BRICS

Basic Research in Computer Science

Refocusing in Reduction Semantics

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series RS-04-26

ISSN 0909-0878 November 2004

Refocusing in Reduction Semantics [*]

Olivier Danvy and Lasse R. Nielsen [†]
BRICS [‡]
Department of Computer Science
University of Aarhus [§]

November 2004

Abstract

The evaluation function of a reduction semantics (i.e., a small-step operational semantics with an explicit representation of the reduction context) is canonically defined as the transitive closure of (1) decomposing a term into a reduction context and a redex, (2) contracting this redex, and (3) plugging the contractum in the context. Directly implementing this evaluation function therefore yields an interpreter with a worst-case overhead, for each step, that is linear in the size of the input term.

We present sufficient conditions over the constituents of a reduction semantics to circumvent this overhead, by replacing the composition of (3) plugging and (1) decomposing by a single "refocus" function mapping a contractum and a context into a new context and a new redex, if any. We also show how to construct such a refocus function, we prove the correctness of this construction, and we analyze the complexity of the resulting refocus function.

The refocused evaluation function of a reduction semantics implements the transitive closure of the refocus function, i.e., a "pre-abstract machine." Fusing the refocus function with the trampoline function computing the transitive closure gives a state-transition function, i.e., an abstract machine. This abstract machine clearly separates between the transitions implementing the congruence rules of the reduction semantics and the transitions implementing its reduction rules. The construction of a refocus function therefore shows how to mechanically obtain an abstract machine out of a reduction semantics, which was done previously on a case-by-case basis.

We illustrate refocusing by mechanically constructing Felleisen et al.'s CK machine from a call-by-value reduction semantics of the lambda-calculus, and by constructing a substitution-based version of Krivine's machine from a call-by-name reduction semantics of the lambda-calculus. We also mechanically construct three one-pass CPS transformers from three quadratic context-based CPS transformers for the lambda-calculus.

Keywords
Reduction semantics, refocusing, pre-abstract machine, abstract machine, defunctionalization, transformation into continuation-passing style (CPS).

[*]A preliminary version of this article was presented at RULE 2001 [8].
[†]Current affiliation: ACURE A/S, Bytoften 1, DK-8240 Risskov, Denmark
[‡]Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.
[§]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: {`danvy,lrn`}`@brics.dk`

Contents

List of Figures

1 Introduction

A reduction semantics (a.k.a. syntactic theory) is a small-step operational semantics with an explicit representation of the reduction context [10–12]. We consider the problem of implementing the evaluation function of a reduction semantics in the form of an interpreter. Our emphasis, however, is not on automating this process, as in Xiao and Ariola's SL project [28, 29]. Instead, we observe that the direct implementation of an evaluation function entails an overhead and we show how to circumvent this overhead.

1.1 Related work

Probably because every interesting programming language corresponds to an interesting model of computation, the meta-language of formal semantics has consistently inspired a range of programming languages—witness denotational semantics and functional programming [25, 27], Moggi's computational monads and monad-based functional programming [26], operational semantics and logic programming with definite clause grammars [17], algebraic semantics and OBJ [15], etc. These similarities between language and meta-language make it possible to write 'executable specifications.' For reduction semantics, however, executable specifications appear not to scale up in practice [24, 28].

We identify a source of inefficiency in the direct implementation of a reduction semantics as an interpreter, and we show how to bypass it. The result can be used to mechanically construct an abstract machine out of a reduction semantics, an issue that is usually dealt with on a case-by-case basis [11, Chapter 7].

1.2 Interpreters for reduction semantics

A reduction semantics is a small-step operational semantics where evaluation is defined as the transitive closure of single reductions, each performed by (1) decomposing a term into a reduction context and a potential redex, (2) contracting the redex, if possible,[1] and (3) plugging the contractum into the context. Most reduction semantics are developed for deterministic programming languages and satisfy a unique-decomposition property.

The interpreter for a reduction semantics implements its evaluation function. It naturally consists of a decompose-contract-plug loop. Decomposition is usually implemented with a depth-first search in the abstract-syntax tree. The decompose step therefore introduces an overhead that is proportional to the size of the input term. Likewise, plugging takes time linear in the size of the context.

1.3 A canonical example: the λ-calculus

Here is a call-by-value reduction semantics of the λ-calculus:

Terms	$t ::= x \mid \lambda x.t \mid t\ t$	$t \in \Lambda$
Values	$v ::= x \mid \lambda x.t$	$v \in \mathit{Value} \subseteq \Lambda$
Variables		$x \in \mathit{Var}$
Reduction contexts	$C ::= [\,] \mid C[[\,]\ t] \mid C[v\ [\,]]$	$C \in \mathit{RedCont}$
Potential redexes	$r ::= v\ v$	$r \in \mathit{PotRedex} \subseteq \Lambda$

[1]Not all potential redexes are actual reducible expressions—some are stuck terms [20].

Among potential redexes, only β-redexes are actual reducible terms. Therefore, the only reduction rule is the following one:

$$C[(\lambda x.t)\ v] \rightarrow C[t\,\{v/x\}]$$

Plugging the hole of a reduction context with a term is defined by induction on the context, and thus it operates in time proportional to the size of the context:

$$\begin{array}{rcl} ([\,])[t] & = & t \\ (C[[\,]\ t'])[t] & = & C[t\ t'] \\ (C[v\ [\,]])[t] & = & C[v\ t] \end{array}$$

Likewise, decomposition operates in time proportional, in the worst case, to the size of the term to decompose.

Let us illustrate this overhead with Church numerals: the Church numeral for the number n is $\lambda s.\lambda z.\,\underbrace{s(s(s(\ldots(s}_{n}\,z)\ldots)))$ and is noted $\lceil n \rceil$.

Example 1 (Church numeral) *We consider the term $\lceil n \rceil\ (\lambda x.x)\ v$ where v is any value. This term reduces in two steps to $\underbrace{(\lambda x.x)((\lambda x.x)((\lambda x.x)(\ldots((\lambda x.x)}_{n}\ v)\ldots)))$. From then on, each decomposition into a context and a redex (where the redex is always $(\lambda x.x)\ v$) takes time proportional to the number of remaining applications. The total time taken by decomposition during evaluation is thus proportional to n^2.* $\square$

The evaluation function of a reduction semantics is inefficient because it constructs intermediate terms (see the left side of Figure 1). We observe, though, that in the course of evaluation, each decompose step always follows a plug step (even for the first step, since $([\,])[t] = t$). One could therefore deforest intermediate terms by fusing the plug step and the decompose step into one 'refocus' step (see the right side of Figure 1).

Rather than deforesting plug and decompose on a case-by-case basis, we seek general conditions over the constituents of a reduction semantics to construct such an efficient refocus step.

1.4 This work

We state conditions under which consecutive plug-and-decompose operations can be replaced by a more efficient 'refocus' operation where the intermediate term is deforested away. These conditions pertain to the order in which a redex occurs in the depth-first traversal of the decompose operation. We show that these conditions hold if the reduction semantics is given in the "standard" way, i.e., by a context-free grammar of values and reduction contexts, and if it satisfies a unique-decomposition property. We also show how to mechanically construct such a refocus function.

Overview: The rest of this article is structured as follows. We first list sufficient conditions to obtain the refocus function of a reduction semantics (Section 2). The proof that these conditions are sufficient is constructive. It indicates how to mechanically rephrase the evaluation function of a reduction semantics to circumvent the plug-and-decompose overhead. These conditions are general enough to make refocusing useful

$$\begin{tikzcd}
t_0 \arrow[ddd, dashed, "\text{reduce}"'] \arrow[dr, "\text{decompose}"] & & t_0 \arrow[d, "\text{refocus}"] \\
& C_0[r_0] \arrow[d, "\text{contract}"] & C_0[r_0] \arrow[d, "\text{contract}"] \\
& C_0[t'_0] \arrow[dl, "\text{plug}"] & C_0[t'_0] \arrow[dd, "\text{refocus}"] \\
t_1 \arrow[ddd, dashed, "\text{reduce}"'] \arrow[dr, "\text{decompose}"] & & \\
& C_1[r_1] \arrow[d, "\text{contract}"] & C_1[r_1] \arrow[d, "\text{contract}"] \\
& C_1[t'_1] \arrow[dl, "\text{plug}"] & C_1[t'_1] \arrow[dd, "\text{refocus}"] \\
t_2 \arrow[ddd, dashed, "\text{reduce}"'] \arrow[dr, "\text{decompose}"] & & \\
& C_2[r_2] \arrow[d, "\text{contract}"] & C_2[r_2] \arrow[d, "\text{contract}"] \\
& C_2[t'_2] \arrow[dl, "\text{plug}"] & C_2[t'_2] \arrow[dd, "\text{refocus}"] \\
t_3 \arrow[d, dashed] \arrow[dr, "\text{decompose}"] & & \\
\vdots & \vdots & \vdots
\end{tikzcd}$$

Figure 1: Canonical vs. refocused evaluation

in practice—for example, they are fulfilled in all the deterministic reduction semantics described in Felleisen and Flatt's lecture notes [11] and all the examples documented in the SL project [28, 29].

We then illustrate refocusing with two reduction semantics for the λ-calculus, one for call by value and one for call by name (Sections 3 and 4). In the call-by-value case, the refocused evaluation function implements Felleisen et al.'s CK machine, and in the call-by-name case, the refocused evaluation function implements a substitution-based version of Krivine's machine. We also consider three context-based, quadratic CPS transformers (two for call by value and one for call by name) and we make them operate in one pass.

2 Refocusing in a reduction semantics

In this section we show how to construct from scratch a *refocus* function for a reduction semantics, as an alternative to plugging and decomposing for defining an evaluation function. We prove that this function computes the correct result and we give an exact measure of its computational complexity. Finally we show that refocusing is always at least as efficient as plugging and then decomposing.

2.1 The language

A programming language is defined by its syntax and its semantics. Here, the semantics is specified by a reduction semantics. Without loss of generality we assume that the abstract syntax is specified by a context-free grammar (or BNF) with only productions of the form

$$t ::= \mathsf{c}(t_1, \ldots, t_n)$$

where c ranges over a set of constructor names and t, $t_1, \ldots, t_n$ are non-terminals corresponding to the syntactic categories of the language. Each constructor may occur only once in a grammar.

2.2 Reduction semantics

A reduction semantics is a specification of a small-step operational semantics, i.e., it specifies a reduction relation between terms. Reduction semantics usually come with context-free grammars for values and reduction contexts, a notion of decomposition and potential redex, often a unique-decomposition lemma, and finally reduction rules of the form $C[r] \rightarrow t$, where the term r denotes a redex. Let us review each of these concepts in turn.

2.2.1 Values

The grammar of values extends the BNF of the language with one new non-terminal, v, per non-terminal t. The set of values ranged over by a v is included in the set of terms ranged over by the matching t. The productions for values are all of the form

$$v ::= \mathsf{c}(x_1, \ldots, x_n)$$

where $t ::= \mathsf{c}(t_1, \ldots, t_n)$ is a production of the language grammar and each x_i is either t_i or v_i.

If a reduction semantics of the language contains only values, then it is irrelevant whether we use v_i or t_i in place of x_i. From here on we will consistently use a t_i for such a syntactic category.

2.2.2 Reduction contexts

Reduction contexts (often called 'evaluation contexts') are also specified by context-free grammars, and their meaning is given by an associated *plug* function. For example, the reduction contexts of the λ-calculus under call by value and the associated *plug* function were shown in Section 1.3.

All the reduction contexts of a reduction semantics can be written as the composition of *elementary* reduction contexts. Composition is a binary function on reduction contexts, $\circ$, that together with the *plug* function satisfies $(C_1 \circ C_2)[t] = C_1[C_2[t]]$, and is thus associative.[2] Elementary contexts are those that cannot themselves be written as compositions of other non-empty contexts.

For example, for the λ-calculus under call by value (see Section 3), the elementary contexts are $\mathsf{app}([\ ],\ t)$ and $\mathsf{app}(v,\ [\ ])$. The construction of reduction contexts corresponds to composing an elementary context to the right of another context, e.g., $C[\mathsf{app}([\ ],\ t)] = C \circ \mathsf{app}([\ ],\ t)$. In many articles about reduction semantics, though, compound contexts are constructed by composing an elementary context on the left rather than on the right [11, 13]. For the λ-calculus under call by value, the grammar of reduction contexts then reads as follows.

$$C \quad ::= \quad [\,] \mid \mathsf{app}(C,\ t) \mid \mathsf{app}(v,\ C)$$

This notation induces the same elementary contexts.

2.2.3 Decompositions and potential redexes

If a term t' is the result of plugging another term t into a reduction context C, i.e., if $C[t] = t'$, then we say that C and t form a *decomposition* (into a reduction context and a term) of t', or alternatively that t' can be decomposed into C and t. In general, a term can be decomposed in many different ways.

If $t' = C[t]$ then the decomposition is *trivial* if either C is the empty context or t is a value. We assume that values are normal forms: they cannot have a non-value sub-term in a position where it can be evaluated, so all their decompositions must be trivial.[3] Non-value terms that can only be decomposed trivially are called *potential redexes*. For the λ-calculus under call by value, the potential redexes are exactly the terms of the form $\mathsf{app}(v,\ v)$.

If $t' = C[t]$ then the decomposition is *complete* if t is a potential redex. The reduction rules of a reduction semantics contain only complete decompositions. If a term has a decomposition, $C_1[t_1]$, that is not complete, then either t_1 is a value or it can be further decomposed in a non-trivial way as $C_2[t_2]$. Then $(C_1 \circ C_2)[t_2]$ is again a non-trivial decomposition of the original term. The context of a complete decomposition is maximal in the sense that further decompositions would be trivial.

2.2.4 Unique decomposition

Unique decomposition is a property of reduction semantics that guarantees the existence and uniqueness of complete decompositions for arbitrary terms.

Definition 1 (Unique decomposition) *A reduction semantics is said to satisfy the* unique-decomposition property *if any non-value term t can be uniquely decomposed as $t = C[r]$ where r is a potential redex.*

Unique decomposition is so fundamental to reduction semantics for deterministic languages that it is almost always the first property to be established. Its proof is

[2]We note that reduction contexts, together with composition and the empty context, form a monoid, which explains why they can indifferently be represented "outside in" or "inside out" [7, 11, 13]. Representing contexts inside out leads one to a stack implementation: extending a context is implemented by pushing, and decomposing the innermost context is implemented by popping and dispatching.

[3]This does not preclude, e.g., weak head normal forms or lazy pairs.

often technically simple, but because of its many small cases, it tends to be tedious and error-prone. This state of affairs motivated Xiao, Sabry, and Ariola to develop automated support for proving unique-decomposition properties [29].[4]

2.2.5 Reduction rules

The reduction rules of a reduction semantics are of the form $C[r] \rightarrow t$ where r is a potential redex and t depends only on C and on the sub-terms of r. Often t is written as $C[t']$ for some t' depending on r.

The potential redexes that occur in a reduction rule are the actual reducible terms of the reduction semantics. The remaining potential redexes are stuck, to use Plotkin's terminology [20]. Stuck terms encompass type-incorrect terms (e.g., the application of a literal) and undefined terms (e.g., the application of an identifier).

The only reduction rule of the λ-calculus under call by value is

$$C[\mathsf{app}(\mathsf{lam}(x,\ t),\ v)] \rightarrow C[t\,\{v/x\}]$$

and thus $\mathsf{app}(\mathsf{lam}(x,\ t),\ v)$ is a reducible term. The remaining potential redexes (of the form $\mathsf{app}(x,\ v)$) are stuck terms.

2.3 Requirements

We state three requirements that are sufficient for constructing a *refocus* function. We consider reduction semantics whose structure is as described in Section 2.2 and that satisfy unique decomposition.

2.3.1 No redundant constructs

A reduction semantics specifies a reduction relation. Any part of a specification that does not affect the specified relation is redundant and can be ignored. Specifically:

- If the language has a grammar production $t ::= \mathsf{c}(t_1,\ \ldots,\ t_i,\ \ldots,\ t_n)$ and the syntactic category ranged over by t_i contains only values, then an elementary reduction context of the form $\mathsf{c}(x_1,\ \ldots,\ x_{i-1},\ [\ ],\ x_{i+1},\ \ldots,\ x_n)$ is redundant. Since only values can be plugged in the hole, and values can only be trivially decomposed, then no reduction context constructed by composing this elementary context can ever be part of a complete decomposition.

- If a syntactic category ranged over by t contains no values, i.e., the corresponding values ranged over by v is the empty set, then any occurrence of v in a rule makes that rule redundant. (Syntactic categories with no values are not useless; they can occur meaningfully in, e.g., languages with control effects.)

- If the syntactic category ranged over by t is empty, then that entire syntactic category and all other productions containing t are redundant and can safely be ignored.

[4]Xiao, Sabry, and Ariola also point out that proving unique decomposition reduces to proving equivalence and unambiguity of context-free grammars—two undecidable properties. In their system SL, they reduce the problem to regular tree languages, for which equivalence and unambiguity are decidable. Similarly, the grammars we consider here are regular tree grammars.

The above properties are all decidable. Deciding whether the set of terms ranged over by a non-terminal is empty is decidable for context-free grammars. The specific structure of the productions of values guarantees that values must be a subset of terms, and that it is decidable whether t and v generate the same set of terms, i.e., whether t only ranges over values.

2.3.2 Distinct value constructors and potential-redex constructors

By definition, potential redexes are distinct from values, and values are defined by a context-free grammar of the form shown in Section 2.2.1. If an instance of a syntactic construct can become a value by evaluating its sub-terms, then another instance must not also be able to become a redex, and vice-versa.

The only way a syntactic construct can become both a value and a potential redex is if the value is given by a production of the form $v ::= \mathsf{c}(\ldots,\ v_i,\ \ldots)$ and there is no elementary context of the form $\mathsf{c}(\ldots,\ [\ ],\ \ldots)$ with a hole in the ith position. Then $\mathsf{c}(\ldots,\ t_i,\ \ldots)$ is a potential redex when t_i is *not* a value.

There are several ways to verify the property that no syntactic construct can become both a value and a potential redex. We use the equivalent requirement that the potential redexes can be specified by a context-free grammar with productions of the same type as for values. In other words, the productions are of the form $r ::= \mathsf{c}(x_1,\ \ldots,\ x_n)$ where x_i is either v_i or t_i.

2.3.3 Left-to-right evaluation of sub-terms

Without loss of generality, we assume that sub-terms of a syntactic construct are ordered so that evaluating them proceeds from left to right. In other words, the unique decomposition into reduction context and potential redex will always have the hole of the context occurring in the left-most non-value sub-term, in the ordering of sub-terms chosen for the abstract syntax.

Finally, we assume that in an elementary context, if the syntactic category v_i occurs to the left of the hole, then the corresponding category t_i actually contains non-value terms.

2.4 Consequences of the requirements

The requirements of Section 2.3 guarantee that a reduction semantics contains elementary reduction contexts, values, and potential redexes of a specific form. This form is described in Figure 2.

An example where the m in Figure 2 is strictly smaller than n is a conditional expression such as $\mathsf{if}(t,\ t,\ t)$. The only elementary reduction context is $\mathsf{if}([\ ],\ t,\ t)$, since the other two sub-terms should not be evaluated until the conditional expression has itself been reduced. Another example is the λ-calculus under call by name where only the function part of an application is evaluated.

Lemma 1 *A reduction semantics satisfying the requirements stated in Section 2.3 contains productions corresponding to Figure 2.*

Proof (sketch): Take the production $t ::= \mathsf{c}(t_1,\ \ldots,\ t_n)$.

If there is a value production $v ::= \mathsf{c}(t_1,\ \ldots,\ t_n)$ or a potential-redex production $r ::= \mathsf{c}(t_1,\ \ldots,\ t_n)$, then $m = 0$ and c becomes a value or a potential redex.

For a syntactic construct with n sub-terms, $t ::= \mathsf{c}(t_1, \ldots, t_n)$, there is a number $0 \leq m \leq n$ such that the elementary reduction contexts for c are exactly:

$$\begin{array}{l} \mathsf{c}([\,], t_2, \ldots, t_n) \\ \mathsf{c}(v_1, [\,], t_3, \ldots, t_n) \\ \vdots \\ \mathsf{c}(v_1, \ldots, v_{m-1}, [\,], t_{m+1}, \ldots, t_n) \end{array}$$

If terms of the form $\mathsf{c}(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$ exist (i.e., the syntactic categories ranged over by $t_1, \ldots\ t_m$ all contain values) then they are either all values or all potential redexes. In other words, either

$$v ::= \mathsf{c}(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$$

or

$$r ::= \mathsf{c}(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$$

is a production, but not both.

We then say that c *needs to evaluate m sub-terms.* If $\mathsf{c}(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)$ is a value we say that c *becomes a value.* If it is a potential redex we say that c *becomes a potential redex.*

Figure 2: Elementary contexts, values, and potential redexes

If there is no such production then, since t is not redundant, there must exist a term $\mathsf{c}(t_1, \ldots, t_n)$ that has a non-trivial decomposition and thus there must exist an elementary reduction context $\mathsf{c}(t_1, \ldots, t_{i-1}, [\,], t_{i+1}, \ldots, t_n)$. We assume that evaluation proceeds from left to right, so the hole must be in the left-most term, i.e., the elementary context is $\mathsf{c}([\,], t_2, \ldots, t_n)$.

Assume that there exists an elementary reduction context of the form

$$\mathsf{c}(v_1, \ldots, v_{i-1}, [\,], t_{i+1}, \ldots, t_n).$$

Three cases occur:

1. If t_i only ranges over non-values then $m = i$ and c becomes neither a value nor a potential redex.
2. If t_i ranges over values and there exists a production

   $$v ::= \mathsf{c}(v_1, \ldots, v_i, t_{i+1}, \ldots, t_n)$$

   or

   $$r ::= \mathsf{c}(v_1, \ldots, v_i, t_{i+1}, \ldots, t_n)$$

   then $m = i$ and c becomes either a value or a potential redex (since it cannot be both).
3. If t_i ranges over values, but terms of the form $\mathsf{c}(v_1, \ldots, v_i, t_{i+1}, \ldots, t_n)$ are not values or potential redexes, then they must have non-trivial decompositions, so

there is a corresponding elementary reduction context. Under left-to-right evaluation, the context must be $\mathsf{c}(v_1,\ \ldots,\ v_i,\ [\,],\ t_{i+2},\ \ldots,\ t_n)$.

In other words, either (1) $\mathsf{c}(v_1,\ \ldots,\ v_i,\ t_{i+1},\ \ldots,\ t_n)$ does not exist, or it is a value or a potential redex, and then $m = i$, or (2) there exists an elementary reduction context $\mathsf{c}(v_1,\ \ldots,\ v_i,\ [\,],\ t_{i+2},\ \ldots,\ t_n)$, and then $m > i$.

An induction argument shows that m is the smallest i such that (1) holds. There exists such a smallest i: a term of the form $\mathsf{c}(v_1,\ \ldots,\ v_n)$ either does not exist or it has only trivial decompositions, i.e., is either a value or a potential redex, and thus $m \leq n$ as required. □

We have shown that a reduction semantics satisfying the requirements contains the productions of Figure 2. We do not show that there are no further productions. However, we show in the next section that one can compute the unique decomposition of any term using only the productions of Figure 2, and thus any further productions would either be redundant or violate the unique-decomposition property.

2.5 Constructing a refocus function

We construct a function, *refocus*, that is extensionally equivalent to the composition of the *plug* function and the *decompose* function. We choose to make it use a stack of elementary contexts to represent reduction contexts. Such a stack allows an efficient implementation of composing an elementary reduction context and plugging a term in a context. We use $[\,]$ for the empty context/stack and $C \circ \mathsf{c}(v_1,\ \ldots,\ v_{i-1},\ [\,],\ t_{i+1},\ \ldots,\ t_n)$ for composing/pushing an elementary context.

The refocusing function is defined with two mutually recursive functions. The first, *refocus*, takes a term and a stack of elementary reduction contexts, and is defined by cases on the term; it has the same type as *decompose*. The other function, $\mathit{refocus}_{aux}$, takes a stack of elementary contexts and a value as arguments, and is defined by cases on the top-most elementary reduction context on the stack.

For each $t ::= \mathsf{c}(t_1,\ \ldots,\ t_n)$ in the language grammar, there exists one corresponding rule in *refocus*. If c needs to evaluate m sub-terms then there are m rules in $\mathit{refocus}_{aux}$ corresponding to the elementary contexts $\mathsf{c}([\,],\ t_2,\ \ldots,\ t_n)$ through $\mathsf{c}(v_1,\ \ldots,\ v_{m-1},\ [\,],\ t_{m+1},\ \ldots,\ t_n)$. If *refocus* and $\mathit{refocus}_{aux}$ are implemented in a typed setting (e.g., ML), there is one *refocus* and one $\mathit{refocus}_{aux}$ for each syntactic category of the language. The rules of *refocus* mentioned above go in the refocus functions corresponding to the syntactic category ranged over by t. The rules of $\mathit{refocus}_{aux}$ go in the auxiliary refocus functions corresponding to the syntactic category of the hole, which is the same as the syntactic category of the value argument.

1. If c needs to evaluate zero sub-terms, then we know that $\mathsf{c}(t_1,\ \ldots,\ t_n)$ is a value or a potential redex.

   (a) If c becomes a value, then

   $$\mathit{refocus}(\mathsf{c}(t_1,\ \ldots,\ t_n), C) = \mathit{refocus}_{aux}(C, \mathsf{c}(t_1,\ \ldots,\ t_n))$$

   (b) If instead c becomes a potential redex, then

   $$\mathit{refocus}(\mathsf{c}(t_1,\ \ldots,\ t_n),\ C) = (C,\ \mathsf{c}(t_1,\ \ldots,\ e_n))$$

   since we have found the decomposition.

2. If c needs to evaluate more than zero sub-terms then the first term must be reduced first, and we simply refocus on it:

$$\mathit{refocus}(\mathsf{c}(t_1,\, \ldots,\, t_n), C) = \mathit{refocus}(t_1, C \circ \mathsf{c}([\,],\, t_2,\, \ldots,\, t_n))$$

Likewise, the $\mathit{refocus}_{aux}$ function is defined by cases on the reduction context on top of the stack. If $\mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, [\,],\, t_{i+1},\, \ldots,\, t_n)$ is the top-most elementary context on the stack, then the corresponding case is one of the following.

1. If c needs to evaluate exactly i sub-terms, then $\mathsf{c}(v_1,\, \ldots,\, v_i,\, t_{i+1},\, \ldots,\, t_n)$, if such a term exists, is either a value or a potential redex.

   (a) If c becomes a value then the case is

   $$\begin{array}{l}\mathit{refocus}_{aux}(C \circ \mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, [\,],\, t_{i+1},\, \ldots,\, t_n), v_i)\\ = \mathit{refocus}_{aux}(C, \mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, v_i,\, t_{i+1},\, \ldots,\, t_n))\end{array}$$

   (b) If c becomes a potential redex then we have found a decomposition into reduction context and potential redex, and the case is

   $$\begin{array}{l}\mathit{refocus}_{aux}(C \circ \mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, [\,],\, t_{i+1},\, \ldots,\, t_n), v_i)\\ = (C,\, \mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, v_i,\, t_{i+1},\, \ldots,\, t_n))\end{array}$$

   (c) If t_i ranges over only non-values then there is no rule. In a typed setting there is no $\mathit{refocus}_{aux}$ corresponding to the constructors of that syntactic category.

2. If c needs to evaluate more than i sub-terms, then we continue searching for a potential redex in the next sub-term, and the rule is:

   $$\begin{array}{l}\mathit{refocus}_{aux}(C \circ \mathsf{c}(v_1,\, \ldots,\, v_{i-1},\, [\,],\, t_{i+1},\, \ldots,\, t_n), v_i)\\ = \mathit{refocus}(t_{i+1}, C \circ \mathsf{c}(v_1,\, \ldots,\, v_i,\, [\,],\, t_n))\end{array}$$

3. Finally there is one rule for the empty context.

   $$\mathit{refocus}_{aux}([\,], v) = v$$

   This base case accounts for the situation where the entire term is a value, so the decomposition has to return a value rather than a context and a potential redex.

In Appendix A, we illustrate the construction of a *refocus* function for a reduction semantics of arithmetic expressions with precedence.

In the following section we show that the *refocus* function generated by the above rules computes a correct decomposition of a term into a reduction context and a potential redex, and that it does so at least as efficiently as the combination of *plug* and *decompose*.

2.6 Correctness

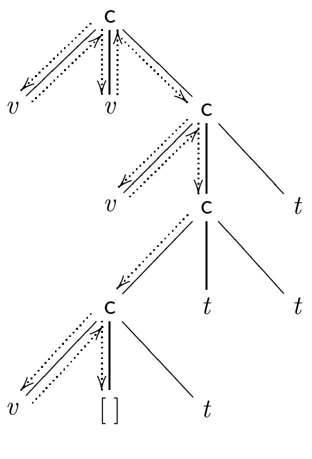
By construction, *refocus* can only return either a value or a pair of a reduction context and a potential redex, since the remaining cases all perform a tail call. Both *refocus* and $refocus_{aux}$ are passed a decomposition of a term, i.e., a reduction context and another term, and if they make a recursive call, it is on another decomposition of the same term. Therefore, if $refocus(t, C)$ returns a decomposition, then it is a complete decomposition of $C[t]$.

It thus suffices to prove that *refocus* terminates on any argument. Let us show that if $C[t] = C'[r]$, where r is a potential redex, then $refocus(t, C)$ performs $\mathcal{T}(C') + \mathcal{T}_{aux}(r) - \mathcal{T}(C)$ tail-recursive calls, where $\mathcal{T}$ and $\mathcal{T}_{aux}$ are defined as follows:

$$\begin{aligned} \mathcal{T} : \textsc{RedCont} &\to N \\ \mathcal{T}([\,]) &= 0 \\ \mathcal{T}(C \circ \mathsf{c}(v_1, \ldots, v_{k-1}, [\,], t_{k+1}, \ldots, t_{e_n})) &= 1 + \mathcal{T}(C) + \textstyle\sum_{i=1}^{k+1}(1 + \mathcal{T}_{aux}(v_i)) \end{aligned}$$

$$\begin{aligned} \mathcal{T}_{aux} : \textsc{Val} + \textsc{PotRedex} &\to N \\ \mathcal{T}_{aux}(\mathsf{c}(v_1, \ldots, v_m, t_{m+1}, \ldots, t_n)) &= 1 + \textstyle\sum_{i=1}^{m}(1 + \mathcal{T}_{aux}(v_i)) \end{aligned}$$

Given a reduction context, $\mathcal{T}$ computes the number of edges traversed to go from the root to the hole in a depth-first left-to-right traversal, counting the edges to values both on the way down and on the way up. This traversal is illustrated by the following picture:



As for $\mathcal{T}_{aux}$, it computes the number of edges followed by a full traversal of a composite value.

All fully traversed sub-terms must be values. With this definition, the result of $\mathcal{T}(C)$ is at most twice the number of nodes visited on such a traversal.

We use the $\mathcal{T}$ function to measure how far the *refocus* function has progressed in its traversal of its argument.

Lemma 2 (Termination) *If $C[t] = C'[r]$ then $refocus(t, C)$ terminates after $\mathcal{T}(C')+$ $\mathcal{T}_{aux}(r) - \mathcal{T}(C)$ tail-recursive calls.*

Proof: Let us use $\mathcal{T}$ to define a *progress measure* on all the left-hand sides and right-hand sides of the definition of *refocus* and $refocus_{aux}$:

$$\begin{aligned}
\textsc{Progress}(\mathit{refocus}(t, C))) &= \mathcal{T}(C) \\
\textsc{Progress}(\mathit{refocus}_{aux}(C, v)) &= \mathcal{T}(C) + \mathcal{T}_{aux}(v) \\
\textsc{Progress}(C, r) &= \mathcal{T}(C) + \mathcal{T}_{aux}(r) \\
\textsc{Progress}(v) &= \mathcal{T}_{aux}(v)
\end{aligned}$$

With this definition, all choices of arguments to *refocus* and $\mathit{refocus}_{aux}$ give a right-hand side with a progress value one greater than the left-hand side. We omit the details and give only one case as example.

If c needs to evaluate m sub-terms to become a value and $0 < k < m$, then $\mathit{refocus}_{aux}$ contains the following case:

$$\begin{aligned}
&\mathit{refocus}_{aux}(C \circ \mathsf{c}(v_1, \ldots, v_{k-1}, [\,], t_{k+1}, t_{k+2}, \ldots, t_n), v_k) \\
&= \mathit{refocus}(t_{k+1}, C \circ \mathsf{c}(v_1, \ldots, v_{k-1}, v_k, [\,], t_{k+2}, \ldots, t_n))
\end{aligned}$$

Taking the progress measure on both the left-hand side and the right-hand side gives the expected one-greater value on the right-hand side:

$$\begin{aligned}
&\textsc{Progress}(\mathit{refocus}_{aux}(C \circ \mathsf{c}(v_1, \ldots, v_{k-1}, [\,], t_{k+1}, \ldots, t_n), v_i)) \\
&= \mathcal{T}(C \circ \mathsf{c}(v_1, \ldots, v_{k-1}, [\,], t_{k+1}, \ldots, t_n)) + \mathcal{T}_{aux}(v_k) \\
&= 1 + \mathcal{T}(C) + \textstyle\sum_{i=1}^{k-1}(1 + \mathcal{T}_{aux}(v_i)) + \mathcal{T}_{aux}(v_k)
\end{aligned}$$

and

$$\begin{aligned}
&\textsc{Progress}(\mathit{refocus}(t_{k+1}, C \circ \mathsf{c}(v_1, \ldots, v_k, [\,], t_{k+2}, \ldots, t_n))) \\
&= \mathcal{T}(C \circ \mathsf{c}(v_1, \ldots, v_k, [\,], t_{k+2}, \ldots, t_n)) \\
&= 1 + \mathcal{T}(C) + \textstyle\sum_{i=1}^{k}(1 + \mathcal{T}_{aux}(v_i)) \\
&= 1 + \mathcal{T}(C) + \textstyle\sum_{i=1}^{k-1}(1 + \mathcal{T}_{aux}(v_i)) + \mathcal{T}_{aux}(v_k) + 1
\end{aligned}$$

Since each recursive call increases the progress measure by one, and there is only a finite possible number of different such calls (each call is to one of two functions on a decomposition of the same term, and a term has only finitely many decompositions), the recursion must eventually end. When this happens, the cumulative increase in the progress measure is also the number of recursive calls performed to reach it. This number is exactly

$$\textsc{Progress}((C', r)) - \textsc{Progress}(\mathit{refocus}(t, C)) = \mathcal{T}(C') + \mathcal{T}_{aux}(r) - \mathcal{T}(C).$$

□

Lemma 2 tells us both that *refocus* is a total function and how many calls it takes to find the result. In Section 2.7, we use it to show that *refocus* is always at least as efficient as using *plug* and *decompose*.

2.7 Complexity

In order to show that *refocus* is more efficient than the composition of *plug* and *decompose*, we must first decide on a relevant complexity measure, and we must find the complexity of *plug* and *decompose*.

For measuring the time complexity of *refocus* we use the number of recursive calls, as in Section 2.6. Each call performs a bounded number of basic operations on the syntax (either constructing a term or deconstructing a term while naming the sub-terms—the latter is performed implicitly since the function is defined by cases). If each such operation takes constant time, the time taken to compute the result of *refocus* is proportional to the number of recursive calls (i.e., bounded by a constant factor times the number of calls).

Let us informally argue that any implementation of a function *decompose* : $\textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})$ finding the unique complete decomposition of terms (in a syntactic category satisfying our requirements) must perform at least $(\mathcal{T}(C) + \mathcal{T}_{aux}(r))/2$ basic operations on the syntax to compute $\mathit{decompose}(C[r]) = (C, r)$. If it uses fewer operations, it cannot inspect all the values of the reduction context and of the potential redex. If we changed such a value to a non-value, then the hole of the reduction context should be in the new non-value term, but the function cannot see this without inspecting the value and thus it must generate an erroneous result, contradicting the assumption that it computes the unique complete decomposition.

The *plug* function is implemented in time proportional to the depth of the hole in the reduction context. Since the time complexity of *decompose* is always greater, we ignore *plug* in our comparison.

In summary, the complexity of computing $\mathit{decompose}(C[t]) = C'[r]$ is at least proportional to $\mathcal{T}(C')+\mathcal{T}_{aux}(r)$, whereas the complexity of computing $\mathit{refocus}(t, C) = C'[r]$ is proportional to $\mathcal{T}(C') + \mathcal{T}_{aux}(r) - \mathcal{T}(C)$. For reduction semantics where rewrites are often local, the difference between $\mathcal{T}(C)$ and $\mathcal{T}(C')$ is often significantly smaller than the value of $\mathcal{T}(C')$ itself. For example, in the Church-numeral example of Section 1.3 (Example 1), the difference between $\mathcal{T}(C)$ and $\mathcal{T}(C')$ is constant whereas $\mathcal{T}(C')$ alone is proportional to the size of the input term.

In some cases, *refocus* does no better than *decompose*, e.g., when the context argument to *refocus* is always empty (e.g., for tail-recursive or continuation-passing style λ-terms, and for trampoline-like programs with control operators where the context is discarded in each step). Likewise, if the size of the context is bounded, the saving is at most a constant factor.

2.8 Summary

We have presented a few mild requirements to construct a *refocus* function for a reduction semantics. We have also proven the correctness of this construction and characterized the complexity of the constructed function. Refocusing is at least as efficient as first plugging and then decomposing, and it is often significantly more efficient.

3 The λ-calculus under call by value

Let us go back to the λ-calculus as initially considered in Section 1.3. We first restate the grammars of the language and of the reduction semantics in the format used in Section 2. These grammars can be expressed directly as ML data types and the corresponding plug and decompose functions as ML functions (see Appendix A).

Terms	$t ::= \mathsf{var}(x) \mid \mathsf{lam}(x,\, t) \mid \mathsf{app}(t,\, t)$	$t \in \textsc{Term}$
Values	$v ::= \mathsf{var}(x) \mid \mathsf{lam}(x,\, t)$	$v \in \textsc{Val}$
Variables		$x \in \textsc{Var}$
Reduction contexts	$C ::= [\,] \mid C[\mathsf{app}([\,],\, t)] \mid C[\mathsf{app}(v,\, [\,])]$	$C \in \textsc{RedCont}$
Potential redexes	$r ::= \mathsf{app}(v,\, v)$	$r \in \textsc{PotRedex}$

We make plugging and decomposition explicit with two functions:

$$\begin{aligned} \mathit{plug} &: \textsc{Term} \times \textsc{RedCont} \to \textsc{Term} \\ \mathit{decompose} &: \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \end{aligned}$$

The refocusing function *refocus* is defined as the composition of *decompose* and *plug*. Its type is thus:

$$\mathit{refocus} : \textsc{Term} \times \textsc{RedCont} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})$$

We now consider a call-by-value interpreter and then two call-by-value CPS transformers for the λ-calculus (Sections 3.1, 3.2, and 3.3). Each uses *decompose* and *plug*, which we fuse into a more efficient *refocus* function (Section 3.4).

3.1 A call-by-value interpreter

We start from a traditional call-by-value reduction semantics for the λ-calculus [10–12]. The corresponding interpreter is defined as the transitive closure of (1) decomposing a term into a reduction context and a potential redex, (2) contracting this redex if possible, and (3) plugging the contractum into the context. We restate this interpreter so that the decomposition function is always applied to the result of the plug function.

3.1.1 The original specification

The following interpreter implements the evaluation function of the reduction semantics. It takes a term as argument and repeatedly decomposes, contracts, and plugs until either a value or a stuck term is reached, if any:

$$\begin{aligned} \mathit{evaluate} &: \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{evaluate}(t) &= \mathit{iterate}(\mathit{decompose}(t)) \end{aligned}$$

$$\begin{aligned} \mathit{iterate} : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{iterate}(v) &= v \\ \mathit{iterate}(C,\, \mathsf{app}(\mathsf{lam}(x,\, t),\, v)) &= \mathit{evaluate}(\mathit{plug}(t\,\{v/x\},\, C)) \\ \mathit{iterate}(C,\, \mathsf{app}(\mathsf{var}(x),\, v)) &= (C,\, \mathsf{app}(\mathsf{var}(x),\, v)) \end{aligned}$$

The evaluation of a term t is $\mathit{evaluate}(t)$. (NB: The last clause of *iterate* is used for stuck terms.)

The interpreter contains two calls to *evaluate*: one in the second clause of *iterate* and one in the initial call. In the second clause of *iterate*, the argument of *evaluate* is the result of *plug*, and the same holds in the initial call, since $t = plug(t, [\,])$. In that sense, *decompose* is always called with the result of *plug*.

Let us re-express the interpreter with a refocusing function that combines the effects of *decompose* and *plug*.

3.1.2 The refocused specification

We change the interpreter to use *refocus* instead of *decompose* and *plug*. Instead of $decompose(t)$ above, we write $decompose(plug(t, [\,]))$, i.e., $refocus(t, [\,])$; and instead of $evaluate(plug(t\,\{v/x\}, C))$, we inline the (old) definition of *evaluate* and write $iterate(decompose(plug(t\,\{v/x\}, C)))$, i.e., $iterate(refocus(t\,\{v/x\}, C))$:

$$\begin{aligned} evaluate &: \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ evaluate(t) &= iterate(refocus(t, [\,])) \end{aligned}$$

$$\begin{aligned} iterate : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ iterate(v) &= v \\ iterate(C, \mathsf{app}(\mathsf{lam}(x, t), v)) &= iterate(refocus(t\,\{v/x\}, C)) \\ iterate(C, \mathsf{app}(\mathsf{var}(x), v)) &= (C, \mathsf{app}(\mathsf{var}(x), v)) \end{aligned}$$

The evaluation of a term t is $evaluate(t)$.

Proposition 1 *The original interpreter and the refocused interpreter implement the same evaluation function.*

Proof: Immediate. □

We are now free to use any implementation of *refocus* that is extensionally equivalent to $decompose \circ plug$.

3.2 Sabry and Felleisen's CPS transformer

In their work on reasoning about programs in continuation-passing style (CPS), Sabry and Felleisen designed a new CPS transformation [23, Definition 5]. This CPS transformation integrates a notion of generalized reduction and thus yields very compact CPS programs [6]. It is also unusual in the sense that it builds on the notion of a reduction semantics. Therefore, it is defined as the transitive closure of decomposing, performing an elementary CPS transformation, and plugging, which makes it a candidate for refocusing.

3.2.1 The original specification

Definition 2 (Sabry and Felleisen, 1993) *The following CPS transformer uses three mutually recursive functions: $\mathcal{C}_k$ (and the auxiliary function $\mathcal{C}'_k$) to transform terms, $\mathbf{\Phi}$ to transform values, and $\mathcal{K}_k$ to transform reduction contexts. The functions $\mathcal{C}_k$, $\mathcal{C}'_k$, and $\mathcal{K}_k$ are parameterized over a variable k that represents the current continuation.*

$$\begin{aligned} \mathcal{C}_k &: \textsc{Term} \to \textsc{Term} \\ \mathcal{C}_k(t) &= \mathcal{C}'_k(decompose(t)) \end{aligned}$$

$$\begin{array}{rcl}
\mathcal{C}'_k \ : \ \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Term} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(\mathsf{var}(k),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(\mathsf{var}(x),\ v)) & = & \mathsf{app}(\mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(\mathsf{lam}(x,\ t),\ v)) & = & \mathsf{app}(\mathsf{lam}(x,\ \mathcal{C}_k(\mathit{plug}(t,\ C))),\ \mathbf{\Phi}(v))
\end{array}$$

$$\begin{array}{rcl}
\mathbf{\Phi} \ : \ \textsc{Val} & \rightarrow & \textsc{Val} \\
\mathbf{\Phi}(\mathsf{var}(x)) & = & \mathsf{var}(x) \\
\mathbf{\Phi}(\mathsf{lam}(x,\ t)) & = & \mathsf{lam}(k,\ \mathsf{lam}(x,\ \mathcal{C}_k(t))) \\
 & & \textit{for a fresh } k
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}_k \ : \ \textsc{RedCont} & \rightarrow & \textsc{Term} \\
\mathcal{K}_k([\,]) & = & \mathsf{var}(k) \\
\mathcal{K}_k(C[\mathsf{app}(\mathsf{var}(x),\ [\,])]) & = & \mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)) \\
\mathcal{K}_k(C[\mathsf{app}(\mathsf{lam}(x,\ t),\ [\,])]) & = & \mathsf{lam}(x,\ \mathcal{C}_k(\mathit{plug}(t,\ C))) \\
\mathcal{K}_k(C[\mathsf{app}([\,],\ t)]) & = & \mathsf{lam}(u_i,\ \mathcal{C}_k(\mathit{plug}(\mathsf{app}(\mathsf{var}(u_i),\ t),\ C))) \\
 & & \textit{for a fresh } u_i
\end{array}$$

The CPS transformation of a term t is $\mathsf{lam}(k,\ \mathcal{C}_k(t))$*, for a fresh k.* □

The CPS transformer contains five calls to $\mathcal{C}_k$ (counting the initial one). In three cases, the argument of $\mathcal{C}_k$ is the result of *plug*, and in two cases, the argument is t. As in Section 3.1.1, and since $t = \mathit{plug}(t,\ [\,])$, we can say that *decompose*, in the definition of $\mathcal{C}_k$, is always called with the result of *plug*.

If they are implemented literally, decomposition and plugging entail a time factor for each transformation step that is linear in the size of the input term, in the worst case. Overall, the worst-case time complexity of this CPS transformation is quadratic in the size of the input term.

Let us re-express the CPS transformer with a refocusing function that combines the effects of *decompose* and *plug*.

3.2.2 The refocused specification

We change the CPS transformer to use *refocus* instead of *decompose* and *plug*.

$$\begin{array}{rcl}
\mathcal{C}_k \ : \ \textsc{Term} & \rightarrow & \textsc{Term} \\
\mathcal{C}_k(t) & = & \mathcal{C}'_k(\mathit{refocus}(t,\ [\,]))
\end{array}$$

$$\begin{array}{rcl}
\mathcal{C}'_k \ : \ \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Term} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(\mathsf{var}(k),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(\mathsf{var}(x),\ v)) & = & \mathsf{app}(\mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(\mathsf{lam}(x,\ t),\ v)) & = & \mathsf{app}(\mathsf{lam}(x,\ \mathcal{C}'_k(\mathit{refocus}(t,\ C))),\ \mathbf{\Phi}(v))
\end{array}$$

$$\begin{array}{rcl}
\mathbf{\Phi} \ : \ \textsc{Val} & \rightarrow & \textsc{Val} \\
\mathbf{\Phi}(\mathsf{var}(x)) & = & \mathsf{var}(x) \\
\mathbf{\Phi}(\mathsf{lam}(x,\ t)) & = & \mathsf{lam}(k,\ \mathsf{lam}(x,\ \mathcal{C}_k(t))) \\
 & & \text{for a fresh } k
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}_k \ : \ \textsc{RedCont} & \rightarrow & \textsc{Term} \\
\mathcal{K}_k([\,]) & = & \mathsf{var}(k) \\
\mathcal{K}_k(C[\mathsf{app}(\mathsf{var}(x),\ [\,])]) & = & \mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)) \\
\mathcal{K}_k(C[\mathsf{app}(\mathsf{lam}(x,\ t),\ [\,])]) & = & \mathsf{lam}(x,\ \mathcal{C}'_k(\mathit{refocus}(t,\ C))) \\
\mathcal{K}_k(C[\mathsf{app}([\,],\ t)]) & = & \mathsf{lam}(u_i,\ \mathcal{C}'_k(\mathit{refocus}(\mathsf{app}(\mathsf{var}(u_i),\ t),\ C))) \\
 & & \text{for a fresh } u_i
\end{array}$$

The CPS transformation of a term t is $\mathsf{lam}(k,\, \mathcal{C}_k(t))$, for a fresh k.

Proposition 2 *The original CPS transformer and the refocused CPS transformer implement the same transformation function.*

Proof: Immediate. $\square$

We are now free to use any implementation of *refocus* that is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$.

3.3 A simpler CPS transformer

We present a simplified, less compacting, version of Sabry and Felleisen's context-based CPS transformer, one that produces CPS terms à la Plotkin ($\lambda x.\lambda k.\ldots$) rather than à la Fischer ($\lambda k.\lambda x.\ldots$) [6]. We then restate it so that the decomposition function is always applied to the result of the plug function.

3.3.1 The original specification

The following CPS transformation repeatedly decomposes a source term into a context and the application of one value to another value, CPS transforms the application, and plugs a fresh variable in the context. This process continues until the source term is a value.

Definition 3 (Context-based CPS transformation)

$$\begin{array}{rcl}
\mathcal{C}_k : \textsc{Term} & \rightarrow & \textsc{Term} \\
\mathcal{C}_k(t) & = & \mathcal{C}'_k(\mathit{decompose}(t)) \\[1em]
\mathcal{C}'_k : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Term} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(\mathsf{var}(k),\, \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\, \mathsf{app}(v,\, v')) & = & \mathsf{app}(\mathsf{app}(\mathbf{\Phi}(v),\, \mathbf{\Phi}(v')),\, \mathcal{K}_k(C)) \\[1em]
\mathbf{\Phi} : \textsc{Val} & \rightarrow & \textsc{Val} \\
\mathbf{\Phi}(\mathsf{var}(x)) & = & \mathsf{var}(x) \\
\mathbf{\Phi}(\mathsf{lam}(x,\, t)) & = & \mathsf{lam}(x,\, \mathsf{lam}(k,\, \mathcal{C}_k(t))) \\
& & \textit{for a fresh } k \\[1em]
\mathcal{K}_k : \textsc{RedCont} & \rightarrow & \textsc{Val} \\
\mathcal{K}_k(C) & = & \mathsf{lam}(u_i,\, \mathcal{C}_k(\mathit{plug}(\mathsf{var}(u_i),\, C))) \\
& & \textit{for a fresh } u_i
\end{array}$$

The CPS transformation of a term t is $\mathsf{lam}(k,\, \mathcal{C}_k(t))$, for a fresh k. $\square$

Again, and for the same reasons as in Section 3.2.1, the worst-case time complexity of this CPS transformation is quadratic in the size of the input term.

3.3.2 The refocused specification

We change the CPS transformer to use *refocus* instead of *decompose* and *plug*. Instead of $\mathit{decompose}(t)$ above, we write $\mathit{decompose}(\mathit{plug}(t,\, [\,]))$, i.e., $\mathit{refocus}(t,\, [\,])$; and instead of $\mathcal{C}_k(\mathit{plug}(\mathsf{var}(u_i),\, C))$, we write $\mathcal{C}'_k(\mathit{decompose}(\mathit{plug}(\mathsf{var}(u_i),\, C)))$, i.e., we write $\mathcal{C}'_k(\mathit{refocus}(\mathsf{var}(u_i),\, C))$:

$$
\begin{aligned}
\mathcal{C}_k : \textsc{Term} &\to \textsc{Term}\\
\mathcal{C}_k(t) &= \mathcal{C}'_k(\mathit{refocus}(t,\ [\,]))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}'_k : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) &\to \textsc{Term}\\
\mathcal{C}'_k(v) &= \mathsf{app}(\mathsf{var}(k),\ \mathbf{\Phi}(v))\\
\mathcal{C}'_k(C,\ \mathsf{app}(v,\ v')) &= \mathsf{app}(\mathsf{app}(\mathbf{\Phi}(v),\ \mathbf{\Phi}(v')),\ \mathcal{K}_k(C))
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{\Phi} : \textsc{Val} &\to \textsc{Val}\\
\mathbf{\Phi}(\mathsf{var}(x)) &= \mathsf{var}(x)\\
\mathbf{\Phi}(\mathsf{lam}(x,\ t)) &= \mathsf{lam}(x,\ \mathsf{lam}(k,\ \mathcal{C}_k(t)))\\
&\ \text{for a fresh } k
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{K}_k : \textsc{RedCont} &\to \textsc{Val}\\
\mathcal{K}_k(C) &= \mathsf{lam}(u_i,\ \mathcal{C}'_k(\mathit{refocus}(\mathsf{var}(u_i),\ C)))\\
&\ \text{for a fresh } u_i
\end{aligned}
$$

The CPS transformation of a term t is $\mathsf{lam}(k,\ \mathcal{C}_k(t))$, for a fresh k.

Proposition 3 *The original CPS transformer and the refocused CPS transformer implement the same transformation function.*

Proof: Immediate. □

We are now free to use any implementation of *refocus* that is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$.

3.4 Refocusing efficiently

Applying the construction of Section 2.5 yields the following functions:

$$
\begin{aligned}
\mathit{refocus}\ :\ \textsc{Term} \times \textsc{RedCont} &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})\\
\mathit{refocus}(v,\ C) &= \mathit{refocus}_{aux}(C,\ v)\\
\mathit{refocus}(\mathsf{app}(t,\ t'),\ C) &= \mathit{refocus}(t,\ C[\mathsf{app}([\,],\ t')])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{refocus}_{aux}\ :\ \textsc{RedCont} \times \textsc{Val} &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})\\
\mathit{refocus}_{aux}([\,],\ v) &= v\\
\mathit{refocus}_{aux}(C[\mathsf{app}([\,],\ t')],\ v) &= \mathit{refocus}(t',\ C[\mathsf{app}(v,\ [\,])])\\
\mathit{refocus}_{aux}(C[\mathsf{app}(v,\ [\,])],\ v') &= (C,\ \mathsf{app}(v,\ v'))
\end{aligned}
$$

By construction, *refocus* implements a depth-first search for the first application of one value to another, by recursively descending into a term and accumulating the corresponding context. As for $\mathit{refocus}_{aux}$, it dispatches on the accumulated context.

The following definition of *decompose* holds as a corollary:

$$
\begin{aligned}
\mathit{decompose} : \textsc{Term} &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})\\
\mathit{decompose}(t) &= \mathit{refocus}(t,\ [\,])
\end{aligned}
$$

3.4.1 The call-by-value interpreter

Proposition 4 *The new refocused interpreter and the previous one implement the same evaluation function.*

Proof: A corollary of the correctness of the refocus construction. □

Let us return to Church-numeral example of Section 1.3. When evaluating a term such as $\lceil n \rceil \, (\lambda x.x) \, v$, the time taken to efficiently refocus from a contractum and a context to a new context and a redex is independent of the number of remaining applications. The new interpreter therefore operates more efficiently than the original one.

3.4.2 Sabry and Felleisen's CPS transformer

Proposition 5 *The new refocused CPS transformer and the previous one implement the same transformation function.*

Proof: A corollary of the correctness of the refocus construction. □

In contrast to the original CPS transformer, the new refocused CPS transformer operates in one pass over its input.

Incidentally, the transformation can be optimized to read as follows:

$$\mathcal{K}_k(C[\mathsf{app}([\,],\, t)]) = \mathsf{lam}(u_i,\, \mathcal{C}_k(\mathit{refocus}(t,\, C[\mathsf{app}(\mathsf{var}(u_i),\, [\,])])))$$

This way, each $\mathsf{var}(u_i)$ is not inspected as a term.

3.4.3 The simpler CPS transformer

Proposition 6 *The new refocused CPS transformer and the previous one implement the same transformation function.*

Proof: A corollary of the correctness of the refocus construction. □

In contrast to the original CPS transformer, the new refocused CPS transformer operates in one pass over its input.

3.5 Analysis of the refocused call-by-value interpreter

The refocused interpreter takes the form of a "pre-abstract machine" (Section 3.5.1). We successively show how to transform this pre-abstract machine into an abstract machine (Section 3.5.2), how to put this abstract machine in eval/apply form (Section 3.5.3), and how to put it in push/enter form (Section 3.5.4). We also exhibit the corresponding big-step semantics (Section 3.5.5).

3.5.1 A pre-abstract machine

In Section 3.4.1, the refocused interpreter in a particular form, which we name *pre-abstract machine*, and which involves a transition function and a 'trampoline' [14]: *refocus* and $\mathit{refocus}_{aux}$ (in Section 3.4) define one transition function and *iterate* (in Section 3.1.2) defines a trampoline, i.e., another transition function that keeps activating the other until a value is obtained, if there is any. The trampoline function computes the transitive closure of the transition function.

3.5.2 From the pre-abstract machine to an abstract machine

In the pre-abstract machine, *iterate* is always called on the result of *refocus*. Let us fuse these two functions into one transition function so that *refocus* is called tail-recursively and that *iterate* is applied to its result (and to the result of $\mathit{refocus}_{aux}$):

$$
\begin{array}{rcl}
\mathit{evaluate} \;:\; \textsc{Term} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t,\ [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{iterate} \;:\; \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{iterate}(v) & = & v \\
\mathit{iterate}(C,\ \mathsf{app}(\mathsf{lam}(x,\ t),\ v)) & = & \mathit{refocus}(t\,\{v/x\},\ C) \\
\mathit{iterate}(C,\ \mathsf{app}(\mathsf{var}(x),\ v)) & = & (C,\ \mathsf{app}(\mathsf{var}(x),\ v))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{refocus} \;:\; \textsc{Term} \times \textsc{RedCont} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v,\ C) & = & \mathit{refocus}_{\mathit{aux}}(C,\ v) \\
\mathit{refocus}(\mathsf{app}(t,\ t'),\ C) & = & \mathit{refocus}(t,\ C[\mathsf{app}([\,],\ t')])
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{refocus}_{\mathit{aux}} \;:\; \textsc{RedCont} \times \textsc{Val} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}_{\mathit{aux}}([\,],\ v) & = & \mathit{iterate}(v) \\
\mathit{refocus}_{\mathit{aux}}(C[\mathsf{app}([\,],\ t')],\ v) & = & \mathit{refocus}(t',\ C[\mathsf{app}(v,\ [\,])]) \\
\mathit{refocus}_{\mathit{aux}}(C[\mathsf{app}(v,\ [\,])],\ v') & = & \mathit{iterate}(C,\ \mathsf{app}(v,\ v'))
\end{array}
$$

The result is a (tail-recursive) state-transition function, i.e., an abstract machine [21].

Proposition 7 *The pre-abstract machine and the abstract machine implement the same evaluation function.*

Proof: Immediate. □

The form of the abstract machine is rather ideal because *iterate* implements the reduction rules of the reduction semantics and *refocus* and $\mathit{refocus}_{\mathit{aux}}$ implement its congruence rules—a distinction that usually requires a non-trivial analysis to establish for existing abstract machines [16].

3.5.3 An eval/apply abstract machine

Inlining *iterate* yields an eval/apply abstract machine [18] which we recognize to be Felleisen et al.'s CK machine [10–12]:

$$
\begin{array}{rcl}
\mathit{evaluate} \;:\; \textsc{Term} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t,\ [\,])
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{refocus} \;:\; \textsc{Term} \times \textsc{RedCont} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v,\ C) & = & \mathit{refocus}_{\mathit{aux}}(C,\ v) \\
\mathit{refocus}(\mathsf{app}(t,\ t'),\ C) & = & \mathit{refocus}(t,\ C[\mathsf{app}([\,],\ t')])
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{refocus}_{\mathit{aux}} \;:\; \textsc{RedCont} \times \textsc{Val} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}_{\mathit{aux}}([\,],\ v) & = & v \\
\mathit{refocus}_{\mathit{aux}}(C[\mathsf{app}([\,],\ t')],\ v) & = & \mathit{refocus}(t',\ C[\mathsf{app}(v,\ [\,])]) \\
\mathit{refocus}_{\mathit{aux}}(C[\mathsf{app}(\mathsf{lam}(x,\ t),\ [\,])],\ v) & = & \mathit{refocus}(t\,\{v/x\},\ C) \\
\mathit{refocus}_{\mathit{aux}}(C[\mathsf{app}(\mathsf{var}(x),\ [\,])],\ v) & = & (C,\ \mathsf{app}(\mathsf{var}(x),\ v))
\end{array}
$$

Proposition 8 *The abstract machine and the eval/apply abstract machine implement the same evaluation function.*

Proof: Immediate. □

3.5.4 A push/enter abstract machine

Inlining $\mathit{refocus}_{\mathit{aux}}$ yields a push/enter [18] version of the CK machine:

$$\begin{array}{rcl}
\mathit{evaluate} & : & \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t,\ [\,])
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus} & : & \textsc{Term} \times \textsc{RedCont} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v,\ [\,]) & = & v \\
\mathit{refocus}(v,\ C[\mathsf{app}([\,],\ t')]) & = & \mathit{refocus}(t',\ C[\mathsf{app}(v,\ [\,])]) \\
\mathit{refocus}(v,\ C[\mathsf{app}(\mathsf{lam}(x,\ t),\ [\,])]) & = & \mathit{refocus}(t\,\{v/x\},\ C) \\
\mathit{refocus}(v,\ C[\mathsf{app}(\mathsf{var}(x),\ [\,])]) & = & (C,\ \mathsf{app}(\mathsf{var}(x),\ v)) \\
\mathit{refocus}(\mathsf{app}(t,\ t'),\ C) & = & \mathit{refocus}(t,\ C[\mathsf{app}([\,],\ t')])
\end{array}$$

Proposition 9 *The eval/apply abstract machine and the push/enter abstract machine implement the same evaluation function.*

Proof: Immediate. □

3.5.5 From the eval/apply abstract machine to a big-step semantics

As already observed elsewhere [1, 5], the eval/apply CK machine of Section 3.5.3 is in defunctionalized form [7, 22]: the reduction contexts, together with $\mathit{refocus}_{\mathit{aux}}$, are the first-order counterpart of a function (or, more precisely, of a continuation). The higher-order counterpart of this abstract machine is a continuation-passing evaluation function. Writing it in direct style [4] yields an evaluation function that implements a traditional big-step operational semantics of the λ-calculus under call by value [19, 24]. We state it below without proof and in a form where the contexts are constructed inside-out in case of stuck terms:

$$\frac{}{v \Downarrow v}$$

$$\frac{t \Downarrow \mathsf{lam}(x,\ t'') \quad t' \Downarrow v' \quad t''\,\{v'/x\} \Downarrow a}{\mathsf{app}(t,\ t') \Downarrow a} \qquad \frac{t \Downarrow \mathsf{var}(x) \quad t' \Downarrow v'}{\mathsf{app}(t,\ t') \Downarrow ([\,],\ \mathsf{app}(\mathsf{var}(x),\ v'))}$$

$$\frac{t \Downarrow (C,\ r)}{\mathsf{app}(t,\ t') \Downarrow (\mathsf{app}(C,\ t'),\ r)} \qquad \frac{t \Downarrow v \quad t' \Downarrow (C,\ r)}{\mathsf{app}(t,\ t') \Downarrow (\mathsf{app}(v,\ C),\ r)}$$

3.6 Analysis of the refocused CPS transformers

The refocused CPS transformers also take the form of pre-abstract machines, i.e., of a trampoline function computing the transitive closure of a transition function. We can fuse these two functions into an abstract machine. This abstract machine is in defunctionalized form, and we can write its higher-order counterpart, obtaining a traditional one-pass CPS transformer, as we have shown elsewhere [9].

3.7 Summary

We have described how to efficiently implement the evaluation function of a call-by-value reduction semantics for the λ-calculus, by deforesting the intermediate terms produced in each decompose-contract-plug cycle. As a byproduct, we have shown how refocusing makes it possible to mechanically obtain an array of abstract machines—including the CK machine—out of this reduction semantics. From these abstract machines, we have exhibited the corresponding big-step operational semantics. It is also simple to walk back from a big-step operational semantics to an abstract machine, from this abstract machine to a refocused interpreter, and from this refocused interpreter to a reduction semantics.

We have also shown how to refocus quadratic-time context-based CPS transformers into linear-time ones.

In the literature [11], the decompose function is usually left unspecified. Our work suggests that an iterative definition with an accumulator is convenient since it connects one-step reduction and evaluation in the form of an abstract machine.

4 The λ-calculus under call by name

We briefly illustrate the call-by-name counterpart of Section 3: a call-by-name interpreter and a simple call-by-name CPS transformer for the λ-calculus. We first state the grammars of the language and of the reduction semantics.

Terms	$t ::= \mathsf{var}(x) \mid \mathsf{val}(x) \mid \mathsf{lam}(x,\ t) \mid \mathsf{app}(t,\ t)$	$t \in \textsc{Term}$
Values	$v ::= \mathsf{val}(x) \mid \mathsf{lam}(x,\ t)$	$v \in \textsc{Val}$
Variables		$x \in \textsc{Var}$
Reduction contexts	$C ::= [\,] \mid C[\mathsf{app}([\,],\ t)]$	$C \in \textsc{RedCont}$
Potential redexes	$r ::= \mathsf{app}(v,\ t) \mid \mathsf{var}(x)$	$r \in \textsc{PotRedex}$

The val constructor is used for variables that denote values (e.g., literals) rather than computations—for those, the var constructor is used.

Plugging and decomposition are achieved with the two following functions:

$$\begin{aligned} \mathit{plug} &: \textsc{Term} \times \textsc{RedCont} \rightarrow \textsc{Term} \\ \mathit{decompose} &: \textsc{Term} \rightarrow \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \end{aligned}$$

We now consider a call-by-name interpreter and then a call-by-name CPS transformer for the λ-calculus. Each uses *decompose* and *plug*, which we fuse into a more efficient *refocus* function:

$$\mathit{refocus} : \textsc{Term} \times \textsc{RedCont} \rightarrow \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})$$

4.1 A call-by-name interpreter

We start from a traditional reduction semantics for the λ-calculus under call by name [11]. The corresponding interpreter is defined as the transitive closure of (1) decomposing a term into a reduction context and a potential redex, (2) contracting this redex, if possible, and (3) plugging the contractum in the context. We restate this interpreter so that the decomposition function is always applied to the result of the plug function.

4.1.1 The original specification

The following interpreter implements the evaluation function of the reduction semantics. It takes a term as argument and repeatedly decomposes, contracts, and plugs until either a value or a stuck term is reached, if any.

$$\begin{aligned} \mathit{evaluate} &: \textsc{Term} \rightarrow \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{evaluate}(t) &= \mathit{iterate}(\mathit{decompose}(t)) \end{aligned}$$

$$\begin{aligned} \mathit{iterate} : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) &\rightarrow \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{iterate}(v) &= v \\ \mathit{iterate}(C,\ \mathsf{app}(\mathsf{val}(x),\ t)) &= (C,\ \mathsf{app}(\mathsf{val}(x),\ t)) \\ \mathit{iterate}(C,\ \mathsf{app}(\mathsf{lam}(x,\ t),\ t')) &= \mathit{evaluate}(\mathit{plug}(t\,\{t'/x\},\ C)) \\ \mathit{iterate}(C,\ \mathsf{var}(x)) &= (C,\ \mathsf{var}(x)) \end{aligned}$$

The evaluation of a term t is $\mathit{evaluate}(t)$. (NB: The second and fourth clauses of *iterate* are used for stuck terms.)

The interpreter contains two calls to *evaluate*: one in the third clause of *iterate* and one in the initial call. In the third clause of *iterate*, the argument of *evaluate* is the result of *plug*, and the same holds in the initial call, since $t = \mathit{plug}(t, [\,])$. Therefore, *decompose* is always called with the result of *plug*.

Let us re-express the interpreter with a refocusing function that combines the effects of *decompose* and *plug*.

4.1.2 The refocused specification

We change the interpreter to use *refocus* instead of *decompose* and *plug*. Instead of $\mathit{decompose}(t)$ above, we write $\mathit{decompose}(\mathit{plug}(t, [\,]))$, i.e., $\mathit{refocus}(t, [\,])$; and instead of $\mathit{evaluate}(\mathit{plug}(t\{t'/x\}, C))$, we write $\mathit{iterate}(\mathit{decompose}(\mathit{plug}(t\{t'/x\}, C)))$, i.e., $\mathit{iterate}(\mathit{refocus}(t\{t'/x\}, C))$:

$$\begin{aligned} \mathit{evaluate} &: \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{evaluate}(t) &= \mathit{iterate}(\mathit{refocus}(t, [\,])) \end{aligned}$$

$$\begin{aligned} \mathit{iterate} : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) &\to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\ \mathit{iterate}(v) &= v \\ \mathit{iterate}(C, \mathsf{app}(\mathsf{val}(x), t)) &= (C, \mathsf{app}(\mathsf{val}(x), t)) \\ \mathit{iterate}(C, \mathsf{app}(\mathsf{lam}(x, t), t')) &= \mathit{iterate}(\mathit{refocus}(t\{t'/x\}, C)) \\ \mathit{iterate}(C, \mathsf{var}(x)) &= (C, \mathsf{var}(x)) \end{aligned}$$

The evaluation of a term t is $\mathit{evaluate}(t)$.

Proposition 10 *The original interpreter and the refocused interpreter implement the same evaluation function.*

Proof: Immediate. □

We are now free to use any implementation of *refocus* that is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$.

4.2 A simple CPS transformer

We present the call-by-name analogue of the context-based CPS transformer of Section 3.3. We then restate it so that the decomposition function is always applied to the result of the plug function.

4.2.1 The original specification

The following CPS transformation repeatedly decomposes a source term into a context and the application of one value to another value, CPS transforms the application, and plugs a fresh variable (one that denotes a value) into the context. This process continues until the source term is a value.

Definition 4 (Context-based CPS transformation)

$$\begin{aligned} \mathcal{C}_k &: \textsc{Term} \to \textsc{Term} \\ \mathcal{C}_k(t) &= \mathcal{C}'_k(\mathit{decompose}(t)) \end{aligned}$$

$$\begin{array}{rcl}
\mathcal{C}'_k : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Term} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(\mathsf{var}(k),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(v,\ t)) & = & \mathsf{app}(\mathsf{app}(\mathbf{\Phi}(v),\ \mathsf{lam}(k',\ \mathcal{C}_{k'}(t))),\ \mathcal{K}_k(C)) \\
& & \textit{for a fresh } k' \\
\mathcal{C}'_k(C,\ \mathsf{var}(x)) & = & \mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)) \\
\\
\mathbf{\Phi} : \textsc{Val} & \rightarrow & \textsc{Val} \\
\mathbf{\Phi}(\mathsf{val}(x)) & = & \mathsf{val}(x) \\
\mathbf{\Phi}(\mathsf{lam}(x,\ t)) & = & \mathsf{lam}(x,\ \mathsf{lam}(k,\ \mathcal{C}_k(t))) \\
& & \textit{for a fresh } k \\
\\
\mathcal{K}_k : \textsc{RedCont} & \rightarrow & \textsc{Val} \\
\mathcal{K}_k(C) & = & \mathsf{lam}(u_i,\ \mathcal{C}_k(\mathit{plug}(\mathsf{val}(u_i),\ C))) \\
& & \textit{for a fresh } u_i
\end{array}$$

The CPS transformation of a term t is $\mathsf{lam}(k,\ \mathcal{C}_k(t))$*, for a fresh k.* □

As in Section 3, the worst-case time complexity of this CPS transformation is quadratic in the size of the input term.

4.2.2 The refocused specification

We change the CPS transformer to use *refocus* instead of *decompose* and *plug*. Instead of $\mathit{decompose}(t)$ above, we write $\mathit{decompose}(\mathit{plug}(t,\ [\,]))$, i.e., $\mathit{refocus}(t,\ [\,])$; and instead of $\mathcal{C}_k(\mathit{plug}(\mathsf{val}(u_i),\ C))$, we write $\mathcal{C}'_k(\mathit{decompose}(\mathit{plug}(\mathsf{val}(u_i),\ C)))$, i.e., $\mathcal{C}'_k(\mathit{refocus}(\mathsf{val}(u_i),\ C))$:

$$\begin{array}{rcl}
\mathcal{C}_k : \textsc{Term} & \rightarrow & \textsc{Term} \\
\mathcal{C}_k(t) & = & \mathcal{C}'_k(\mathit{refocus}(t,\ [\,])) \\
\\
\mathcal{C}'_k : \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Term} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(\mathsf{var}(k),\ \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(C,\ \mathsf{app}(v,\ t)) & = & \mathsf{app}(\mathsf{app}(\mathbf{\Phi}(v),\ \mathsf{lam}(k',\ \mathcal{C}_{k'}(t))),\ \mathcal{K}_k(C)) \\
& & \text{for a fresh } k' \\
\mathcal{C}'_k(C,\ \mathsf{var}(x)) & = & \mathsf{app}(\mathsf{var}(x),\ \mathcal{K}_k(C)) \\
\\
\mathbf{\Phi} : \textsc{Val} & \rightarrow & \textsc{Val} \\
\mathbf{\Phi}(\mathsf{val}(x)) & = & \mathsf{val}(x) \\
\mathbf{\Phi}(\mathsf{lam}(x,\ t)) & = & \mathsf{lam}(x,\ \mathsf{lam}(k,\ \mathcal{C}_k(t))) \\
& & \text{for a fresh } k \\
\\
\mathcal{K}_k : \textsc{RedCont} & \rightarrow & \textsc{Val} \\
\mathcal{K}_k(C) & = & \mathsf{lam}(u_i,\ \mathcal{C}'_k(\mathit{refocus}(\mathsf{val}(u_i),\ C))) \\
& & \text{for a fresh } u_i
\end{array}$$

The CPS transformation of a term t is $\mathsf{lam}(k,\ \mathcal{C}_k(t))$, for a fresh k.

Proposition 11 *The original CPS transformer and the refocused CPS transformer implement the same transformation function.*

Proof: Immediate. □

We are now free to use any implementation of *refocus* that is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$.

4.3 Refocusing efficiently

Applying the construction of Section 2.5 yields the following functions:

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{refocus} \;:\; \textsc{Term} \times \textsc{RedCont} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})} \\ \mathit{refocus}(v,\, C) & = & \mathit{refocus}_{aux}(C,\, v) \\ \mathit{refocus}(\mathsf{app}(t,\, t'),\, C) & = & \mathit{refocus}(t,\, C[\mathsf{app}([\,],\, t')]) \\ \mathit{refocus}(\mathsf{var}(x),\, C) & = & (C,\, \mathsf{var}(x)) \end{array}$$

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{refocus}_{aux} \;:\; \textsc{RedCont} \times \textsc{Val} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})} \\ \mathit{refocus}_{aux}([\,],\, v) & = & v \\ \mathit{refocus}_{aux}(C[\mathsf{app}([\,],\, t)],\, v) & = & (C,\, \mathsf{app}(v,\, t)) \end{array}$$

By construction, *refocus* implements a depth-first search for the first application of one value to another, by recursively descending into a term and accumulating the corresponding context. As for $\mathit{refocus}_{aux}$, it dispatches on the accumulated context.

The following definition of *decompose* holds as a corollary:

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{decompose} : \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex})} \\ \mathit{decompose}(t) & = & \mathit{refocus}(t,\, [\,]) \end{array}$$

4.3.1 The call-by-name interpreter

Proposition 12 *The new refocused interpreter and the previous one implement the same evaluation function.*

Proof: A corollary of the correctness of the refocus construction. □

As in Section 3.4.1, the new refocused interpreter operates more efficiently than the original one, e.g., on the Church-numerals example of Section 1.3.

4.3.2 The simple CPS transformer

Proposition 13 *The new refocused CPS transformer and the previous one implement the same transformation function.*

Proof: A corollary of the correctness of the refocus construction. □

In contrast to the original CPS transformer, the new refocused CPS transformer operates in one pass over its input.

4.4 Analysis of the refocused call-by-name interpreter

As in Section 3.5, the refocused interpreter takes the form of a "pre-abstract machine". We successively show how to transform this pre-abstract machine into an abstract machine, how to put this abstract machine in eval/apply form, and how to put it in push/enter form. We also exhibit the corresponding big-step semantics.

4.4.1 A pre-abstract machine

As in Section 3.1.2, the refocused interpreter is in the form of a pre-abstract machine: *iterate* is a trampoline function computing the transitive closure of the transition function defined by *refocus* and $\mathit{refocus}_{aux}$.

4.4.2 From the pre-abstract machine to an abstract machine

In the pre-abstract machine, *iterate* is always called on the result of *refocus*. Let us fuse these two functions into one transition function so that *refocus* is called tail-recursively and that *iterate* is applied on its result (and to the result of $\mathit{refocus}_{aux}$):

$$\begin{array}{rcl}
\mathit{evaluate} \ : \ \textsc{Term} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t, [\,])
\end{array}$$

$$\begin{array}{rcl}
\mathit{iterate} \ : \ \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{iterate}(v) & = & v \\
\mathit{iterate}(C, \mathsf{app}(\mathsf{val}(x), t)) & = & (C, \mathsf{app}(\mathsf{val}(x), t)) \\
\mathit{iterate}(C, \mathsf{app}(\mathsf{lam}(x, t), t')) & = & \mathit{refocus}(t\{t'/x\}, C) \\
\mathit{iterate}(C, \mathsf{var}(x)) & = & (C, \mathsf{var}(x))
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus} \ : \ \textsc{Term} \times \textsc{RedCont} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v, C) & = & \mathit{refocus}_{aux}(C, v) \\
\mathit{refocus}(\mathsf{app}(t, t'), C) & = & \mathit{refocus}(t, C[\mathsf{app}([\,], t')]) \\
\mathit{refocus}(\mathsf{var}(x), C) & = & \mathit{iterate}(C, \mathsf{var}(x))
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus}_{aux} \ : \ \textsc{RedCont} \times \textsc{Val} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}_{aux}([\,], v) & = & \mathit{iterate}(v) \\
\mathit{refocus}_{aux}(C[\mathsf{app}([\,], t)], v) & = & \mathit{iterate}(C, \mathsf{app}(v, t))
\end{array}$$

Proposition 14 *The pre-abstract machine and the abstract machine implement the same evaluation function.*

Proof: Immediate. □

As in Section 3.5.2, the form of the abstract machine is rather ideal because *iterate* implements the reduction rules of the reduction semantics and *refocus* and $\mathit{refocus}_{aux}$ implement its congruence rules.

4.4.3 An eval/apply abstract machine

Inlining *iterate* yields the following eval/apply abstract machine:

$$\begin{array}{rcl}
\mathit{evaluate} \ : \ \textsc{Term} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t, [\,])
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus} \ : \ \textsc{Term} \times \textsc{RedCont} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v, C) & = & \mathit{refocus}_{aux}(C, v) \\
\mathit{refocus}(\mathsf{app}(t, t'), C) & = & \mathit{refocus}(t, C[\mathsf{app}([\,], t')]) \\
\mathit{refocus}(\mathsf{var}(x), C) & = & (C, \mathsf{var}(x))
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus}_{aux} \ : \ \textsc{RedCont} \times \textsc{Val} & \rightarrow & \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}_{aux}([\,], v) & = & v \\
\mathit{refocus}_{aux}(C[\mathsf{app}([\,], t')], \mathsf{val}(x)) & = & (C, \mathsf{app}(\mathsf{val}(x), t)) \\
\mathit{refocus}_{aux}(C[\mathsf{app}([\,], t')], \mathsf{lam}(x, t)) & = & \mathit{refocus}(t\{t'/x\}, C)
\end{array}$$

Proposition 15 *The abstract machine and the eval/apply abstract machine implement the same evaluation function.*

Proof: Immediate. □

4.4.4 A push/enter abstract machine

Inlining $\mathit{refocus}_{aux}$ and obtain a push/enter abstract machine which we recognize to be (a substitution-based version of) Krivine's machine [2, 3]:

$$\begin{array}{rcl}
\mathit{evaluate} & : & \textsc{Term} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{evaluate}(t) & = & \mathit{refocus}(t, [\,])
\end{array}$$

$$\begin{array}{rcl}
\mathit{refocus} & : & \textsc{Term} \times \textsc{RedCont} \to \textsc{Val} + (\textsc{RedCont} \times \textsc{PotRedex}) \\
\mathit{refocus}(v, [\,]) & = & v \\
\mathit{refocus}(\mathsf{var}(x), C[\mathsf{app}([\,], t)]) & = & (C, \mathsf{var}(x)) \\
\mathit{refocus}(\mathsf{lam}(x, t), C[\mathsf{app}([\,], t')]) & = & \mathit{refocus}(t\{t'/x\}, C) \\
\mathit{refocus}(\mathsf{app}(t, t'), C) & = & \mathit{refocus}(t, C[\mathsf{app}([\,], t')]) \\
\mathit{refocus}(\mathsf{var}(x), C) & = & (C, \mathsf{var}(x))
\end{array}$$

Proposition 16 *The eval/apply abstract machine and the push/enter abstract machine implement the same evaluation function.*

Proof: Immediate. □

4.4.5 From the eval/apply abstract machine to a big-step semantics

As in Section 3.5.5, the eval/apply machine of Section 4.4.3 is in defunctionalized form: the reduction contexts, together with $\mathit{refocus}_{aux}$, are the first-order counterpart of a function (or, more precisely, of a continuation). The higher-order counterpart of this abstract machine is a continuation-passing evaluation function. Writing it in direct style yields an evaluation function that implements a traditional big-step operational semantics of the λ-calculus under call by name, as in Section 3.5.5. We state it below without proof and in a form where the contexts are constructed inside-out in case of stuck terms:

$$\frac{}{v \Downarrow v}$$

$$\frac{t \Downarrow \mathsf{lam}(x, t'') \quad t''\{t'/x\} \Downarrow a}{\mathsf{app}(t, t') \Downarrow a} \qquad \frac{t \Downarrow \mathsf{val}(x)}{\mathsf{app}(t, t') \Downarrow ([\,], \mathsf{app}(\mathsf{val}(x), t'))}$$

$$\frac{t \Downarrow (C, r)}{\mathsf{app}(t, t') \Downarrow (\mathsf{app}(C, t'), r)} \qquad \frac{}{\mathsf{var}(x) \Downarrow ([\,], \mathsf{var}(x))}$$

4.5 Analysis of the refocused CPS transformer

As in Section 3.6, the refocused CPS transformer also takes the form of a pre-abstract machine. We can fuse the trampoline function and the two refocus functions into an abstract machine. Again, this abstract machine is in defunctionalized form, and we can write its higher-order counterpart, obtaining a traditional one-pass CPS transformer.

4.6 Summary

We have described how to efficiently implement the evaluation function of a reduction semantics for the λ-calculus under call by name, by deforesting the intermediate terms produced in each decompose-contract-plug cycle. As a byproduct, we have shown how refocusing makes it possible to mechanically obtain an array of abstract machines—including (a substitution-based version of) Krivine's machine—out of this reduction semantics. From these abstract machines, we have exhibited the corresponding big-step operational semantics. As in Section 3, it is also simple to walk back from a big-step operational semantics to an abstract machine, from this abstract machine to a refocused interpreter, and from this refocused interpreter to a reduction semantics.

We have also shown how to refocus a quadratic-time context-based CPS transformer into a linear-time one.

5 Conclusion

We have presented a structural result about reduction semantics with context-free grammars of values, reduction contexts, and redexes, and satisfying a unique-decomposition property. These conditions are quite general: they hold for deterministic languages and also for oracle-based non-deterministic languages, as illustrated in Appendix A; they however do not hold when the next redex is searched non-deterministically.

Our structural result enables one to mechanically construct a refocus function that iteratively goes from redex site to redex site and avoids the overhead of first plugging and then decomposing in an evaluation function, as illustrated in Figure 1. The refocused evaluation function implements a state-transition function, i.e., an abstract machine. We have illustrated this property by mechanically constructing Felleisen et al.'s CK machine and a substitution-based version of Krivine's machine out of two reduction semantics of the λ-calculus. We have also mechanically turned context-based quadratic-time program transformations into program transformations that operate in one pass.

The construction of the refocus function suggests a convenient definition of the decompose function that directly connects one-step reduction and evaluation in the form of an abstract machine. It also suggests a practical method to obtain a reduction semantics out of an abstract machine.

Acknowledgments: We are grateful to Daniel Damian, Bernd Grobauer, Fritz Henglein, and Julia Lawall for commenting the initial version of this article. Thanks are also due to Mads Sig Ager, Julia Lawall, Kevin Millikin, Kristian Støvring Sørensen, and the anonymous reviewers for further comments; to Matthias Felleisen for pointing out that refocusing systematizes the construction of an abstract machine out of a reduction semantics; to Amr Sabry for encouraging us to stress the connection between *refocus* and *decompose*; to Mitchell Wand for encouraging us to stress that a refunctionalized and direct-style-transformed abstract machine implements a big-step semantics; and to Małgorzata Biernacka and Mitchell Wand for rekindling our interest in documenting this work properly.

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

A Arithmetic expressions with precedence

This section treats a comprehensive example illustrating all the cases of the construction of a *refocus* function as presented in Section 2.5. We consider arithmetic expressions with addition, multiplication, conditional expressions checking whether their first argument is zero, parenthesized expressions, literals, and oracles returning 0 or 1, nondeterministically. (The oracles are only there to illustrate a case of the construction.)

The grammar is unambiguous and hierarchic, as often given in undergraduate compiler courses. It specifies expressions, terms, and factors, and reads as follows:

$$\begin{array}{rl@{\qquad}rcl}
e \in & \textsc{Expr} & e & ::= & t + e \mid \textsc{Ifz}\ e\ e\ e \mid t \\
t \in & \textsc{Term} & t & ::= & f \times t \mid f \\
f \in & \textsc{Fact} & f & ::= & n \mid \mathsf{flip} \mid (e) \\
n \in & \textsc{Lit} & & &
\end{array}$$

A program is an expression.

The rest of this appendix is structured as follows. We first present a reduction semantics for arithmetic expressions (Section A.1), we review how to represent reduction contexts outside-in or inside-out (Section A.2), and we present a detailed implementation of the reduction semantics (Section A.3). We then apply the algorithm of Section 2.5 and construct a refocus function (Section A.4).

A.1 A reduction semantics

We define the reduction semantics of arithmetic expressions by specifying its values, its computations, its reduction contexts, its redexes and its reduction rules.

Values (trivial terms):

$$\begin{array}{rl@{\qquad}rcl}
v_e \in & \textsc{ExprVal} & v_e & ::= & v_t \\
v_t \in & \textsc{TermVal} & v_t & ::= & v_f \\
v_f \in & \textsc{FactVal} & v_f & ::= & n
\end{array}$$

Computations (serious terms):

$$\begin{array}{rl@{\qquad}rcl}
c_e \in & \textsc{ExprComp} & c_e & ::= & t + e \mid \textsc{Ifz}\ e\ e\ e \mid c_t \\
c_t \in & \textsc{TermComp} & c_t & ::= & f \times t \mid c_f \\
c_f \in & \textsc{FactComp} & c_f & ::= & \mathsf{flip} \mid (e)
\end{array}$$

Reduction contexts:

$$\begin{array}{rl@{\qquad}rcl}
E_e \in & \textsc{ExprEvCont} & E_e & ::= & [\,]_e \mid E_t + e \mid v_t + E_e \mid \textsc{Ifz}\ E_e\ e\ e \mid E_t \\
E_t \in & \textsc{TermEvCont} & E_t & ::= & [\,]_t \mid E_f \times t \mid v_f \times E_t \mid E_f \\
E_f \in & \textsc{FactEvCont} & E_f & ::= & [\,]_f \mid (E_e)
\end{array}$$

Redexes:

$$\begin{array}{rlrcl}
r_e \in & \textsc{ExprRedex} & r_e & ::= & v_t + v_e \mid \textsc{Ifz}\ v_e\ e\ e \mid r_t \\
r_t \in & \textsc{TermRedex} & r_t & ::= & v_f \times v_t \mid r_f \\
r_f \in & \textsc{FactRedex} & r_f & ::= & \mathsf{flip} \mid (v_e)
\end{array}$$

Reduction rules:

$$\begin{array}{rcll}
E_e[n_1 + n_2] & \rightarrow & E_e[n_3] & \text{where } n_3 \text{ is the sum of } n_1 \text{ and } n_2 \\
E_e[\textsc{Ifz}\ n\ e_1\ e_2] & \rightarrow & E_e[e_1] & \text{if } n = 0 \\
E_e[\textsc{Ifz}\ n\ e_1\ e_2] & \rightarrow & E_e[e_2] & \text{if } n \neq 0 \\
E_e[n_1 \times n_2] & \rightarrow & E_e[n_3] & \text{where } n_3 \text{ is the product of } n_1 \text{ and } n_2 \\
E_e[\mathsf{flip}] & \rightarrow & E_e[0] & \\
E_e[\mathsf{flip}] & \rightarrow & E_e[1] & \\
E_e[(n)] & \rightarrow & E_e[n] &
\end{array}$$

The reduction rules are defined only on decompositions that "make sense", i.e., only an expression is plugged into $[\,]_e$, only a term into $[\,]_t$ and a factor into $[\,]_f$. The result of a reduction is an expression.

This reduction semantics satisfies three unique-decomposition lemmas, i.e., decomposing an expression (resp. a term and a factor) into a reduction context and a redex yields a unique result. We prove these lemmas by structural induction on the syntax. The language is simple enough that the number of cases is manageable.

There are no stuck terms, and reduction always yields an expression. The reduction relation, however, is not a function from expressions to expressions, even though decompositions are unique, because of the oracles.

A.2 An alternative representation of contexts

In Section A.1, the grammar of reduction contexts embodies left composition, i.e., the construction of contexts by composition with an elementary context on the left. In other words, each production except for the empty context, e.g., $E_e ::= v_t + E_e$, could be written as $E_e ::= (v_t + [\,]_e) \circ E_e$. Since composition of contexts is associative, we could define the same contexts using right composition, giving a production of the form $E_e ::= E_e \circ (v_t + [\,]_e)$. We write it, however, in the more common way: $E_e ::= E_e[v_t + [\,]_e]$.

We transform the grammar of contexts into one representing composition on the right, and we do this in a completely mechanical way. The example just above shows the general idea, though it does not illustrate the case where the sub-context is not of the same type as the generated context.

We have three groups of reduction contexts, grouped by the syntactic category they produce when plugged. Take the elementary context $[\,]_t + e$. If we left-compose another reduction context with this elementary context, we require that the other context *produces* terms. If we right-compose this elementary context, however, we require the other context to *accept* expressions in the hole. To capture this restriction in the grammar, we group the productions by the type of the hole instead. So for example, the production $E_e ::= E_t + e$ is transformed into $E_t ::= E_e[[\,]_t + e]$.

In general, the transformation rewrites a production of the form $E_a ::= c(x_1, \ldots, E_b, \ldots, x_n)$ into $E_b ::= E_a[c(x_1, \ldots, [\,]_b, \ldots, x_n)]$, and it keeps the productions of empty contexts. Performing this transformation on the grammar of reduction contexts above gives the following grammar.

Reduction contexts:

$$\begin{array}{llcl}
E_e \in \textsc{ExprEvCont} & E_e & ::= & [\,]_e \mid E_e[v_t + [\,]_e] \mid E_e[\textsc{Ifz}\ [\,]_e\ e\ e] \mid E_f[([\,])] \\
E_t \in \textsc{TermEvCont} & E_t & ::= & [\,]_t \mid E_e[[\,]_t + e] \mid E_t[v_f \times [\,]_t] \mid E_e \\
E_f \in \textsc{FactEvCont} & E_f & ::= & [\,]_f \mid E_t[[\,]_f \times t] \mid E_t
\end{array}$$

In the original representation, E_e ranged over all contexts that generated expression, but made no restriction on the type of the hole. The second representation likewise lets E_e range over all contexts with a hole accepting an expression, but makes no restriction on the type of the result of a plug. Using this second representation, we must require that the reduction contexts appearing in the reduction rules must output expressions, i.e., we rule out the empty contexts from E_t and E_f in the grammar of reduction contexts. The resulting reduction contexts and reduction rules read as follows.

Reduction contexts:

$$\begin{array}{llcl}
E_e \in \textsc{ExprEvCont} & E_e & ::= & [\,]_e \mid E_e[v_t + [\,]_e] \mid E_e[\textsc{Ifz}\ [\,]_e\ e\ e] \mid E_f[([\,])] \\
E_t \in \textsc{TermEvCont} & E_t & ::= & E_e[[\,]_t + e] \mid E_t[v_f \times [\,]_t] \mid E_e \\
E_f \in \textsc{FactEvCont} & E_f & ::= & E_t[[\,]_f \times t] \mid E_t
\end{array}$$

Reduction rules:

$$\begin{array}{rcll}
E_e[n_1 + n_2] & \rightarrow & E_e[n_3] & \text{where } n_3 \text{ is the sum of } n_1 \text{ and } n_2 \\
E_e[\textsc{Ifz}\ n\ e_1\ e_2] & \rightarrow & E_e[e_1] & \text{if } n = 0 \\
E_e[\textsc{Ifz}\ n\ e_1\ e_2] & \rightarrow & E_e[e_2] & \text{if } n \neq 0 \\
E_t[n_1 \times n_2] & \rightarrow & E_t[n_3] & \text{where } n_3 \text{ is the product of } n_1 \text{ and } n_2 \\
E_f[\mathsf{flip}] & \rightarrow & E_f[0] & \\
E_f[\mathsf{flip}] & \rightarrow & E_f[1] & \\
E_f[(n)] & \rightarrow & E_f[n] &
\end{array}$$

A.3 Implementation

Figure 3 displays the BNF of arithmetic expressions in Standard ML. As usual with reduction semantics, we distinguish between values (trivial terms) and computations (serious terms). An expression (`expr`) is thus trivial (`expr_val`) or serious (`expr_comp`); a term (`term`) is trivial (`term_val`) or serious (`term_comp`); and a factor (`fact`) is trivial (`fact_val`) or serious (`fact_comp`). The only values are integers, hence values are defined with the hierarchy of types `expr_val`, `term_val`, `fact_val`, and `int`. Computations are similarly embedded, hence the hierarchy of types `expr_comp`, `term_comp`, and `fact_comp`.

Figure 4 displays the reduction contexts and Figure 5, the corresponding plug functions. Figure 6 displays the implementation of redexes and the result of decomposition, and Figure 7, the corresponding decomposition functions.

```
datatype expr = EXPR_VAL of expr_val
              | EXPR_COMP of expr_comp
 and expr_val = TERM_VAL' of term_val
and expr_comp = ADD of term * expr
              | IFZ of expr * expr * expr
              | TERM_COMP' of term_comp
     and term = TERM_VAL of term_val
              | TERM_COMP of term_comp
 and term_val = FACT_VAL' of fact_val
and term_comp = MUL of fact * term
              | FACT_COMP' of fact_comp
     and fact = FACT_VAL of fact_val
              | FACT_COMP of fact_comp
 and fact_val = INT of int
and fact_comp = FLIP
              | PARENS of expr
```

Figure 3: Abstract syntax of arithmetic expressions

```
datatype expr_evcont = EEC0
                     | EEC1 of term_val * expr_evcont
                     | EEC2 of expr * expr * expr_evcont
                     | EEC3 of fact_evcont
     and term_evcont = TEC1 of expr * expr_evcont
                     | TEC2 of fact_val * term_evcont
                     | TEC3 of expr_evcont
     and fact_evcont = FEC1 of term * term_evcont
                     | FEC2 of term_evcont
```

Figure 4: Reduction contexts for arithmetic expressions

```
(*  plug_expr : expr * expr_evcont -> expr  *)
(*  plug_term : term * term_evcont -> expr  *)
(*  plug_fact : fact * fact_evcont -> expr  *)
fun
    plug_expr (e, EEC0)
    = e
  | plug_expr (e, EEC1 (tt, eec))
    = plug_expr (EXPR_COMP (ADD (TERM_VAL tt, e)), eec)
  | plug_expr (e, EEC2 (e1, e2, eec))
    = plug_expr (EXPR_COMP (IFZ (e, e1, e2)), eec)
  | plug_expr (e, EEC3 fec)
    = plug_fact (FACT_COMP (PARENS e), fec)
and
    plug_term (t, TEC1 (e, eec))
    = plug_expr (EXPR_COMP (ADD (t, e)), eec)
  | plug_term (t, TEC2 (tf, tec))
    = plug_term (TERM_COMP (MUL (FACT_VAL tf, t)), tec)
  | plug_term (TERM_VAL tt, TEC3 eec)
    = plug_expr (EXPR_VAL (TERM_VAL' tt), eec)
  | plug_term (TERM_COMP st, TEC3 eec)
    = plug_expr (EXPR_COMP (TERM_COMP' st), eec)
and
    plug_fact (f, FEC1 (t, tec))
    = plug_term (TERM_COMP (MUL (f, t)), tec)
  | plug_fact (FACT_VAL tf, FEC2 tec)
    = plug_term (TERM_VAL (FACT_VAL' tf), tec)
  | plug_fact (FACT_COMP sf, FEC2 tec)
    = plug_term (TERM_COMP (FACT_COMP' sf), tec)
```

Figure 5: Iterative plug functions for arithmetic expressions

```
datatype expr_redex = ADD_REDEX of term_val * expr_val
                    | IFZ_REDEX of expr_val * expr * expr

datatype term_redex = MUL_REDEX of fact_val * term_val

datatype fact_redex = FLIP_REDEX
                    | PARENS_REDEX of expr_val

datatype decomposed = VALUE of expr_val
                    | EXPR_DECOMPOSITION of expr_evcont * expr_redex
                    | TERM_DECOMPOSITION of term_evcont * term_redex
                    | FACT_DECOMPOSITION of fact_evcont * fact_redex
```

Figure 6: Redexes and the result of decomposition for arithmetic expressions

```
(*  decompose_expr      : expr                   -> decomposed  *)
(*  decompose_expr_comp : expr_comp * expr_evcont -> decomposed  *)
(*  decompose_term_comp : term_comp * term_evcont -> decomposed  *)
(*  decompose_fact_comp : fact_comp * fact_evcont -> decomposed  *)
fun
    decompose_expr (EXPR_VAL te)
    = VALUE te
  | decompose_expr (EXPR_COMP se)
    = decompose_expr_comp (se, EEC0)
and
    decompose_expr_comp (ADD (TERM_VAL tt, EXPR_VAL te), eec)
    = EXPR_DECOMPOSITION (eec, ADD_REDEX (tt, te))
  | decompose_expr_comp (ADD (TERM_VAL tt, EXPR_COMP se), eec)
    = decompose_expr_comp (se, EEC1 (tt, eec))
  | decompose_expr_comp (ADD (TERM_COMP st, e), eec)
    = decompose_term_comp (st, TEC1 (e, eec))
  | decompose_expr_comp (IFZ (EXPR_VAL te, e1, e2), eec)
    = EXPR_DECOMPOSITION (eec, IFZ_REDEX (te, e1, e2))
  | decompose_expr_comp (IFZ (EXPR_COMP se, e1, e2), eec)
    = decompose_expr_comp (se, EEC2 (e1, e2, eec))
  | decompose_expr_comp (TERM_COMP' st, eec)
    = decompose_term_comp (st, TEC3 eec)
and
    decompose_term_comp (MUL (FACT_VAL tf, TERM_VAL tt), tec)
    = TERM_DECOMPOSITION (tec, MUL_REDEX (tf, tt))
  | decompose_term_comp (MUL (FACT_VAL tf, TERM_COMP st), tec)
    = decompose_term_comp (st, TEC2 (tf, tec))
  | decompose_term_comp (MUL (FACT_COMP sf, t), tec)
    = decompose_fact_comp (sf, FEC1 (t, tec))
  | decompose_term_comp (FACT_COMP' sf, tec)
    = decompose_fact_comp (sf, FEC2 tec)
and
    decompose_fact_comp (FLIP, fec)
    = FACT_DECOMPOSITION (fec, FLIP_REDEX)
  | decompose_fact_comp (PARENS (EXPR_VAL te), fec)
    = FACT_DECOMPOSITION (fec, PARENS_REDEX te)
  | decompose_fact_comp (PARENS (EXPR_COMP se), fec)
    = decompose_expr_comp (se, EEC3 fec)
```

Figure 7: Iterative decomposition of an arithmetic expression

```
(*  evaluate :       expr -> expr_val  *)
(*   iterate : decomposed -> expr_val  *)
fun
    evaluate e
    = iterate (decompose_expr e)
and
    iterate (VALUE te)
    = te
  | iterate (EXPR_DECOMPOSITION
               (eec, ADD_REDEX (FACT_VAL' (INT n1),
                                TERM_VAL' (FACT_VAL' (INT n2)))))
    = evaluate (plug_expr
                 (EXPR_VAL (TERM_VAL' (FACT_VAL' (INT (n1 + n2)))),
                  eec))
  | iterate (EXPR_DECOMPOSITION
               (eec, IFZ_REDEX (TERM_VAL' (FACT_VAL' (INT 0)), e1, e2)))
    = evaluate (plug_expr (e1, eec))
  | iterate (EXPR_DECOMPOSITION
               (eec, IFZ_REDEX (TERM_VAL' (FACT_VAL' (INT n)), e1, e2)))
    = evaluate (plug_expr (e2, eec))
  | iterate (TERM_DECOMPOSITION
               (tec, MUL_REDEX (INT n1, FACT_VAL' (INT n2))))
    = evaluate (plug_term (TERM_VAL (FACT_VAL' (INT (n1 * n2))), tec))
  | iterate (FACT_DECOMPOSITION
               (fec, FLIP_REDEX))
    = evaluate (plug_fact (FACT_VAL (INT (Oracle.flip ())), fec))
  | iterate (FACT_DECOMPOSITION
               (fec, PARENS_REDEX (TERM_VAL' (FACT_VAL' (INT n)))))
    = evaluate (plug_fact (FACT_VAL (INT n), fec))
```

Figure 8: Canonical evaluation of an arithmetic expression

Figure 8 displays the evaluation function, which attempts to decompose its input expression into a reduction context and a redex. If the expression is a value (and thus cannot be decomposed), then the result is found. Otherwise, the redex is contracted, the contractum is plugged into the context, and evaluation is iterated. Contracting a redex amounts to adding two integers, selecting between two expressions, multiplying two integers, calling an oracle (which is unspecified here; our implementation is state-based), or skipping parentheses. No terms are stuck.

A.4 Refocusing

We now construct the refocus functions. We start from the ML definition of the grammar, in Figure 3, which fits the format expected for the construction.

Let us determine the reduction sequence of each syntactic construct.

The main syntactic constructs: Both the addition and the multiplication evaluate their arguments from left to right. The conditional expression is special in that it only evaluates its first argument. The oracle has no argument. The parentheses have one argument and evaluate it. All of these syntactic constructs create redexes.

The auxiliary syntactic constructs: The coercions between terms and expressions and between factors and expressions have one argument. This argument is evaluated.

The refocus functions follow the grammatical structure of the source language. Their skeleton is thus as follows.

```
fun refocus_expr (EXPR_VAL te, eec)
    = refocus_expr_val (te, eec)
  | refocus_expr (EXPR_COMP se, eec)
    = refocus_expr_comp (se, eec)
and refocus_expr_val (te, eec)
    = ...
and refocus_expr_comp (ADD (t, e), eec)
    = ...
  | refocus_expr_comp (IFZ (e0, e1, e2), eec)
    = ...
  | refocus_expr_comp (TERM_COMP' st, eec)
    = ...
and refocus_term (TERM_VAL tt, tec)
    = refocus_term_val (tt, tec)
  | refocus_term (TERM_COMP st, tec)
    = refocus_term_comp (st, tec)
and refocus_term_val (tt, tec)
    = ...
and refocus_term_comp (MUL (f, t), tec)
    = ...
  | refocus_term_comp (FACT_COMP' sf, tec)
    = ...
and refocus_fact (FACT_VAL tf, fec)
    = refocus_fact_val (tf, fec)
  | refocus_fact (FACT_COMP sf, fec)
    = refocus_fact_comp (sf, fec)
and refocus_fact_val (tf, fec)
    = ...
and refocus_fact_comp (FLIP, fec)
    = ...
  | refocus_fact_comp (PARENS e, fec)
    = ...
```

In the rest of this section, we complete each missing case in this definition.

1. If the length of the reduction sequence of a production is zero, then the result of the production is a value or a redex.

   (a) The case of values is already taken care of because the grammar differentiates between values and computations. All functions `refocus_x` call `refocus_x_val` if given a value, and call `refocus_x_comp` if given a computation.

   (b) If the result of a production is a redex, then we have found a decomposition. Here, only the oracle fits the bill. We thus return a decomposition.

   ```
   refocus_fact_comp (FLIP, fec)
   = FACT_DECOMP (fec, FLIP_REDEX)
   ```

2. If the length of the reduction sequence is non-zero, then the first sub-expression must be evaluated.

```
      refocus_expr_comp (ADD (t, e), eec)
      = refocus_term (t, TEC1 (e, eec))
    | refocus_expr_comp (IFZ (e0, e1, e2), eec)
      = refocus_expr (e0, EEC2 (e1, e2, eec))
    | refocus_expr_comp (TERM_COMP' st, eec)
      = refocus_term_comp (st, TEC3 eec)

      refocus_term_comp (MUL (f, t), tec)
      = refocus_fact (f, FEC1 (t, tec))
    | refocus_term_comp (FACT_COMP' sf, tec)
      = refocus_fact_comp (sf, FEC2 tec)

    | refocus_fact_comp (PARENS e, fec)
      = refocus_expr (e, EEC3 fec)
```

3. Likewise, each `refocus_x_val` is defined by cases on the inner elementary reduction context.

```
      refocus_expr_val (te, EEC0)
      = ...
    | refocus_expr_val (te, EEC1 (tt, eec))
      = ...
    | refocus_expr_val (te, EEC2 (e1, e2, eec))
      = ...
    | refocus_expr_val (te, EEC3 fec)
      = ...

      refocus_term_val (tt, TEC1 (e, eec))
      = ...
    | refocus_term_val (tt, TEC2 (tf, tec))
      = ...
    | refocus_term_val (tt, TEC3 eec)
      = ...
      refocus_fact_val (tf, FEC1 (t, tec))
      = ...
    | refocus_fact_val (tf, FEC2 tec)
      = ...
```

   (a) If the length of the reduction sequence of a production is reached and the production constructs a value, we refocus on this value.

```
    | refocus_term_val (tt, TEC3 eec)
      = refocus_expr_val (TERM_VAL' tt, eec)

    | refocus_fact_val (tf, FEC2 tec)
      = refocus_term_val (FACT_VAL' tf, tec)
```

   (b) If the length of the reduction sequence of a production is reached and the production constructs a redex, we have found a decomposition.

```
| refocus_expr_val (te, EEC1 (tt, eec))
  = EXPR_DECOMP (eec, ADD_REDEX (tt, te))
| refocus_expr_val (te, EEC2 (e1, e2, eec))
  = EXPR_DECOMP (eec, IFZ_REDEX (te, e1, e2))
| refocus_expr_val (te, EEC3 fec)
  = FACT_DECOMP (fec, PARENS_REDEX te)

| refocus_term_val (tt, TEC2 (tf, tec))
  = TERM_DECOMP (tec, MUL_REDEX (tf, tt))
```

(c) If the length of the reduction sequence of a production is not reached, we refocus on the next sub-expression to be reduced.

```
refocus_term_val (tt, TEC1 (e, eec))
= refocus_expr (e, EEC1 (tt, eec))

refocus_fact_val (tf, FEC1 (t, tec))
= refocus_term (t, TEC2 (tf, tec))
```

4. Finally, we turn to the empty context:

```
refocus_expr_val (te, EEC0)
= VALUE te
```

The refocus functions are now complete (see Figures 9 and 10).

```
(*  refocus_expr      : expr      * expr_evcont -> decomposed  *)
(*  refocus_expr_val  : expr_val  * expr_evcont -> decomposed  *)
(*  refocus_expr_comp : expr_comp * expr_evcont -> decomposed  *)
(*  refocus_term      : term      * term_evcont -> decomposed  *)
(*  refocus_term_val  : term_val  * term_evcont -> decomposed  *)
(*  refocus_term_comp : term_comp * term_evcont -> decomposed  *)
(*  refocus_fact      : fact      * fact_evcont -> decomposed  *)
(*  refocus_fact_val  : fact_val  * fact_evcont -> decomposed  *)
(*  refocus_fact_comp : fact_comp * fact_evcont -> decomposed  *)
```

Figure 9: Refocusing over an arithmetic expression (signature)

```
fun refocus_expr (EXPR_VAL te, eec)
    = refocus_expr_val (te, eec)
  | refocus_expr (EXPR_COMP se, eec)
    = refocus_expr_comp (se, eec)
and refocus_expr_val (te, EEC0)
    = VALUE te
  | refocus_expr_val (te, EEC1 (tt, eec))
    = EXPR_DECOMPOSITION (eec, ADD_REDEX (tt, te))
  | refocus_expr_val (te, EEC2 (e1, e2, eec))
    = EXPR_DECOMPOSITION (eec, IFZ_REDEX (te, e1, e2))
  | refocus_expr_val (te, EEC3 fec)
    = FACT_DECOMPOSITION (fec, PARENS_REDEX te)
and refocus_expr_comp (ADD (t, e), eec)
    = refocus_term (t, TEC1 (e, eec))
  | refocus_expr_comp (IFZ (e0, e1, e2), eec)
    = refocus_expr (e0, EEC2 (e1, e2, eec))
  | refocus_expr_comp (TERM_COMP' st, eec)
    = refocus_term_comp (st, TEC3 eec)
and refocus_term (TERM_VAL tt, tec)
    = refocus_term_val (tt, tec)
  | refocus_term (TERM_COMP st, tec)
    = refocus_term_comp (st, tec)
and refocus_term_val (tt, TEC1 (e, eec))
    = refocus_expr (e, EEC1 (tt, eec))
  | refocus_term_val (tt, TEC2 (tf, tec))
    = TERM_DECOMPOSITION (tec, MUL_REDEX (tf, tt))
  | refocus_term_val (tt, TEC3 eec)
    = refocus_expr_val (TERM_VAL' tt, eec)
and refocus_term_comp (MUL (f, t), tec)
    = refocus_fact (f, FEC1 (t, tec))
  | refocus_term_comp (FACT_COMP' sf, tec)
    = refocus_fact_comp (sf, FEC2 tec)
and refocus_fact (FACT_VAL tf, fec)
    = refocus_fact_val (tf, fec)
  | refocus_fact (FACT_COMP sf, fec)
    = refocus_fact_comp (sf, fec)
and refocus_fact_val (tf, FEC1 (t, tec))
    = refocus_term (t, TEC2 (tf, tec))
  | refocus_fact_val (tf, FEC2 tec)
    = refocus_term_val (FACT_VAL' tf, tec)
and refocus_fact_comp (FLIP, fec)
    = FACT_DECOMPOSITION (fec, FLIP_REDEX)
  | refocus_fact_comp (PARENS e, fec)
    = refocus_expr (e, EEC3 fec)
```

Figure 10: Refocusing over an arithmetic expression (definition)

```
(*  evaluate : expr       -> expr_val  *)
(*   iterate : decomposed -> expr_val  *)
fun
    evaluate e
    = iterate (refocus_expr (e, EEC0))
and
    iterate (VALUE te)
    = te
  | iterate (EXPR_DECOMPOSITION
               (eec, ADD_REDEX (FACT_VAL' (INT n1),
                                TERM_VAL' (FACT_VAL' (INT n2)))))
    = iterate (refocus_expr
                 (EXPR_VAL (TERM_VAL' (FACT_VAL' (INT (n1 + n2)))), eec))
  | iterate (EXPR_DECOMPOSITION
               (eec, IFZ_REDEX (TERM_VAL' (FACT_VAL' (INT 0)), e1, e2)))
    = iterate (refocus_expr (e1, eec))
  | iterate (EXPR_DECOMPOSITION
               (eec, IFZ_REDEX (TERM_VAL' (FACT_VAL' (INT n)), e1, e2)))
    = iterate (refocus_expr (e2, eec))
  | iterate (TERM_DECOMPOSITION
               (tec, MUL_REDEX (INT n1, FACT_VAL' (INT n2))))
    = iterate (refocus_term (TERM_VAL (FACT_VAL' (INT (n1 * n2))), tec))
  | iterate (FACT_DECOMPOSITION
               (fec, FLIP_REDEX))
    = iterate (refocus_fact (FACT_VAL (INT (Oracle.flip ())), fec))
  | iterate (FACT_DECOMPOSITION
               (fec, PARENS_REDEX (TERM_VAL' (FACT_VAL' (INT n)))))
    = iterate (refocus_fact (FACT_VAL (INT n), fec))
```

Figure 11: Refocused evaluation of an arithmetic expression

The refocused evaluation function is displayed in Figure 11. Rather than decomposing the result of the plug functions, as in Figure 8, `iterate` refocuses each contractum and iterates. Fusing `iterate` and each of the refocus functions yields an abstract machine for arithmetic expressions.

A.5 Summary

We have considered arithmetic expressions with precedence and we have specified their meaning with a reduction semantics. We have then applied the algorithm of Section 2.5 to write refocusing functions that map a reduction context and a syntactic entity into either a value or a decomposition into a reduction context and a potential redex. The construction suggests an alternative definition of decomposition:

```
fun decompose_expr_alt e
    = refocus_expr (e, EEC0)
```

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.

[2] Pierre Crégut. An abstract machine for lambda-terms normalization. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[3] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[4] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[5] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[6] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, Technical report 545, Computer Science Department, Indiana University, pages 35–39, London, England, January 2001. Accepted for publication in Information Processing Letters (2004).

[7] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[8] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[9] Olivier Danvy and Lasse R. Nielsen. On one-pass CPS transformations. Technical Report BRICS RS-02-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. Accepted for publication in the Journal of Functional Programming.

[10] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[11] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[12] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[13] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[14] 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*, pages 18–27, Paris, France, September 1999. ACM Press.

[15] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. The MIT Press, 1996.

[16] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[17] Gilles Kahn. Natural semantics. Technical Report 601, INRIA, Sophia Antipolis, France, February 1987.

[18] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In Kathleen Fischer, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 4–15, Snowbird, Utah, September 2004. ACM Press.

[19] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992.

[20] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[22] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[23] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[24] Peter Sestoft. Demonstrating lambda calculus reduction. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 420–435. Springer-Verlag, 2002.

[25] Joseph Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[26] Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[27] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[28] Yong Xiao, Zena M. Ariola, and Michel Mauny. From syntactic theories to interpreters: A specification language and its compilation. In Nachum Dershowitz and Claude Kirchner, editors, *Informal proceedings of the First International Workshop on Rule-Based Programming (RULE 2000)*, Montréal, Canada, September 2000. Available online at `http://www.loria.fr/~ckirchne/=rule2000/proceedings/`.

[29] 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.

Recent BRICS Report Series Publications