54

[79] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006. A preliminary version was presented at the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP 2004). 130

[80] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960. 127

[81] John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62*, pages 21–28. North-Holland, August 1962. 95

[82] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81:721–781, 2012. 54

[83] Jan Midtgaard. *Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs*. PhD thesis, BRICS PhD School, Aarhus University, Aarhus, Denmark, June 2007. 5

[84] Kevin Millikin. *A Structured Approach to the Transformation, Normalization and Execution of Computer Programs*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2007. 5, 27, 54, 80, 104, 107

[85] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 133

[86] Eugenio Moggi. An abstract view of programming languages. Course notes ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1990. 95

[87] Peter D. Mosses. *Mathematical Semantics and Compiler Generation*. PhD thesis, Computing Laboratory, Oxford University, Oxford, UK, 1975. 115

[88] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000. 145

[89] 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. 5, 27, 54, 80, 104, 107

[90] 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. 5

[91] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992. 59, 78, 114

[92] Atsushi Ohori. The logical abstract machine: A Curry-Howard isomorphism for machine code. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming, 4th International Symposium, FLOPS 1999*, number 1722 in Lecture Notes in Computer Science, pages 300–318, Tsukuba, Japan, November 1999. Springer. 95

[93] 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. 28, 68, 70, 98, 129

[94] 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. 136, 138

[95] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. 42

[96] 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. 4, 5, 97

[97] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. 11, 34, 43, 98, 126

[98] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977. 129

[99] 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 [100]. 22, 41, 133

[100] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. 143

[101] Matthias Puech. Proofs, upside down – a functional correspondence between natural deduction and the sequent calculus. In Chung-chieh Shan, editor, *Programming Languages and Systems – 11th Asian Symposium, APLAS 2013*, number 8301 in Lecture Notes in Computer Science, pages 365–380, Melbourne, VIC, Australia, December 2013. Springer. 5, 98

[102] 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 [104]. 3, 33, 34, 35, 58, 63, 126, 128

[103] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6 (3/4):233–247, 1993. 33, 125

[104] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998. 143

[105] 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. 144

[106] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986. 95

[107] David A. Schmidt. State-transition machines for lambda-calculus expressions. *Higher-Order and Symbolic Computation*, 20(3):319–332, 2007. Journal version of [105], with a foreword [108]. 4

[108] David A. Schmidt. State-transition machines, revisited. *Higher-Order and Symbolic Computation*, 20(3):319–332, 2007. 144

[109] Heinrich Scholz and Gisbert Hasenjaeger. *Grundzüge der Mathematischen Logik*. Springer-Verlag, 1961. 124

[110] Tom Schrijvers and Ricardo Peña, editors. *Proceedings of the 15th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'13)*, Madrid, Spain, September 2013. ACM Press. 139, 140, 141, 144

[111] 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. 127

[112] 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. 5

[113] 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. 43, 98

[114] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7 (3):231–264, May 1997. 95

[115] Ravi Sethi and Adrian Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980. 95

[116] 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. 68

[117] Robert J. Simmons and Ian Zerny. A logical correspondence between natural semantics and abstract machines. In Schrijvers and Peña [110], pages 109–119. 98

[118] James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, October 1974. 54

[119] 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. 63, 91, 126

[120] Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. 129

[121] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977. 95, 115

[122] Christopher Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 198–220. North-Holland, 1966. 127

[123] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000, with a foreword [88]. 95

[124] 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 [130]. 125

[125] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998. 95

[126] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003. 7, 54

[127] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In Anatol O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, Seminars in Mathematics (translated from Russian), pages 115–125. Consultants Bureau, New York, NY, 1968. 90

[128] David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54(9):101–109, 2011. 5, 98

[129] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989. 126

[130] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000. 145

[131] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980. 90

[132] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241, Albuquerque, New Mexico, January 1982. ACM Press. 95

[133] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. 42

[134] 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. 12, 133

[135] Ian Zerny. *The Interpretation and Inter-derivation of Small-step and Big-step Specifications*. PhD thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, June 2013. 5, 139