Higher-Order Rewriting and Partial Evaluation

Olivier Danvy
Kristoffer H. Rose

BRICS Report Series RS-97-46

ISSN 0909-0878 December 1997

Higher-Order Rewriting and Partial Evaluation

Olivier Danvy
BRICS,[*] Dept. of Computer Science, University of Aarhus[†]

Kristoffer Høgsbro Rose
LIP,[‡] Ecole Normale Supérieure de Lyon[§]

December 1997
Extended version of RTA '98 article

Abstract

We demonstrate the usefulness of higher-order rewriting techniques for specializing programs, *i.e.*, for *partial evaluation.* More precisely, we demonstrate how casting *program specializers* as *combinatory reduction systems* (CRSs) makes it possible to formalize the corresponding program transformations as *meta-reductions*, *i.e.*, reductions in the internal "substitution calculus." For partial-evaluation problems, this means that instead of having to prove on a case-by-case basis that one's "two-level functions" operate properly, one can concisely formalize them as a combinatory reduction system and obtain as a corollary that static reduction does not go wrong and yields a well-formed residual program.

We have found that the CRS *substitution calculus* provides an adequate expressive power to formalize partial evaluation: it provides sufficient termination strength while avoiding the need for additional restrictions such as types that would complicate the description unnecessarily (for our purpose). We also review the benefits and penalties entailed by more expressive higher-order formalisms.

In addition, partial evaluation provides a number of *examples* of higher-order rewriting where being higher order is a central (rather than an occasional or merely exotic) property. We illustrate this by demonstrating how standard but non-trivial partial-evaluation examples are handled with higher-order rewriting.

[*]Basic Research in Computer Science (Centre of the Danish Research Foundation).
[†]Building 540, Ny Munkegade, DK–8000 Aarhus C, Denmark; ⟨danvy@brics.dk⟩.
[‡]Laboratoire de l'Informatique du Parallélisme.
[§]46, Allée d'Italie, F–69364 Lyon 07, France; ⟨Kristoffer.Rose@ens-lyon.fr⟩.

Contents

1 Introduction 2

2 Partial Evaluation 3
2.1 A first-order example 3
2.2 A higher-order example 4

3 Combinatory Reduction Systems and Functional Programming 5
3.1 Combinatory Reduction Systems 5
3.2 Comparing to functional programs 8

4 Synthesis 10
4.1 Derivors 10
4.2 The call-by-value CPS transform 12
4.3 Deriving specializers 13

5 Conclusion 16

1 Introduction

Most programs are overly general: they usually run with some invariants (*e.g.*, part of their input is constant). Partial evaluation aims at specializing programs with respect to these invariants [5, 13]. According to Kleene's S^m_n-theorem [15], specializing a program with respect to [an invariant] part of its input is computable, and running the corresponding specialized program on the remaining input should yield the same result as running the source program on the complete input, provided of course that the source program, the partial evaluator, and the residual program all terminate. The practical appeal of partial evaluation is that specialized programs are usually more efficient, so that running them amortizes the cost of partial evaluation.

What we find curious is that while in effect the renewal of partial evaluation originates in the area of rewriting techniques [14], there has been virtually no work to continue bridging the two areas. In this article we address this by formalizing the rewriting technique underlying standard partial-evaluation examples. Our prime target is the removal of interpretive overhead. In that view, any program traversing a data structure is an "interpreter" for that data structure (Abelson makes this point comprehensively in his foreword to "Essentials of Programming Languages" [11]). We thus focus on inductive data types and the associated functionals (or "derivors").

Our starting-point is the following analogy:

- In *partial evaluation* (PE) we specialize programs by performing part of them "in advance." This is achieved by using "two-level functions" that perform a mix of static evaluation and dynamic code generation.
- In rewriting we model (functional) programs by rewrite systems with a special "application" operator which is the root symbol of all (functional) rewrite rules. Since each application means we have to do work, we are interested in reducing the number of applications that are built, ideally to zero. We can do this using higher-order techniques in the framework of *combinatory reduction systems* (CRSs).

By recasting PE in the CRS formalism we should thus be able to exploit the meta-reductions in CRSs to perform the static reductions of our two-level functions.

Road map: The rest of the article is organized as follows: Section 2 reviews the two-level programming technique used in partial evaluation. Section 3 provides the necessary background in combinatory reduction systems, showing how simple program transformations can be cast as higher-order rewrite systems. Section 4 presents our synthesis between rewriting and partial evaluation, culminating with the detailed description of how to derive a specializer, and exemplified by a formal treatement of a classical example of specialization, a continuation-passing style (CPS) transformation for the λ-calculus. Section 5 concludes and briefly mentions some related work and future directions.

2 Partial Evaluation

In this section we illustrate partial evaluation through two examples: a first-order one and the higher-order one of traversing binary trees. The latter one hints at the style of transformation techniques used in the following sections.

2.1 A first-order example

So what is specialization? Specializing a program amounts to parameterizing it with code-generating (two-level) functions and running it. Let us demonstrate this with a first-order example: natural numbers and the exponentiation program, which is standard in partial evaluation. It is expressed as a simple conditional recursive equation as follows:

$$x^n = \begin{cases} x & \text{if } n = 1 \\ x^{n/2} \times x^{n/2} & \text{if } n \text{ even} \\ x \times x^{n-1} & \text{otherwise} \end{cases}$$

Seen as a function definition, *i.e.*, reading the equation from left to right, the exponentiation program is a *derivor*: it decomposes – or interprets – the exponent, in a trail of multiplications. Our aim is to specialize this derivor with respect to a particular exponent, which we achieve by interpreting the static exponent, in a trail of residual multiplications.

Here is the annotated derivor, where we have overlined the static parts that we can compute immediately when n becomes available:

$$x^{\overline{n}} = \begin{cases} x & \overline{\text{if } n = 1} \\ x^{\overline{n/2}} \times x^{\overline{n/2}} & \overline{\text{if } n \text{ even}} \\ x \times x^{\overline{n-1}} & \overline{\text{otherwise}} \end{cases}$$

Such a program is variously known as a *generating extension* [13] or a *backquote interpreter* [11] in the literature.

In any case, repeatedly evaluating the overlined subexpressions of $x^{\overline{5}}$ gives the *residual* expression $x \times \big((x \times x) \times (x \times x)\big)$, which is in normal form since there are no further overlined expressions to evaluate. The trail of residual multiplications is all that remains of the static reductions.

2.2 A higher-order example

Consider the data type αBT of binary trees over some (unspecified) type α:

$$\alpha BT \ ::= \ \mathit{Leaf}(\alpha) \ \mid \ \mathit{Node}(\alpha BT, \alpha BT)$$

and its associated fold functional *Fold* typed as follows:

$$\mathit{Fold} \ : \ (\alpha \to \beta) \times (\beta \times \beta \to \beta) \to \alpha BT \to \beta$$

As is customary in functional programming, we instantiate this fold functional with two functions: one for processing the leaves, and one for processing the nodes. For example, the application

$$\mathit{Fold}\ (\ \lambda x.1\ ,\ \lambda\langle l, r\rangle.(l + r)\ )$$

yields a function computing the number of leaves in a binary tree.

As is customary in partial evaluation, we instantiate this fold functional with three two-level functions: one for processing the leaves and one for processing the nodes, plus one to initialize the static computation. As before we overline static parts, here λs and applications.[1] Supplying a given binary tree

[1]An application is denoted by the space between two subexpressions, so $x \overline{\;\;} y$ is a static application whereas $x\,y$ is not.

to this fold function yields a residual program where the interpretive overhead of the fold function has been eliminated. For example, given unspecified function names L and N, the expression

$$\mathit{Fold}^{-}\left(\,\overline{\lambda}x.\,L\,x\ ,\ \overline{\lambda}\langle l,r\rangle.\,N\,l\,r\,\right)$$

yields a residual program combining L and N in a way that is isomorphic to the structure of the given binary tree. Applying the above to a binary tree such as

$$\mathit{Node}(\mathit{Node}(\mathit{Leaf}(1),\mathit{Leaf}(2)),\mathit{Leaf}(3))$$

yields the residual program

$$N\,\big(\,N\,(L\,1)\,(L\,2)\big)\,\big(L\,3\big)$$

which is well-formed since neither overlines nor $\langle\cdot,\cdot\rangle$-pairs remain.

3 Combinatory Reduction Systems and Functional Programming

In this article, we use Klop's *Combinatory Reduction System* (CRS) formalism. In this section we first summarize the definition of CRSs [16, 18] before relating them to functional programming with a simple example demonstrating the use of higher-order rewriting to express improvements to functional programs.

3.1 Combinatory Reduction Systems

The following is a brief summary of the definition of CRSs. To avoid notational overloading of ordinary parentheses, we slightly modify the standard presentation of CRSs [18, §11–12]. We write $\cdot.\cdot$ and $\cdot[\cdot]$ instead of $[\cdot]\cdot$ and $\cdot(\cdot)$ for abstraction and meta-application, respectively.

3.1.1. Definition (many-sorted CRS). Assume a signature Σ of ranked symbols F^n, variables x, and ranked *meta-variables* Z^n (in both cases the superscript n is the rank).

1. CRS *terms* have the form

$$t\ ::=\ x\ \mid\ x.t\ \mid\ F^n(t_1,\ldots,t_n)$$

   and must be *closed* (that is, $\mathrm{fv}(t)=\{\}$ where $\mathrm{fv}(x)=\{x\}$, $\mathrm{fv}(x.t)=\mathrm{fv}(t)\setminus\{x\}$, and $\mathrm{fv}(F^n)=\bigcup_{i=1}^{n}\mathrm{fv}(t_i)$). The three forms are respectively called *variable*, *abstraction*, and *construction*.

2. CRS *meta-terms* extend CRS terms to

$$t \ ::= \ x \ \mid \ x.t \ \mid \ F^n(t_1,\ldots,t_n) \ \mid \ \mathrm{Z}^n[t_1,\ldots,t_n]$$

   The new form is called a *meta-application*.

3. An *assignment* σ specifies how to eliminate meta-applications that use specific meta-variables. It is a collection of pairs $(\mathrm{Z}^n[x_1,\ldots,x_n],t')$ with distinct x_i; $\sigma(t)$ is the resulting term where everywhere in t, $\mathrm{Z}^n[t_1,\ldots,t_n]$ is replaced by $t'\{x_1 := t_1,\ldots,x_n := t_n\}$ (which denotes an ordinary simultaneous substitution). The assignment σ is *safe* if

$$\forall \mathrm{Z},\mathrm{Z}' : \mathrm{fv}(\sigma(\mathrm{Z})) \cap \mathrm{bv}(\sigma(\mathrm{Z}')) = \varnothing$$

   (with $\mathrm{bv}(\cdot)$ denoting the *bound variables* of a preterm defined inductively as $\mathrm{bv}(x) = \varnothing$, $\mathrm{bv}([x]t) = \{x\} \cup \mathrm{bv}(t)$, and $\mathrm{bv}(F^n(\vec{t})) = \mathrm{bv}(\mathrm{Z}^n(\vec{t})) = \bigcup_{i=1}^{n} \mathrm{fv}(t_i)$).

4. CRS *rules*, written $\ell \to r$, are constructed from two meta-terms ℓ and r with the following additional restrictions:

   (a) ℓ (the left-hand side) must be a "pattern:" a construction where all meta-applications have the form $\mathrm{Z}^n[x_1,\ldots,x_n]$ with distinct x_i, and

   (b) r (the right-hand side) can only contain meta-applications with meta-variables occurring in the left-hand side.

   The rewrite rule $\ell \to r$ is *safe* for an assignment σ if ℓ and r (considered as preterms) satisfy

$$\forall \mathrm{Z} \in \mathrm{mv}(p)\ \forall x \in \mathrm{fv}(\sigma(\mathrm{Z})) : x \notin (\mathrm{bv}(\ell) \cup \mathrm{bv}(r))$$

   (with $\mathrm{mv}(\cdot)$ denoting the meta-variables of a term).

5. We say that a term t *matches* a pattern ℓ if an assignment σ exists such that $t = \sigma(\ell)$ (the intuition being that each pattern meta-application $\mathrm{Z}^n[x_1,\ldots,x_n]$ becomes part of the assignment of σ). The assignment must be safe (we can ensure this by renaming).

6. The rules define the CRS *rewrite relation*: $s \to t$ iff s and t are identical except for one subterm: In s, it must be $\sigma(\ell)$ for some assignment σ, the "redex." In t, it must be $\sigma(r)$, the "contractum." The rule must be safe for the assignment (again we can ensure this by renaming).

7. A *sorted CRS* is the subsystem obtained by restricting terms to be "well-sorted" according to some syntax specification, and assigning to each meta-variable a sort that it must match.

We use the usual abbreviations for CRSs. In particular, we omit the rank superscript and abbreviate $F^1(x.F^1(y.t))$, $F^0()$, and $\mathrm{Z}^0[\,]$, as $Fxy.t$, F, and Z, respectively. We also exploit conventions introduced in syntax productions to bind meta-variables to sorts and introduce infix binary constructors.

We do not delve further into the exciting details of the properties of rewriting systems in general and CRS in particular but refer the reader to the comprehensive literature on the subject [17, 18]. Instead we go straight to our basic example.

3.1.2. Example (2-level λ-calculus). The *2-level λ-calculus*, denoted $\overline{\boldsymbol{\lambda}}$, is the single-sorted CRS over the $\overline{\boldsymbol{\lambda}}$*-terms*

$$\mathrm{E} \ ::= \ x \ \mid \ \lambda x.\mathrm{E} \ \mid \ \mathrm{E}_0\ \mathrm{E}_1 \ \mid \ \overline{\lambda} x.\mathrm{E} \ \mid \ \mathrm{E}_0 \,\overline{\ }\, \mathrm{E}_1$$

where concatenation denotes "application" (the invisible infix application function symbol sometimes written as @), $\mathrm{E}_0 \,\overline{\ }\, \mathrm{E}_1$ is "overlined application" (also $\overline{@}$), and both associate to the left as usual. Its rewrite rules read

$$(\lambda x.\mathrm{E}[x])\ \mathrm{E}' \to \mathrm{E}[\mathrm{E}'] \qquad (\beta)$$
$$(\overline{\lambda} x.\mathrm{E}[x]) \,\overline{\ }\, \mathrm{E}' \to \mathrm{E}[\mathrm{E}']. \qquad (\overline{\beta})$$

(The subset with no overlines and with just β as reduction is the usual $\lambda\beta$-calculus denoted $\boldsymbol{\lambda}$ [2].)

Thus as a CRS, $\overline{\boldsymbol{\lambda}}$ has the constructors λ^1, $\overline{\lambda}^1$, $@^2$, and $\overline{@}^2$, with the restriction on $\lambda^1(t)$ and $\overline{\lambda}^1(t)$ that t must be a CRS abstraction $x.t'$.

3.1.3. Definition (abstract rewriting). Binary relations are denoted by arrows. Relational *composition* is written $\underset{1}{\to} \cdot \underset{2}{\to}$; the *inverse* of $\to$ is $\leftarrow$, its *transitive reflexive closure* is $\twoheadrightarrow$, and its *normalisation function* is $\longrightarrow\!\!\!\!\!\twoheadrightarrow\!\!\shortmid$ (the restriction of $\twoheadrightarrow$ to just the reductions ending in a normal form). Two relations *commute* if $(\underset{1}{\leftarrow} \cdot \underset{2}{\to}) \subseteq (\underset{2}{\to} \cdot \underset{1}{\leftarrow})$; a relation $\to$ is *confluent* if $\twoheadrightarrow$ self-commutes. Finally a relation is *convergent* if it is confluent and *terminating*, *i.e.*, has no infinite reduction sequences.

3.1.4. Definition (CRS restrictions). A CRS is *left-linear* if all meta-variables occurring in each left-hand side are distinct. A CRS is *non-overlapping* if it is impossible for a symbol in a term to be part of two redexes in the term. A CRS is *orthogonal* if it is left-linear and non-overlapping. A *constructor* CRS's symbols are in two disjoint sets: *functions* that occur at the root of left-hand sides and *constructors* that do not. Finally, a CRS is a *term-rewriting system* (TRS) if all meta-variables used in rules are nullary.

3.1.5. Example. $\overline{\lambda}$ is a left-linear and non-overlapping (hence orthogonal) constructor CRS (with functions $@, \overline{@}$ and constructors $\lambda, \overline{\lambda}$) but not a TRS.

3.1.6. Theorem. *Orthogonal CRSs are confluent [16, 18].*

3.1.7. Example. $\overline{\lambda}$ is orthogonal, hence confluent.

3.2 Comparing to functional programs

First-order functional programs are usually said to correspond to left-linear constructor TRSs. We observe that untyped higher-order functional programming corresponds to adding β to the underlying formalism, thus interpreting the special relationship between the application function symbol and the λ constructor.

Functional programming languages do not enforce orthogonality but usually obtain uniqueness of the result by fixing a *deterministic reduction strategy*, permitting only one rule at any point according to some priority principle. However, typically the *model* used for programming does not enforce a particular reduction strategy, so therefore program transformations *do* benefit from being orthogonal because they involve reducing out of the usual order (such reductions are typically called "non-standard").

3.2.1. Example (binary tree folding, functional style). A *binary tree* of integers has two sorts: trees,

$$\mathrm{T} \ ::= \ \mathit{Leaf}(\mathrm{I}) \ \mid \ \mathit{Node}(\mathrm{T}_1, \mathrm{T}_2)$$

and integers, I. "Folding" over the tree means replacing each $\mathit{Node}(\mathrm{T}_1, \mathrm{T}_2)$ with an application $N\ \mathrm{T}_1\ \mathrm{T}_2$ and each leaf $\mathit{Leaf}(\mathrm{I})$ with $L\ \mathrm{I}$, as discussed in section 2.2. A typical "functional program" rewrite system to do this is the following (left-linear constructor) TRS over trees extended with the symbol *Fold*:

$$\mathit{Fold}(\mathrm{L}, \mathrm{N}, \mathit{Leaf}(\mathrm{I})) \to \mathrm{L}\ \mathrm{I} \tag{1}$$
$$\mathit{Fold}(\mathrm{L}, \mathrm{N}, \mathit{Node}(\mathrm{T}_1, \mathrm{T}_2)) \to \mathrm{N}\ \big(\mathit{Fold}(\mathrm{L}, \mathrm{N}, \mathrm{T}_1)\big)\ \big(\mathit{Fold}(\mathrm{L}, \mathrm{N}, \mathrm{T}_2)\big) \tag{2}$$

For any tree T this system rewrites $\mathit{Fold}(L, N, \mathrm{T})$ to the folding of T with L and N. Running it[2] on an example term gives

$$\begin{aligned}&\mathit{Fold}(L, N, \mathit{Node}(\mathit{Node}(\mathit{Leaf}(1), \mathit{Node}(\mathit{Leaf}(2), \mathit{Leaf}(3))),\\ &\qquad\qquad \mathit{Node}(\mathit{Node}(\mathit{Leaf}(4), \mathit{Leaf}(5)), \mathit{Leaf}(6))))\\ &\qquad\qquad\quad \twoheadrightarrow N\,(N\,(L\,1)(N\,(L\,2)(L\,3)))(N\,(N\,(L\,4)(L\,5))(L\,6))\end{aligned}$$

[2]All examples were run with the CRS implementation of the second author's PhD thesis [26, chapter6], adapted to the present syntax.

as should be expected.

Assume now that we wish to *flatten* trees just as we programmed it in the previous section, converting the tree to a list of the leaf integers. The following makes $\mathit{Flatten}(\mathrm{T})$ rewrite to this effect when combined with *Fold* and $\boldsymbol{\lambda}$ to achieve the function reduction (thus this program is higher-order):

$$\mathit{Flatten}(\mathrm{T}) \to \mathit{Fold}(\ \lambda ia.\mathit{Cons}(i,a)\ ,\ \lambda c_1c_2a.c_1(c_2\ a)\ ,\ \mathrm{T}\ )\ \mathit{Nil} \tag{3}$$

Running the system on an example term gives

$$\begin{aligned}&\mathit{Flatten}(\mathit{Node}(\mathit{Node}(\mathit{Leaf}(1), \mathit{Node}(\mathit{Leaf}(2), \mathit{Leaf}(3))),\\ &\qquad\qquad \mathit{Node}(\mathit{Node}(\mathit{Leaf}(4), \mathit{Leaf}(5)), \mathit{Leaf}(6))))\\ &\qquad\qquad \twoheadrightarrow \mathit{Cons}(1, \mathit{Cons}(2, \mathit{Cons}(3, \mathit{Cons}(4, \mathit{Cons}(5, \mathit{Cons}(6, \mathit{Nil})))))).\end{aligned}$$

The inconvenience of the above system (and of functional programming in general) is that the only rule doing actual rearrangement is β: the folding itself does nothing but build applications. This can be fixed by exploiting the possibilities of the CRS formalism for matching functions in any rule rather than just in β. In the next example, we thus reconsider flattening.

3.2.2. Example (binary tree folding, CRS). Consider binary trees as before with the additional auxiliary symbols L^1 and N^2. We then define folding by pattern matching on the functions to let the CRS formalism do the work of unfolding the leaf- and node-functions; furthermore we add a root function applied by the wrapper *Fold*$'$:

$$\mathit{Fold}'(\lambda t.\mathrm{R}[t], \mathrm{L}, \mathrm{N}, \mathrm{T}) \to \mathrm{R}[\mathit{Fold}(\mathrm{L}, \mathrm{N}, \mathrm{T})] \tag{4}$$

$$\mathit{Fold}(\lambda i.\mathrm{L}[i], \mathrm{N}, \mathit{Leaf}(\mathrm{T})) \to \mathrm{L}[\mathrm{T}] \tag{5}$$

$$\begin{aligned}&\mathit{Fold}(\mathrm{L}, \lambda st.\mathrm{N}[s,t], \mathit{Node}(\mathrm{T}_1, \mathrm{T}_2)) \to\\ &\qquad \mathrm{N}\big[\mathit{Fold}(\mathrm{L}, \lambda st.\mathrm{N}[s,t], \mathrm{T}_1), \mathit{Fold}(\mathrm{L}, \lambda st.\mathrm{N}[s,t], \mathrm{T}_2)\big]\end{aligned} \tag{6}$$

Then flattening is merely dispersing yet another "continuation" wrapped in special $\langle\cdot,\cdot\rangle$ brackets which are pattern-matched by the fold functions:

$$\mathit{Flatten}(\mathrm{T}) \to \mathit{Fold}'(\ \lambda x.x(\lambda z.\langle z, \mathit{Nil}\rangle),\ \lambda i.L\ i,\ \lambda lr.N\ l\ r,\ \mathrm{T}\ ) \tag{7}$$

$$L\ \mathrm{A}\ (\lambda z.\langle z, \mathrm{B}\rangle) \to \mathit{Cons}(\mathrm{A}, \mathrm{B}) \tag{8}$$

$$N\ \mathrm{A}\ \mathrm{B}\ \mathrm{C} \to \mathrm{A}\ (\lambda v.\langle v, \mathrm{B}\ \mathrm{C}\rangle) \tag{9}$$

Notice that there are two levels of functions involved: the function constructed with an explicit λ in the arguments to *Fold*$'$, and the function encoded by the L and N rules. The solution is elegant and very efficient since all the constructed abstractions are known to have the form $\lambda x.\langle x, \mathrm{A}\rangle$ where x is not free in A, so substitution is not costly – in fact we exploit this in the pattern of (8) where

B is not written as B$[z]$ because we know it does not contain a free occurrence of z. Folding the sample tree gives the same result as in the previous example, of course. The only remaining inconvenience is the fact that we still have to prove that all $\langle\cdot\rangle$ brackets are eliminated.

4 Synthesis

In this section we apply the higher-order rewriting technology discussed in the last section to partial evaluation, and we formalise the notion of a program specializer accordingly. We use this to prove a general theorem of well-annotatedness of specializers when given in the form of two-level derivors.

We first set the scene in section 4.1 by defining the notion of "derivor" formally, and proving the properties we need, before stating our main example in section 4.2, the continuation-passing style transformation, and showing how concisely it is accounted for using higher-order rewriting. Finally, section 4.3 presents our main result: how one can mechanically transform a two-level derivor into a specializer where all "administrative reductions" are achieved through the CRS substitution calculus. The transformation can even be used as a test of well-annotatedness since it only works if the two-level derivor was "well-annotated" to start with.

4.1 Derivors

We only consider syntax-directed program transformations. They are usually specified compositionally in the following sense.

4.1.1. Definition. A constructor CRS is *compositional* if each function symbol is compositional in one of its arguments, *i.e.*, if it has a distinguished argument such that the distinguished argument of all function constructions in the right-hand side of rules is always a strict subterm of the distinguished argument of the function construction on the left-hand side. Only one exception is permitted (to facilitate "root" rules): if a function symbol occurs only on left-hand sides, then the rules where it occurs are exempted from the constraint.

4.1.2. Example. The system of Example 3.2.2 is compositional: the symbol *Fold*$'$ occurs only on the left-hand side, in (4) and the occurrences of *Fold* in the other two rules, (5) and (6), are compositional in the third argument.

4.1.3. Lemma. *A compositional constructor CRS is terminating.*

Proof. We first consider the case with just one function F^n, compositional in the first argument, and one constructor C^m. Consider

$$\begin{aligned}
[\![F^n(\mathrm{T}_1,\dots,\mathrm{T}_n)]\!] &\to \wr|\mathrm{T}_1|\int \uplus [\![\mathrm{T}_1]\!] \uplus \cdots \uplus [\![\mathrm{T}_n]\!] &(10)\\
[\![C^m(\mathrm{T}_1,\dots,\mathrm{T}_m)]\!] &\to [\![\mathrm{T}_1]\!] \uplus \cdots \uplus [\![\mathrm{T}_m]\!] &(11)\\
|F^n(\mathrm{T}_1,\dots,\mathrm{T}_n)| &\to 1 + |\mathrm{T}_1| + \cdots + |\mathrm{T}_n| &(12)\\
|C^m(\mathrm{T}_1,\dots,\mathrm{T}_m)| &\to 1 + |\mathrm{T}_1| + \cdots + |\mathrm{T}_m| &(13)
\end{aligned}$$

where $\wr\cdot\int$ denotes a multiset and $\uplus$ is multiset union. This interprets any term t into a multiset expression obtained as (the normal form of) $[\![t]\!]$. Now the compositionality condition ensures that for every rewrite $t \to s$ in the original system, $[\![t]\!] >^\mu [\![s]\!]$, where $>^\mu$ is the *multiset ordering* induced by ordinary natural number comparison $>$ (so one multiset is larger than another where one of its elements is replaced by any number of smaller elements). Since $>^\mu$ is terminating [9], so is the CRS. The argument generalises easily to systems with any number of functions and constructors, and terms with a function symbols only in the left-hand side of rules contribute a single element with value equal to the sum of all subterm sizes, multiplied by the largest number of copies made of a subterm by any single rule. □

Now we can characterize the program transformers under consideration.

4.1.4. Definition. A *derivor* is an orthogonal and compositional constructor CRS where the normal forms contain only constructors.

4.1.5. Example. The interpretation system used in the proof of Lemma 4.1.3 is a derivor if $+$ and $\uplus$ are seen as constructors.

4.1.6. Theorem. *Derivors are convergent.*

Proof. Use Theorem 3.1.6 and Lemma 4.1.3. □

With this we can express an interesting class of systems, which is directly relevant to partial evaluation.

4.1.7. Definition. A *two-level derivor* is a derivor producing $\overline{\boldsymbol{\lambda}}$-terms (of Definition 3.1.2) with the restrictions that $\overline{@}$ does not occur on any left-hand side and that $\overline{\lambda}$ is a constructor. A *well-annotated* two-level derivor is one producing $\overline{\boldsymbol{\lambda}}$-terms for which the $\overline{\beta}$-normal form is a $\boldsymbol{\lambda}$-term, *i.e.*, all overlines are eliminated.

Two-level derivors have interesting properties: first of all it is easy to see that "static reduction does not go wrong," which is mandatory in partial evaluation [13].

4.1.8. Proposition. *For any two-level derivor, $\mathcal{D}$, $\mathcal{D} \cup \boldsymbol{\lambda}$ is confluent.*

Proof. The restrictions on the occurrences of $\overline{@}$ and $\overline{\lambda}$ ensure that the combined system remains orthogonal. $\square$

However, it remains difficult to prove well-annotatedness and termination of a two-level derivor because $\overline{\beta}$ has the full Turing-complete power in it. Both can be proven if one restricts the permitted $\overline{\beta}$ to a subset known to terminate, such as the simply typed λ-calculus. Then the entire construction of $\overline{\boldsymbol{\lambda}}$-terms has to be shown well-typed, a property that is easy to lose by even minute changes to the system. (This is related to why we have chosen CRSs as our basis formalism; we comment on this in the conclusion.)

Both properties can be shown for the first-order flatten in Example 3.2.1 but are trivial for the higher-order flatten of Example 3.2.2 since it produces no overlines at all. We exploit this property in the following example.

4.2 The call-by-value CPS transform

Compiler implementors are always in search for a good intermediate language, *i.e.*, a language that is both simple, concise, and expressive. The λ-calculus is such a candidate, but it is too expressive in that λ-terms can be evaluated with various reduction strategies (call-by-name, by-value, *etc*). There is however a sublanguage of the λ-calculus that is insensitive to its order of evaluation: the sub-language of continuation-passing style (CPS). A CPS transformation translates λ-terms into CPS (and in so doing, it encodes an evaluation order). As such, it is useful in compilers both for functional languages such as Scheme [4, 19, 30] and ML [1, 20] with a translation encoding "eager" semantics, and for pure languages such as Haskell [10, 23] with a transformation encoding "lazy" evaluation. There is therefore a strong interest in having efficient CPS transformations.

The traditional way amounts to (1) performing the translation following Plotkin's seminal specification [25]; and (2) performing so-called "administrative reductions" to simplify the resulting term. A more direct way, however, exists that combines (1) and (2) in one pass [6]. This method is crucially higher-order and two-level. Two-level because it combines static simplifications and dynamic code generation; and higher-order because it is expressed in the λ-calculus. The technique used to obtain this system resembles the technique we used in the flattening Example 3.2.2: The derivor is an interpreter that traverses its input λ-term, in a trail of continuations. The idea is to specialize the interpreter with respect to a particular λ-term. The corresponding derivor also traverses the fixed λ-term, in a trail of dynamic continuations. The result of specialization is a λ-term in continuation-passing style.

4.2.1. Definition (Call-by-Value CPS transformation). The eager, or *Call-by-Value*, CPS transformation can be expressed as a derivor over the two-sorted syntax

$$\mathrm{V} ::= x \tag{14}$$

$$\mathrm{E} ::= \mathrm{V} \mid \lambda x.\mathrm{E} \mid \mathrm{E}_0\ \mathrm{E}_1 \mid \overline{\lambda} x.\mathrm{E} \mid \mathrm{E}_0 \mathbin{\overline{\ }} \mathrm{E}_1 \mid \mathit{CPS1}(\mathrm{E}) \mid \langle \mathrm{E} \rangle \tag{15}$$

(the first sort just contains variables) with rules

$$\mathit{CPS1}(\mathrm{E}) \to \lambda k.\langle \mathrm{E} \rangle \mathbin{\overline{\ }} (\overline{\lambda} m.km) \tag{16}$$

$$\langle \mathrm{V} \rangle \to \overline{\lambda} k.k \mathbin{\overline{\ }} \mathrm{V} \tag{17}$$

$$\langle \lambda x.\mathrm{E}[x] \rangle \to \overline{\lambda} k.k \mathbin{\overline{\ }} (\lambda x.\lambda k.\langle \mathrm{E}[x] \rangle \mathbin{\overline{\ }} (\overline{\lambda} m.km)) \tag{18}$$

$$\langle \mathrm{E}_0 \mathrm{E}_1 \rangle \to \overline{\lambda} k.\langle \mathrm{E}_0 \rangle \mathbin{\overline{\ }} (\overline{\lambda} m.\langle \mathrm{E}_1 \rangle \mathbin{\overline{\ }} (\overline{\lambda} n.mn(\lambda a.k \mathbin{\overline{\ }} a))) \tag{19}$$

(where we exploit the sorting to ensure that (17) is only applied to variables).

The *CPS1* system is obviously a two-level derivor. It is possible to prove its well-annotatedness and termination directly using a typing argument [6, 24]. Instead, let us integrate the "administrative" $\overline{\beta}$-contractions in the transformation, making it truly one-pass in a rewriting sense; this will mechanically lead us to Sabry and Felleisen's "compacting" CPS transformation [27]. That this integration is well-defined is clear from Proposition 4.1.8. What remains is to express the transformation as a derivor that does not require "post-processing" in the form of static reductions or erasure to make it obvious that it cannot generate static applications or static abstractions.

4.3 Deriving specializers

4.3.1. Definition. A two-level derivor is a *specializer* if its normal forms are $\boldsymbol{\lambda}$-terms.

A specializer thus encodes static reductions into the derivor (so specializers are trivially well-annotated). The "Holy Grail of Partial Evaluation" follows:

4.3.2. Corollary. *Specializers are convergent (since they are derivors).*

So merely *expressing* a program transformation as a two-level derivor whose normal forms are $\boldsymbol{\lambda}$-terms ensures both that "static reduction does not go wrong" and that static normal forms (*i.e.*, specialized programs) exist and are unique.

The remainder of this section is devoted to show how one can mechanically obtain a specializer from a well-annotated two-level derivor and vice versa.

4.3.3. Theorem. *Let $\mathcal{D}$ be a two-level derivor. Then there is a specializer realizing $\underset{\mathcal{D}}{\to} \cdot \underset{\overline{\beta}}{\twoheadrightarrow\!\!|}$ if and only if $\mathcal{D}$ is well-annotated.*

Before we prove this by actually constructing the specializer, let us illustrate the method for our example system. First we observe that the problematic function symbol is $\langle\cdot\rangle$ because it has the (normalization function) type $\boldsymbol{\lambda} \to \overline{\boldsymbol{\lambda}} \to \overline{\boldsymbol{\lambda}}$ which is of order 2, and we want to change it to make the result belong to $\boldsymbol{\lambda}$. The technique we use is to *uncurry* the uses of $\langle\cdot\rangle$ to obtain something which has the type $\boldsymbol{\lambda} \times (\overline{\boldsymbol{\lambda}} \to \overline{\boldsymbol{\lambda}}) \to \boldsymbol{\lambda}$ and thus creates no overlines anywhere. This is expressed by the *representation shift* from the curried $\langle \textsc{e}_1\rangle \overline{\ } (\overline{\lambda} m.\textsc{e}_2[m])$ to the uncurried $\langle \textsc{e}_1, \overline{\lambda} m.\textsc{e}_2[m]\rangle$ which gives the following set of rules (the first one again for initialisation):

$$\mathit{CPS2}(\textsc{e}) \to \lambda k.\langle \textsc{e}, \overline{\lambda} m.km\rangle \tag{20}$$

$$\langle \textsc{v}, \overline{\lambda} k.\textsc{f}[k]\rangle \to \textsc{f}[\textsc{v}] \tag{21}$$

$$\langle \lambda x.\textsc{e}[x], \overline{\lambda} k.\textsc{f}[k]\rangle \to \textsc{f}[\lambda x.\lambda k.\langle \textsc{e}[x], \overline{\lambda} m.km\rangle] \tag{22}$$

$$\langle \textsc{e}_0\textsc{e}_1, \overline{\lambda} k.\textsc{f}[k]\rangle \to \langle \textsc{e}_0, \overline{\lambda} m.\langle \textsc{e}_1, \overline{\lambda} n.mn(\lambda a.\textsc{f}[a])\rangle\rangle \tag{23}$$

This is sufficient since there are now no $\overline{@}$s on any right-hand side and all $\overline{\lambda}$s are eliminated by $\langle\cdot,\cdot\rangle$.

4.3.4. Proposition. *CPS2 is a specializer.*

Proof. *CPS2* is clearly a compositional derivor, hence $\langle\cdot,\cdot\rangle$ is well-defined as a normalisation function with with type $\boldsymbol{\lambda} \times (\overline{\boldsymbol{\lambda}} \to \overline{\boldsymbol{\lambda}}) \to \boldsymbol{\lambda}$; from this the proposition follows, which is easy since in the constructor CRS with $\langle\cdot,\cdot\rangle$, the only function symbol is closed with respect to the sub-$\overline{\boldsymbol{\lambda}}$ system with terms

$$\textsc{a} ::= x \mid \lambda x.\textsc{a} \mid \textsc{a}_0\textsc{a}_1 \mid \langle \textsc{a}, \overline{\lambda} x.\textsc{a}\rangle \tag{24}$$

which degenerates to $\boldsymbol{\lambda}$ for normal forms because all possible variations of $\langle \textsc{a}, \overline{\lambda} x.\textsc{a}\rangle$ match one of (21–23). $\square$

This integration of administrative reductions into the CPS transformation is known for several years now [6, 27, 28]. What we have done here is to derive it using our rewriting account of partial evaluation. In particular, the resulting term need no post-processing (such as erasing remaining annotations).

Even systems with higher-order types can be handled by first reducing the type order with supercombinator extraction [12], which one could call "meta-λ-lifting" since it is targeted at lifting out all higher-order applications of $\overline{@}$. This is, in fact, what we did with the tree flattening Example 3.2.2, and what we use in the following general construction.

Proof of Theorem 4.3.3.

Case $\Rightarrow$. Given the two-level derivor, $\mathcal{D}$, and a specializer, $\mathcal{S}$, such that $\underset{\mathcal{S}}{\to} = \underset{\mathcal{D}}{\to} \cdot \underset{\overline{\beta}}{\twoheadrightarrow}$. Then $\mathcal{D}$ is well-annotated because we know static reduction will finish with a term containing no $\overline{@}$s.

Case $\Leftarrow$. Given $\mathcal{D}$ a well-annotated two-level derivor. Then $\underset{\mathcal{D}}{\to} \cdot \underset{\overline{\beta}}{\twoheadrightarrow}$ is a function into $\boldsymbol{\lambda}$. Let us specify how to transform the rules $\mathcal{D}$ into a specializer. Clearly, the problem is to get rid of $\overline{@}$s on the right-hand sides. Thus we have three subcases:

Base subcase: If there are no $\overline{@}$s at all then the system is already a specializer because all $\overline{\lambda}$s must be eliminated in some way by the system even without $(\overline{\beta})$ because no $\overline{\beta}$-redexes are created.

Uncurry: If the system has a rule of the form

$$F^n(\vec{t}) \to \overline{\lambda}k.s$$

then add the new rule

$$F^n(\vec{\mathrm{T}}) \to F_1^{n+1}(\vec{\mathrm{T}}, \overline{\lambda}k.k)$$

with F_1 a fresh function symbol, and replace the rule with

$$F_1^{n+1}(\vec{t}, \overline{\lambda}k.\mathrm{E}[k]) \to s'$$

where s' is obtained from s by replacing

- all occurrences of $k\overline{\ }t'$, for some t', by $\mathrm{E}[t']$, and
- all occurrences of $(F(\vec{t'}))\overline{\ }t''$, for some $\vec{t'}, t''$, by $F_1(\vec{t'}, t'')$.

with E a new meta-variable (of the appropriate sort).

$\overline{\boldsymbol{\lambda}}$-lift step: If the system has a rule of the form

$$F^n(\vec{t}) \to C\{\overline{\lambda}k.s\}$$

which was not generated by uncurrying and where $C\{\cdot\}$ is a non-empty context, then add the (generic) rule

$$A^2(\overline{\lambda}x.\mathrm{Z}[x], \mathrm{T}) \to \mathrm{Z}[\mathrm{T}]$$

and replace the rule with

$$F^n(\vec{t}) \to C\{\overline{\lambda}k.s'\}$$

where s' is obtained by replacing in s all occurrences of $k\overline{\ }t$ by $A^2(k, t)$.

The iteration terminates (with a number of iterations corresponding to the order of the involved two-level types). The resulting system has no $\overline{@}$s left because the well-annotated $\mathcal{D}$ cannot have other instances of $\overline{@}$ which would not be $\overline{\beta}$-reducible.

From the two cases we conclude that $\xrightarrow[\mathcal{D}]{} \cdot \xrightarrow[\overline{\beta}]{}\!\!\!\!\rightarrow$ is a specializer if and only if the two-level derivor $\mathcal{D}$ is well-annotated. □

5 Conclusion

Foremost we report a success: using higher-order rewriting, we have been able to formalize the partial-evaluation technique of two-level programming, and we have illustrated it with two non-trivial examples: flattening a binary tree in Section 3 and the so-called "one-pass" CPS transformation in Section 4. The immediate benefit, from a partial-evaluation point of view, is obvious: the formalization comes with a generic proof technique to establish the correctness of program specialization. A dual benefit also holds, from the rewriting point of view: the idea of tapping into a source of examples where being higher-order is what makes the examples work.

Why CRSs? One question immediately arises: Why have we used CRSs rather than any of the other formalisms for higher-order rewriting? In particular the complexity of Definition 3.1.1, due to the fact that it is "stand-alone," seems excessive. The major reason was that we have found CRSs were very easy to understand in an informal and intuitive way, first of all due to Klop, van Oostrom, and van Raamsdonk's survey [18]. Once we had worked with a few examples, CRSs have posed few problems. It is perhaps an significant factor here that program transformation is a very "syntactic" activity and the purpose of CRSs was to provide a syntactic theory of systems with binding [16].

One could instead use a formalism founded on known systems: such are usually much more concisely defined (for better and worse). A good candidate for this is HRS [22] where the "substitution calculus" used to describe the mechanics of rewrite steps is Church's λ^{τ} (simply typed λ-terms with β). Two difficulties need to be overcome: (1) The notion of "binding" in HRSs is more semantic, which makes it nonobvious to work with free variables as we did in (14), something most program transformers do. (2) The syntactic constructors of the source language need to be typed in HRSs to ensure that the notion of substitution is well-defined. The (weaker) "calculus of developments" used by CRSs guarantees that substitution terminates no matter which constructions are used so no special considerations are needed [31]. One could see the demand for typing as an advantage, in particular in our last proof where the "uncurrying" is type directed: it would be nice if the underlying formalism provided support for system transformations involving type changes.

An even more drastic approach would be to use a formalism where the substitution calculus is a "plug-in" such as HORS [32]: this could provide for more advanced notions of "static"reduction, for example including arithmetic as needed by the first-order "power" example. One worry remains, however: the typed systems (including HRSs) work on $\beta\eta$-long normal forms. It is not clear to which extent this interferes with the transformations and syntactic constraints we have discussed.

Related work. We only know of three lines of work relating rewriting and partial evaluation, and none that establish a common ground between them. (They focus more on highlighting the fact that TRSs can be seen as a fully functional programming language but did not exploit rewriting technology for the formalization of partial evaluation.) (1) In his M.Sc. thesis [3], Bondorf investigated the (self-applicable) partial evaluation of TRSs. He thus wrote a partial evaluator for TRSs, using a TRS. (2) Sherman and Strandh [29] use partial evaluation to optimize the implementation of term-rewriting systems. (3) Dershowitz [8] uses rewriting as the basic mechanism for abstracting and instantiating program schemas. Furthermore, higher-order systems such as λ-prolog or Elf can also be used for program transformation. For example, Danvy and Pfenning have formalized the CPS transformation in Elf [7].

Future work. In addition to the understanding better the rôle of types in higher-order rewriting, we plan to investigate the relation to specific published notions of reduction and λ-lifting, specifically 2-level λ-lifting [21].

Acknowledgements. We thank the anonymous RTA '98 referees for perceptive comments, and Tobias Nipkow for encouraging us not to stop at CRSs to formalize partial evaluation.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[3] Anders Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, 1988.

[4] William Clinger and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

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

[7] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.

[8] Nachum Dershowitz. Program abstraction and instantiation. *ACM Transactions on Programming Languages and Systems*, 7(3):446–477, 1985.

[9] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[10] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

[11] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press and McGraw-Hill, 1991.

[12] John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, August 1982.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[14] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 124–140, Dijon, France, May 1985.

[15] Stephen C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[16] Jan Willem Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[17] Jan Willem Klop. Term rewriting systems. In Samson Abramsky, Dov M. Gabby, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, chapter 1, pages 2–116. Oxford University Press, Oxford, 1992.

[18] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[19] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[21] Flemming Nielson and Hanne Riis Nielson. 2-level λ-lifting. In Harald Ganzinger, editor, *Proceedings of the Second European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 328–343, Nancy, France, March 1988.

[22] Tobias Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 306–317, Utrecht, The Netherlands, March 1993.

[23] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. In Carolyn L. Talcott, editor, *Special issue on continuations (Part II)*, LISP and Symbolic Computation, Vol. 7, No. 1, pages 57–81. Kluwer Academic Publishers, January 1994.

[24] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, July 1993.

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

[26] Kristoffer Høgsbro Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Universitetsparken 1, DK-2100

København Ø, February 1996. DIKU report 96/1, available from ⟨URL: http://www.diku.dk/research/published/96-1.ps.gz⟩.

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

[28] Amr Sabry and Philip Wadler. Compiling with reflections. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 13–24, Philadelphia, Pennsylvania, May 1996. ACM Press.

[29] David Sherman and Robert Strandh. Optimization of equational programs using partial evaluation. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 72–82, New Haven, Connecticut, June 1991. ACM Press.

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

[31] Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *HOA-93*, volume 816 of *LNCS*, pages 276–304. Springer-Verlag, 1993.

[32] Vincent van Oostrom and Femke van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. Technical Report CS-R9501, CWI, 1995.

Recent BRICS Report Series Publications