Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 17 pages

Syntactic Theories in Practice

Olivier Danvy and Lasse R. Nielsen

BRICS [1]

Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
E-mail: {danvy,lrn}@brics.dk

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 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, and illustrate the method with an interpreter for the call-by-value λ-calculus and a transformation into continuation-passing style (CPS). In particular, we mechanically change the time complexity of this CPS transformation from potentially quadratic to linear.

An extended version is available as the technical report BRICS RS-01-31 [4].

1 Introduction

A syntactic theory provides a uniform, concise, and elegant framework to specify a programming language and to reason about it [5]. 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 [10,11]. [2] Instead, we show how to circumvent the overhead entailed by a direct implementation of an evaluation function.

We first identify the overhead. Then, we list sufficient conditions to rephrase a syntactic theory so that implementing its evaluation function does not incur this overhead. The proof that these conditions are sufficient is constructive in that it indicates how to mechanically rephrase the syntactic

[1] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

[2] www.cs.uoregon.edu/~ariola/SL/

theory to circumvent the overhead. We consider two examples: a syntactic theory for the call-by-value λ-calculus and a transformation of λ-terms into continuation-passing style due to Sabry and Felleisen [8].

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:

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

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

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

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 may take a quadratic time. Let us illustrate this complexity using Church numerals: $\lceil n \rceil$ is the Church numeral for the number n, i.e., $\lambda s.\lambda z.\ \underbrace{s(s(s(\ldots(s}_{n}\ z)\ldots)))$.

Example 2.1 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 2.2 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 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 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 generic production on the form $e ::= c(e_1, \ldots, e_n)$, the set associated to e is called Exp and the ones ranged over by e_1 through e_n are called Exp_1 through Exp_n, respectively.

We assume that there is no trivial syntactic category, i.e., one where 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 [6, 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 [11].)

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 then 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:

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

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.

$$|[\,]| = 0$$
$$|c(e_1, \ldots, E, \ldots, e_n)| = 1 + |E|$$

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 2.3 [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 [11].

If the syntactic theory satisfies a unique-decomposition property, then its redexes are exactly the non-value terms that can only be trivially decomposed. A decomposition, $E[e]$ is trivial if either e is a value or E is the empty context. That redexes are exactly the non-value terms that can only be trivially decomposed follows from 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 2.4 [Left-to-right reduction sequence] A syntactic theory is said to have 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.

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

(ii) 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. □

For simplicity, in the present version of this article, we only consider syntactic theories that have such a left-to-right reduction sequence. (In an extended version of this article [4], we consider arbitrary reduction sequences.)

In the remainder of this section, 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*.

(i) 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.

(ii) 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.

(iii) 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 sub-expression, 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:

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

(iv) Finally there is one rule for the empty context.

$$refocus_{aux}([\,], v) = v$$

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 2.5 [Ordering on decompositions of a term] Let $e = E_1[e_1] = E_2[e_2]$ be two decompositions of the *same* term. 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 are 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 three cases:

(i) $E_1 \sqcap E_2 = E_1$, in which case $E_1 < E_2$ and thus $(E_1, e_1) < (E_2, e_2)$ (by the first case of Definition 2.5),

(ii) $E_1 \sqcap E_2 = E_2$, in which case $E_2 < E_1$ and thus $(E_2, e_2) < (E_1, e_1)$ (by the first case of Definition 2.5), or

(iii) $E_1 \sqcap E_2$ is strictly smaller than both E_1 an E_2, so by the second case of Definition 2.5, 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 $refocus_{aux}$ happen on smaller and smaller decompositions of the same term, so such calls must eventually terminate.

Lemma 2.6 (Totality) *Any call to refocus terminates, i.e., refocus is total.*

Proof. We prove that $refocus_{aux}$ is total and that it yields a decomposition of its argument into an evaluation context and a redex.

The proof is by well-founded induction on the arguments of $refocus_{aux}$.

There are four cases, one base case and one case for each possible behavior

of $\mathit{refocus}_{aux}$ on an argument $(E \circ c_i(e_1, \ldots, [\,], \ldots, e_n), v)$.

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

(ii) 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

$$(E, c(e_1, \ldots, v, \ldots, e_n)) \sqsubset (E \circ c_i(e_1, \ldots, [\,], \ldots, e_n), v).$$

(iii) 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.

(iv) 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 $E_1 \not< E'$ (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 $refocus_{aux}$ is a total function, and another induction, similar to the sub-induction in Case 3, shows that *refocus* is total as well. □

The function *refocus* is total, and it can only terminate yielding either a value (from the decomposition $[\ ][v]$) or a decomposition into an evaluation context and a redex ($E[v_1\ v_2]$), both smaller decompositions of the original term. Since such decompositions are unique, *refocus* 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.[3] In an implementation of a syntactic theory that uses *refocus*, the time taken to perform a reduction sequence like $E[e] \rightarrow E[e'] \rightarrow E[e'']$ is thus independent of E. Using *refocus* thus makes it possible to avoid the overhead identified at the start of this section.

3 An interpreter for the call-by-value lambda-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.

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

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

[3] In fact, as we observed elsewhere [3], the refocus function corresponds to a CPS 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.

The interpreter is defined as follows.

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

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

given

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

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.

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

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

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

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

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

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 any overhead. For example (cf. Example 2.1 in Section 2), it evaluates a term such as $\lceil n \rceil\ (\lambda x.x)\ v$ in time $O(n)$.

4 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 [8, 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 [1,7] or denotational semantics [9]. 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 overhead 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 4.1). We first make decomposition and plugging explicit, indexing this specification with 1 (Section 4.2). Then, we present a version that explicitly uses a refocus function, indexing this specification with 2 (Section 4.3). Using the construction of

Section 2, we then define an efficient version of the refocus function.

Terms, values, and contexts are defined as follows.

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

4.1 The original specification

Definition 4.1 [Sabry and Felleisen, 1993] The following CPS transformation uses three mutually recursive functions: $\mathcal{C}_k^0$ to transform terms, Φ^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.

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

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

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

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

4.2 Making decomposition and plugging explicit

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

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

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 $\widetilde{\mathcal{C}}_k^1$.

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

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

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

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

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 re-express the CPS transformation using a refocusing function combining *decompose* and *plug*.

4.3 Refocusing

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

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

The CPS transformation can then be expressed as follows.

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

$$\begin{aligned}
\Phi^2 &: \mathit{Values} \to \Lambda \\
\Phi^2[\![x]\!] &= x \\
\Phi^2[\![\lambda x.M]\!] &= \lambda k.\lambda x.\mathcal{C}_k^2 \circ \mathit{refocus}[\![[\ ],\ M]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{K}_k^2 &: \mathit{EvCont} \to \Lambda \\
\mathcal{K}_k^2[\![[\ ]]\!] &= k \\
\mathcal{K}_k^2[\![E[x\ [\ ]]]\!] &= x\ \mathcal{K}_k^2[\![E]\!]
\end{aligned}$$

$$\mathcal{K}_k^2[\![E[(\lambda x.M)\ [\,]]]\!] = \lambda x.\mathcal{C}_k^2 \circ \mathit{refocus}[\![E,\ M]\!]$$
$$\mathcal{K}_k^2[\![E[[\,]\ M]]\!] = \lambda u_i.\mathcal{C}_k^2 \circ \mathit{refocus}[\![E,\ u_i\ M]\!]$$

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 $\mathit{decompose} \circ \mathit{plug}$.

4.4 Efficiency

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

$$\mathit{refocus}[\![E,\ M_0\ M_1]\!] = \mathit{refocus}[\![E[[\,]\ M_1],\ M_0]\!]$$
$$\mathit{refocus}[\![E[[\,]\ M_1],\ V_0]\!] = \mathit{refocus}[\![E[V_0\ [\,]],\ M_1]\!]$$
$$\mathit{refocus}[\![E[V_0\ [\,]],\ V_1]\!] = [\![E,\ V_0\ V_1]\!]$$

That this function is extensionally equivalent to $\mathit{decompose} \circ \mathit{plug}$ and more efficient follows from the more general proof 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 $(\mathcal{C}_k^2,\ \Phi^2,\ \mathcal{K}_k^2)$ produces the same compact terms as $(\mathcal{C}_k^0,\ \Phi^0,\ \mathcal{K}_k^0)$, but it operates in linear time, or more precisely, in one pass, over its input.

5 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 linear-time overhead for each reduction step. We have illustrated the result with an interpreter for the call-by-value λ-calculus and by mechanically turning a quadratic-time program transformation into a program transformation that operates in one pass. In both cases, contexts are best represented inside out.

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

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. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[4] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. Technical Report BRICS RS-01-31, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, July 2001. 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.

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

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

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

[9] 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.

[10] 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/`.

[11] 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.