BRICS

Basic Research in Computer Science

BRICS RS-02-4 Danvy & Nielsen: Syntactic Theories in Practice

Syntactic Theories in Practice

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series RS-02-4

ISSN 0909-0878 January 2002

Copyright © 2002, Olivier Danvy & Lasse R. Nielsen.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

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/02/4/`

Syntactic Theories in Practice [*]

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

January 31, 2002

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 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 program. We present sufficient conditions over a syntactic theory to circumvent this overhead, by replacing the composition of (3) plugging and (1) decomposing by a single "refocusing" function mapping a contractum and a context into a new context and a new redex, if there are any. We also show how to construct such a refocusing function, we prove its correctness, and we analyze its complexity.

We illustrate refocusing with two examples: a programming-language interpreter and a transformation into continuation-passing style. As a byproduct, the time complexity of this program transformation is mechanically changed from quadratic in the worst case to linear.

Keywords: syntactic theories, interpreters, refocusing.

[*] A preliminary version of this work was presented at RULE 2001 [4, 10].
[†] 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

1 Introduction

A syntactic theory provides a uniform, concise, and elegant framework for specifying the semantics of a programming language and reasoning about programs [5, 6, 7]. 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 [17, 18].[1] Instead, we identify that the direct implementation of an evaluation function entails an overhead and we show how to circumvent it.

1.1 Related work

Formal semantics has always offered a tempting format for implementing specifications or even for designing programming languages—witness denotational semantics and functional programming [13], Moggi's computational monads and monad-based functional programming [15], operational semantics and logic programming [9], etc. Formal semantics can also provide cost models for computations [8]. The present article describes how to bypass the cost of consecutive plug-and-decompose operations when implementing an interpreter for a syntactic theory, an issue that is usually dealt with on a case-by-case basis to construct abstract machines [6, Chapter 7].

1.2 Interpreters for syntactic theories

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

The interpreter for a syntactic theory 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 a significant overhead, proportional to the program size. Likewise, plugging takes time linear in the size of the context and it always takes at most as long as the following decomposition, if there is one.

1.3 A canonical example: the call-by-value λ-calculus

Here is a syntactic theory of the call-by-value λ-calculus:

$$\begin{array}{rcl@{\qquad}l} e & ::= & x \mid \lambda x.e \mid e\, e & e \in \Lambda \\ v & ::= & x \mid \lambda x.e & v \in \mathit{Value} \subseteq \Lambda \\ & & & x \in \mathit{Var} \end{array}$$

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

[2] Some potential redexes are not actual ones—they are stuck terms [11].

$$\begin{array}{rcll} E & ::= & [\,] \mid E[[\,]\,e] \mid E[v\;[\,]] & E \in \mathit{EvCont} \\ r & ::= & v\;v & r \in \mathit{PotRedex} \subseteq \Lambda \end{array}$$

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

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

Plugging the hole of an evaluation context with an expression is defined by induction on the evaluation context, and thus it operates in time proportional to the depth of the context:

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

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 *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 at least proportional to n^2.* □

1.4 This work

We state conditions under which consecutive plug-and-decompose operations can be replaced by a more efficient refocus operation. 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 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 mechanically construct such a refocus function.

The rest of this article is structured as follows. We first illustrate refocusing with two concrete examples: a syntactic theory for the call-by-value λ-calculus and a transformation of call-by-value λ-terms into continuation-passing style due to Sabry and Felleisen [12]. We then list sufficient conditions to refocus a syntactic theory. The proof that these conditions are sufficient is constructive. It indicates how to mechanically rephrase the syntactic theory to circumvent the overhead. These conditions are general enough to make refocusing practically useful—for example, they are fulfilled by all the examples documented in the SL project [17, 18].

2 The call-by-value λ-calculus

Let us go back to the call-by-value λ-calculus, initially considered in Section 1.3. First, we uniformly restate the grammars of the language and of the syntactic theory in the format used in the rest of this article. These grammars can be expressed directly as ML data types and the corresponding functions as ML functions.

$$\begin{array}{rcll}
e & ::= & \mathsf{var}(x) \mid \mathsf{lam}(x,\, e) \mid \mathsf{app}(e,\, e) & e \in \textsc{Exp} \\
v & ::= & \mathsf{var}(x) \mid \mathsf{lam}(x,\, e) & v \in \textsc{Val} \\
 & & & x \in \textsc{Var} \\
E & ::= & [\,] \mid E[\mathsf{app}([\,],\, e)] \mid E[\mathsf{app}(v,\, [\,])] & E \in \textsc{EvCont} \\
r & ::= & \mathsf{app}(v,\, v) & r \in \textsc{PotRedex}
\end{array}$$

$\textsc{Var}$ is the syntactic category of variables.

We make decomposition and plugging explicit with two functions:

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

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

$$\mathit{refocus} \;:\; \textsc{Exp} \times \textsc{EvCont} \to \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex})$$

We now consider an interpreter and then a CPS transformer for the call-by-value λ-calculus (Sections 2.1 and 2.2). Each uses *decompose* and *plug*, which we fuse into *refocus* (Section 2.3).

2.1 An interpreter

2.1.1 The original specification

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

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

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

The evaluation of an expression e is $\mathit{eval}(e)$. (NB: The last rule of eval' is used for stuck terms.)

The interpreter contains two calls to *eval*: one in the second rule of eval' and one in the initial call. In the second rule of eval', the argument of *eval* is the result of *plug*. In some sense, so is it too in the initial call, since $e = plug(e, [\,])$. In that sense, *decompose* (in the definition of *eval*) 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*.

2.1.2 The refocused specification

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

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

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

The evaluation of an expression e is $\mathit{eval}(e)$.

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

2.2 A CPS transformer

In their work on reasoning about programs in continuation-passing style (CPS), Sabry and Felleisen designed a new CPS transformation [12, Definition 5]. This CPS transformation integrates a notion of generalized reduction and thus yields very compact CPS programs [3]. It is also unusual in the sense that it builds on the notion of a syntactic theory, rather than on operational semantics [2, 11] or denotational semantics [16]. Therefore, it is defined as the transitive closure of decomposing, performing an elementary CPS transformation, and plugging, which makes it a candidate for refocusing.

2.2.1 The original specification

Definition 1 (Sabry and Felleisen, 1993) *The following CPS transformation uses three mutually recursive functions: $\mathcal{C}_k$ (and the auxiliary function $\mathcal{C}'_k$) to transform expressions, $\mathbf{\Phi}$ to transform values, and $\mathcal{K}_k$ to transform evaluation contexts. The functions $\mathcal{C}_k$, $\mathcal{C}'_k$, and $\mathcal{K}_k$ are parameterized over a variable k that represents the current continuation.*

$$\begin{array}{rcl} \mathcal{C}_k & : & \textsc{Exp} \to \textsc{Exp} \\ \mathcal{C}_k(e) & = & \mathcal{C}'_k(\mathit{decompose}(e)) \end{array}$$

$$\begin{array}{rcl} \mathcal{C}'_k & : & \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex}) \to \textsc{Exp} \\ \mathcal{C}'_k(v) & = & \mathsf{app}(k, \mathbf{\Phi}(v)) \\ \mathcal{C}_k(E, \mathsf{app}(x, v)) & = & \mathsf{app}(\mathsf{app}(x, \mathcal{K}_k(E)), \mathbf{\Phi}(v)) \\ \mathcal{C}_k(E, \mathsf{app}(\mathsf{lam}(x, e), v)) & = & \mathsf{app}(\mathsf{lam}(x, \mathcal{C}_k(\mathit{plug}(e, E))), \mathbf{\Phi}(v)) \end{array}$$

$$\begin{array}{rcl}
\mathbf{\Phi} & : & \textsc{Val} \rightarrow \textsc{Exp} \\
\mathbf{\Phi}(x) & = & x \\
\mathbf{\Phi}(\mathsf{lam}(x,\, e)) & = & \mathsf{lam}(k,\, \mathsf{lam}(x,\, \mathcal{C}_k(e))) \\
 & & \textit{where } k \textit{ is fresh}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}_k & : & \textsc{EvCont} \rightarrow \textsc{Exp} \\
\mathcal{K}_k([\,]) & = & k \\
\mathcal{K}_k(E[\mathsf{app}(x,\, [\,])]) & = & \mathsf{app}(x,\, \mathcal{K}_k(E)) \\
\mathcal{K}_k(E[\mathsf{app}(\mathsf{lam}(x,\, e),\, [\,])]) & = & \mathsf{lam}(x,\, \mathcal{C}_k(\mathit{plug}(e,\, E))) \\
\mathcal{K}_k(E[\mathsf{app}([\,],\, e)]) & = & \mathsf{lam}(u_i,\, \mathcal{C}_k(\mathit{plug}(\mathsf{app}(u_i,\, e),\, E))) \\
 & & \textit{where } u_i \textit{ is fresh}
\end{array}$$

The CPS counterpart of an expression e is $\mathsf{lam}(k,\, \mathcal{C}_k(e))$, where k is fresh. □

The CPS transformer contains five calls to $\mathcal{C}_k$. In three cases, the argument of $\mathcal{C}_k$ is the result of *plug*, and in two cases, the argument is e. As in Section 2.1.1, and since $e = \mathit{plug}(e,\, [\,])$, we can say that *decompose* (in the definition of $\mathcal{C}_k$) is always called with the result of *plug*.

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

2.2.2 The refocused specification

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

$$\begin{array}{rcl}
\mathcal{C}_k & : & \textsc{Exp} \rightarrow \textsc{Exp} \\
\mathcal{C}_k(e) & = & \mathcal{C}'_k(\mathit{refocus}(e,\, [\,]))
\end{array}$$

$$\begin{array}{rcl}
\mathcal{C}'_k & : & \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex}) \rightarrow \textsc{Exp} \\
\mathcal{C}'_k(v) & = & \mathsf{app}(k,\, \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(E,\, \mathsf{app}(x,\, v)) & = & \mathsf{app}(\mathsf{app}(x,\, \mathcal{K}_k(E)),\, \mathbf{\Phi}(v)) \\
\mathcal{C}'_k(E,\, \mathsf{app}(\mathsf{lam}(x,\, e),\, v)) & = & \mathsf{app}(\mathsf{lam}(x,\, \mathcal{C}'_k(\mathit{refocus}(e,\, E))),\, \mathbf{\Phi}(v))
\end{array}$$

$$\begin{array}{rcl}
\mathbf{\Phi} & : & \textsc{Val} \rightarrow \textsc{Exp} \\
\mathbf{\Phi}(x) & = & x \\
\mathbf{\Phi}(\mathsf{lam}(x,\, M)) & = & \mathsf{lam}(k,\, \mathsf{lam}(x,\, \mathcal{C}_k(e))) \\
 & & \text{where } k \text{ is fresh}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}_k & : & \textsc{EvCont} \rightarrow \textsc{Exp} \\
\mathcal{K}_k([\,]) & = & k \\
\mathcal{K}_k(E[\mathsf{app}(x,\, [\,])]) & = & \mathsf{app}(x,\, \mathcal{K}_k(E)) \\
\mathcal{K}_k(E[\mathsf{app}(\mathsf{lam}(x,\, e),\, [\,])]) & = & \mathsf{lam}(x,\, \mathcal{C}'_k(\mathit{refocus}(e,\, E))) \\
\mathcal{K}_k(E[\mathsf{app}([\,],\, e)]) & = & \mathsf{lam}(u_i,\, \mathcal{C}'_k(\mathit{refocus}(\mathsf{app}(u_i,\, e),\, E))) \\
 & & \text{where } u_i \text{ is fresh}
\end{array}$$

The CPS transformation of an expression e is $\mathsf{lam}(k,\, \mathcal{C}_k(e))$, where k is fresh.

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

2.3 Refocusing efficiently

Let us fuse *plug* and *decompose* into one function. First, here is their definition.

$$\begin{aligned}
\mathit{plug}(e,\,[\,]) &= e \\
\mathit{plug}(e,\,E[\mathsf{app}([\,],\,e_2)]) &= \mathit{plug}(\mathsf{app}(e,\,e_2),\,E) \\
\mathit{plug}(e,\,E[\mathsf{app}(v_1,\,[\,])]) &= \mathit{plug}(\mathsf{app}(v_1,\,e),\,E)
\end{aligned}$$

$$\begin{aligned}
\mathit{decompose}(e) &= \mathit{decompose}'(e,\,[\,])
\end{aligned}$$

$$\begin{aligned}
\mathit{decompose}' &: \textsc{Exp} \times \textsc{EvCont} \rightarrow \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex}) \\
\mathit{decompose}'(v,\,E) &= \mathit{decompose}'_{aux}(E,\,v) \\
\mathit{decompose}'(\mathsf{app}(e_1,\,e_2),\,E) &= \mathit{decompose}'(e_1,\,E[\mathsf{app}([\,],\,e_2)])
\end{aligned}$$

$$\begin{aligned}
\mathit{decompose}'_{aux} &: \textsc{EvCont} \times \textsc{Val} \rightarrow \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex}) \\
\mathit{decompose}'_{aux}([\,],\,v) &= v \\
\mathit{decompose}'_{aux}(E[\mathsf{app}([\,],\,e_2)],\,v) &= \mathit{decompose}'(e_2,\,E[\mathsf{app}(v,\,[\,])]) \\
\mathit{decompose}'_{aux}(E[\mathsf{app}(v_1,\,[\,])],\,v) &= (E,\,\mathsf{app}(v_1,\,v))
\end{aligned}$$

The definition of *plug* is a transliteration of the one in Section 1.3. As for *decompose*, it implements a depth-first search for the first application of one value to another, and uses two auxiliary functions: one, $\mathit{decompose}'$, recursively descends into an expression and accumulates the corresponding context, and the other, $\mathit{decompose}'_{aux}$, dispatches on the accumulated context.

Let us now construct a more efficient version of *refocus* = *decompose* ∘ *plug*. Our key observation is a consequence of the unique-decomposition property of a syntactic theory of the λ-calculus and of the fact that the grammar of evaluation contexts is context-free:

$$\mathit{decompose}(\mathit{plug}(e,\,E)) = \mathit{decompose}'(e,\,E)$$

In words: E uniquely determines the partial decomposition of the result of $\mathit{plug}(e,\,E)$ into the expression e and the evaluation context E. (In general, this decomposition is partial because e is not necessarily a potential redex.) A similar consequence holds for values:

$$\mathit{decompose}(\mathit{plug}(v,\,E)) = \mathit{decompose}'_{aux}(E,\,v)$$

A simple fold-unfold calculation[3] lets us calculate the following version of *refocus*.

[3]For example, $$\begin{aligned}
\mathit{refocus}(\mathsf{app}(e_1,\,e_2),\,E) &= \mathit{decompose}(\mathit{plug}(\mathsf{app}(e_1,\,e_2),\,E)) \\
&= \mathit{decompose}'(\mathsf{app}(e_1,\,e_2),\,E) \\
&= \mathit{decompose}'(e_1,\,E[\mathsf{app}([\,],\,e_2)]) \\
&= \mathit{decompose}(\mathit{plug}(e_1,\,E[\mathsf{app}([\,],\,e_2)])) \\
&= \mathit{refocus}(e_1,\,E[\mathsf{app}([\,],\,e_2)])
\end{aligned}$$

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

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

The resulting function *refocus* is extensionally equivalent to *decompose* $\circ$ *plug* because fold-unfold transformations are correct [1]. It is also more efficient because it avoids building an intermediate expression for the sole purpose of decomposing it straight away.[4]

2.3.1 The interpreter

The refocused interpreter produces the same result as the original one, but it operates more efficiently. For example (cf. Example 1 in Section 1.3), when evaluating a term such as $\lceil n \rceil\ (\lambda x.x)\, v$, the time taken to refocus from a contractum and a context to a new context and a redex is independent of the number of remaining applications.

2.3.2 The CPS transformer

The refocused CPS transformer produces the same compact terms as the original one, but it operates in linear time. More precisely, it operates in one pass over its input.[5]

2.4 Summary

We have described how to refocus a syntactic theory for the call-by-value λ-calculus, by deforesting the intermediate expressions produced in each decompose-contract-plug cycle. As a byproduct, we have shown how refocusing makes it possible to derive a linear-time program transformation out of a quadratic-time one.

[4]Replacing the composition of two functions that produce and consume an intermediate data structure is an instance of *deforestation* [14].

[5]Actually, the transformation could have been optimized to read as follows:

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

This way, each u_i would not be inspected as an expression.

3 Refocusing in a syntactic theory

In this section we show how to construct a *refocus* function for a syntactic theory, as an alternative to plugging and decomposing. 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.

3.1 The language

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

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

where c ranges over a set of constructor names and e, $e_1, \ldots, e_n$ are non-terminals corresponding to the syntactic categories of the language. Each constructor may occur only once in a grammar.

3.2 Syntactic theories

A syntactic theory is a specification of a small-steps operational semantics, i.e., it specifies a reduction relation between programs. Syntactic theories traditionally come with context-free grammars for values and evaluation contexts, along with reduction rules of the form $E[r] \to e$, where the term r is called a redex (short for *reducible expression*). Let us review each of these concepts in turn.

3.2.1 Values

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

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

where $e ::= \mathsf{c}(e_1, \ldots, e_n)$ is a production of the language grammar and each x_i is either e_i or v_i.

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

3.2.2 Evaluation contexts

Evaluation contexts are also specified by context-free grammars, and their meaning is given by the associated *plug* function. For example, the evaluation contexts of the call-by-value λ-calculus and the associated *plug* function are shown in Section 2.

All the evaluation contexts of a syntactic theory can be written as the composition of *elementary* evaluation contexts. Composition is a binary function on evaluation contexts, $\circ$, that together with the *plug* function satisfies $(E_1 \circ E_2)[e] = E_1[E_2[e]]$, and is thus associative. Elementary contexts can themselves not be written as compositions of other non-empty contexts.[6]

For the call-by-value λ-calculus, the elementary contexts are $\mathsf{app}([\,],\ e)$ and $\mathsf{app}(v,\ [\,])$. The construction of evaluation contexts corresponds to composing an elementary context to the right of another context, e.g., $E[\mathsf{app}([\,],\ e)] = E \circ \mathsf{app}([\,],\ e)$. In many articles on syntactic theories, though, compound contexts are constructed by composing an elementary context on the left rather than on the right. For the λ-calculus, the grammar of evaluation contexts then reads as follows.

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

This notation induces the same elementary contexts.

3.2.3 Decompositions and potential redexes

If a term, e, is the result of plugging term e' into an evaluation context E, i.e., $E[e'] = e$, then we say that E and e form a *decomposition* (into an evaluation context and a term) of e', or alternatively that e' can be decomposed into E and e. A term can be decomposed in many different ways.

If $e = E[e']$ then the decomposition is *trivial* if either E is the empty context or e' is a value.

Values are normal forms: they cannot have a non-value sub-term in a position where it can be evaluated, so all decompositions must be trivial.[7] Non-value terms that can only be decomposed trivially are called *potential redexes*. For the λ-calculus, the potential redexes are exactly the terms of the form $\mathsf{app}(v,\ v)$.

If $e = E[e']$ then the decomposition is *complete* if e' is a potential redex. The reduction rules of a syntactic theory contain only complete decompositions.

If a term has a decomposition, $E[e]$, that is not complete, then e can be further decomposed in a non-trivial way as $E'[e']$. Then $(E \circ E')[e']$ 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.

3.2.4 Unique decomposition

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

Definition 2 (Unique decomposition) *A syntactic theory is said to satisfy the* unique-decomposition property *if any non-value term e can be uniquely decomposed as $e = E[r]$ where r is a potential redex.*

[6]We note that evaluation contexts, together with composition and the empty context, form a monoid.

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

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 [18].[8]

3.2.5 Reduction rules

The reduction rules of a syntactic theory are of the form $\mathrm{E}[r] \to e$ where r is a potential redex and e depends only on E and on the sub-terms of r. Often e is written as $\mathrm{E}[e']$ for some e' depending on r.

The potential redexes that occur in a reduction rule are the actual redexes of the syntactic theory. The remaining potential redexes stick, to use Plotkin's word [11]. 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 call-by-value λ-calculus is $E[\mathsf{app}(\mathsf{lam}(x,\ e),\ x)] \to e\,[v/x]$, and thus $\mathsf{app}(\mathsf{lam}(x,\ e),\ x)$ is a redex. The remaining potential redexes (of the form $\mathsf{app}(x,\ v)$) are stuck terms.

3.3 Requirements

We state three requirements that are sufficient for constructing a *refocus* function. We consider syntactic theories whose structure is as described in Section 3.2 and that satisfy unique decomposition.

3.3.1 No redundant constructs

Syntactic theories specify 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 $e ::= \mathsf{c}(e_1,\ \ldots,\ e_n)$ and the syntactic category ranged over by e_i contains only values, then an elementary evaluation 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 evaluation context constructed by composing this elementary context can ever be part of a complete decomposition.
- If a syntactic category, ranged over by e, 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 this rule redundant. (Syntactic categories with no values are not useless; they can occur meaningfully in, e.g., languages with control effects.)

[8]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 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.

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

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 e and v generate the same set of terms, i.e., whether e only ranges over values.

3.3.2 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 3.2.1. 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 i'th position. Then $\mathsf{c}(\ldots,\, e_i,\, \ldots)$ is a potential redex when e_i is *not* a value.

There are several ways to achieve 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 e_i.

3.3.3 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 evaluation context and potential redex will always have the hole of the context occurring in the left-most non-evaluated sub-term, in the ordering of sub-terms chosen for the abstract syntax.

3.4 Consequences of the requirements

The requirements of Section 3.3 guarantee that a syntactic theory contains elementary evaluation contexts, values, and potential redexes of a specific form. This form is described in Figure 1.

An example where the m in Figure 1 is strictly smaller than n is a conditional expression such as $\mathsf{if}(e,\, e,\, e)$. The only elementary evaluation context is $\mathsf{if}([\,],\, e,\, e)$, since the other two sub-terms should not be evaluated until the conditional expression has itself been reduced. Another example is the call-by-name λ-calculus where only the function part of an application is evaluated.

Lemma 1 *A syntactic theory satisfying the requirements stated in Section 3.3 contains productions corresponding to Figure 1.*

For a syntactic construct with n sub-terms, $e ::= \mathsf{c}(e_1, \ldots, e_n)$, there is a number $0 \leq m \leq n$ such that the elementary evaluation contexts for c are exactly:

$$\begin{array}{l}\mathsf{c}([\,], e_2, \ldots, e_n)\\ \mathsf{c}(v_1, [\,], e_3, \ldots, e_n)\\ \vdots\\ \mathsf{c}(v_1, \ldots, v_{m-1}, [\,], e_{m+1}, \ldots, e_n)\end{array}$$

If terms of the form $\mathsf{c}(v_1, \ldots, v_m, e_{m+1}, \ldots, e_n)$ exist (i.e., the syntactic category ranged over by e_m contains values) then they are either all values or all potential redexes. In other words, either

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

or

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

is a production, but not both.

We then say that c *needs to evaluate m sub-terms.* If $\mathsf{c}(v_1, \ldots, v_m, e_{m+1}, \ldots, e_n)$ 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 1: Elementary contexts, values, and potential redexes

Proof (sketch): Take the production $e ::= \mathsf{c}(e_1, \ldots, e_n)$.

If there is a value production, $v ::= \mathsf{c}(e_1, \ldots, e_n)$ or potential-redex production $r ::= \mathsf{c}(e_1, \ldots, e_n)$, then $m = 0$ and c becomes a value or a potential redex.

If there is no such production then since e is not redundant, there must exist a term, $\mathsf{c}(e_1, \ldots, e_n)$, that has a non-trivial decomposition and thus there must exist an elementary evaluation context $\mathsf{c}(e_1, \ldots, e_{i-1}, [\,], e_{i+1}, \ldots, e_n)$. 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 $\mathsf{c}([\,], e_2, \ldots, e_n)$.

Assume that there exists an elementary evaluation context of the form

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

Three cases occur:

1. If e_i only ranges over non-values then $m = i$ and c becomes neither a value nor a potential redex.

2. If e_i ranges over values and there exists a production $v ::= \mathsf{c}(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)$ or $r ::= \mathsf{c}(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)$ then $m = i$ and c becomes either a value or a potential redex.

3. If e_i ranges over values, but terms of the form $\mathsf{c}(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)$ are not values or potential redexes, then they must have non-trivial decompositions, so there is a corresponding elementary evaluation context. Since we assume left-to-right evaluation, the context must be $\mathsf{c}(v_1, \ldots, v_i, [\,], e_{i+2}, \ldots, e_n)$.

In other words, either (1) $\mathsf{c}(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)$ does not exist, or it is a value or a potential redex, and then $m = i$, or (2) there exists an elementary evaluation context $\mathsf{c}(v_1, \ldots, v_i, [\,], e_{i+2}, \ldots, e_n)$, 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 syntactic theory satisfying the requirements contain the productions of Figure 1. 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 1, and thus any further productions would either be redundant or violate the unique-decomposition property.

3.5 Constructing a refocus function

We construct a function, *refocus*, that is extensionally equivalent to the composition of the *plug* function and the *decompose* function. It uses a stack of elementary contexts to represent evaluation contexts. Such a stack directly corresponds to the representation of contexts of Section 2, and it allows an efficient implementation of composing an elementary evaluation context and plugging a term in a context. We use $[\,]$ for the empty context/stack and $E \circ \mathsf{c}(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)$ for composing/pushing an elementary context.

The refocusing function is defined with two mutually recursive functions. The first, *refocus*, takes a stack of elementary evaluation contexts and a term, and is defined by cases of the term. 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 evaluation context on the stack.

For each $e ::= \mathsf{c}(e_1, \ldots, e_n)$ 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 $\mathsf{c}([\,], e_2, \ldots, e_n)$ through $\mathsf{c}(v_1, \ldots, v_{m-1}, [\,], e_{m+1}, \ldots, e_n)$. If *refocus* and $\mathit{refocus}_{aux}$ are implemented in a typed setting, 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 functions corresponding to the syntactic category ranged over by e. The rules of $\mathit{refocus}_{aux}$ go in the 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 $\mathsf{c}(e_1, \ldots, e_n)$ is a value or a potential redex.

(a) If c becomes a value, then

$$\mathit{refocus}(\mathsf{c}(e_1, \ldots, e_n), E) = \mathit{refocus}_{aux}(E, \mathsf{c}(e_1, \ldots, e_n))$$

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

$$\mathit{refocus}(\mathsf{c}(e_1, \ldots, e_n), E) = (E, \mathsf{c}(e_1, \ldots, e_n))$$

since we have found the decomposition.

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

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

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

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

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

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

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

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

(c) If e_i ranges over only non-values then there is no rule. In a typed setting there is no $\mathit{refocus}_{aux}$ corresponding to 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:

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

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 program is a value, so the decomposition has to return a value rather than a a context and a potential redex.

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

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

3.6 Correctness

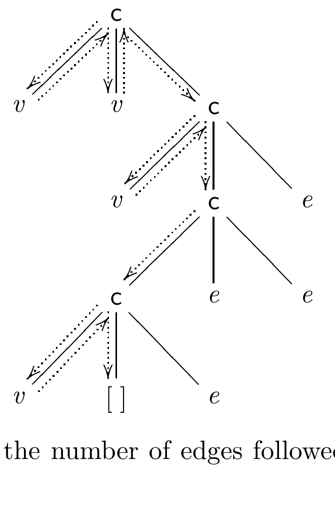
By construction, *refocus* can only return either a value or a pair of evaluation context and potential redex, since the remaining cases all perform a tail-recursive call. Both *refocus* and $\mathit{refocus}_{aux}$ are passed a decomposition of a term, i.e., an evaluation context and another term, and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

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

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

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

Given an evaluation 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 $\mathcal{T}(E)$ 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 $E[e] = E'[r]$ then refocus(e, E) terminates after $\mathcal{T}(E') + \mathcal{T}_{aux}(r) - \mathcal{T}(E)$ 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*$_{aux}$:

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

With this definition, all choices of arguments to *refocus* and *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 *refocus*$_{aux}$ contains the following case:

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

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:

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

and

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

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

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

□

Lemma 2 tells us both that *refocus* is a total function and how many calls it takes to find the result. In Section 3.7, we use it to show that *refocus* is always at least as efficient as using *plug* and *decompose*.

3.7 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 3.6. 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* : $\textsc{Exp} \to \textsc{Val} + (\textsc{EvCont} \times \textsc{PotRedex})$ finding the unique complete decomposition of terms (in a syntactic category satisfying our requirements) must perform at least $(\mathcal{T}(E) + \mathcal{T}_{aux}(r))/2$ basic operations on the syntax to compute $decompose(E[r]) = (E, r)$. If it uses fewer operations, it cannot inspect all the values of the evaluation context and of the potential redex. If we changed such a value to a non-value, then the hole of the evaluation 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 evaluation context. Since the time complexity of *decompose* is always greater, we ignore *plug* in our comparison.

In summary, the complexity of computing $decompose(E[e]) = E'[r]$ is at least proportional to $\mathcal{T}(E') + \mathcal{T}_{aux}(r)$, whereas the complexity of computing $refocus(e, E) = E'[r]$ is proportional to $\mathcal{T}(E') + \mathcal{T}_{aux}(r) - \mathcal{T}(E)$. For syntactic theories where rewrites are often local, the difference between $\mathcal{T}(E)$ and $\mathcal{T}(E')$ is often significantly smaller than the value of $\mathcal{T}(E')$ itself.

For example, in Example 1 of Section 1.3 the difference between $\mathcal{T}(E)$ and $\mathcal{T}(E')$ is constant whereas $\mathcal{T}(E')$ alone is proportional to the size of the program.

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.

3.8 Summary

We have presented a few mild requirements to construct a *refocus* function for a syntactic theory. We have also proven the correctness of this construction and 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.

4 Conclusion and issues

We have presented a general result about syntactic theories with context-free grammars of values, evaluation contexts, and redexes, and satisfying a unique-decomposition property. This result enables one to mechanically derive an interpreter that does not incur a linear-time overhead for each reduction step. 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.

Acknowledgments: We are grateful to Daniel Damian, Bernd Grobauer, Fritz Henglein, and Julia Lawall for commenting a previous version of this article. Thanks are also due to the anonymous RULE'01 referees.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`).

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 3.5. 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.

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

A program is an expression.

A.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):

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

Computations (serious terms):

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

Evaluation contexts:

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

Redexes:

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

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 syntactic theory satisfies three unique-decomposition lemmas, i.e., decomposing an expression (resp. a term and 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 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 Evaluation contexts with right-composition

In Section A.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., $E_e ::= v_t + E_e$, can be written as $E_e ::= (v_t + [\,]_e) \circ E_e$. Since composition of contexts is associative, we can 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 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 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:

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

```
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 2: Abstract syntax of arithmetic expressions

A.3 Implementation

Figure 2 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 3 displays the evaluation contexts and Figure 4, the corresponding plug functions. Figure 5 displays the implementation of redexes and the result of decomposition, and Figure 6, the corresponding decomposition functions.

```
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 3: Evaluation 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 4: 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 5: 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 6: 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
              (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 (plug_expr (e1, eec))
  | eval' (EXPR_DECOMPOSITION
             (eec, IFZ_REDEX (TERM_VAL' (FACT_VAL' (INT n)), e1, e2)))
    = eval (plug_expr (e2, eec))
  | eval' (TERM_DECOMPOSITION
             (tec, MUL_REDEX (INT n1, FACT_VAL' (INT n2))))
    = eval (plug_term (TERM_VAL (FACT_VAL' (INT (n1 * n2))), tec))
  | eval' (FACT_DECOMPOSITION
             (fec, FLIP_REDEX))
    = eval (plug_fact (FACT_VAL (INT (Oracle.flip ())), fec))
  | eval' (FACT_DECOMPOSITION
             (fec, PARENS_REDEX (TERM_VAL' (FACT_VAL' (INT n)))))
    = eval (plug_fact (FACT_VAL (INT n), fec))
```

Figure 7: Evaluation of an arithmetic expression

Figure 7 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 terms are stuck.

A.4 Refocusing

We now construct the refocus functions. We start from the ML definition of the grammar, in Figure 2, 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 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 a constructor needs to evaluate zero sub-terms, 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 constructor needs to evaluate more than zero sub-terms 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 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 all the sub-terms needed by a constructor are evaluated 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 all the sub-terms needed by a constructor are evaluated 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 constructor needs to evaluate more sub-terms, 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 8).

The main evaluation function is displayed in Figure 9. Rather than decomposing the result of the plug functions, `eval'` refocuses from a contractum to the next decomposition and iterates.

```
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 8: 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 9: Evaluation of an arithmetic expression, refocused

A.5 Summary

We have considered arithmetic expressions with precedence and we have specified their meaning with a syntactic theory. We then applied the algorithm of Section 3.5 to write refocusing functions that map an evaluation context and a contractum into either a value or a decomposition into an evaluation context and a new redex. The result is a compositional evaluator that operates in one pass over its input.

References

[1] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

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

[3] 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.

[4] 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. Extended version available as the technical report BRICS RS-02-04.

[5] 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.

[6] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2001.

[7] 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.

[8] John Greiner. *Semantics-based parallel cost models and their use in provably efficient implementations.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1997. Technical Report CMU-CS-97-113.

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

[10] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

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

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

[13] 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.

[14] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989.

[15] 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.

[16] 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.

[17] 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/`.

[18] 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):387–409, 2001.

Recent BRICS Report Series Publications

RS-02-4 **Olivier Danvy and Lasse R. Nielsen.** ***Syntactic Theories in Practice*****. January 2002. 34 pp. This revised report supersedes the earlier BRICS report RS-01-31.**

RS-02-3 **Olivier Danvy and Lasse R. Nielsen.** ***On One-Pass CPS Transformations*****. January 2002. 18 pp.**

RS-02-2 **Lasse R. Nielsen.** ***A Simple Correctness Proof of the Direct-Style Transformation*****. January 2002.**

RS-02-1 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***The `<bigwig>` Project*****. January 2002. 36 pp. This revised report supersedes the earlier BRICS report RS-00-42.**

RS-01-55 **Daniel Damian and Olivier Danvy.** ***A Simple CPS Transformation of Control-Flow Information*****. December 2001.**

RS-01-54 **Daniel Damian and Olivier Danvy.** ***Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation*****. December 2001. 41 pp. To appear in the *Journal of Functional Programming*. This report supersedes the earlier BRICS report RS-00-15.**

RS-01-53 **Zoltán Ésik and Masami Ito.** ***Temporal Logic with Cyclic Counting and the Degree of Aperiodicity of Finite Automata*****. December 2001. 31 pp.**

RS-01-52 **Jens Groth.** ***Extracting Witnesses from Proofs of Knowledge in the Random Oracle Model*****. December 2001. 23 pp.**

RS-01-51 **Ulrich Kohlenbach.** ***On Weak Markov's Principle*****. December 2001. 10 pp.**

RS-01-50 **Jiří Srba.** ***Note on the Tableau Technique for Commutative Transition Systems*****. December 2001. 19 pp. To appear in Nielsen and Engberg, editors, *Foundations of Software Science and Computation Structures*, FoSSaCS '02 Proceedings, LNCS 2303, 2002.**

RS-01-49 **Olivier Danvy and Lasse R. Nielsen.** ***A First-Order One-Pass CPS Transformation*****. December 2001. 21 pp. Extended version of a paper to appear in Nielsen and Engberg, editors, *Foundations of Software Science and Computation Structures*, FoSSaCS '02 Proceedings, LNCS 2303, 2002.**