AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

# From Outermost Reduction Semantics to Abstract Machine

Department of Computer Science

Olivier Danvy and
Jacob Johannsen

Report

ISBN 978-87-7507-311-5

Cover painting sketch by Irene Danvy, Aarhus, Summer 2014

Aarhus University Library

# From Outermost Reduction Semantics to Abstract Machine [*]

Olivier Danvy and Jacob Johannsen
Aarhus University [†]

January 2015

**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 article, 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.

[*] Extended version of [6].

[†] Department of Computer Science, Aabogade 34, DK-8200 Aarhus N, Denmark
Email: danvy@cs.au.dk and cnn@cs.au.dk

# Contents

# List of Figures

# 1 Introduction

A Structural Operational Semantics [28] 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 [14] 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 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 [33], 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 [3, 14]. Today reduction semantics are in common use in the area of programming languages [15, 26].

## 1.1 Reduction-based vs. reduction-free evaluation

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

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

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, the first author and his students have been putting forward a methodology for systematically constructing such abstract machines [10, 5, 4]: 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:

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

This shortcut works for deterministic reduction strategies where after recomposition, decomposition always comes back to the contractum and its reduction context before continuing [10]. In particular, it always works for innermost reduction, and has given rise to a 'syntactic correspondence' between reduction semantics and abstract machines [2, 3].

This syntactic correspondence has proved successful to reconstruct many preexisting abstract machines as well as to construct new ones [1, 7, 17, 30], even in the presence of control operators [3, 8]. For a class of examples, it applies to all the reduction semantics of Felleisen et al.'s latest textbook [15]. More generally, it concretizes Plotkin's connection between calculi and programming languages [27] 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 [12, 19, 18], 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.

### 1.2 Overview

We first illustrate reduction semantics for arithmetic expressions with an innermost reduction strategy (Section 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 3). With respect to refocusing, the only problematic combination of overlaps and strategies is backward-overlapping rules and outermost strategy (Section 4). To solve the problem, we suggest to backtrack after contracting a redex, which enables refocusing (Section 5). For symmetry, we also consider foretracking (Section 6). We then review related work (Section 7).

## 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:

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

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

$$pr ::= A(0, t_2) \mid A(S(t_1^{\mathrm{nf}}), t_2)$$

The reduction rules read as follows:

$$\begin{array}{rcl} A(0, t_2) & \mapsto & t_2 \\ A(S(t_1^{\mathrm{nf}}), t_2) & \mapsto & S(A(t_1^{\mathrm{nf}}, t_2)) \end{array}$$

Note the occurrence of $t_1^{\mathrm{nf}}$, 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:

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

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

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

We obtained this grammar by CPS-transforming a search function implementing the innermost reduction strategy and then defunctionalizing its continuation [11].

**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:

$$\begin{array}{rcl} \langle \Box[t] \rangle^{\mathrm{rec}} & \uparrow & t \\ \langle C[S[t]] \rangle^{\mathrm{rec}} & \uparrow & \langle C[S(t)] \rangle^{\mathrm{rec}} \\ \langle C[A([t_1], t_2)] \rangle^{\mathrm{rec}} & \uparrow & \langle C[A(t_1, t_2)] \rangle^{\mathrm{rec}} \end{array}$$

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:

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

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:

$$\begin{array}{rcl} \langle C[0] \rangle^{\mathrm{dec}}_{\mathrm{term}} & \downarrow & \langle C[0] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \\ \langle C[S(t)] \rangle^{\mathrm{dec}}_{\mathrm{term}} & \downarrow & \langle C[S[t]] \rangle^{\mathrm{dec}}_{\mathrm{term}} \\ \langle C[A(t_1, t_2)] \rangle^{\mathrm{dec}}_{\mathrm{term}} & \downarrow & \langle C[A([t_1], t_2)] \rangle^{\mathrm{dec}}_{\mathrm{term}} \end{array}$$

$$\begin{array}{rcl} \langle \Box[t^{\mathrm{nf}}] \rangle^{\mathrm{dec}}_{\mathrm{cont}} & \downarrow & t^{\mathrm{nf}} \\ \langle C[S[t^{\mathrm{nf}}]] \rangle^{\mathrm{dec}}_{\mathrm{cont}} & \downarrow & \langle C[S(t^{\mathrm{nf}})] \rangle^{\mathrm{dec}}_{\mathrm{cont}} \\ \langle C[A([0], t_2)] \rangle^{\mathrm{dec}}_{\mathrm{cont}} & \downarrow & C[A(0, t_2)] \\ \langle C[A([S(t^{\mathrm{nf}})], t_2)] \rangle^{\mathrm{dec}}_{\mathrm{cont}} & \downarrow & C[A(S(t^{\mathrm{nf}}), t_2)] \end{array}$$

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:

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

**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:

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

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

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

**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 short-cut 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

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

$$\begin{tikzcd}
A(A(S(0),0),0) \arrow[r, "\text{decomposition}"'] \arrow[dd, dashed, "\text{reduction}"'] & \square[A(\llbracket A(S(0),0)\rrbracket,0)] \arrow[d, "\text{contraction}"] \\
 & \square[A(\llbracket S(A(0,0))\rrbracket,0)] \arrow[dl, "\text{recomposition}"'] \arrow[d, dotted, "\text{refocusing}"] \\
A(S(A(0,0)),0) \arrow[r, "\text{decomposition}"'] \arrow[dd, dashed, "\text{reduction}"'] & \square[A([S\llbracket A(0,0)\rrbracket],0)] \arrow[d, "\text{contraction}"] \\
 & \square[A([S\llbracket 0\rrbracket],0)] \arrow[dl, "\text{recomposition}"'] \arrow[d, dotted, "\text{refocusing}"] \\
A(S(0),0) \arrow[r, "\text{decomposition}"'] \arrow[dd, dashed, "\text{reduction}"'] & \square\llbracket A(S(0),0)\rrbracket \arrow[d, "\text{contraction}"] \\
 & \square\llbracket S(A(0,0))\rrbracket \arrow[dl, "\text{recomposition}"'] \arrow[d, dotted, "\text{refocusing}"] \\
S(A(0,0)) \arrow[r, "\text{decomposition}"'] \arrow[dd, dashed, "\text{reduction}"'] & \square[S\llbracket A(0,0)\rrbracket] \arrow[d, "\text{contraction}"] \\
 & \square[S\llbracket 0\rrbracket] \arrow[dl, "\text{recomposition}"'] \arrow[d, dotted, "\text{refocusing}"] \\
S(0) \arrow[r, "\text{decomposition}"'] & \square\llbracket S(0)\rrbracket
\end{tikzcd}$$

Figure 1: Innermost reduction sequence for $A(A(S(0),0),0)$

where $([\mapsto];\downarrow^*)$ denotes contraction in context followed by decomposition (and was noted $\rightsquigarrow$ in Section 1.1).

**An example:** See Figure 1.

**Reduction-free evaluation:** After applying refocusing, we follow the steps of the syntactic correspondence [2, 3, 5], fusing the iteration and refocus functions, inlining the contract function, and compressing corridor transitions. The resulting normalizer implements a transition system described by the following abstract machine:

$$\begin{aligned}
t &\rightarrowtail \langle\square\llbracket t\rrbracket\rangle_{\text{term}} \\[1ex]
\langle C\llbracket 0\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C\llbracket 0\rrbracket\rangle_{\text{cont}} \\
\langle C\llbracket S(t)\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C[S\llbracket t\rrbracket]\rangle_{\text{term}} \\
\langle C\llbracket A(t_1,t_2)\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C[A(\llbracket t_1\rrbracket,t_2)]\rangle_{\text{term}} \\[1ex]
\langle\square\llbracket t^{\text{nf}}\rrbracket\rangle_{\text{cont}} &\rightarrowtail t^{\text{nf}} \\
\langle C[S\llbracket t^{\text{nf}}\rrbracket]\rangle_{\text{cont}} &\rightarrowtail \langle C\llbracket S(t^{\text{nf}})\rrbracket\rangle_{\text{cont}} \\
\langle C[A(\llbracket 0\rrbracket,t_2)]\rangle_{\text{cont}} &\rightarrowtail \langle C\llbracket t_2\rrbracket\rangle_{\text{term}} \\
\langle C[A(\llbracket S(t^{\text{nf}})\rrbracket,t_2)]\rangle_{\text{cont}} &\rightarrowtail \langle C[S[A(\llbracket t^{\text{nf}}\rrbracket,t_2)]]\rangle_{\text{cont}}
\end{aligned}$$

# 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 2** (Backward-overlapping rules)**.** *Let $l_1 \to r_1$ and $l_2 \to r_2$ 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 *[12, 19, 18].*

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 3.1 | Section 3.2 |
| backward-overlapping rules | Section 3.3 | Section 3.4 |

## 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.

### 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.

### 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.

### 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.

## 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:

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

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

$$pr ::= A(0, t_2) \mid A(S(t_1), t_2)$$

The reduction rules now read as follows:

$$\begin{aligned} A(0, t_2) &\mapsto t_2 \\ A(S(t_1), t_2) &\mapsto S(A(t_1, t_2)) \end{aligned}$$

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:

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

**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:

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

As in Section 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 2, a term not in normal form can be decomposed into more than one reduction context and one potential redex. For example, the term $A(S(A(S(t_0), t_1)), t_2)$ can be decomposed into $\Box[A(S(A(S(t_0), t_1)), t_2)]$ and $\Box[A([S[A(S(t_0), t_1)]], t_2)]$. This non-unique decomposition puts us outside the validity conditions of refocusing [10], so we are on our own here.

**Recomposition:** It is defined as in Section 2.

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

[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.

$$\begin{aligned}
\langle C[0]\rangle^{\text{dec}}_{\text{term}} &\downarrow \langle C[0]\rangle^{\text{dec}}_{\text{cont}}\\
\langle C[S(t)]\rangle^{\text{dec}}_{\text{term}} &\downarrow \langle C[S[t]]\rangle^{\text{dec}}_{\text{term}}\\
\langle C[A(t_1, t_2)]\rangle^{\text{dec}}_{\text{term}} &\downarrow \langle C[A(t_1, t_2)]\rangle^{\text{dec}}_{\text{add}}
\end{aligned}$$

$$\begin{aligned}
\langle C[A(0, t_2)]\rangle^{\text{dec}}_{\text{add}} &\downarrow C[A(0, t_2)]\\
\langle C[A(S(t_1), t_2)]\rangle^{\text{dec}}_{\text{add}} &\downarrow C[A(S(t_1), t_2)]\\
\langle C[A(A(t_{11}, t_{12}), t_2)]\rangle^{\text{dec}}_{\text{add}} &\downarrow \langle C[A([A(t_{11}, t_{12})], t_2)]\rangle^{\text{dec}}_{\text{add}}
\end{aligned}$$

$$\begin{aligned}
\langle \square[t^{\text{nf}}]\rangle^{\text{dec}}_{\text{cont}} &\downarrow t^{\text{nf}}\\
\langle C[S[t^{\text{nf}}]]\rangle^{\text{dec}}_{\text{cont}} &\downarrow \langle C[S(t^{\text{nf}})]\rangle^{\text{dec}}_{\text{cont}}
\end{aligned}$$

As in Section 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:

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

**A notable property:** Due to the invariant of the abstract machine implementing decomposition, as in Section 2, the recomposition function is still a left inverse of the decomposition function.

**One-step reduction:** It is defined as in Section 2.

**Reduction-based evaluation:** It is defined as in Section 2.

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

$$\begin{aligned} A(0, t_2) &\mapsto t_2 \\ A(S(t_1), t_2) &\mapsto S(A(t_1, t_2)) \end{aligned}$$

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 shortcut the recomposition and decomposition to this point, and just continue the decomposition *in situ*.

More formally, we have

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

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 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 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:

$$\begin{tikzcd}
A(A(S(0),0),0) \arrow[r, "\text{decomposition}"'] \arrow[ddd, dashed, "\text{reduction}"'] & \Box[A(\llbracket A(S(0),0)\rrbracket,0)] \arrow[d, "\text{contraction}"] \\
 & \Box[A(\llbracket S(A(0,0))\rrbracket,0)] \arrow[d, dotted, "\text{backtracking}"] \arrow[ddl, "\text{recomposition}"'] \\
 & \Box\llbracket A(S(A(0,0)),0)\rrbracket \arrow[d, dotted, "\text{refocus}"] \\
A(S(A(0,0)),0) \arrow[r, "\text{decomposition}"'] \arrow[ddd, dashed, "\text{reduction}"'] & \Box\llbracket A(S(A(0,0)),0)\rrbracket \arrow[d, "\text{contraction}"] \\
 & \Box\llbracket S(A(A(0,0),0))\rrbracket \arrow[d, dotted, "\text{backtracking}"] \arrow[ddl, "\text{recomposition}"'] \\
 & \Box\llbracket S(A(A(0,0),0))\rrbracket \arrow[d, dotted, "\text{refocusing}"] \\
S(A(A(0,0),0)) \arrow[r, "\text{decomposition}"'] \arrow[ddd, dashed, "\text{reduction}"'] & \Box[S[A(\llbracket A(0,0)\rrbracket,0)]] \arrow[d, "\text{contraction}"] \\
 & \Box[S[A(\llbracket 0\rrbracket,0)]] \arrow[d, dotted, "\text{backtracking}"] \arrow[ddl, "\text{recomposition}"'] \\
 & \Box[S\llbracket A(0,0)\rrbracket] \arrow[d, dotted, "\text{refocusing}"] \\
S(A(0,0)) \arrow[r, "\text{decomposition}"'] \arrow[ddd, dashed, "\text{reduction}"'] & \Box[S\llbracket A(0,0)\rrbracket] \arrow[d, "\text{contraction}"] \\
 & \Box[S\llbracket 0\rrbracket] \arrow[d, dotted, "\text{backtracking}"] \arrow[ddl, "\text{recomposition}"'] \\
 & \Box\llbracket S(0)\rrbracket \arrow[d, dotted, "\text{refocusing}"] \\
S(0) \arrow[r, "\text{decomposition}"'] & S(0)
\end{tikzcd}$$

Figure 2: Outermost reduction sequence for $A(A(S(0),0),0)$

$$\begin{aligned}
t &\rightarrowtail \langle \Box\llbracket t\rrbracket\rangle_{\text{term}} \\[1ex]
\langle C\llbracket 0\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C\llbracket 0\rrbracket\rangle_{\text{cont}} \\
\langle C\llbracket S(t)\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C[S\llbracket t\rrbracket]\rangle_{\text{term}} \\
\langle C\llbracket A(t_1,t_2)\rrbracket\rangle_{\text{term}} &\rightarrowtail \langle C\llbracket A(t_1,t_2)\rrbracket\rangle_{\text{add}} \\[1ex]
\langle \Box\llbracket A(0,t_2)\rrbracket\rangle_{\text{add}} &\rightarrowtail \langle \Box\llbracket t_2\rrbracket\rangle_{\text{term}} \\
\langle C[S\llbracket A(0,t_2)\rrbracket]\rangle_{\text{add}} &\rightarrowtail \langle C[S\llbracket t_2\rrbracket]\rangle_{\text{term}} \\
\langle C[A(\llbracket A(0,t_2)\rrbracket,t_2')]\rangle_{\text{add}} &\rightarrowtail \langle C\llbracket A(t_2,t_2')\rrbracket\rangle_{\text{add}} \\
\langle \Box\llbracket A(S(t_1),t_2)\rrbracket\rangle_{\text{add}} &\rightarrowtail \langle \Box[S\llbracket A(t_1,t_2)\rrbracket]\rangle_{\text{add}} \\
\langle C[S\llbracket A(S(t_1),t_2)\rrbracket]\rangle_{\text{add}} &\rightarrowtail \langle C[S[S\llbracket A(t_1,t_2)\rrbracket]]\rangle_{\text{add}} \\
\langle C[A(\llbracket A(S(t_1),t_2)\rrbracket,t_2')]\rangle_{\text{add}} &\rightarrowtail \langle C\llbracket A(S(A(t_1,t_2)),t_2')\rrbracket\rangle_{\text{add}} \\
\langle C\llbracket A(A(t_{11},t_{12}),t_2)\rrbracket\rangle_{\text{add}} &\rightarrowtail \langle C[A(\llbracket A(t_{11},t_{12})\rrbracket,t_2)]\rangle_{\text{add}} \\[1ex]
\langle \Box\llbracket t^{\text{nf}}\rrbracket\rangle_{\text{cont}} &\rightarrowtail t^{\text{nf}} \\
\langle C[S\llbracket t^{\text{nf}}\rrbracket]\rangle_{\text{cont}} &\rightarrowtail \langle C\llbracket S(t^{\text{nf}})\rrbracket\rangle_{\text{cont}}
\end{aligned}$$

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.

# 5 Backtracking

## 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 2. 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.

## 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.

### 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 $\lambda$-calculus [2, 10]. In this case, the contractum that gives rise to a backward overlap is in normal form: it is a $\lambda$-abstraction that occurs on the left of an application node; this application node forms a new $\beta$-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 [5]. In this case, the backward overlap only occurs when contracting a redex which is not outermost, so backtracking is not needed.

## 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.

# 7 Related work

Refocusing has mainly be applied for weak reduction in the $\lambda$-calculus, for normal order, applicative order, etc. For full reduction in the $\lambda$-calculus, a backward overlap exists. As analyzed in Section 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 $\lambda$-calculus: one by Danvy, Millikin and Munk [9, 24, 25], in the mid-2000's, and a recent one by García-Pérez and Nogueira [17, 16]:

- Danvy, Millikin and Munk overcome the backward overlap (without identifying it as such) by backtracking *after* applying the refocus function. In a more general setting, backtracking after refocusing would change the reduction order, so this solution does not scale. Our solution does not change the reduction order, and therefore it applies in a more general setting.

- 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 [12, 19, 20, 13, 18], and more recently in Jiresch's thesis [21]. 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 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* [32], which is a concept used in term rewriting [22, 31]. However, narrowing is used to solve equations [23], and hence it is unrelated to our goal here.

# 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 [10] 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.

# References

[1] 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. 2

[2] 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. 2, 6, 14

[3] 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. 1, 2, 6

[4] 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. 2

[5] 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. 2, 6, 14

[6] 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. i

[7] 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. 2

[8] 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. 2

[9] Olivier Danvy, Kevin Millikin, and Johan Munk. A correspondence between full normalization by reduction and full normalization by evaluation. Manuscript, July 2007. 15

[10] 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 was presented at the 2nd International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4. 2, 9, 14, 15

[11] Olivier Danvy and Ian Zerny. A prequel to reduction semantics. Chapter 5 of [34], June 2013. 4

[12] 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. 2, 7, 15

[13] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116, 1987. 15

[14] Matthias Felleisen. *The Calculi of $\lambda$-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. 1

[15] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009. 1, 2

[16] 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. 15

[17] 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. 2, 15

[18] 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. 2, 7, 15

[19] 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. 2, 7, 15

[20] 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. 15

[21] 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. 15

[22] Dallas S. Lankford. Canonical inference. Technical Report ATP-32, Department of Mathematics, Southwestern University, Georgetown, Texas, December 1975. 15

[23] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81:721–781, 2012. 15

[24] 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. 15

[25] 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. 15

[26] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002. 1

[27] Gordon D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975. 2

[28] 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 [29]. 1

[29] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. 19

[30] 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. 2

[31] James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, October 1974. 15

[32] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003. 15

[33] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. 1

[34] 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. 17