Basic Research in Computer Science

Syntactic Theories in Practice

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series **RS-01-31**

ISSN 0909-0878 **July 2001**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/01/31/`

Syntactic Theories in Practice *

Olivier Danvy and Lasse R. Nielsen

BRICS †
Department of Computer Science
University of Aarhus ‡

July, 2001

Abstract

The evaluation function of a syntactic theory is canonically defined as the transitive closure of (1) decomposing a program into an evaluation context and a redex, (2) contracting this redex, and (3) plugging the result in the context. Directly implementing this evaluation function therefore yields an interpreter with a quadratic time factor over its input. We present sufficient conditions over a syntactic theory to circumvent this quadratic factor, and we illustrate the method with two programming-language interpreters and a transformation into continuation-passing style (CPS). As a byproduct, the time complexity of this CPS transformation is mechanically changed from quadratic to linear.

We also flesh out a new connection between continuations and evaluation contexts, using Reynolds's defunctionalization.

*Extended version of an article to appear in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Firenze, Italy, September 4, 2001.

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail: {`danvy,lrn`}`@brics.dk`

Contents

List of Figures

1 Introduction

A syntactic theory provides a uniform, concise, and elegant framework to specify a programming language and to reason about it [4]. We consider the issue of implementing the evaluation function of a syntactic theory in the form of an interpreter. Our emphasis, however, is not on automating this process, as in Xiao and Ariola's SL project [12, 13].[1] Instead, we show how to circumvent the quadratic time factor over source programs entailed by a direct implementation of an evaluation function.

We first identify the quadratic factor (Section 2). Then, we list sufficient conditions to rephrase a syntactic theory so that implementing its evaluation function does not incur this factor. The proof that these conditions are sufficient is constructive in that it indicates how to mechanically rephrase the syntactic theory to circumvent the quadratic factor. We then consider three examples: a syntactic theory for the call-by-value λ-calculus (Section 3), a syntactic theory for arithmetic expressions with precedence (Section 4), and a transformation of λ-terms into continuation-passing style due to Sabry and Felleisen [9] (Section 5).

2 Refocusing in a syntactic theory

A syntactic theory is a small-step semantics where evaluation is defined as the transitive closure of single reductions, each performed by (1) decomposing a program into a context and a redex, (2) contracting the redex, and (3) plugging the result of contraction in the context. Most syntactic theories satisfy a *unique decomposition property*.

The interpreter for a syntactic theory corresponding to its evaluation function naturally consists of a decompose-contract-plug loop. Often, the only viable implementation of decomposition is a depth-first search in the abstract syntax tree. The decompose step therefore introduces a significant overhead, proportional to the program size. Likewise, plugging can also take time linear in the program size, although it always takes at most as long as the following decomposition, if there is one, and as illustrated below.

For example, here is a syntactic theory of the call-by-value λ-calculus:

$$\begin{array}{lcllcl} e & \in & \Lambda & e & ::= & x \mid \lambda x.e \mid e\ e \\ v & \in & \mathit{Values} & v & ::= & x \mid \lambda x.e \\ x & \in & \mathit{Vars} & & & \\ E & \in & \mathit{EvCont} & E & ::= & [\,] \mid E\ e \mid v\ E \\ r & \in & \mathit{Redex} & r & ::= & v\ v \end{array}$$

Plugging the hole of an evaluation context with an expression is defined as usual:

$$\begin{array}{rcl} ([\,])[e] & = & e \\ (E\ e')[e] & = & E[e]\ e' \\ (v\ E)[e] & = & v\ E[e] \end{array}$$

[1] `http://www.cs.uoregon.edu/~ariola/SL/`

The only reduction rule is the following one:

$$E[(\lambda x.e)\ v] \to E[e[v/x]]$$

Decomposition induces a linear overhead and thus evaluation takes a quadratic time. Let us illustrate this complexity using Church numerals: $\lceil n \rceil$ is the Church numeral for the number n, i.e., $\lambda z.\lambda s.\, \underbrace{s(s(s(\ldots\,(s\,z)\,\ldots)))}_{n}$.

Example 1 *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, always* $(\lambda x.x)\, v$*, takes time proportional to the number of remaining applications. The total evaluation time is thus* $O(n^2)$. □

Example 2 *More generally, let us consider the term*

$$\underbrace{(\lambda x.x)((\lambda x.x)((\lambda x.x)(\ldots\,((\lambda x.x)}_{n}\, e)\,\ldots)))$$

where e *reduces to* v *in* m *steps. The time complexity of reducing this term to a value is at least* $O(m \times n + n^2)$*. Indeed, the factor* n *is attached to each of the* m *reduction steps and thus the time complexity for reducing* e *to* v *in this context is at least* $O(m \times n)$*, after which we are back at the previous example.* □

We propose an alternative implementation of consecutive plug-and-decompose operations that avoids the quadratic overhead. In this alternative implementation, the composition of plug and decompose is replaced by a single function that we call *refocus*. This replacement is only possible if the syntactic theory satisfies some properties that essentially amount to the next redex occurring, in the depth-first traversal of decompose, later than any other expression that can occur in an evaluation context. We show that these properties hold if the syntactic theory is given in the "standard" way, i.e., by a context-free grammar of values and evaluation contexts, and if it satisfies a unique-decomposition property. We also show how to construct a refocus function that avoids the quadratic overhead.

2.1 Context-free syntactic theories (terms)

First, we can assume some properties of the grammar of the language. We are working with abstract syntax, i.e., a program is an abstract-syntax *tree* where each node is created by a production in the language grammar. Because the abstract syntax need not correspond to the concrete syntax, we can, without loss of generality, assume (1) that all productions are of the form

$$e ::= c(e_1, \ldots, e_n)$$

for some terminal symbol c and non-terminals $e_1, \ldots, e_n$, and (2) that there is only one production using c. We call the terminal symbols, c, *constructors* and the non-terminals, e, *term identifiers*. The set of terms generated from each non-terminal of the grammar is associated to this non-terminal, and the non-terminal is used to refer to any element of that set. When we refer to a production on the form $e ::= c(e_1, \ldots, e_n)$, the set associated to e is called $\textsc{Exp}$ and the ones ranged over by e_1, ..., and e_n are called $\textsc{Exp}_1$, ..., and $\textsc{Exp}_n$, respectively.

We assume that there is no trivial syntactic category, i.e., one where $\textsc{Exp}_i$ contains only zero or one element. There are standard ways to transform any grammar into an equivalent grammar with no trivial syntactic categories [5, Section 4.4].

We also require the syntactic theory to be defined by context-free grammars of values, evaluation contexts, and redexes. (Xiao, Sabry, and Ariola also make this assumption for terms and evaluation contexts [13].)

2.2 Context-free syntactic theories (values)

If $e ::= c(e_1, \ldots, e_n)$ is a production in the language grammar, a production for values is thus of the form

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

where the x_i are either term identifiers, e_i, or non-terminals that, like v, represent values. We call such non-terminals *value identifiers*. The set of value terms for a value identifier v_i is called $\textsc{Val}_i$ and is necessarily a subset of $\textsc{Exp}_i$.

2.3 Context-free syntactic theories (contexts)

Likewise, evaluation contexts are given by a grammar of the form:

$$E ::= [\,] \mid c(x_1, \ldots, x_{i-1}, E_i, x_{i+1} \ldots, x_n) \mid \ldots$$

where again the x_j's are either value- or term identifiers, and E_i and E are non-terminals representing evaluation contexts. We call such non-terminals *context identifiers*. The terminal $[\,]$ is called the *hole* of a context.

The binary operator $\circ$ constructs the composition of two contexts. It is defined inductively on the structure of its first argument as follows.

$$\begin{aligned} [\,] \circ E_2 &= E_2 \\ c(e_1, \ldots, E, \ldots, e_n) \circ E_2 &= c(e_1, \ldots, E \circ E_2, \ldots, e) \end{aligned}$$

Composition thus satisfies $(E_1 \circ E_2)[e] = E_1[E_2[e]]$.

Contexts with composition form a monoid where the empty context, $[\,]$, is the unit, and all other evaluation contexts can be constructed by composing elementary contexts, i.e., contexts where the immediate sub-context is a hole, e.g., $c(x_1, \ldots, x_{i-1}, [\,], x_{i+1}, \ldots, x_n)$.

Because composition is associative, we can define evaluation contexts corresponding to composition on the left or on the right. For example, at the

beginning of Section 2, we gave a traditional definition of *EvCont* where evaluation contexts are created by composition on the left. We can also specify *EvCont* so that evaluation contexts are created by composition on the right:

$$E \in \mathit{EvCont} \qquad E ::= [\,] \mid E[[\,]\ e] \mid E[v\ [\,]]$$

Plugging the hole of an evaluation context with an expression is then defined iteratively as follows:

$$\begin{array}{rcl} ([\,])[e] & = & e \\ (E[[\,]\ e'])[e] & = & E[e\ e'] \\ (E[v\ [\,]])[e] & = & E[v\ e] \end{array}$$

In the sense that an evaluation context represents a function from terms to terms, it is injective, i.e., if $E[e] = E[e']$ then $e = e'$. This injectivity can be proven by structural induction over E.

We define the *depth* of an evaluation context E as the number of productions used to create it. We write it $|E|$, and define the depth function $|\cdot|$ inductively over the structure of evaluation contexts as follows.

$$\begin{array}{rcl} |[\,]| & = & 0 \\ |c(e_1, \ldots, E, \ldots, e_n)| & = & 1 + |E| \end{array}$$

One can then easily show that $|E_1 \circ E_2| = |E_1| + |E_2|$ and $|E| = 0 \Leftrightarrow E = [\,]$.

We also define a partial ordering on evaluation contexts based on the composition operation: $E_1 \leq E_2$ if there exists an E' such that $E_2 = E_1 \circ E'$. This relation is reflexive, anti-symmetric, and transitive (simple proof omitted).

If two evaluation contexts are both smaller than a third one, then the two are themselves related. This follows from the structure of the grammar of evaluation contexts, and therefore we can uniquely define the greatest lower bound of any set of contexts with respect to the ordering. We write $E_1 \sqcap E_2$ for the binary greatest lower bound of E_1 and E_2.

If E_1 is strictly smaller than E_2, i.e., if $E_1 \leq E_2$ and $E_1 \neq E_2$, we use the traditional notation $E_1 < E_2$. This strict ordering is well-founded, since the $|\cdot|$ function maps the evaluation contexts into the natural numbers ordered by size and it is monotone with respect to $<$.

Some syntactic categories contain only values, e.g., the syntactic categories of literals. We assume that there are no evaluation contexts for those syntactic categories, since such evaluation contexts could never occur in a decomposition into context and redex anyway, and as such they are irrelevant to the semantics of a language. Likewise, there is no reason to distinguish between values and expressions, so we only represent the syntactic category by the term identifier and never by the associated value identifier.

2.4 Context-free syntactic theories (redexes)

We require redexes to be defined by a context-free grammar using only constructors, term identifiers, and value identifiers. More precisely, productions must be of the form $r ::= c(x_1, \ldots, x_n)$ where x_i is either v_i or e_i. The set of redexes

ranged over by r_i is called $\textsc{Redex}_i$. It is a subset of $\textsc{Exp}_i$. We require $\textsc{Redex}_i$ to be disjoint from the set of values, $\textsc{Val}_i$.

Some syntactic theories have more than these groups of terms and include groups of terms that are considered errors or that are stuck. With an abuse of language, we group all these terms under the name "redex". Indeed, we are primarily interested in the plugging and decomposition taking place between reductions, independently of what the contractions do, inasmuch as we can distinguish values from non-value expressions.

2.5 Properties of syntactic theories

We require one property of the syntactic theory, that of unique decomposition.

Definition 1 (Unique decomposition) *A syntactic theory satisfies a* unique-decomposition property *if any term, e, can be uniquely decomposed into either a value, if e itself is a value, or an evaluation context E and a redex r such that $e = E[r]$.* □

Unique decomposition is so fundamental to syntactic theories for deterministic languages that it is almost always the first property to be established. Its proof is 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 an automated support for proving unique-decomposition properties [13].

If the syntactic theory satisfies a unique-decomposition property, then its redexes are exactly the non-value terms that can only be trivially decomposed, as shown by the following two inclusions.

- Let r be a redex. If $E[e]$ is a decomposition of r then either e is a value, and the decomposition is trivial, or e is not a value, and it can be decomposed uniquely into $e = E'[r']$. Then $(E \circ E')[r']$ is a decomposition of r into a context and a redex. Since $[\,][r]$ is also a decomposition into an evaluation context and a redex, and it is unique, we know that $E \circ E' = [\,]$ and $r' = r$. The only way $E \circ E'$ can be $[\,]$ is if both E and E' are $[\,]$, and so the decomposition was trivial.
- Let e be a non-value term that can only be trivially decomposed. Then by unique decomposition e can be uniquely decomposed as $E[r]$. Since this decomposition must be trivial, and r is a redex and thus not a value, it follows that $E = [\,]$ and thus $e = r$ is itself a redex.

The following property of syntactic theories with unique decomposition allows us to show the correctness of the refocus function.

Definition 2 (Left-to-right reduction sequence) *A syntactic theory has a* left-to-right reduction sequence *if for each production $e ::= c(e_1, \ldots, e_n)$ of the language grammar, there exists a number $0 \leq m \leq n$, called the* length of the reduction sequence of c, *such that if $e = c(e_1, \ldots, e_n)$ then the following two properties hold.*

1. *If all of e_1 through e_m are values then e is either a value or a redex, depending only on c.*

2. *If $1 \leq i < m$ is the first i such that e_i is not a value, with a (unique) decomposition of $e_i = E[r]$, then the decomposition of e is*

$$c(e_1, \ldots, e_{i-1}, E, e_{i+1}, \ldots, e_n)[r].$$

Depending on whether e is a value or a redex in the first property, we say that c constructs a value *or c* constructs a redex*, respectively.*

In words, the length of a reduction sequence is the number of immediate sub-expressions that must be reduced to produce a value or a redex. □

Not all syntactic theories have such a left-to-right reduction sequence, but any syntactic theory with unique decomposition can be transformed into one that does by only reordering the arguments of the constructors

We show that there is a reduction sequence, not necessarily left-to-right, for any production of the language grammar.

Definition 3 (Reduction sequence) *A syntactic theory is said to have a* reduction sequence *if for each production $e ::= c(e_1, \ldots, e_n)$ of the language grammar, there exists a number $0 \leq m \leq n$, called the* length of the reduction sequence of c*, and a sequence of length m of indices between 1 and n, represented by an injective mapping $\sigma : \{1, \ldots, m\} \to \{1, \ldots, n\}$ (called the reduction sequence of c), such that if $e = c(e_1, \ldots, e_n)$ then the following two properties hold.*

1. *If all of $e_{\sigma(1)}$ through $e_{\sigma(m)}$ are values then e is either a value or a redex, depending only on c.*

2. *If $1 \leq i < m$ is the first i such that $e_{\sigma(i)}$ is not a value, with a (unique) decomposition of $e_{\sigma(i)} = E[r]$, then the decomposition of e is*

$$c(e_1, \ldots, e_{\sigma(i)-1}, E, e_{\sigma(i)+1}, \ldots, e_n)[r].$$

A left-to-right reduction sequence is a reduction sequence where the σ-function is the identity.

Theorem 1 *A syntactic theory that satisfies "unique decomposition" has a reduction sequence.*

Proof: Let $e ::= c(e_1, \ldots, e_n)$ be a production of the grammar of the language.

The proof constructs the reduction sequence (m, σ). First we formalize and name the two properties above.

$$P_1(m,\sigma) = \begin{cases} \forall e_1,\ldots,e_n. \\ \quad (\forall i \in \{1,\ldots,m\}\,.e_{\sigma(i)} \in \textsc{Val}_{\sigma(i)}) \Rightarrow c(e_1,\ldots,e_n) \in \textsc{Val} \cup \textsc{Redex} \end{cases}$$

$$P_2(m,\sigma) = \begin{cases} \forall e_1,\ldots,e_n. \forall i \in \{1,\ldots,m\}\,. \\ \quad ((\forall j \in \{1,\ldots,i-1\}\,.e_{\sigma(j)} \in \textsc{Val}_{\sigma(j)}) \wedge e_{\sigma(i)} \in \textsc{Exp}_{\sigma(i)} \setminus \textsc{Val}_{\sigma(i)} \\ \quad\quad \wedge\ \mathit{decompose}(e_{\sigma(i)}) = (E,r)) \\ \quad \Rightarrow \mathit{decompose}(c(e_1,\ldots,e_n)) = (c(e_1,\ldots,e_{\sigma(i)-1},E,e_{\sigma(i)+1},\ldots,e_n),r) \end{cases}$$

We show three facts about these properties:

Fact 1: $P_2(0,\sigma)$**.** It holds vacuously. The σ must be the empty function, since it is defined over the empty set.

Fact 2: $P_2(m,\sigma) \Rightarrow \ P_1(m,\sigma)$
$\vee\ \exists\sigma' : \{1,\ldots,m+1\} \overset{inj}{\to} \{1,\ldots,n\}\,.P_2(m+1,\sigma')$.

Three cases occur.

Assume $P_2(m,\sigma)$. Let e_1 through e_n be given such that $e_{\sigma(1)}$ through $e_{\sigma(m)}$ are values and the remaining are not (if at all possible, some syntactic categories might contain only values).

Then either $c(e_1,\ldots,e_n)$ is a value or not.

- If $c(e_1,\ldots,e_n)$ is a value, then there must be some production $v ::= c(x_1,\ldots,x_n)$ such that x_i is not a value identifier, v_i, unless i is in the range of σ (if e_i is a value only because it comes from a syntactic category that contains only values, then x_i is assumed to be a term identifier). In that case, any choice of e_1 through e_n where $e_{\sigma(1)}\ldots e_{\sigma(m)}$ are values must also be a value, so $P_1(m,\sigma)$ must hold and we say that c *constructs a value*.
- If $c(e_1,\ldots,e_n)$ is not a value, then it has a unique decomposition into evaluation context and redex. Let $(E,r) = \mathit{decompose}(c(e_1,\ldots,e_n))$ be that decomposition. Then either E is the empty context or not.
  - If E is the empty context, then $c(e_1,\ldots,e_2)$ is itself a redex. That means that there is a production for redexes on the form $r ::= c(x_1,\ldots,x_n)$ that matches this choice of values and non-values (x_i being a value identifier only if i is in the range of σ). In that case, any choice of e_1 through e_n where $e_{\sigma(1)}\ldots e_{\sigma(m)}$ are values must also be a redex, so $P_1(m,\sigma)$ must hold and we say that c *constructs a redex*.
  - If E is not the empty context, then E must be of the form

$$c(x_1,\ldots,x_{i-1},E_i,x_{i+1},\ldots,x_n)$$

There must then be a production $E ::= c(x_1, \ldots, x_{i-1}, E_i, x_{i+1}, \ldots, x_n)$ where x_j is only a value identifier if j is in the range of σ. In that case, for any choice of e_1 through e_n with $e_{\sigma(1)} \ldots e_{\sigma(m)}$ being values and $e_i = E_i[r]$ being a non value, $c(e_1, \ldots, e_n)$ will also be decomposable into $c(e_1, \ldots, e_{i-1}, E_i, e_{i+1}, \ldots, e_n)$ and r. If we extend σ with $(m+1) \mapsto i$, it must then satisfy $P_2(m+1, \sigma[m+1 \mapsto i])$.

Fact 3: $P_1(n, \sigma)$**.** If $\sigma : \{1, \ldots, n\} \xrightarrow{inj} \{1, \ldots, n\}$ then σ is a permutation. Assuming that all of $e_{\sigma(1)}$ through $e_{\sigma(n)}$ are values is then the same as assuming all of e_1 through e_n are values. Then $c(e_1, \ldots, e_n)$ can have no non-trivial decomposition, so it is either a value or a redex.

If it is a value, then there must be a production $v ::= c(x_1, \ldots, x_n)$, which means that any choice of values, $e_1 \ldots e_n$ will make $c(e_1, \ldots, e_n)$ a value, and c constructs a value.

If $c(e_1, \ldots, e_n)$ is a redex, then there must be a production on the form $r ::= c(x_1, \ldots, x_n)$, so any choice of values $e_1 \ldots e_n$ will also make $c(e_1, \ldots, e_n)$ a redex, and c constructs a redex.

By repeating Fact 2 up to n times, starting with Fact 1, we can see that there must be a reduction sequence of some length, (m, σ), satisfying $P_1(m, \sigma)$ and $P_2(m, \sigma)$, and indeed the proof suggests how to build such a σ. Thus, any syntactic construct in the language has a reduction sequence. The definition guarantees that a reduction sequence is unique, since the existence of a longer reduction sequence requires an expression to both be a value or redex and to be decomposable into a non-trivial evaluation context and a redex. □

In the remainder of this section, we assume that the subterms of the constructors are ordered according to the reduction sequence, i.e., the reduction sequence is left-to-right in the abstract syntax. This is only for ease of representation, since it lets us abstract away the mapping σ. Any argument based on left-to-right reduction sequences still hold in the more general case when σ is inserted in the appropriate places.

Also, for ease of reference, we subscript the evaluation-context constructors by the index of the argument that is an evaluation context. That is, $E ::= c_i(v_1, \ldots, v_{i-1}, E_i, e_{i+1}, \ldots, e_n)$, since all evaluation contexts that are used must be of this form.

2.6 Construction of a refocus function

We now define a function, *refocus*, that is extensionally equivalent to the composition of the *plug* function and the *decompose* function. It uses a different representation of evaluation contexts—a stack of elementary contexts—that allows us to inexpensively compose an elementary context on the right of a context, i.e., to plug an elementary context in a context. Since the evaluation context is only ever accessed in *plug* and *decompose*, we are free to choose our own representation when implementing these functions.

The refocus function is defined with two mutually recursive functions. The first, *refocus*, takes an evaluation context—represented as a stack of elementary evaluation contexts—and an expression. It then tries to find the redex in that expression by decomposing the expression. If it fails, then the expression must be a value, and it calls an auxiliary function, $refocus_{aux}$, defined by cases on the top-most evaluation context on the stack.

For each $e ::= c(e_1, \ldots, e_n)$ in the language grammar, there exists one corresponding rule in *refocus*.

1. If the length of the reduction sequence of c is 0, then we know that $c(e_1, \ldots, e_n)$ is a value or a redex.

   (a) If c constructs a value, then

   $$refocus(c(e_1, \ldots, e_n), E) = refocus_{aux}(E, c(e_1, \ldots, e_n)).$$

   (b) If instead c constructs a redex, then

   $$refocus(c(e_1, \ldots, e_n), E) = (E, c(e_1, \ldots, e_n))$$

   since we have found the decomposition.

2. If the length of the reduction sequence is non-zero, then the first expression must be reduced, and we simply refocus on it:

   $$refocus(c(e_1, \ldots, e_n), E) = refocus(e_1, E \circ c([\,], e_2, \ldots, e_n))$$

   where we write $E \circ c(\ldots, [\,], \ldots)$ for "pushing" the elementary context, $c(\ldots, [\,], \ldots)$, on the "stack" of contexts, E.

3. Likewise, the $refocus_{aux}$ function is defined by cases on the evaluation context on top of the stack. Let us take, e.g., the evaluation context $c_i(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)$ as the top-most elementary context on the stack.

   (a) If the length of the reduction sequence of c is i and c constructs a value, then

   $$\begin{aligned} &refocus_{aux}(E \circ c_i(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n), v_i) \\ &= refocus_{aux}(E, c(v_1, \ldots, v_{i-1}, v_i, e_{i+1}, \ldots, e_n)) \end{aligned}$$

   The auxiliary function tries to find the next expression in the reduction sequence by plugging the value given and picking the next subexpression in the reduction sequence. In this case there is no next subexpression, so $refocus_{aux}$ iterates with the newly constructed value.

   (b) If the length of the reduction sequence of c is i, but c constructs a redex, then we have found a decomposition. Thus, the rule is:

   $$\begin{aligned} &refocus_{aux}(E \circ c_i(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n), v_i) \\ &= (E, c(v_1, \ldots, v_{i-1}, v_i, e_{i+1}, \ldots, e_n)). \end{aligned}$$

(c) If the length of the reduction sequence of c is bigger than i, then we are not finished evaluating the c-expression, so we refocus on the next subexpression in the reduction sequence, and the rule is:

$$\begin{aligned} &\mathit{refocus}_{aux}(E \circ c_i(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, e_{i+2}, \ldots, e_n), v_i) \\ &= \mathit{refocus}(e_{i+1}, E \circ c_{i+1}(v_1, \ldots, v_{i-1}, v_i, [\,], e_{i+2}, \ldots, e_n)). \end{aligned}$$

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

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

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

With one such rule for each construction in the language syntax and in the evaluation-context syntax, the refocus function terminates only when finding a decomposition into an evaluation context and a redex or if the entire program is found to be a value. An ordering argument using the reduction sequence shows that *refocus* visits expressions in an order corresponding to a depth-first post-order traversal of the syntax tree, skipping the branches that are not in position to be evaluated (such as the conditional branches of an "if" construct). As such, *refocus* necessarily terminates, and it does so exactly with a decomposition, which has to be unique. We define this ordering as follows.

Definition 4 (Ordering on decompositions of a term) *Let $e = E_1[e_1] = E_2[e_2]$ be decompositions. There are two cases where $(E_1, e_1) \sqsubseteq (E_2, e_2)$.*

- *$(E_1, e_1) \sqsubseteq (E_2, e_2)$ if $E_1 \leq E_2$ (i.e., if $E_2 = E_1 \circ E'$ for some E'). Two equivalent ways of stating the requirement is that $E_1 \sqcap E_2 = E_1$ or that e_2 is a subterm of e_1. When $E_1 \leq E_2$ we write $E_2 \setminus E_1$ for the E' satisfying $E_2 = E_1 \circ E'$. This E' is in fact uniquely determined by E_1 and E_2.*

- *If neither $E_1 < E_2$ nor $E_2 < E_1$ then $E = E_1 \sqcap E_2$ is strictly smaller than both E_1 and E_2. Let us look at where the evaluation contexts differ; let $E_1' = E_1 \setminus E$ and $E_2' = E_2 \setminus E$.*

  Since E_1' and E_2' are both non-empty and $E_1'[e_1] = E_2'[e_2]$, there must be some constructor c such that they are of the form $E_1 \setminus E = c_i(e_1, \ldots, e_{i-1}, E_i, e_{i+1}, \ldots, e_n)$ and $E_2 \setminus E = c_j(e_1, \ldots, e_{j-1}, E_j, e_{j+1}, \ldots, e_n)$. The i and j (which we will call the index of the sub-context) must be different, since otherwise $E \circ (c_i(v_1, \ldots, E_i, \ldots, e_n))$ would be a lower bound of both E_1 and E_2, contradicting that E is the greatest lower bound.

  Then $(E_1, e_1) \sqsubseteq (E_2, e_2)$ if $i > j$, i.e., if e_1 is "later in the reduction sequence" than e_2.

No decompositions are related by $\sqsubseteq$ unless they satisfy one of these two cases. □

The relation $\sqsubseteq$ is reflexive, symmetric, and transitive (proof omitted), i.e., it is an ordering relation. It is also total, i.e., for any two decompositions of the same term, $E_1[e_1] = E_2[e_2]$, either $(E_1, e_1) \sqsubseteq (E_2, e_2)$ or $(E_2, e_2) \sqsubseteq (E_1, e_1)$. This follows from any two decompositions, $E_1[e_1] = E_2[e_2]$, satisfying at least one of the following four cases (the third one is compound):

1. $E_1 \sqcap E_2 = E_1$, in which case $E_1 < E_2$ and thus $(E_1, e_1) < (E_2, e_2)$ (by the first case of Definition 4),

2. $E_1 \sqcap E_2 = E_2$, in which case $E_2 < E_1$ and thus $(E_2, e_2) < (E_1, e_1)$ (by the first case of Definition 4), or

3. $E_1 \sqcap E_2$ is strictly smaller than both E_1 an E_2, so by the second case of Definition 4, either $(E_1, e_1) \sqsubseteq (E_2, e_2)$ or $(E_2, e_2) \sqsubseteq (E_1, e_1)$.

We define the strict ordering, $\sqsubset$, as $(E_1, e_1) \sqsubset (E_2, e_2)$ if $(E_1, e_1) \sqsubseteq (E_2, e_2)$ and $(E_1, e_1) \neq (E_2, e_2)$. Then $\sqsubset$ is well-founded, since the number of decompositions of a term is finite.

Now we are in position to prove that *refocus* terminates. The proof shows that consecutive calls to $\mathit{refocus}_{aux}$ happen on smaller and smaller decompositions, so such calls must eventually terminate.

Lemma 1 (Totality) *Any call to refocus terminates, i.e., refocus is total.*

Proof: We prove that $\mathit{refocus}_{aux}$ is total.

The proof is by well-founded induction on the arguments of $\mathit{refocus}_{aux}$.

There are four cases, one base case and one case for each possible behavior of $\mathit{refocus}_{aux}$ on an argument $(E \circ c_i(e_1, \ldots, [\,], \ldots, e_n), v)$.

1. If the argument to $\mathit{refocus}_{aux}$ is an empty context and a value, then it stops immediately with that value as result.

2. If the reduction sequence of c has length $m = i$ and c constructs a value, then
$$\begin{aligned}&\mathit{refocus}_{aux}(E \circ c_i(e_1, \ldots, [\,], \ldots, e_n), v)\\ &= \mathit{refocus}_{aux}(E, c(e_1, \ldots, v, \ldots, e_n)).\end{aligned}$$
Since $E < E \circ c_i(e_1, \ldots, [\,], \ldots, e_n)$, the first case of the definition of $\sqsubseteq$ shows that
$$(E, c(e_1, \ldots, v, \ldots, e_n)) \sqsubset (E \circ c_i(e_1, \ldots, [\,], \ldots, e_n), v).$$

3. If the reduction sequence of c has length $m = i$ and c constructs redexes, then $\mathit{refocus}_{aux}$ stops with a decomposition into an evaluation context and a redex.

4. If the reduction sequence of c has length $m > i$ then
$$\begin{aligned}&\mathit{refocus}_{aux}(E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)\\ &= \mathit{refocus}(e_{i+1}, E \circ c_1(e_1, \ldots, v, [\,], \ldots, e_n)).\end{aligned}$$

NB: $(E \circ c_1(e_1, \ldots, v, [\,], \ldots, e_n), e_{i+1}) \sqsubset (E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)$.

Here we need a sub-induction to show that $\mathit{refocus}(e_{i+1}, E \circ c(e_1, \ldots, v, [\,], \ldots, e_n))$ eventually either computes a value, or calls $\mathit{refocus}_{aux}$ with an argument that is smaller than $(E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)$.

The proof is by structural induction on e_{i+1}. We show that for any argument, (e, E), to *refocus*, either the above happens, or *refocus* calls itself with an argument that is still smaller (wrt. $\sqsubset$) than $(E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)$ and structurally smaller than e, guaranteeing eventual termination. The three cases are as follows.

- Either *refocus* calls $\mathit{refocus}_{aux}$ with the same arguments that it received itself (in the opposite order). These arguments are smaller than $(E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)$ by induction hypothesis.
- *refocus* terminates with a result that is an evaluation context and a redex.
- *refocus* calls itself with an argument that is structurally smaller, and which is also a smaller decomposition than $(E \circ c_i(e_1, \ldots, [\,], e_{i+1}, \ldots, e_n), v)$.

  It suffices to show that if $(E_1, e_1) \sqsubset (E', e')$ and $E_1 \not\leq E'$ (i.e., it is due to the second case in the definition of $\sqsubseteq$), then any other decomposition (E_2, e_2) with $E_1 < E_2$ also satisfies $(E_2, e_2) \sqsubset (E', e')$. This follows directly from the definition, since $E' \sqcap E_1 = E' \sqcap E_2$, and thus $E_1 \setminus E' \sqcap E_1 < E_2 \setminus E' \sqcap E_2$.

The above induction argument proves that $\mathit{refocus}_{aux}$ is a total function, and another induction, similar to the sub-induction in Case 3, shows that *refocus* is total as well. □

Since *refocus* can only terminate yielding a decomposition into either a value or an evaluation context and a redex, and such a decomposition is unique, it must compute the same function as the composition of plugging and decomposing.

We make only a short argument that using *refocus* leads to a more efficient implementation of finding the next redex than plugging and then decomposing using a depth-first search for the redex. The refocus function visits the nodes of the syntax tree in the same order as a recursive-descent decomposition. However, it does not start from scratch, but from the node that is about to be plugged.[2] In an implementation of a syntactic theory that uses *refocus*, the time taken to perform a reduction sequence like $E[e] \to E[e'] \to E[e'']$ is thus independent of E. Using *refocus* thus makes it possible to avoid the quadratic overhead identified at the start of this section.

[2] In fact, as we observed elsewhere [3], the refocus function corresponds to a continuation-passing implementation of recursive descent with a first-order representation of the continuation: the evaluation context. Applying the auxiliary function to a context and a value corresponds to applying the continuation to that value. We come back to this point in appendix.

3 Application: the call-by-value λ-calculus

Let us get back to the call-by-value λ-calculus, as initially considered in the beginning of Section 2. First, we state the grammars of the language and of the syntactic theory, but this time in the format used in Section 2.1 and onwards.

$$\begin{array}{rcl@{\qquad}rcl} e & \in & \textsc{Exp} & e & ::= & \mathsf{var}(x) \mid \mathsf{lam}(x,e) \mid \mathsf{app}(e,e) \\ x & \in & \textsc{Ide} & & & \\ v & \in & \textsc{Val} & v & ::= & \mathsf{var}(x) \mid \mathsf{lam}(x,e) \\ E & \in & \textsc{EvCont} & E & ::= & [\,] \mid \mathsf{app}(E,e) \mid \mathsf{app}(v,E) \\ r & \in & \textsc{Redex} & r & ::= & \mathsf{app}(v,v) \end{array}$$

Ide is a syntactic category of identifiers, containing only values, so there is no value- and evaluation-context definition for it.

The interpreter is defined as follows.

$$\begin{array}{rcl} \mathit{eval} : \textsc{Exp} & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\ \mathit{eval}(e) & = & \mathit{eval}'(\mathit{decompose}(e)) \end{array}$$

$$\begin{array}{rcl} \mathit{eval}' : \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\ \mathit{eval}'(v) & = & v \\ \mathit{eval}'(E, \mathsf{app}(\mathsf{lam}(x,e), v)) & = & \mathit{eval}(\mathit{plug}(E, e[v/x])) \\ \mathit{eval}'(E, \mathsf{app}(x, v_2)) & = & (E, \mathsf{app}(x, v_2)) \end{array}$$

given

$$\begin{array}{rcl} \mathit{decompose} : \textsc{Exp} & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\ \mathit{plug} : \textsc{EvCont} \times \textsc{Exp} & \to & \textsc{Exp} \end{array}$$

The interpreter takes one expression as argument, and attempts to repeatedly decompose, contract, and plug, until either a value or a stuck redex is reached, if any.

The syntactic theory given has a left-to-right reduction sequence, where the length of the reduction sequence of app is 2 and app constructs redexes. Therefore we define the *refocus* and $\mathit{refocus}_{aux}$ functions as follows.

$$\begin{array}{rcl} \mathit{refocus} : \textsc{Exp} \times \textsc{EvCont} & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\ \mathit{refocus}(\mathsf{var}(x), E) & = & \mathit{refocus}_{aux}(E, \mathsf{var}(x)) \\ \mathit{refocus}(\mathsf{lam}(x,e), E) & = & \mathit{refocus}_{aux}(E, \mathsf{lam}(x,e)) \\ \mathit{refocus}(\mathsf{app}(e_1,e_2), E) & = & \mathit{refocus}(e_1, E \circ (\mathsf{app}([\,], e_2)) \end{array}$$

$$\begin{array}{rcl} \mathit{refocus}_{aux} : \textsc{EvCont} \times \textsc{Val} & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\ \mathit{refocus}_{aux}([\,], v) & = & v \\ \mathit{refocus}_{aux}(E \circ (\mathsf{app}([\,], e_2)), v) & = & \mathit{refocus}(e_2, E \circ (\mathsf{app}(v, [\,]))) \\ \mathit{refocus}_{aux}(E \circ (\mathsf{app}(v_1, [\,])), v) & = & (E, \mathsf{app}(v_1, v)) \end{array}$$

The interpreter is then changed to using *refocus* instead of *decompose* and *plug*.

$$
\begin{array}{rcl}
\mathit{eval} : \textsc{Exp} & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\
\mathit{eval}(e) & = & \mathit{eval}'(\mathit{refocus}(e, [\,]))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{eval}' : \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) & \to & \textsc{Val} + (\textsc{EvCont} \times \textsc{Redex}) \\
\mathit{eval}'(v) & = & v \\
\mathit{eval}'(E, \mathsf{app}(\mathsf{lam}(x, e), v)) & = & \mathit{eval}'(\mathit{refocus}(e[v/x], E)) \\
\mathit{eval}'(E, \mathsf{app}(x, v_2)) & = & (E, \mathsf{app}(x, v_2))
\end{array}
$$

NB: The last rule of *eval′* is used for stuck redexes.

To implement this interpreter, e.g., in ML, the grammars of the syntactic theory can be expressed directly as ML data types. As for the *refocus* function, it can use a stack of elementary contexts and a push operation to efficiently implement the composition of an elementary context to the right of a context. It is however more convenient then to represent evaluation contexts "inside out":

$$E ::= [\,] \mid E[[\,]e] \mid E[v[\,]].$$

Because it uses refocusing instead of consecutive plugging and decomposing, the resulting interpreter does not incur a quadratic overhead over its input. For example (cf. Example 1 in Section 2), it evaluates a term such as $\lceil n \rceil\ (\lambda x.x)\ v$ in time $O(n)$.

4 Application: arithmetic expressions with precedence

We give a comprehensive example illustrating all the cases of the construction of a refocus function. We consider arithmetic expressions with additions, multiplications, conditional expressions checking whether their first argument is zero, parenthesized expressions, literals, and oracles returning 0 or 1. (The oracles are only there to illustrate a case of the construction.)

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

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

A program is an expression.

4.1 A syntactic theory

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

Values (trivial terms):

$$\begin{array}{lll} v_e \in \textsc{ExprVal} & & v_e ::= v_t \\ v_t \in \textsc{TermVal} & & v_t ::= v_f \\ v_f \in \textsc{FactVal} & & v_f ::= n \end{array}$$

Computations (serious terms):

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

Evaluation contexts:

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

Redexes:

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

Reduction rules:

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

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 syntactic theory satisfies three unique-decomposition lemmas, i.e., decomposing an expression (resp. of a term and of a factor) into an evaluation 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 redexes, 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.

4.2 An alternative representation of contexts

In Section 4.1, the grammar of evaluation 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 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 evaluation contexts, grouped by the syntactic category they produce when plugged. Take the elementary context $[\,]_t + e$. If we left-compose another evaluation 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 on 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 evaluation contexts above gives the following grammar.

Evaluation contexts:

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

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 evaluation 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 evaluation contexts. The resulting evaluation contexts and reduction rules read as follows.

Evaluation contexts:

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

Reduction rules:

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

4.3 Implementation

Figure 1 displays the BNF of arithmetic expressions in Standard ML. As usual with syntactic theories, 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 2 displays the evaluation contexts and Figure 3, the corresponding plug functions. Figure 4 displays the implementation of redexes and the result of decomposition, and Figure 5, 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 1: 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 2: Evaluation contexts for arithmetic expressions

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

Figure 3: 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 4: 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 5: Decomposition of an arithmetic expression

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

Figure 6: Evaluation of an arithmetic expression

Figure 6 displays the evaluation function, which attempts to decompose its input expression into an evaluation 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 redexes are stuck.

4.4 Refocusing

We now construct the refocus functions. We start from the ML definition of the grammar, in Figure 1, which fits the format expected for the construction.

Let us determine the reduction sequence of each syntactic construct.

The main syntactic constructs: Both addition and 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 has one argument and evaluates 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 differenciates betwen 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 subexpression 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 evaluation 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 Figure 7).

```
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 7: Refocusing over an arithmetic expression

```
(*  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  *)

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

Figure 8: Evaluation of an arithmetic expression, refocused

The main evaluation function is displayed in Figure 8. Rather than decomposing the result of the plug functions, `eval'` refocuses each contractum and iterates.

4.5 Summary and conclusion

We have considered arithmetic expressions with precedence and specified their meaning with a syntactic theory. We then used the method of Section 2 to write refocusing functions that map an evaluation context and syntactic entity into either a value or a decomposition into an evaluation context and a redex. The result is a compositional evaluator that operates in one pass over its input.

5 Application: Sabry and Felleisen's CPS transformation

In their work on reasoning about programs in continuation-passing style (CPS), Sabry and Felleisen designed a new CPS transformation [9, Definition 5]. This CPS transformation integrates a notion of generalized reduction and thus yields very compact CPS programs [2]. It is also unusual in the sense that it builds on the notion of a syntactic theory, rather than on operational semantics [1, 6] or denotational semantics [11]. Therefore, and unlike all the other formalized CPS transformations we are aware of, it is not defined by structural induction over its input. Instead, it is defined as the transitive closure of decomposing, performing an elementary CPS transformation, and plugging. Therefore, its direct implementation incurs the same quadratic factor as considered in Section 2. In the rest of this section, we derive an implementation that operates in linear time over the source program.

The original specification is indexed with 0 (Section 5.1). We first make decomposition and plugging explicit, indexing this specification with 1 (Section 5.2). Then, we present a version that explicitly uses a refocus function, indexing this spefication with 2 (Section 5.3). Using the construction of Section 2, we then define an efficient version of the refocus function.

Terms, values, and contexts are defined as follows.

$$\begin{array}{rcl@{\qquad}rcl} M & \in & \Lambda & M & ::= & V \mid M\,M \\ V & \in & \mathit{Values} & V & ::= & x \mid \lambda x.M \\ x & \in & \mathit{Vars} & & & \\ E & \in & \mathit{EvCont} & E & ::= & [\,] \mid E[V\,[\,]] \mid E[[\,]\,M] \end{array}$$

5.1 The original specification

Definition 5 (Sabry and Felleisen, 1993) *The following CPS transformation uses three mutually recursive functions: $\mathcal{C}_k^0$ to transform terms, $\mathbf{\Phi}^0$ to transform values, and $\mathcal{K}_k^0$ to transform evaluation contexts. Let $k, u_i \in \mathit{Vars}$ be fresh. The functions $\mathcal{C}_k^0$ and $\mathcal{K}_k^0$ are parameterized over a variable k that represents the current continuation.*

$$\begin{array}{rcl} \mathcal{C}_k^0 : \Lambda & \to & \Lambda \\ \mathcal{C}_k^0[\![V]\!] & = & k\,\mathbf{\Phi}^0[\![V]\!] \\ \mathcal{C}_k^0[\![E[x\,V]]\!] & = & x\,\mathcal{K}_k^0[\![E]\!]\,\mathbf{\Phi}^0[\![V]\!] \\ \mathcal{C}_k^0[\![E[(\lambda x.M)\,V]]\!] & = & (\lambda x.\mathcal{C}_k^0[\![E[M]]\!])\,\mathbf{\Phi}^0[\![V]\!] \end{array}$$

$$
\begin{array}{rcl}
\mathbf{\Phi}^0 : \mathit{Values} & \to & \Lambda \\
\mathbf{\Phi}^0[\![x]\!] & = & x \\
\mathbf{\Phi}^0[\![\lambda x.M]\!] & = & \lambda k.\lambda x.\mathcal{C}_k^0[\![M]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}_k^0 : \mathit{EvCont} & \to & \Lambda \\
\mathcal{K}_k^0[\![[\,]]\!] & = & k \\
\mathcal{K}_k^0[\![E[x\ [\,]]]\!] & = & x\ \mathcal{K}_k^0[\![E]\!] \\
\mathcal{K}_k^0[\![E[(\lambda x.M)\ [\,]]]\!] & = & \lambda x.\mathcal{C}_k^0[\![E[M]]\!] \\
\mathcal{K}_k^0[\![E[[\,]\ M]]\!] & = & \lambda u_i.\mathcal{C}_k^0[\![E[u_i\ M]]\!]
\end{array}
$$

The CPS transformation of a complete program M is $\lambda k.\mathcal{C}_k^0[\![M]\!]$. □

5.2 Making decomposition and plugging explicit

We make decomposition and plugging explicit using the two following functions.

$$
\begin{array}{rcl}
\mathit{decompose} & : & \Lambda \to \mathit{Values} + \mathit{EvCont} \times \Lambda \\
\mathit{plug} & : & \mathit{EvCont} \times \Lambda \to \Lambda
\end{array}
$$

Therefore, instead of $\mathcal{C}[\![E[M]]\!]$, we explicitly write $\mathcal{C} \circ \mathit{plug}[\![E,\ M]\!]$.

The original CPS transformation can then be expressed as follows, using an auxiliary function $\widetilde{\mathcal{C}}_k^1$.

$$
\begin{array}{rcl}
\mathcal{C}_k^1 : \Lambda & \to & \Lambda \\
\mathcal{C}_k^1[\![M]\!] & = & \widetilde{\mathcal{C}}_k^1 \circ \mathit{decompose}[\![M]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\widetilde{\mathcal{C}}_k^1 : \mathit{Values} + \mathit{EvCont} \times \Lambda & \to & \Lambda \\
\widetilde{\mathcal{C}}_k^1[\![V]\!] & = & k\ \mathbf{\Phi}^1[\![V]\!] \\
\widetilde{\mathcal{C}}_k^1[\![E,\ x\ V]\!] & = & x\ \mathcal{K}_k^1[\![E]\!]\ \mathbf{\Phi}^1[\![V]\!] \\
\widetilde{\mathcal{C}}_k^1[\![E,\ (\lambda x.M)\ V]\!] & = & (\lambda x.\mathcal{C}_k^1 \circ \mathit{plug}[\![E,\ M]\!])\ \mathbf{\Phi}^1[\![V]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{\Phi}^1 : \mathit{Values} & \to & \Lambda \\
\mathbf{\Phi}^1[\![x]\!] & = & x \\
\mathbf{\Phi}^1[\![\lambda x.M]\!] & = & \lambda k.\lambda x.\mathcal{C}_k^1[\![M]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}_k^1 : \mathit{EvCont} & \to & \Lambda \\
\mathcal{K}_k^1[\![[\,]]\!] & = & k \\
\mathcal{K}_k^1[\![E[x\ [\,]]]\!] & = & x\ \mathcal{K}_k^1[\![E]\!] \\
\mathcal{K}_k^1[\![E[(\lambda x.M)\ [\,]]]\!] & = & \lambda x.\mathcal{C}_k^1 \circ \mathit{plug}[\![E,\ M]\!] \\
\mathcal{K}_k^1[\![E[[\,]\ M]]\!] & = & \lambda u_i.\mathcal{C}_k^1 \circ \mathit{plug}[\![E,\ u_i\ M]\!]
\end{array}
$$

The CPS transformation of a complete program M is $\lambda k.\mathcal{C}_k^1[\![M]\!]$.

Inlining $\mathcal{C}_k^1$ makes it clear that *decompose* is mostly called with the result of *plug*—in fact always, since $M = \mathit{plug}[\![[\,], M]\!]$, initially and in the CPS transformation of a λ-abstraction. Therefore, we can reexpress the CPS transformation using a refocusing function combining *decompose* and *plug*.

5.3 Refocusing

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

$$\mathit{refocus} \quad : \quad \mathit{EvCont} \times \Lambda \to \mathit{Values} + \mathit{EvCont} \times \Lambda$$

The CPS transformation can then be expressed as follows.

$$\begin{array}{rcl}
\mathcal{C}_k^2 : \mathit{Values} + \mathit{EvCont} \times \Lambda & \to & \Lambda \\
\mathcal{C}_k^2[\![V]\!] & = & k\ \mathbf{\Phi}^2[\![V]\!] \\
\mathcal{C}_k^2[\![E,\ x\ V]\!] & = & x\ \mathcal{K}_k^2[\![E]\!]\ \mathbf{\Phi}^2[\![V]\!] \\
\mathcal{C}_k^2[\![E,\ (\lambda x.M)\ V]\!] & = & (\lambda x.\mathcal{C}_k^2 \circ \mathit{refocus}[\![E,\ M]\!])\ \mathbf{\Phi}^2[\![V]\!] \\
 & & \\
\mathbf{\Phi}^2 : \mathit{Values} & \to & \Lambda \\
\mathbf{\Phi}^2[\![x]\!] & = & x \\
\mathbf{\Phi}^2[\![\lambda x.M]\!] & = & \lambda k.\lambda x.\mathcal{C}_k^2 \circ \mathit{refocus}[\![[\,], M]\!] \\
 & & \\
\mathcal{K}_k^2 : \mathit{EvCont} & \to & \Lambda \\
\mathcal{K}_k^2[\![[\,]]\!] & = & k \\
\mathcal{K}_k^2[\![E[x\ [\,]]]\!] & = & x\ \mathcal{K}_k^2[\![E]\!] \\
\mathcal{K}_k^2[\![E[(\lambda x.M)\ [\,]]]\!] & = & \lambda x.\mathcal{C}_k^2 \circ \mathit{refocus}[\![E,\ M]\!] \\
\mathcal{K}_k^2[\![E[[\,]\ M]]\!] & = & \lambda u_i.\mathcal{C}_k^2 \circ \mathit{refocus}[\![E,\ u_i\ M]\!]
\end{array}$$

The CPS transformation of a complete program M is $\lambda k.\mathcal{C}_k^2 \circ \mathit{refocus}[\![[\,], M]\!]$.

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

5.4 Efficiency

The following 'deforested' implementation of *refocus* avoids redundant plugging and decomposition. We write it by following the guidelines of Section 2, inlining $\mathit{refocus}_{aux}$.

$$\begin{array}{rcl}
\mathit{refocus}[\![E,\ M_0\ M_1]\!] & = & \mathit{refocus}[\![E[[\,]\ M_1],\ M_0]\!] \\
\mathit{refocus}[\![E[[\,]\ M_1],\ V_0]\!] & = & \mathit{refocus}[\![E[V_0\ [\,]],\ M_1]\!] \\
\mathit{refocus}[\![E[V_0\ [\,]],\ V_1]\!] & = & [\![E,\ V_0\ V_1]\!]
\end{array}$$

That this function is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$ and more efficient follows from the proof of its construction in Section 2.6. Also, that this

refocus function is similar to the one in Section 3 is unsurprising since it is based on the same syntactic theory of the λ-calculus.

With this definition of *refocus*, the CPS transformation ($\mathcal{C}_k^2$, $\mathbf{\Phi}^2$, $\mathcal{K}_k^2$) produces the same compact terms as ($\mathcal{C}_k^0$, $\mathbf{\Phi}^0$, $\mathcal{K}_k^0$), but it operates in linear time, or more precisely, in one pass, over its input.

6 Conclusion and issues

We have presented a general result about syntactic theories with context-free grammars of values, evaluation contexts, and redexes, and with a unique-decomposition property. This result enables one to mechanically derive an interpreter that does not incur a quadratic-time overhead. We have illustrated the result with two interpreters, one for the call-by-value λ-calculus and one for arithmetic expressions with precedence, and by mechanically turning a quadratic-time program transformation into a program transformation that operates in one pass. In all cases, contexts are best represented inside out.

In appendix, we also flesh out a new connection between evaluation contexts and continuations.

Acknowledgments: We are grateful to Daniel Damian, Bernd Grobauer, and Julia Lawall for commenting the initial version of this article. Thanks are also due to the RULE'01 anonymous referees. This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`).

A A note on defunctionalization

A.1 Defunctionalization before refocusing

Elsewhere [3, Section 4], we observed that in a syntactic theory, the evaluation contexts, the plug functions, and the decomposition functions are the defunctionalized counterpart [7, 8] of one collection of decomposition functions written in continuation-passing style. For three reasons we have found this observation to be consistently useful in practice: (1) writing one collection of decomposition functions is simpler than writing three interconnected definitions; (2) the decomposition functions implement a straightforward recursive descent; and (3) a correct grammar of evaluation contexts follows for free.

Our case in point is Figure 9, which displays the higher-order version of Figures 2, 3, and 5. Initially, `decompose_expr_comp` is called with the identity function instead of with `EEC0`.

A.2 Defunctionalization after refocusing

Our observation also holds after refocusing. Figures 2, 7, and 8 are the defunctionalized counterpart of one collection of refocusing functions written in

```
(*  decompose_expr      : expr                        -> decomposed  *)
(*  decompose_expr_comp : expr_comp * (expr -> expr) -> decomposed  *)
(*  decompose_fact_comp : fact_comp * (fact -> expr) -> decomposed  *)
(*  decompose_term_comp : term_comp * (term -> expr) -> decomposed  *)

fun decompose_expr (EXPR_VAL te)
    = VALUE te
  | decompose_expr (EXPR_COMP se)
    = decompose_expr_comp (se, fn e => e)
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, fn e => eec (EXPR_COMP (ADD (TERM_VAL tt, e))))
  | decompose_expr_comp (ADD (TERM_COMP st, e), eec)
    = decompose_term_comp (st, fn t => eec (EXPR_COMP (ADD (t, e))))
  | 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, fn e => eec (EXPR_COMP (IFZ (e, e1, e2))))
  | decompose_expr_comp (TERM_COMP' st, eec)
    = decompose_term_comp (st, fn (TERM_VAL tt)
                                  => eec (EXPR_COMP (TERM_COMP' st))
                                | (TERM_COMP st)
                                  => eec (EXPR_COMP (TERM_COMP' st)))
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, fn t => tec (TERM_COMP (MUL (FACT_VAL tf, t))))
  | decompose_term_comp (MUL (FACT_COMP sf, t), tec)
    = decompose_fact_comp (sf, fn f => tec (TERM_COMP (MUL (f, t))))
  | decompose_term_comp (FACT_COMP' sf, tec)
    = decompose_fact_comp (sf, fn (FACT_VAL tf)
                                  => tec (TERM_VAL (FACT_VAL' tf))
                                | (FACT_COMP sf)
                                  => tec (TERM_COMP (FACT_COMP' sf)))
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, fn e => fec (FACT_COMP (PARENS e)))
```

Figure 9: Higher-order decomposition of an arithmetic expression

continuation-passing style, and using the following higher-order data type.

```
datatype decomposed
= VALUE of expr_val
  | EXPR_DECOMPOSITION of (expr_val -> decomposed) * expr_redex
  | TERM_DECOMPOSITION of (term_val -> decomposed) * term_redex
  | FACT_DECOMPOSITION of (fact_val -> decomposed) * fact_redex
```

Our case in point is Figure 10, which displays the higher-order version of Figures 2, 7, and 8. Initially, `refocus_expr` is called with `VALUE` instead of with `EEC0`.

The refocus functions, in Figure 10, implement a straightforward recursive descent over the source expression. Their types read as follows.

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

These types witness that the continuation of each refocus function expects a value. The fact that each refocus function is compositional implies that evaluation operates in one pass.

We also observe that because of the intermediate decompositions, the refocus functions do not strictly conform to continuation-passing style, where all calls are tail calls [10]. Therefore, to write them in direct style, one needs control operators for composing continuations such as shift and reset [1].

Another possibility would be to inline `eval'` and to localize the contraction of redexes where they occur in the source term. The result conforms to continuation-passing style and can be mapped back to direct style, yielding a completely usual, compositional, and efficient recursive-descent evaluator.

A.3 Conclusion

Defunctionalization therefore contributes to connecting evaluation contexts and continuations in two ways:

1. In the decomposition of an expression into an evaluation context and a redex, the evaluation context and the plug functions are a defunctionalized continuation.

2. In a refocused evaluator, the evaluation context and the refocus functions are also a defunctionalized continuation.

These two views correspond to the two traditional ways of presenting continuations in the literature: as a representation of the current context and as a representation of the rest of the computation.

```
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)
    = eec te
and refocus_expr_comp (ADD (t, e), eec)
    = refocus_term
        (t, fn tt => refocus_expr
                        (e, fn te => EXPR_DECOMPOSITION
                                       (eec, ADD_REDEX (tt, te))))
  | refocus_expr_comp (IFZ (e0, e1, e2), eec)
    = refocus_expr (e0, fn te => EXPR_DECOMPOSITION
                                   (eec, IFZ_REDEX (te, e1, e2)))
  | refocus_expr_comp (TERM_COMP' st, eec)
    = refocus_term_comp
        (st, fn tt => refocus_expr_val (TERM_VAL' tt, 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)
    = tec tt
and refocus_term_comp (MUL (f, t), tec)
    = refocus_fact
        (f, fn tf => refocus_term
                        (t, fn tt => TERM_DECOMPOSITION
                                       (tec, MUL_REDEX (tf, tt))))
  | refocus_term_comp (FACT_COMP' sf, tec)
    = refocus_fact_comp
        (sf, fn tf => refocus_term_val (FACT_VAL' tf, 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)
    = fec tf
and refocus_fact_comp (FLIP, fec)
    = FACT_DECOMPOSITION (fec, FLIP_REDEX)
  | refocus_fact_comp (PARENS e, fec)
    = refocus_expr
        (e, fn te => FACT_DECOMPOSITION (fec, PARENS_REDEX te))
```

Figure 10: Higher-order refocusing over an arithmetic expression

References

[1] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[2] 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. Also available as the technical report BRICS RS-00-35.

[3] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. Technical Report BRICS RS-01-23, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2001. Extended version of an article to appear in the proceedings of the Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001), Firenze, Italy, September 5-7, 2001.

[4] 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.

[5] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

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

[7] 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).

[8] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

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

[10] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[11] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.

[12] 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/`.

[13] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of decomposition lemma. *Higher-Order and Symbolic Computation*, 14(4), 2001. To appear.

Recent BRICS Report Series Publications

RS-01-31 Olivier Danvy and Lasse R. Nielsen. *Syntactic Theories in Practice*. July 2001. 37 pp. Extended version of an article to appear in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001 (Firenze, Italy, September 4, 2001).

RS-01-30 Lasse R. Nielsen. *A Selective CPS Transformation*. July 2001. 24 pp. To appear in Brookes and Mislove, editors, *27th Annual Conference on the Mathematical Foundations of Programming Semantics*, MFPS '01 Proceedings, ENTCS 45, 2000. A preliminary version appeared in Brookes and Mislove, editors, *Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01*, (Aarhus, Denmark, May 24–27, *2001)*, BRICS Notes Series NS-01-2, 2001, pages 201–222.

RS-01-29 Olivier Danvy, Bernd Grobauer, and Morten Rhiger. *A Unifying Approach to Goal-Directed Evaluation*. July 2001. 23 pp. To appear in *New Generation Computing*, 20(1), November 2001. A preliminary version appeared in Taha, editor, *2nd International Workshop on Semantics, Applications, and Implementation of Program Generation*, SAIG '01 Proceedings, LNCS 2196, 2001, pages 108–125.

RS-01-28 Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. *A Fully Equational Proof of Parikh's Theorem*. June 2001.

RS-01-27 Mario Jose Cáccamo and Glynn Winskel. *A Higher-Order Calculus for Categories*. June 2001. 24 pp. Appears in Boulton and Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference*, TPHOLs '01 Proceedings, LNCS 2152, 2001, pages 136–153.

RS-01-26 Ulrik Frendrup and Jesper Nyholm Jensen. *A Complete Axiomatization of Simulation for Regular CCS Expressions*. June 2001. 18 pp.

RS-01-25 Bernd Grobauer. *Cost Recurrences for DML Programs*. June 2001. 51 pp. Extended version of a paper to appear in Leroy, editor, *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, 2001.