Three Syntactic Theories for Combinatory Graph Reduction

Olivier Danvy and Ian Zerny

Department of Computer Science, Aarhus University
Aabogade 34, DK-8200 Aarhus N, Denmark
{danvy,zerny}@cs.au.dk

Abstract. We present a purely syntactic theory of graph reduction for the canonical combinators S, K, and I, where graph vertices are represented with evaluation contexts and let expressions. We express this syntactic theory as a reduction semantics. We then factor out the introduction of let expressions to denote as many graph vertices as possible *upfront* instead of *on demand*, resulting in a second syntactic theory, this one of term graphs in the sense of Barendregt et al. We then interpret let expressions as operations over a global store (thus shifting, in Strachey's words, from denotable entities to storable entities), resulting in a third syntactic theory, which we express as a reduction semantics. This store-based reduction semantics corresponds to a store-based abstract machine whose architecture *coincides with that of Turner's original reduction machine.* The three syntactic theories presented here therefore properly account for combinatory graph reduction As We Know It.

1 Introduction

Fifteen years ago [3, 4, 24], Ariola, Felleisen, Maraist, Odersky and Wadler presented a purely syntactic theory for the call-by-need λ-calculus. In retrospect, their key insight was to syntactically represent 'def-use chains' for identifiers with evaluation contexts. For example, here is one of their contraction rules:

$$(\lambda x.E[x])\, v \rightarrow (\lambda x.E[v])\, v$$

In the left-hand side, an identifier, x, occurs (i.e., is 'used') in the eye of an evaluation context, E: its denotation is therefore needed.[1] This identifier is declared (i.e., is 'defined') in a λ-abstraction that is applied to a (syntactic) value v. In the right-hand side, v hygienically replaces x in the eye of E. There may be other occurrences of x in E: if another such one is needed later in the reduction sequence, this contraction rule will intervene again—it implements memoization.

In this article, we take a next logical step and present a purely syntactic theory of graph reduction for the canonical combinators S, K and I. Our key technique is to syntactically represent def-use chains for graph vertices using evaluation contexts and let expressions declaring unique references. For example,

[1] The notation $E[t]$ stands for a term that uniquely decomposes into a reduction context, E, and a subterm, t.

the K combinator is traditionally specified as $K\,t_1\,t_2 = t_1$, for any terms t_1 and t_2, a specification that does not account for sharing of subterms before and after contraction. In contrast, our specification does account for sharing, algebraically:

$$\begin{array}{ll}
\mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\,t_1\ \mathsf{in} & \to \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_3 = t_1\ \mathsf{in} \\
\qquad\qquad E_1[\mathsf{let}\ x_0 = x_1\,t_0\ \mathsf{in} & \qquad\qquad \mathsf{let}\ x_1 = x_2\,x_3\ \mathsf{in} \\
\qquad\qquad\quad E_0[x_0]]] & \qquad\qquad E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in} \\
 & \qquad\qquad\quad E_0[x_0]]] \\
 & \text{where } x_3 \text{ is fresh}
\end{array}$$

This contraction rule should be read inside-out. In the left-hand side, i.e., in the redex, a reference, x_0, occurs in the eye of an evaluation context: its denotation is therefore needed. The definiens of x_0 is the application of a second reference, x_1, to a term t_0: the denotation of x_1 is therefore also needed. The definiens of x_1 is the application of a third reference, x_2, to a term t_1: the denotation of x_2 is therefore also needed. The definiens of x_2 is the K combinator. In the right-hand side, i.e., in the contractum, a fresh (and thus unique) reference, x_3, is introduced to denote t_1: it replaces the application of x_1 to t_0. Reducing the K combinator is thus achieved (1) by creating a fresh reference to t_1 to share any subsequent reduction in t_1,[2] and (2) by replacing the reference to the application $x_1\,t_0$ by x_3. There may be other occurrences of x_0 in E_0: if another such one is needed later in the reduction sequence, this contraction rule for K will not intervene again—its result has been memoized.

In Section 2, we fully specify our syntactic theory of combinatory graph reduction as a reduction semantics, and we then apply the first author's programme [10, 11] to derive the first storeless abstract machine for combinatory graph reduction, in a way similar to what we recently did for the call-by-need λ-calculus [13].

Our syntactic theory introduces let expressions for applications on demand. In Section 3, we preprocess source terms by introducing let expressions upfront for all source applications, and we present the corresponding reduction semantics and storeless abstract machine. The preprocessed terms essentially coincide with the term graphs of Barendregt et al. [7].

In Section 4, we map the explicit representation of graph vertices as let headers to implicit references in a global store. Again, we present the corresponding store-based reduction semantics and store-based abstract machine. This store-based abstract machine essentially coincides with Turner's original graph-reduction machine [30]. This coincidence provides an independent, objective bridge between the modern theory of combinatory graph reduction and its classical computational practice.

Prerequisites and notations: We expect an elementary awareness of the S, K and I combinators and how combinatory terms can be reduced to head normal form, either in principle (as a formal property in Combinatory Logic [6]) or in practice (as a stack-based graph-reduction machine [26, 30]). We also assume a basic familiarity with the format of reduction semantics and abstract machines

[2] References ensure sharing of subterms: they are uniquely defined, but they can have many uses. References can be freely duplicated, but what they refer to, i.e., their denotation, is not duplicated and is thus shared.

as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [11]; and with the concept of term graphs, as pedagogically presented in Blom's PhD dissertation [8]. In particular, we use the terms 'reduction context' and 'evaluation context' interchangeably.

For a notion of reduction $\mathcal{R}$ defined with a set of contraction rules, we define the contraction of a redex r into a contractum t as $(r, t) \in \mathcal{R}$. We use $\rightarrow_{\mathcal{R}}$ for one-step $\mathcal{R}$-reduction, and $\twoheadrightarrow_{\mathcal{R}}$ for the transitive-reflexive closure of $\rightarrow_{\mathcal{R}}$. A term t is in $\mathcal{R}$-normal form ($\mathcal{R}$-nf) if t does not contain a redex of $\mathcal{R}$.

Pictorial overview:

	terms	graphs	store
reduction semantics	Section 2.1	Section 3.1	Section 4.1
abstract machine	Section 2.2	Section 3.2	Section 4.2

2 Two inter-derivable semantic artifacts for storeless combinatory graph reduction

Our starting point is the following grammar of combinatory terms:

$$t ::= I \mid K \mid S \mid t\,t$$

So a combinatory term is a combinator or a combination, i.e., the application of a term to another term.

We embed this grammar into a grammar of terms where sub-terms can be referred to through let expressions and where a program p is an initial term t denoted by a reference whose denotation is needed:

$$\begin{aligned} t &::= I \mid K \mid S \mid t\,t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid x \\ p &::= \mathsf{let}\ x = t\ \mathsf{in}\ x \end{aligned}$$

In this grammar, a term is a combinatory term (i.e., a combinator or a combination), the declaration of a reference to a term in another term, or the occurrence of a declared reference. This grammar of terms is closed under contraction.

In our experience, however, there is a better fit for the contraction rules, namely the following sub-grammar where a denotable term is an original combinatory term or a term that generalizes the original program into declarations nested around a declared reference:

$$\begin{aligned} p &::= \mathsf{let}\ x = d\ \mathsf{in}\ x \\ d &::= I \mid K \mid S \mid d\,d \mid t \\ t &::= \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \end{aligned}$$

This grammar of terms excludes terms with let expressions whose body is a combinator or a combination. It is closed under contraction.

At this point, it would be tempting to define reduction contexts to reflect how references are needed in the reduction process:

$$\textsf{Reduction Context} \ni E ::= [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\,d\ \mathsf{in}\ E[x]$$

The constructor "$\mathsf{let}\ x = d\ \mathsf{in}\ E$" accounts for the recursive search for the innermost reference in a term. The constructor "$\mathsf{let}\ x = E\,d\ \mathsf{in}\ E[x]$" accounts for the need of intermediate references.

In our experience, however, there is a better grammatical fit for contexts, namely one which separates the search for the innermost reference in a term and the subsequent construction that links needed references to their declaration, i.e., usage to definition. The former gives rise to delimited reduction contexts and the latter to def-use chains:

$$\begin{array}{rcl}\mathsf{Reduction\ Context} \ni E & ::= & [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\ \mathsf{Def\text{-}use\ Chain} \ni C & ::= & [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]\end{array}$$

2.1 A reduction semantics

Here is the full definition of the syntax:

$$\begin{array}{rcl}\mathsf{Program} \ni p & ::= & \mathsf{let}\ x = d\ \mathsf{in}\ x \\ \mathsf{Denotable\ Term} \ni d & ::= & I \mid K \mid S \mid d\,d \mid t \\ \mathsf{Term} \ni t & ::= & \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \\ \mathsf{Reduction\ Context} \ni E & ::= & [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\ \mathsf{Def\text{-}use\ Chain} \ni C & ::= & [\,] \mid \mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]\end{array}$$

Reduction contexts reflect the recursive search for the innermost reference in a term. While returning from this search, def-use chains are constructed to connect each reference whose denotation is needed with its declaration site. We abbreviate $\mathsf{let}\ x = [\,]\,d\ \mathsf{in}\ E[C[x]]$ as $(x,\, d,\, E) \cdot C$ and we write $\mathit{\Pi}_0^{i=n}(x_i,\, d_i,\, E_i) \cdot C$ as short hand for $(x_n,\, d_n,\, E_n) \cdot \ldots \cdot (x_0,\, d_0,\, E_0) \cdot C$, and $|C|$ for the length of C, so that $|\mathit{\Pi}_0^{i=n}(x_i,\, d_i,\, E_i) \cdot [\,]| = n + 1$.

Axioms (i.e., contraction rules): Figure 1 displays the axioms. Each of (I), (K) and (S) is much as described in Section 1, with the addition of the inner def-use chain: it carries out a particular rearrangement while preserving sharing through common references.[3]

In the left-hand side of (comb), a reference, x_0, occurs in the eye of the current def-use chain, C: its denotation is therefore needed. Its definiens is a combination of two denotable terms, d_0 and d_1. In the right-hand side, a fresh reference, x_1, is introduced to denote d_0. This fresh reference extends the current def-use chain for d_0, thereby ensuring that any subsequent reduction in d_0 is shared.[4] This specific choice of d_0 ensures that any redex found in a subsequent search will be on the left of this combination, thereby enforcing left-most reduction.

The axiom (assoc) is used to flatten let expressions,[5] and the axiom (ref) to resolve indirect references.

[3] On the right-hand side of (I), (K) and (S), we have kept $E_0[C[x_0]]$ in order to highlight each particular rearrangement. It would be simple to "optimize" these right-hand sides by taking advantage of [a subsequent use of] (ref) and (comb), so that, e.g., the right-hand side of (K) contains $E_0[C[x_3]]$ instead.

[4] If d_0 is already a reference, it is already part of a def-use chain and no contraction need take place: $\mathsf{let}\ x_0 = x\,d_1\ \mathsf{in}\ E_0[C[x_0]]$ is not a redex.

[5] There is no need for the condition "x_1 does not occur free in E_0" since each reference is unique.

$$
\begin{array}{rrcl}
(I) & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & \rightarrow & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = d_0\ \mathsf{in} \\
 & E_0[C[x_0]]] & & \qquad\qquad E_0[C[x_0]]] \\
(K) & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & \rightarrow & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_3 = d_1\ \mathsf{in} \\
 & E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & & \qquad\qquad \mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in} \\
 & E_0[C[x_0]]]] & & \qquad\qquad E_1[\mathsf{let}\ x_0 = x_3\ \mathsf{in} \\
 & & & \qquad\qquad\quad E_0[C[x_0]]]] \\
 & & & \text{where } x_3 \text{ is fresh} \\
(S) & \mathsf{let}\ x_3 = S\ \mathsf{in} & \rightarrow & \mathsf{let}\ x_3 = S\ \mathsf{in} \\
 & E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} & & E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\
 & E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} & & \quad \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\
 & E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} & & \quad E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\
 & E_0[C[x_0]]]]] & & \qquad \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\
 & & & \qquad E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\
 & & & \qquad\quad \mathsf{let}\ x_0 = (x_4\, x_6)\, (x_5\, x_6)\ \mathsf{in} \\
 & & & \qquad\quad E_0[C[x_0]]]]] \\
 & & & \text{where } x_4,\ x_5 \text{ and } x_6 \text{ are fresh} \\
(comb) & \mathsf{let}\ x_0 = d_0\, d_1\ \mathsf{in}\ E_0[C[x_0]] & \rightarrow & \mathsf{let}\ x_1 = d_0\ \mathsf{in}\ \mathsf{let}\ x_0 = x_1\, d_1\ \mathsf{in}\ E_0[C[x_0]] \\
 & & & \text{where } d_0 \text{ is not a reference} \\
 & & & \quad \text{and } x_1 \text{ is fresh} \\
(assoc) & \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]] & \rightarrow & \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ \mathsf{let}\ x_0 = t_0\ \mathsf{in}\ E_0[C[x_0]] \\
(ref) & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \rightarrow & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]
\end{array}
$$

Fig. 1. Reduction semantics for combinatory graph reduction: axioms

Inside-out recomposition of a reduction context with a term:

$$
\frac{}{\langle [\,],\ t\rangle_{io} \Uparrow_{\text{rec}} t}
\qquad\qquad
\frac{\langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{io} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{io} \Uparrow_{\text{rec}} t'}
$$

Outside-in recomposition of a reduction context with a term:

$$
\frac{}{\langle [\,],\ t\rangle_{oi} \Uparrow_{\text{rec}} t}
\qquad\qquad
\frac{\langle E,\ t\rangle_{oi} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{oi} \Uparrow_{\text{rec}} \mathsf{let}\ x = d\ \mathsf{in}\ t'}
$$

Recomposition of a def-use chain:

$$
\frac{}{\langle [\,],\ x\rangle_{chain} \Uparrow_{\text{rec}} x}
\qquad\qquad
\frac{\langle C,\ x'\rangle_{chain} \Uparrow_{\text{rec}} t \qquad \langle E,\ t\rangle_{oi} \Uparrow_{\text{rec}} t'}{\langle \mathsf{let}\ x' = [\,]\, d\ \mathsf{in}\ E[C[x']],\ x\rangle_{chain} \Uparrow_{\text{rec}} \mathsf{let}\ x' = x\, d\ \mathsf{in}\ t'}
$$

Fig. 2. Reduction semantics for combinatory graph reduction: recompositions

The corresponding notion of reduction is $\mathcal{T}$:

$$\mathcal{T} = (I) \cup (K) \cup (S) \cup (comb) \cup (assoc) \cup (ref)$$

Reduction strategy: The reduction strategy is left-most outermost (and thus not optimal [23]): the def-use chains force us to only consider the outermost combination, and $(comb)$ ensures that this outermost combination is the left-most one.

Recompositions: Figure 2 displays the recompositions of a reduction context with a term, and of a def-use chain with a reference:

Definition 1 (inside-out recomposition of contexts). *A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E,\ t\rangle_{io} \Uparrow_{\text{rec}} t'$ holds. (See Figure 2 and also Footnote 1 for the notation $E[t]$.)*

Definition 2 (outside-in recomposition of contexts). *A context E is recomposed around a term t into a term $t' = E[t]$ whenever $\langle E,\, t\rangle_{oi} \Uparrow_{\text{rec}} t'$ holds. (See Figure 2.)*

Outside-in recomposition of contexts is used as an auxiliary judgment in the recomposition of def-use chains:

Definition 3 (recomposition of def-use chains). *A def-use chain C is recomposed around a reference x into a term $t = C[x]$ whenever $\langle C,\, x\rangle_{chain} \Uparrow_{\text{rec}} t$ holds.*[6] *(See Figure 2.)*

Decomposition: Decomposition implements the reduction strategy by searching for a redex and its reduction context in a term. Figure 3 displays this search as a transition system:

term-**transitions:** Given a term, we recursively dive into the bodies of its nested let expressions until its innermost reference x, which is therefore needed.

cont-**transitions:** Having found a reference x that is needed, we backtrack in search of its declaration, incrementally constructing a def-use chain for it.[7] If we do not find any redex, the term is in $\mathcal{T}$-normal form.

den-**transitions:** Having found the declaration of the reference that was needed, we check whether we have also found a redex and thus a decomposition. Otherwise, a combinator is not fully applied or a new reference is needed. We then resume a *cont*-transition, either on our way to a $\mathcal{T}$-normal form or extending the current def-use chain for this new reference.

Definition 4 (decomposition). *For any t,*

$$\langle t,\, [\,]\rangle_{term} \downarrow^{*}_{\text{dec}} \begin{cases} \langle t\rangle_{nf} & \textit{if } t \in \mathcal{T}\textit{-nf} \\ \langle E,\, r\rangle_{dec} & \textit{otherwise} \end{cases}$$

where r is the left-most outermost redex in t and E is its reduction context.

The transition system implementing decomposition can be seen as a big-step abstract machine [12]. As repeatedly pointed out in the first author's lecture notes at AFP 2008 [11], such a big-step abstract machine is often in defunctionalized form—as is the case here. In the present case, it can be refunctionalized into a function over source terms which is compositional. Ergo, it is expressible as a catamorphism over source terms. Further, the parameters of this catamorphism are total functions. Therefore, the decomposition function is total. It yields either the given term if this term is in $\mathcal{T}$-normal form, or its left-most outermost redex and the corresponding reduction context.

[6] As already pointed out in Footnote 1, the notation $C[x]$ stands for a term that uniquely decomposes into a def-use chain, C, and a reference, x.

[7] There is no transition for $\langle [\,],\, (x_0,\, E_0,\, C)\rangle_{cont}$ because all references are declared.

$$
\begin{array}{rl}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} \downarrow_{\mathrm{dec}} & \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} \downarrow_{\mathrm{dec}} & \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
\\
\langle [\,],\ t\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} \downarrow_{\mathrm{dec}} & \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
& \text{where } x \neq x_0 \\
\\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\,d_0\ \mathsf{in}\ E_0[C[x_0]]]\rangle_{dec} \\
& \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\,d_1\ \mathsf{in} \\
& \qquad\qquad\qquad E_1[\mathsf{let}\ x_0 = x_1\,d_0\ \mathsf{in} \\
& \qquad\qquad\qquad\quad E_0[C[x_0]]]]\rangle_{dec} \\
& \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_3 = S\ \mathsf{in}\ E_3[\mathsf{let}\ x_2 = x_3\,d_2\ \mathsf{in} \\
& \qquad\qquad\qquad E_2[\mathsf{let}\ x_1 = x_2\,d_1\ \mathsf{in} \\
& \qquad\qquad\qquad\quad E_1[\mathsf{let}\ x_0 = x_1\,d_0\ \mathsf{in} \\
& \qquad\qquad\qquad\qquad E_0[C[x_0]]]]]\rangle_{dec} \\
& \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
& \text{where } d_0 = I \ \text{ and } |C| < 1 \\
& \quad\ \text{or } d_0 = K \text{ and } |C| < 2 \\
& \quad\ \text{or } d_0 = S \ \text{ and } |C| < 3 \\
& \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\,d_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ (x_1,\ [\,],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ d_0\,d_1,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = d_0\,d_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
& \text{where } d_0 \text{ is not a reference} \\
& \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = (\mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0)\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
& \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} \downarrow_{\mathrm{dec}} & \langle E,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]]\rangle_{dec} \\
& \text{where } \langle C,\ x_0\rangle_{chain} \Uparrow_{\mathrm{rec}} C[x_0] \\
& \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\mathrm{rec}} E_0[C[x_0]]
\end{array}
$$

Fig. 3. Reduction semantics for combinatory graph reduction: decomposition

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{T}$-normal form proceeds as follows: (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 5 (standard one-step reduction). *For any t, $t \mapsto_{\mathcal{T}} t''$ if*

$$\langle t,\ [\,]\rangle_{term} \downarrow^*_{\mathrm{dec}} \langle E,\ r\rangle_{dec}\ \wedge\ (r, t') \in \mathcal{T}\ \wedge\ \langle E,\ t'\rangle_{io} \Uparrow_{\mathrm{rec}} t''$$

One-step reduction is a partial function because the given term may already be in $\mathcal{T}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 6 (standard reduction-based evaluation). *For any program p,*

$$p \mapsto^*_{\mathcal{T}} t \ \wedge \ t \in \mathcal{T}\text{-}nf$$

Evaluation is a partial function because it may diverge.

Most of the time, decomposition and recomposition(s) are kept implicit in published reduction semantics. We however observe that what was kept implicit is then progressively revealed as, e.g., one constructs an abstract machine to implement evaluation [16]. We believe that it is better to completely spell out reduction semantics upfront, because one is then in position to systematically calculate the corresponding abstract machines [11], as illustrated in the next section for syntactic graph reduction.

2.2 A storeless abstract machine

Reduction-based evaluation, as defined in Section 2.1, is inefficient because of its repeated decompositions and recompositions that construct each successive term in a reduction sequence. Refocusing [15] deforests these intermediate terms, and is defined very simply as continuing decomposition with the contractum and its reduction context. The reduction semantics of Section 2.1 satisfies the formal requirements for refocusing [15] and so its reduction-based evaluation function can be simplified into a reduction-free evaluation function that does not construct each successive term in reduction sequences. Reflecting the structure of the decomposition function of Figure 3, the result is an abstract machine whose 'corridor' transitions can be compressed into the abstract machine displayed in Figure 4:

***term*-transitions:** The *term*-transitions are the same as for decomposition.
***cont*-transitions:** The *cont*-transitions are the same as for decomposition.
***den*-transitions:** Having found the declaration of the reference that was needed, we check whether we have also found a redex. If so, we contract it and continue with a new needed reference. Otherwise, the *den*-transitions are the same as for decomposition.

This abstract machine uses the following two compositions of evaluation contexts:

Definition 7 (composition of evaluation contexts). *Two contexts that were constructed outside in are composed into an outside-in context as follows:*

$$\begin{aligned} [\,] \circ_{oi} E &= E \\ (\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{oi} E &= \mathsf{let}\ x = d\ \mathsf{in}\ (E' \circ_{oi} E) \end{aligned}$$

This composition function is associative.

$$\begin{aligned}
\langle \mathsf{let}\ x = d\ \mathsf{in}\ t,\ E\rangle_{term} &\rightarrow_{\text{step}} \langle t,\ \mathsf{let}\ x = d\ \mathsf{in}\ E\rangle_{term} \\
\langle x,\ E\rangle_{term} &\rightarrow_{\text{step}} \langle E,\ (x,\ [\,],\ [\,])\rangle_{cont} \\
\\
\langle [\,],\ t\rangle_{cont} &\rightarrow_{\text{step}} \langle t\rangle_{nf} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ t\rangle_{cont} &\rightarrow_{\text{step}} \langle E,\ \mathsf{let}\ x = d\ \mathsf{in}\ t\rangle_{cont} \\
\\
\langle \mathsf{let}\ x_0 = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\rightarrow_{\text{step}} \langle x_0,\ d,\ E_0,\ C,\ E\rangle_{den} \\
\langle \mathsf{let}\ x = d\ \mathsf{in}\ E,\ (x_0,\ E_0,\ C)\rangle_{cont} &\rightarrow_{\text{step}} \langle E,\ (x_0,\ \mathsf{let}\ x = d\ \mathsf{in}\ E_0,\ C)\rangle_{cont} \\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } x \neq x_0 \\
\\
\langle x_1,\ I,\ E_1,\ \Pi_0^{i=0}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle x_0,\ d_0,\ E_0,\ C,\ E_1 \circ_{io} \mathsf{let}\ x_1 = I\ \mathsf{in}\ E\rangle_{den} \\
\langle x_2,\ K,\ E_2,\ \Pi_0^{i=1}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle x_3,\ d_1,\ E',\ C,\ E_2 \circ_{io} \mathsf{let}\ x_2 = K\ \mathsf{in}\ E\rangle_{den} \\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } E' = (\mathsf{let}\ x_1 = x_2\, x_3\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_0 = x_3\ \mathsf{in}\ E_0) \\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } x_3 \text{ is fresh} \\
\langle x_3,\ S,\ E_3,\ \Pi_0^{i=2}(x_i,\ d_i,\ E_i)\cdot C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle x_4,\ d_2,\ E',\ C',\ E_3 \circ_{io} \mathsf{let}\ x_3 = S\ \mathsf{in}\ E\rangle_{den} \\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } E' = (\mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in}\ E_2) \circ_{oi} (\mathsf{let}\ x_5 = d_1\ \mathsf{in}\ \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in}\ E_1) \circ_{oi} (\mathsf{let}\ x_6 = d_0\ \mathsf{in}\ [\,]) \\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } C' = (x_7,\ x_6,\ [\,])\cdot(x_0,\ x_5\, x_6,\ E_0)\cdot C \\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } x_4, x_5, x_6 \text{ and } x_7 \text{ are fresh} \\
\langle x_0,\ d_0,\ E_0,\ C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle E,\ \mathsf{let}\ x_0 = d_0\ \mathsf{in}\ E_0[C[x_0]]\rangle_{cont} \\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } d_0 = I \text{ and } |C| < 1,\ d_0 = K \text{ and } |C| < 2, \text{ or } d_0 = S \text{ and } |C| < 3 \\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } \langle C,\ x_0\rangle_{chain} \Uparrow_{\text{rec}} C[x_0] \\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } \langle E_0,\ C[x_0]\rangle_{oi} \Uparrow_{\text{rec}} E_0[C[x_0]] \\
\langle x_0,\ x_1\, d_0,\ E_0,\ C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle E,\ (x_1,\ [\,],\ (x_0,\ d_0,\ E_0)\cdot C)\rangle_{cont} \\
\langle x_0,\ d_0\, d_1,\ E_0,\ C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle x_1,\ d_0,\ [\,],\ (x_0,\ d_1,\ E_0)\cdot C,\ E\rangle_{den} \\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } d_0 \text{ is not a reference and } x_1 \text{ is fresh} \\
\langle x_0,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ t_0,\ E_0,\ C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle x_0,\ t_0,\ E_0,\ C,\ \mathsf{let}\ x_1 = d_1\ \mathsf{in}\ E\rangle_{den} \\
\langle x_0,\ x_1,\ E_0,\ C,\ E\rangle_{den} &\rightarrow_{\text{step}} \langle E,\ (x_1,\ \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0,\ C)\rangle_{cont}
\end{aligned}$$

Fig. 4. Storeless abstract machine for combinatory graph reduction

Definition 8 (mixed composition of evaluation contexts). *A context that was constructed inside out is composed with a context that was constructed outside in as follows:*

$$\begin{aligned}[\,] \circ_{io} E &= E \\ (\mathsf{let}\ x = d\ \mathsf{in}\ E') \circ_{io} E &= E' \circ_{io} \mathsf{let}\ x = d\ \mathsf{in}\ E\end{aligned}$$

The resulting context is constructed outside in.

Proposition 1 (full correctness). *For any program p,*

$$p \mapsto^*_{\mathcal{T}} t \ \wedge\ t \in \mathcal{T}\text{-}nf \ \Leftrightarrow\ \langle p, [\,]\rangle_{term} \rightarrow^*_{\text{step}} \langle t\rangle_{nf}$$

2.3 Summary and conclusion

Starting from a completely spelled out reduction semantics for combinatory terms with sharing, we have mechanically derived a storeless abstract machine. These two semantic artifacts share the same syntactic representations.

3 Preprocessing combinatory terms into term graphs

In Section 2, references are declared on demand in the reduction sequence. In this section, we factor out all possible such declarations for combinations into a preprocessing phase.

We start by restating the $(comb)$ and (S) axioms:

$$\begin{array}{lll}(comb') & \mathsf{let}\ x = d_0\, d_1\ \mathsf{in}\ E[C[x]] & \rightarrow\ \mathsf{let}\ x_0 = d_0\ \mathsf{in} \\ & & \ \mathsf{let}\ x_1 = d_1\ \mathsf{in} \\ & & \ \mathsf{let}\ x = x_0\, x_1\ \mathsf{in}\ E[C[x]] \\ & & \ \text{where } d_0 \text{ is not a reference} \\ & & \ \text{and } x_1 \text{ and } x_2 \text{ are fresh}\end{array}$$

In contrast to the $(comb)$ axiom, the restated axiom $(comb')$ declares references for both sides of a combination. Unlike in Section 2, there can thus be references to denotable terms whose denotation is not needed. In the same spirit, we restate the (S) axiom so that it declares references to both sides of any combination in the contractum:

$$\begin{array}{lll}(S') & \begin{array}{l}\mathsf{let}\ x_3 = S\ \mathsf{in} \\ \quad E_3[\mathsf{let}\ x_2 = x_3\, d_2\ \mathsf{in} \\ \quad\quad E_2[\mathsf{let}\ x_1 = x_2\, d_1\ \mathsf{in} \\ \quad\quad\quad E_1[\mathsf{let}\ x_0 = x_1\, d_0\ \mathsf{in} \\ \quad\quad\quad\quad E_0[C[x_0]]]]]\end{array} & \rightarrow\ \begin{array}{l}\mathsf{let}\ x_3 = S\ \mathsf{in} \\ E_3[\mathsf{let}\ x_4 = d_2\ \mathsf{in} \\ \quad \mathsf{let}\ x_2 = x_3\, x_4\ \mathsf{in} \\ \quad E_2[\mathsf{let}\ x_5 = d_1\ \mathsf{in} \\ \quad\quad \mathsf{let}\ x_1 = x_2\, x_5\ \mathsf{in} \\ \quad\quad E_1[\mathsf{let}\ x_6 = d_0\ \mathsf{in} \\ \quad\quad\quad \mathsf{let}\ x_7 = x_4\, x_6\ \mathsf{in} \\ \quad\quad\quad \mathsf{let}\ x_8 = x_5\, x_6\ \mathsf{in} \\ \quad\quad\quad \mathsf{let}\ x_0 = x_7\, x_8\ \mathsf{in} \\ \quad\quad\quad E_0[C[x_0]]]]] \\ \text{where } x_4,\ x_5,\ x_6,\ x_7 \text{ and } x_8 \text{ are fresh}\end{array}\end{array}$$

The corresponding notion of reduction is $\mathcal{T}'$:

$$\mathcal{T}' = (I) \cup (K) \cup (S') \cup (comb') \cup (assoc) \cup (ref)$$

We split $\mathcal{T}'$ into two: a compile-time notion of reduction $\mathcal{C}$ and a run-time notion of reduction $\mathcal{R}$:

$$\begin{aligned} \mathcal{C} &= (assoc) \cup (comb') \\ \mathcal{R} &= (I) \cup (K) \cup (S') \cup (ref) \end{aligned}$$

Each of $\mathcal{C}$ and $\mathcal{R}$ contain only left-linear axioms and no critical pairs: they are orthogonal and thus confluent [21]. Furthermore, $\mathcal{C}$ is strongly normalizing.

The $\mathcal{C}$-normal forms are contained within the following sub-grammar of terms:

$$\begin{aligned} \textsf{Denotable Term} \ni d &::= I \mid K \mid S \mid x\,x \mid x \\ \textsf{Term} \ni t &::= \textsf{let}\ x = d\ \textsf{in}\ t \mid x \end{aligned}$$

In this grammar, only combinations of references are admitted and furthermore let expressions are completely flattened, in a way reminiscent of monadic normal forms [18, 19].

Proposition 2 (Preprocessing). *If $t \twoheadrightarrow_{\mathcal{T}'} t' \ \wedge\ t' \in \mathcal{T}'$-nf then $\exists t'' \in \mathcal{C}$-nf . $t \twoheadrightarrow_{\mathcal{C}} t'' \twoheadrightarrow_{\mathcal{R}} t'$.*

Proof. Strong normalization of $\mathcal{C}$ ensures the existence of t''. Confluence of $\mathcal{T}'$ gives $t'' \twoheadrightarrow_{\mathcal{T}'} t'$. $\mathcal{R}$ is closed over $\mathcal{C}$-nf. Thus, only $\mathcal{R}$ is needed in the reduction $t'' \twoheadrightarrow_{\mathcal{R}} t'$.

We observe that a preprocessed term is a syntactic representation of a graph where every denotable term has been declared with a reference. Indeed it is straightforward to interpret preprocessed terms as term graphs:

Definition 9 (term graphs [7, Definition 4.2.6]). *A* term graph *is a tuple $(N, lab, succ, r)$ over a set of function symbols F where*

- *N is a set of unique node identifiers;*
- *$lab : N \rightarrow F$ is a labeling function mapping nodes to function symbols;*
- *$succ : N \rightarrow N^n$ is a successor function mapping nodes to an n-tuple of successor nodes for some natural number n; and*
- *$r \in N$ is the root of the term graph.*

Definition 10 (interpreting combinatory terms as term graphs). *For any preprocessed term, its term graph over the function symbols $F = \{I, K, S, A\}$ is defined as follows:*

- *N is the set of declared references in the term.*
- *lab is defined on the definiens of a reference: for a combinator, it yields the respective function symbols I, K or S; for a combination, it yields the application symbol A; and for a reference, it yields the result of applying lab to this reference, which in effect acts as an alias for a node.*[8]

[8] This application is well behaved since terms are acyclic.

$$
\begin{array}{llcl}
(I) & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in}\ E_0[C[x_0]]] & \to & \mathsf{let}\ x_1 = I\ \mathsf{in}\ E_1[\mathsf{let}\ x_0 = y_0\ \mathsf{in}\ E_0[C[x_0]]] \\
(K) & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} & \to & \mathsf{let}\ x_2 = K\ \mathsf{in}\ E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
 & \quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} & & \quad E_1[\mathsf{let}\ x_0 = y_1\ \mathsf{in} \\
 & \quad\quad E_0[C[x_0]]]] & & \quad\quad E_0[C[x_0]]]] \\
(S) & \mathsf{let}\ x_3 = S\ \mathsf{in} & \to & \mathsf{let}\ x_3 = S\ \mathsf{in} \\
 & E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} & & E_3[\mathsf{let}\ x_2 = x_3\, y_2\ \mathsf{in} \\
 & \quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} & & \quad E_2[\mathsf{let}\ x_1 = x_2\, y_1\ \mathsf{in} \\
 & \quad\quad E_1[\mathsf{let}\ x_0 = x_1\, y_0\ \mathsf{in} & & \quad\quad E_1[\mathsf{let}\ x_4 = y_2\, y_0\ \mathsf{in} \\
 & \quad\quad\quad E_0[C[x_0]]]]] & & \quad\quad\quad \mathsf{let}\ x_5 = y_1\, y_0\ \mathsf{in} \\
 & & & \quad\quad\quad \mathsf{let}\ x_0 = x_4\, x_5\ \mathsf{in} \\
 & & & \quad\quad\quad E_0[C[x_0]]]]] \\
 & & & \text{where } x_4 \text{ and } x_5 \text{ are fresh} \\
(\mathit{ref}) & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_0]] & \to & \mathsf{let}\ x_0 = x_1\ \mathsf{in}\ E_0[C[x_1]]
\end{array}
$$

Fig. 5. Reduction semantics for combinatory graph reduction over preprocessed terms: axioms

- *succ is defined on the definiens of references: for a combination, it yields the corresponding pair of references, and for everything else, the empty tuple.*
- *r is the innermost reference of the term.*

Using the interpretation of Definition 10, we can translate the contraction rules over combinatory terms to graph-rewriting rules [7, Section 4.4.4]. The translation of (I), (K) and (S') gives us rewriting rules with the side condition that the redex is *rooted*, meaning that there is a path from the root of the graph to the redex, which is the case here and is manifested by its def-use chain. Terms in our language are therefore a restricted form of term graphs: directed acyclic graphs with an ordering hierarchy imposed on *succ* by the scoping of nested let expressions. (In his PhD thesis [8], Blom refers to this property of term graphs as 'horizontal sharing.')

3.1 A reduction semantics

Here is the full definition of the syntax after preprocessing terms into $\mathcal{C}$-nf:

$$
\begin{array}{rcl}
\mathsf{Program} \ni p & ::= & \mathsf{let}\ x = d\ \mathsf{in}\ x \\
\mathsf{Denotable\ Term} \ni d & ::= & I \mid K \mid S \mid x\,x \mid x \\
\mathsf{Term} \ni t & ::= & \mathsf{let}\ x = d\ \mathsf{in}\ t \mid x \\
\mathsf{Reduction\ Context} \ni E & ::= & [\,] \mid \mathsf{let}\ x = d\ \mathsf{in}\ E \\
\mathsf{Def\text{-}use\ Chain} \ni C & ::= & [\,] \mid \mathsf{let}\ x = [\,]\,x\ \mathsf{in}\ E[C[x]]
\end{array}
$$

Axioms: Figure 5 displays the axioms. Each of (I) and (K) is much as the corresponding axiom in Section 2, though we have specialized it with respect to the grammar of preprocessed terms. The (S) axiom is a further specialization of the (S') axiom. Specifically, since the right-hand side of any combination is known to be a reference, there is no need to introduce new let expressions to preserve sharing. As for the (ref) axiom, it is unchanged.

The notion of reduction on preprocessed terms is $\mathcal{G}$:

$$\mathcal{G} = (I) \cup (K) \cup (S) \cup (\mathit{ref})$$

Recompositions: The recompositions of contexts and def-use chains are defined in the same way as in Section 2.

Decomposition: Decomposition is much as in Section 2, though we have specialized it with respect to the grammar of preprocessed terms.

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{G}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 11 (standard one-step reduction). *For any t,*

$$t \mapsto_{\mathcal{G}} t'' \quad \textit{if} \quad \begin{cases} \langle t,\, [\,]\rangle_{term} \downarrow^{*}_{\text{dec}} \langle E,\, r\rangle_{dec} \\ \quad (r, t') \;\in\; \mathcal{G} \\ \langle E,\, t'\rangle_{io} \Uparrow_{\text{rec}} t'' \end{cases}$$

One-step reduction is a partial function because the given term may already be in $\mathcal{G}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 12 (standard reduction-based evaluation). *For any program p,*

$$p \mapsto^{*}_{\mathcal{G}} t \;\wedge\; t \in \mathcal{G}\text{-}nf$$

Evaluation is a partial function because it may diverge.

3.2 A storeless abstract machine

The abstract machine is calculated as in Section 2.2.

3.3 Summary and conclusion

Starting from a completely spelled out reduction semantics for preprocessed combinatory term graphs, we have derived a storeless abstract machine. As in Section 2, these two semantic artifacts share the same syntactic representations.

4 Store-based combinatory graph reduction

In this section, we no longer represent graph vertices explicitly as let expressions, but implicitly as locations in a store:

$$\begin{aligned} \textsf{Global Store} &\ni \sigma \\ \textsf{Location} &\ni x,\, y \end{aligned}$$

In the storeless accounts of Sections 2 and 3, let expressions declare references to denotable terms. In the store-based account presented in this section, a global store maps locations to storable terms. Given a store σ, a location x and a storable term s, we write $\sigma[x := s]$ for the store σ' such that $\sigma'(x) = s$ and $\sigma'(x') = \sigma(x')$ for $x' \neq x$.

The syntax of Section 3 therefore specializes as follows:

$$\begin{aligned}\textsf{Program} \ni p &::= (x, \sigma)\\ \textsf{Storable Term} \ni s &::= I \mid K \mid S \mid x\,x \mid x\\ \textsf{Ancestor Stack} \ni a &::= [\,] \mid (x, x) \cdot a\end{aligned}$$

A program now pairs the root of a graph in a store with this store. Denotable terms have been replaced by storable terms. Terms and reduction contexts have been replaced by references in the store. Def-use chains have specialized into what Turner calls *ancestor stacks*. We write $|a|$ for the height of an ancestor stack a.

Translating the preprocessed terms of Section 3 to store-based terms is straightforward:

Definition 13 (let-expression based to store-based).

$$\begin{aligned}[\![\textsf{let}\ x = d\ \textsf{in}\ t]\!]_\sigma &= [\![t]\!]_{\sigma[x := [\![d]\!]']}\\ [\![x]\!]_\sigma &= (x, \sigma)\end{aligned} \qquad \begin{aligned}[\![I]\!]' &= I\\ [\![K]\!]' &= K\\ [\![S]\!]' &= S\end{aligned} \qquad \begin{aligned}[\![x_0\,x_1]\!]' &= [\![x_0]\!]'\,[\![x_1]\!]'\\ [\![x]\!]' &= x\end{aligned}$$

This encoding maps the explicit representation of graph vertices as let expressions to implicit locations in a store.

4.1 A reduction semantics

Axioms: An axiom is of the form $(x, \sigma) \rightarrow (x', \sigma')$ where x and x' are the left and right root respectively. For such an axiom, a redex is a pair (x'', σ'') together with a renaming of locations defined by a structure-preserving function on storable terms, π, such that:

$$\pi(x) = x'' \text{ and } \forall y \in \mathit{dom}(\sigma)\ .\ \pi(\sigma(y)) = \sigma''(\pi(y)).$$

In words, the renaming must map the left root to the root of the redex, and any location in the store of the axiom must have a corresponding location in the store of the redex. As before, we write $\sigma[x := s]$ for a store mapping the location x to the storable term s.

The axioms are displayed in Figure 6. They assume that there is a path from the graph root to the redex root. This assumption mirrors the decomposition conditions in the axioms of Figure 5. Consequently, the reference axiom is split in two cases: one if the redex root is the graph root, corresponding to a decomposition into the empty def-use chain, and one if the redex root is not the graph root, corresponding to a decomposition into a non-empty def-use chain.

The notion of reduction on store-based terms is $\mathcal{H}$:

$$\mathcal{H} = (I) \cup (K) \cup (S) \cup (\mathit{ref}_1) \cup (\mathit{ref}_2)$$

$$
\begin{array}{lrcl}
(I) & (x_0,\ \sigma[x_1 := I][x_0 := x_1\,y_0]) & \to & (x_0,\ \sigma[x_1 := I][x_0 := y_0]) \\
(K) & (x_0,\ \sigma[x_2 := K][x_1 := x_2\,y_1][x_0 := x_1\,y_0]) & \to & (x_0,\ \sigma[x_2 := K][x_1 := x_2\,y_1][x_0 := y_1]) \\
(S) & (x_0,\ \sigma[x_3 := S] & \to & (x_0,\ \sigma[x_3 := S] \\
 & [x_2 := x_3\,y_2] & & \quad [x_2 := x_3\,y_2] \\
 & [x_1 := x_2\,y_1] & & \quad [x_1 := x_2\,y_1] \\
 & [x_0 := x_1\,y_0]) & & \quad [x_4 := y_2\,y_0] \\
 & & & \quad [x_5 := y_1\,y_0] \\
 & & & \quad [x_0 := x_4\,x_5]) \\
 & & & \text{where } x_4 \text{ and } x_5 \text{ are fresh} \\
(\mathit{ref}_1) & (x_0,\ \sigma[x_0 := x_1]) & \to & (x_1,\ \sigma[x_0 := x_1]) \\
 & & & \text{where } x_0 \text{ is the graph root} \\
(\mathit{ref}_2) & (x_0,\ \sigma[x_0 := x_1]) & \to & (x_1,\ \sigma[x_0 := x_1][x := x_1\,y]) \\
 & & & \text{where } x \text{ is reachable from the graph root} \\
 & & & \text{and } \sigma(x) = x_0\,y \text{ for some } y
\end{array}
$$

Fig. 6. Reduction semantics for store-based combinatory graph reduction: axioms

$$
\begin{array}{rcl}
\langle [\,],\ x,\ \sigma\rangle_{stack} & \downarrow_{\text{dec}} & \langle (x,\sigma)\rangle_{nf} \\
\langle (x_0,y_0)\cdot a,\ x_1,\ \sigma\rangle_{stack} & \downarrow_{\text{dec}} & \langle a,\ x_0,\ \sigma\rangle_{stack} \\
 & & \\
\langle x_1,\ I,\ (x_0,y_0)\cdot a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle a,\ (x_0,\ \sigma)\rangle_{dec} \\
\langle x_2,\ K,\ (x_1,y_1)\cdot(x_0,y_0)\cdot a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle a,\ (x_0,\ \sigma)\rangle_{dec} \\
\langle x_3,\ S,\ (x_2,y_2)\cdot(x_1,y_1)\cdot(x_0,y_0)\cdot a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle a,\ (x_0,\ \sigma)\rangle_{dec} \\
\langle x_0,\ s,\ a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle a,\ x_0,\ \sigma\rangle_{stack} \\
 & & \text{where } s = I \text{ and } |a| < 1, \\
 & & \quad \text{or } s = K \text{ and } |a| < 2 \\
 & & \quad \text{or } s = S \text{ and } |a| < 3 \\
\langle x_0,\ x_1\,y_0,\ a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle x_1,\ \sigma(x_1),\ (x_0,y_0)\cdot a,\ \sigma\rangle_{sto} \\
\langle x_0,\ x_1,\ a,\ \sigma\rangle_{sto} & \downarrow_{\text{dec}} & \langle a,\ (x_0,\ \sigma)\rangle_{dec}
\end{array}
$$

Fig. 7. Reduction semantics for store-based combinatory graph reduction: decomposition

Recomposition: The recomposition of an ancestor stack with a store-based term relocates the root of the graph:

$$
\frac{}{\langle [\,],\ x,\ \sigma\rangle_{stack} \Uparrow_{\text{rec}} (x,\sigma)}
\qquad
\frac{\langle a,\ x_0,\ \sigma\rangle_{stack} \Uparrow_{\text{rec}} (x',\sigma')}{\langle (x_0,y_0)\cdot a,\ x,\ \sigma\rangle_{stack} \Uparrow_{\text{rec}} (x',\sigma')}
$$

Decomposition: Decomposition is much as in Section 2 though we have further specialized it with respect to store-based graphs. The search previously done at return time is now done at call time. Starting from the root reference, x, we recursively search for a redex, incrementally constructing an ancestor stack for x. If we do not find any redex, the term is in $\mathcal{H}$-normal form. Figure 7 displays this search as a transition system:

Definition 14 (decomposition). *For any* (x, σ)*,*

$$\langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \downarrow^*_{\text{dec}} \begin{cases} \langle (x,\sigma)\rangle_{nf} & \textit{if } (x,\sigma) \in \mathcal{H}\text{-}nf \\ \langle a,\, (x',\sigma')\rangle_{dec} & \textit{otherwise} \end{cases}$$

where (x', σ') *is the left-most outermost redex in* (x, σ) *and* a *is the ancestor stack from* x *to* x'*.*

One-step reduction: The function of performing one contraction in a term that is not in $\mathcal{H}$-normal form is defined as (1) locating a redex and its context through a number of decomposition steps according to the reduction strategy, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 15 (standard one-step reduction). *For any* (x, σ)*,*

$$(x,\sigma) \mapsto_{\mathcal{H}} (x''',\sigma''') \quad \textit{iff} \quad \begin{cases} \langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \downarrow^*_{\text{dec}} \langle a,\, (x',\ \sigma')\rangle_{dec} \\ ((x',\ \sigma'),(x'',\ \sigma'')) \ \in \ \mathcal{H} \\ \langle a,\, x'',\, \sigma''\rangle_{stack} \Uparrow_{\text{rec}} (x''',\sigma''') \end{cases}$$

One-step reduction is a partial function because the given term may already be in $\mathcal{H}$-normal form.

Reduction-based evaluation: Reduction-based evaluation is defined as the iteration of the standard one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given program:

Definition 16 (standard reduction-based evaluation). *For any program* (x, σ)*,*

$$(x,\sigma) \mapsto^*_{\mathcal{H}} (x',\sigma') \ \wedge \ (x',\sigma') \in \mathcal{H}\text{-}nf$$

Evaluation is a partial function because it may diverge.

4.2 A store-based abstract machine

The abstract machine is calculated as in Section 2.2. We display it in Figure 8. Its architecture is that of Turner's SK-reduction machine [30]: the left-ancestor stack is incrementally constructed at each combination; upon reaching a combinator, its arguments are found on top of the ancestor stack and a graph transformation takes place to rearrange them. In particular, our handling of stored locations coincides with Turner's indirection nodes. The only differences are that our machine accepts the partial application of combinators and that Turner's combinators are unboxed, which is an optimization.

Proposition 3 (full correctness). *For any program* (x, σ)*,*

$$(x,\sigma) \mapsto^*_{\mathcal{H}} (x',\sigma') \ \wedge \ (x',\sigma') \in \mathcal{H}\text{-}nf \ \Leftrightarrow \ \langle x,\, \sigma(x),\, [\,],\, \sigma\rangle_{sto} \rightarrow^*_{\text{step}} \langle (x',\sigma')\rangle_{nf}$$

$$
\begin{aligned}
\langle [\,],\, x,\, \sigma\rangle_{stack} &\rightarrow_{\text{step}} \langle (x,\sigma)\rangle_{nf}\\
\langle (x_0,y_0)\cdot a,\, x_1,\, \sigma\rangle_{stack} &\rightarrow_{\text{step}} \langle a,\, x_0,\, \sigma\rangle_{stack}\\[1ex]
\langle x_1,\, I,\, (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle x_0,\, y_0,\, a,\, \sigma[x_0 := y_0]\rangle_{sto}\\
\langle x_2,\, K,\, (x_1,y_1)\cdot (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle x_0,\, y_1,\, a,\, \sigma[x_0 := y_1]\rangle_{sto}\\
\langle x_3,\, S,\, (x_2,y_2)\cdot(x_1,y_1)\cdot (x_0,y_0)\cdot a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle y_2,\, \sigma'(y_2),\, (x_4,y_0)\cdot(x_0,y_5)\cdot a,\, \sigma'\rangle_{sto}\\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } \sigma' = \sigma[x_4 := y_2\, y_0]\\
&\phantom{\rightarrow_{\text{step}}}\ \phantom{\text{where } \sigma' = \sigma}[x_5 := y_1\, y_0]\\
&\phantom{\rightarrow_{\text{step}}}\ \phantom{\text{where } \sigma' = \sigma}[x_0 := x_4\, x_5]\\
&\phantom{\rightarrow_{\text{step}}}\ \text{and } x_4 \text{ and } x_5 \text{ are fresh}\\
\langle x_0,\, s,\, a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle a,\, x_0,\, \sigma\rangle_{stack}\\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } s = I \text{ and } |a| < 1,\\
&\phantom{\rightarrow_{\text{step}}}\ \quad \text{or } s = K \text{ and } |a| < 2\\
&\phantom{\rightarrow_{\text{step}}}\ \quad \text{or } s = S \text{ and } |a| < 3\\
\langle x_0,\, x_1\, y_0,\, a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle x_1,\, \sigma(x_1),\, (x_0,y_0)\cdot a,\, \sigma\rangle_{sto}\\
\langle x_0,\, x_1,\, [\,],\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle x_1,\, \sigma(x_1),\, [\,],\, \sigma\rangle_{sto}\\
\langle x_0,\, x_1,\, (x,y)\cdot a,\, \sigma\rangle_{sto} &\rightarrow_{\text{step}} \langle x_1,\, \sigma'(x_1),\, (x,y)\cdot a,\, \sigma'\rangle_{sto}\\
&\phantom{\rightarrow_{\text{step}}}\ \text{where } \sigma' = \sigma[x := x_1\, y]
\end{aligned}
$$

Fig. 8. Store-based abstract machine for combinatory graph reduction

4.3 Summary and conclusion

Starting from a completely spelled out reduction semantics for combinatory term graphs in a store, we have derived a store-based abstract machine. The structure of this store-based abstract machine coincides with that of Turner's SK-reduction machine.

5 Related work

It has long been noticed that combinators make it possible to do without variables. For example, in the 1960's, Robinson outlined how this could be done to implement logics [29]. However, it took Turner to realize in the 1970's that combinatory graph reduction could be not only efficiently implementable, but also provided an efficient implementation for lazy functional languages [30]. Turner's work ignited a culture of implementation techniques in the 1980's [26], whose goal in retrospect can be characterized as designing efficient big-step graph reducers.

Due to the increased interest in graph reduction, Barendregt et al. developed term graphs and term graph rewriting [7]. Their work has since been used to model languages with sharing and to reason about program transformations in the presence of sharing [2,17,20,22,27]. Later work by Ariola and Klop provides an equational theory for term graph rewriting with cycles [5], a topic further developed by Blom in his PhD thesis [8] and by Nakata and Hagesawa since [25].

Over the 2000's, the first author and his students have investigated off-the-shelf program-transformation techniques for inter-deriving semantic artifacts [1, 14,31]. The present work is an outgrowth of this investigation.

6 Conclusion and future work

Methodologically, mathematicians who wrote about their art (Godfrey H. Hardy, John E. Littlewood, Paul Halmos, Jacques Hadamard, George Pólya, Alexandre Gothendriek, and Donald E. Knuth for example) clearly describe how their research is typically structured in two stages: (1) an exploratory stage where they boldly move forward, discovering right and left, and (2) a descriptive stage where they retrace their steps and revisit their foray, verify it, structure it, and put it into narrative shape. As far as abstract machines are concerned, tradition has it to seek new semantic artifacts, which is characteristic of the first stage. Our work stems from this tradition, though by now it subscribes to the second stage as we field-test our derivational tools.

The present article reports our field test of combinatory graph reduction. Our main result is that representing def-use chains using reduction contexts and let expressions, which, in retrospect, is at the heart of Ariola et al.'s syntactic theory of the call-by-need λ-calculus, also makes it possible to account for combinatory graph reduction. We have stated in complete detail two reduction semantics and have derived two storeless abstract machines. Interpreting denotable entities as storable ones in a global store, we have rediscovered David Turner's graph-reduction machine.

Currently, we are adding the Y combinator and inter-deriving natural semantics that correspond to the abstract machines. We are also adding literals and strict arithmetic and logic functions, as well as garbage-collection rules such as the following one:

$$\mathsf{let}\ x = d\ \mathsf{in}\ t \ \rightarrow\ t \quad \text{if } x \text{ does not occur in } t$$

At this point of time, we are wondering which kind of garbage collector is fostered by the nested let expressions of the syntactic theories and also the extent to which its references are akin to Curry's apparent variables [9].

Acknowledgments: Thanks are due to Marìa Alpuente for her generous invitation to present this material at LOPSTR 2010. We are also grateful to Kenichi Asai, Mayer Goldberg, Steffen Daniel Jensen, Julia Lawall, and three anonymous reviewers for their comments on an earlier version of this article.

The present study of a computational reality is a tribute to the first author's *directeur de thèse*, Bernard Robinet (1941–2009), who introduced a generation of students to Combinatory Logic and contexts [28].

References

1. M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003. ACM Press.
2. Z. M. Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146(1&2):69–108, 1995.

3. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.
4. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In P. Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, Jan. 1995. ACM Press.
5. Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3/4):207–240, 1996.
6. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.
7. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, number 259 in Lecture Notes in Computer Science, pages 141–158, Eindhoven, The Netherlands, June 1987. Springer-Verlag.
8. S. Blom. *Term Graph Rewriting – Syntax and Semantics*. PhD thesis, Institute for Programming Research and Algorithmics, Vrije Universiteit, Amsterdam, The Netherlands, Mar. 2001.
9. H. B. Curry. Apparent variables from the standpoint of Combinatory Logic. *Annals of Mathematics*, 34:381–404, 1933.
10. O. Danvy. Defunctionalized interpreters for programming languages. In P. Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 131–142, Victoria, British Columbia, Sept. 2008. ACM Press. Invited talk.
11. O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.
12. O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.
13. O. Danvy, K. Millikin, J. Munk, and I. Zerny. Defunctionalized interpreters for call-by-need evaluation. In M. Blume and G. Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, Apr. 2010. Springer.
14. O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.
15. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, Nov. 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.
16. R. Garcia, A. Lumsdaine, and A. Sabry. Lazy evaluation and delimited control. In B. C. Pierce, editor, *Proceedings of the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 44, No. 1, pages 153–164, Savannah, GA, Jan. 2009. ACM Press.
17. J. R. W. Glauert, R. Kennaway, and M. R. Sleep. Dactl: An experimental graph rewriting language. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph-*

Grammars and Their Application to Computer Science, 4th International Workshop, Proceedings, volume 532 of *Lecture Notes in Computer Science*, pages 378–395, Bremen, Germany, Mar. 1990. Springer.

18. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In H.-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, Jan. 1994. ACM Press.
19. J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997.
20. A. Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 82–91, Paris, France, July 1994. IEEE Computer Society Press.
21. J. W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.
22. P. W. M. Koopman. *Functional Programs as Executable Specifications.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1990.
23. J. L. Lawall and H. G. Mairson. On global dynamics of optimal graph reduction. In M. Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 32, No. 8, pages 188–195, Amsterdam, The Netherlands, June 1997. ACM Press.
24. J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
25. K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.
26. S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.
27. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.
28. B. Robinet. *Contribution à l'étude de réalités informatiques.* Thèse d'état, Université Pierre et Marie Curie (Paris VI), Paris, France, May 1974.
29. J. A. Robinson. A note on mechanizing higher order logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 123–133. Edinburgh University Press, 1969.
30. D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.
31. I. Zerny. On graph rewriting, reduction and evaluation. In Z. Horváth, V. Zsók, P. Achten, and P. Koopman, editors, *Trends in Functional Programming Volume 10*, Komárno, Slovakia, June 2009. Intellect Books. Granted the best student-paper award of TFP 2009. To appear.