Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 204 (2008) 147–162

www.elsevier.com/locate/entcs

Computational Soundness of a Call by Name Calculus of Recursively-scoped Records

Elena Machkasova [1]

Division of Science and Mathematics
University of Minnesota, Morris
Morris, MN, U.S.

Abstract

The paper presents a calculus of recursively-scoped records: a two-level calculus with a traditional call-by-name λ-calculus at a lower level and unordered collections of labeled λ-calculus terms at a higher level. Terms in records may reference each other, possibly in a mutually recursive manner, by means of labels. We define two relations: a rewriting relation that models program transformations and an evaluation relation that defines a small-step operational semantics of records. Both relations follow a call-by-name strategy. We use a special symbol called a black hole to model cyclic dependencies that lead to infinite substitution. Computational soundness is a property of a calculus that connects the rewriting relation and the evaluation relation: it states that any sequence of rewriting steps (in either direction) preserves the meaning of a record as defined by the evaluation relation. The computational soundness property implies that any program transformation that can be represented as a sequence of forward and backward rewriting steps preserves the meaning of a record as defined by the small step operational semantics.
In this paper we describe the computational soundness framework and prove computational soundness of the calculus. The proof is based on a novel inductive context-based argument for meaning preservation of substituting one component into another.

Keywords: Calculus, call-by-name, computational soundness, recursively-scoped records

1 Introduction

In this work we present an untyped call-by-name calculus of *recursively-scoped records*. Recursively-scoped records (called *records* for the remainder of the paper) are unordered collections of labeled components that may reference each other, possibly in a mutually recursive manner. Representation of mutual dependencies arises in many calculi that model separate compilation, modules and linking, e.g. [1,15], dynamic code manipulation, e.g. [6], `letrec`, e.g. [13]. While our system has a much more modest set of features, it captures the essence of mutual dependencies – substitution with a possibility of cyclic dependencies.

[1] Email: `elenam@morris.umn.edu`

doi:10.1016/j.entcs.2008.03.059

We use a common approach pioneered by G. Plotkin in [11] of defining two relations in a system: a rewriting relation that represents program transformations and an evaluation relation that defines the meaning of a term via small-step operational semantics. Computational soundness connects the two relations: it implies that any two terms that are equivalent with respect to the rewriting relation have the same meaning as defined by the evaluation relation. Thus any program transformation that can be constructed as a sequence of rewriting steps (forward and/or backward) preserves the meaning of a term.

In our work both the rewriting relation and the evaluation follow the call-by-name strategy. Since this strategy allows β-reduction and substitution of unevaluated terms, the repertoire of transformations represented by the rewriting relation is greatly expanded to include unrestricted common subexpression elimination within a record, specialization, etc. The computational soundness result implies that all such transformations on mutually dependent components are meaning preserving, and thus can be used in a variety of systems modeling modules and linking, mutual dependencies in a `letrec` binding, etc. Our future plan is to investigate whether the meaning preservation result holds if the evaluation is restricted to a more efficient call-by-value strategy, while transformations follow a more liberal call-by-name one.

As demonstrated in section 5.2, our calculus fails to satisfy properties required for some previously known proof methods for computational soundness: it lacks confluence of the rewriting relation required for Plotkin's original proof method and it fails to satisfy *lift* and *project* properties required for the approach in [10]. We use a novel context-based approach to complete the proof. It is an open question whether the proof can also be completed using an alternative diagram-based approach in [14].

The main contribution of the paper is the computational soundness proof of a call-by-name calculus of mutually dependent components in the framework of term meaning defined via a small-step operational semantics. For more detailed presentation of this work, including proofs omitted here, see [8].

2 Related Work

G. Plotkin in [11] has proven a property equivalent to computational soundness for the call-by-name and the call-by-value term calculi. Z. M. Ariola and J. W. Klop studied issues of confluence and meaning preservation in similar systems of mutually dependent components. The straightforward definition of such a system breaks confluence (see [3]). In [4], in order to achieve confluence, substitution on cycles is disallowed. In [2] Z. M. Ariola and S. Blom show that unrestricted cyclic substitution is meaning preserving up to infinite unwindings of terms; their proof uses an approach that they call "skew-confluence".

In our earlier work [10,7], we proved computational soundness of a non-confluent call-by-value calculus of records similar to the one considered here. We developed and used a diagram-based proof method based on properties that we called *lift* and *project*. This approach has been further extended and generalized to a collection of abstract diagram-based proof methods in [14]. However, the system considered here does not meet the requirements of the lift and project method (see Section 5.2) and it is unclear whether it can be handled using diagram-based methods presented in [14]. Nevertheless, the novel inductive context-based method presented here allows us to prove computational soundness of the call-by-name system.

A recent independent work [12] by M. Schmidt-Schauß presents a proof of correctness of a *copy* rule (analogous to our substitution rule) in a call-by-need and a call-by-name settings. The proof approach uses the machinery of infinite trees and is significantly different from our context-based approach. The relation between the applicability of the two proof methods is a subject of future research.

3 Call-by-Name Calculus of Records

Records are unordered collections of labeled terms. Terms are elements of the traditional call-by-name λ-calculus [5], extended with constants, operations, and special symbols that represent interdependencies between terms. Each term in a record is marked by its unique label. The system can be viewed as a two-level calculus, with regular terms at the lower level and records at the upper level.

The term level of the calculus is defined below. We use prefix T for sets at the term level (such as $TTerm$), R is used at the level of records.

Definition 3.1 (Term-Level Calculus Syntax)

$$\begin{array}{lll}
M, N \in TTerm & ::= c \mid x \mid l \mid \bullet \mid \lambda x.M \mid M_1 \mathbin{@} M_2 \mid M_1 + M_2 \\
\mathbb{C} \in TContext & ::= \square \mid \lambda x.\mathbb{C} \mid \mathbb{C} \mathbin{@} M \mid M \mathbin{@} \mathbb{C} \mid \mathbb{C} + M \mid M + \mathbb{C} \\
\mathbb{E} \in TEvalContext & ::= \square \mid \mathbb{E} \mathbin{@} M \mid \mathbb{E} + M \mid c + \mathbb{E} \\
\mathbb{N} \in TNonEvalCntxt & \quad \mathbb{N} \in TContext, \mathbb{N} \notin TEvalContext
\end{array}$$

M, N denote terms, c stands for constants (such as numbers 1, 2, etc.), x, y, z are variables (distinct from constants), l stands for labels (distinct from variables and constants), $\bullet$ is a special symbol that denotes a black hole, i.e. a cyclic dependency of a record component on itself, $\lambda x.M$ is a lambda abstraction, $M_1 \mathbin{@} M_2$ is an application, $+$ is a binary operation on terms. For simplicity we use only addition in our examples, but other operations can be added. The scope of a lambda binding extends as far to the right as possible, unless limited by parentheses. It is straightforward to extend the calculus with booleans, conditionals, and other features, but for simplicity they are not considered here.

The set $FV(M)$ of free variables of a term M is defined as usual. Labels are distinct from variables, and are not included in $FV(M)$. Syntactic equivalence of terms is defined in [8] and follows the standard approach (see [5]). We use $=$ to denote equality up to α-renaming.

Contexts are used as a way of specifying a particular subterm in a term. We use $\mathbb{C}$ as a metavariable for a term context, $\mathbb{E}$ as a metavariable for a subset of general contexts called *evaluation contexts*, and $\mathbb{N}$ for the complement of this subset called *non-evaluation contexts.* The symbol $\square$ denotes a context hole. As an example, in the term $2 + \lambda x.3$ the subterm 2 appears in the context $\square + \lambda x.3$ (an evaluation context) and 3 appears in $2 + \lambda x.\square$ (a non-evaluation context, since $\square$ is under a λ). Definition 3.3 uses evaluation contexts as means to specify a subterm to be evaluated according to the evaluation relation. If a subterm appears in a non-evaluation context, it will not be reduced by evaluation.

$\mathbb{C}\{M\}$ denotes the result of filling the hole in the context $\mathbb{C}$ with the term M (we use the notation $\mathbb{C}\{M\}$ instead of the traditional $\mathbb{C}[M]$ to avoid confusion with record delimiters). For instance, if $\mathbb{C} = \lambda x.\square$ and $M = x+2$ then $\mathbb{C}\{M\} = \lambda x.x+2$. The notations for filling an evaluation context $\mathbb{E}$ and a non-evaluation context $\mathbb{N}$ are analogous. We can also fill a hole in a context with another context (denoted as $\mathbb{C}_1\{\mathbb{C}_2\}$), the result is a context. Note that it is possible to capture free variables of M when filling a context hole. Thus we do not introduce α-renaming of contexts. Definition 5.6 introduces contexts with multiple holes.

Definition 3.2 (Record-Level Calculus)

$$\begin{array}{lll}
D \in \mathit{RTerm} & ::= & [l_1 \mapsto M_1, ..., l_n \mapsto M_n],\ l_i \neq l_j \ \mathit{for}\ i \neq j \\
\mathbb{D} \in \mathit{RContext} & ::= & [l \mapsto \mathbb{C}, l_1 \mapsto M_1, \ldots, l_n \mapsto M_n], \mathbb{C} \in \mathit{TContext} \\
\mathbb{G} \in \mathit{REvalContext} & ::= & [l \mapsto \mathbb{E}, l_1 \mapsto M_1, \ldots, l_n \mapsto M_n], \mathbb{E} \in \mathit{TEvalContext}
\end{array}$$

D denotes a record with bindings of the form $l_i \mapsto M_i$. If $l \mapsto M$ occurs in a record, we say the term M is bound to the label l. We use notation $l \mapsto M \in D$ to indicate that the binding $l \mapsto M$ occurs in D. $L(D)$ denotes the set of all labels of D. We assume that all terms in a record are closed, i.e. for any record $D = [l_1 \mapsto M_1, ..., l_n \mapsto M_n]$ we have $\cup_{i=1}^{n} FV(M_i) = \emptyset$. Recall that labels are separate from variables and are not included in $FV(M)$.

The following is an example of a record: $[l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.x, l_3 \mapsto l_2 \mathbin{@} l_1]$. It has three components, labeled by l_1, l_2, and l_3, respectively. The term $2 + 3$ is bound to l_1, $\lambda x.x$ is bound to l_2, and the component bound to l_3 references the first two by applying one to the other.

Definition 3.2 also introduces two record-level contexts: a general record context $\mathbb{D}$ and record evaluation context $\mathbb{G}$. For instance, $[l_1 \mapsto 2 + \square, l_2 \mapsto \lambda x.x]$ is a record-level evaluation context (and also a general record context since evaluation contexts are a subset of general contexts). Record-level contexts are filled with terms, not with records. For instance, one may fill the above context with a term 3 obtaining the record $[l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.x]$.

Record components are unordered, i.e two records that differ only in the order of their components are considered equivalent. We define α-renaming of records as α-renaming of bound variables in their components (recall that records consist of closed terms). Since records are intended to be embedded in larger systems, such as program modules, record components may be referenced from outside of a record.

Thus there is no label renaming analogous to α-renaming of terms[2].

3.1 Calculus Relations

Both levels of the calculus follow the call-by-name reduction strategy. We define a *rewriting relation* $\rightarrow$ (which we also call *reduction*) and evaluation relation $\Rightarrow$ at the two levels of the calculus in definitions 3.3 and 3.5 respectively. Intuitively, the reduction relation represents transformations (i.e. "optimizations") of terms and records, and the evaluation relation represents the way records are evaluated at run-time by an evaluation engine (such as an interpreter). As discussed in section 5.1, the meaning of a record is defined by the result (called the *outcome*) of its evaluation.

At a more technical level, the difference between the two relations is that the rewriting relation reduces a redex in any context, while the evaluation reduces a redex in an evaluation context.

Definition 3.3 (Relations at the Term Level) *The rewriting relation $\rightarrow$ and the evaluation relation $\Rightarrow$ at the term level are defined as follows:*

$$
\begin{array}{lll}
(\lambda x.M) \mathbin{@} N & \leadsto M[x := N] & (\beta) \\
c_1 \ + \ c_2 & \leadsto c_3 \text{ where } c_3 \text{ is the result of the operation } + & (op) \\
\mathbb{E}\{R\} & \Rightarrow \mathbb{E}\{Q\} \text{ where } R \leadsto Q & \\
\mathbb{C}\{R\} & \rightarrow \mathbb{C}\{Q\} \text{ where } R \leadsto Q &
\end{array}
$$

The $\leadsto$ arrow denotes the "elementary" reduction, i.e. the basic operations at the term level of the calculus: a call-by-name β-reduction ($M[x := N]$ stands for the result of the capture-free substitution of N for x in M) and an operation (op) that replaces an operation on two constants by their result, also a constant. The rewriting relation $\rightarrow$ can perform an elementary reduction in any context $\mathbb{C}$, i.e. anywhere inside a term. The evaluation step $\Rightarrow$ performs the same operations but only in an evaluation context. $TEvalContext \subseteq TContext$ implies $\Rightarrow \subseteq \rightarrow$.

The term that is α-equivalent to the left-hand side of an elementary reduction rule is called *a term redex*. R denotes redexes. Intuitively, a redex is the subterm that gets reduced by the reduction. The redex is enclosed in a context that remains unchanged by the reduction[3]. As an example, in the reduction $\lambda x.2+3 \rightarrow \lambda x.5$ the redex is $2+3$ and the context is $\lambda x.\Box$. In the evaluation step $1+(\lambda x.x) \mathbin{@} 3 \Rightarrow 1+3$ the redex is $(\lambda x.x) \mathbin{@} 3$ and the context is $1+\Box$.

When writing $\mathbb{C}_1\{M_1\} = \mathbb{C}_2\{M_2\}$ we assume that we chose syntactically equivalent representatives of the α-equivalence classes of $\mathbb{C}_1\{M_1\}$ and $\mathbb{C}_2\{M_2\}$. See [8] for details.

[2] It is possible to add hidden components to records that cannot be referenced from outside of a record (see [10]). Records then are identified up to renaming of hidden labels. However, here we focus on computational soundness of mutually recursive components which is independent from the issue of hidden labels.

[3] More precisely, it is possible to find such representatives M, N in the two respective α-equivalence classes that $M \rightarrow N$ by reducing the given redex in the given context, and the context remains unchanged.

Lemma 3.4 states that a term may have at most one redex in an evaluation context or at most one label in such a context, but not both.

Lemma 3.4 *If $\mathbb{E}_1\{R_1\} = \mathbb{E}_2\{R_2\}$, where R_1, R_2 are redexes, then $\mathbb{E}_1 = \mathbb{E}_2$ and $R_1 = R_2$. If $M = \mathbb{E}_1\{l_1\} = \mathbb{E}_2\{l_2\}$ then $\mathbb{E}_1 = \mathbb{E}_2$ and $l_1 = l_2$ and $M \neq \mathbb{E}\{R\}$ for any $\mathbb{E}$ and R.*

3.1.1 Relations at the level of records.

Following the call-by-name strategy, both a reduction of a record component and a substitution from one component into another one may copy an unevaluated term.

Definition 3.5 (Relations at the Level of Records)

$$\begin{array}{lll}
\mathbb{D}\{R\} & \to \mathbb{D}\{Q\} \textit{ where } R \rightsquigarrow Q & (T) \\
\mathbb{D}\{l\} & \to \mathbb{D}\{N\} \textit{ where } l \mapsto N \in \mathbb{D}\{l\},\, \mathbb{D} \neq [l \mapsto \mathbb{E}, \ldots] & (S) \\
\mathbb{G}\{R\} & \Rightarrow \mathbb{G}\{Q\} \textit{ where } R \rightsquigarrow Q & (TE) \\
\mathbb{G}\{l\} & \Rightarrow \mathbb{G}\{N\} \textit{ where } l \mapsto N \in \mathbb{G}\{l\},\, \mathbb{G} \neq [l \mapsto \mathbb{E}, \ldots] & (SE) \\
[l_1 \mapsto \mathbb{E}\{l_1\}, ...] & \Rightarrow [l_1 \mapsto \bullet, ...] & (B1) \\
[l_1 \mapsto \mathbb{E}\{\bullet\}, ...] & \Rightarrow [l_1 \mapsto \bullet, ...] & (B2)
\end{array}$$

Definition 3.5 gives three kinds of reductions on records, two of which have an evaluation and a rewriting version. A *term reduction* simply reduces a term redex in one of the record's components. It is a rewriting step (see rule T) when it happens in a general context and an evaluation step (rule TE) when it is in an evaluation context. For example, $[l_1 \mapsto \lambda x.2 + 3] \to [l_1 \mapsto \lambda x.5]$ is a rewriting step, but not an evaluation step. Such steps are called *non-evaluation* steps (see Definition 3.6).

Substitution replaces a label occurring in a component of a record by the term bound to that label in the record. Analogously to the term reduction, the substitution is a rewriting step (rule S) if the label occurs in a general context, and an evaluation step (rule SE) if it occurs in an evaluation context.

For example, $[l_1 \mapsto 2 + 3, l_2 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto 2 + 3, l_2 \mapsto (2 + 3) + 1]$ is an evaluation step since $\square + 1$ is an evaluation context. The following substitution is a reduction, but not an evaluation step, since l_1 appears under a lambda: $[l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.l_1] \to [l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.(2 + 3)]$. Note that, just like a term reduction, the substitution is call-by-name: the term that gets substituted does not have to be evaluated first.

The side conditions $\mathbb{D}, \mathbb{G} \neq [l \mapsto \mathbb{E}, \ldots]$ eliminate an ambiguity between substitution and the black hole rule (B1) by preventing a substitution into a label that directly depends on itself in an evaluation context. For instance, the following substitution is not allowed: $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto l_1 + 1 + 1]$, the rule (B1) is applied instead (see below).

A *black hole symbol* $\bullet$ denotes apparent infinite substitution cycles that cannot be meaningfully evaluated. The rule (B1) introduces a black hole to replace a label that depends on itself in an evaluation context. For instance, $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto \bullet]$ instead of an infinite substitution $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto l_1 + 1 + 1] \Rightarrow \dots$ The notion of a black hole was first introduced in [3]. In this work it is essential for confluence of $\Rightarrow$ on records.

The rule (B2) turns a component that depends on a black hole into a black hole: $[l_1 \mapsto \bullet, l_2 \mapsto l_1 + 1] \overset{S}{\Longrightarrow} [l_1 \mapsto \bullet, l_2 \mapsto \bullet + 1] \overset{B2}{\Longrightarrow} [l_1 \mapsto \bullet, l_2 \mapsto \bullet]$.

The black hole rules do not have analogous non-evaluation rules since a self-dependency in a non-evaluation context may be a legitimate recursion and does not always lead to infinite substitution cycle or it may be eliminated during evaluation.

Definition 3.6 (Non-evaluation Relation and Closures) *The following notations are used at both the term and the record level:*

(i) *A non-evaluation relation $\hookrightarrow$ is defined as $\hookrightarrow = \rightarrow \setminus \Rightarrow$.*

(ii) *$\rightarrow^*$, $\Rightarrow^*$, $\hookrightarrow^*$ denote reflexive transitive closures of the respective relations; $\leftrightarrow$, $\overset{n}{\leftrightarrow}$ denote the reflexive symmetric transitive closures of $\rightarrow$ and $\hookrightarrow$, respectively.*

The non-evaluation relation $\hookrightarrow$ can be equivalently defined as a reduction in a non-evaluation context.

A normal form of a term with respect to a relation R is a term that cannot be further reduced by R. The definition is applicable to both terms and records.

Definition 3.7 (Normal Form) *Given a relation R on a set of terms, a normal form with respect to (w.r.t.) R is a term M for which there is no M' such that MRM'. The predicate $nf_R(M)$ is true if M is a normal form w.r.t. R and false otherwise. A term N is an R-normal form of M if MR^*N and $nf_R(N)$.*

4 Confluence of Evaluation

It follows from Lemma 3.4 that there is at most one evaluation step in any record component. For instance, if a component is of the form $\mathbb{E}\{R\}$, i.e. it has a term evaluation redex, it may not have a label in an evaluation context.

However, there is no ordering on components in a record, so any component that has a term or a substitution redex may be evaluated. Thus it is possible to have multiple evaluation steps originating at the same record:

$$[l_1 \mapsto 2 + 3, l_2 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto 5, l_2 \mapsto l_1 + 1]$$
$$[l_1 \mapsto 2 + 3, l_2 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto 2 + 3, l_2 \mapsto 2 + 3 + 1]$$

This flexibility opens a way for modeling separate compilation and evaluation of modules: evaluation of known components may start before the entire record becomes available.

Lemma 4.1 (Confluence of Evaluation) $\Rightarrow$ *is confluent on records.*

Proof. Case analysis on pairs of evaluation redexes shows that evaluation satisfies the strip lemma (see [5], Ch. 11) which implies confluence. See [9] for details [4]. □

The presence of a black hole in the calculus is essential for confluence of evaluation. Consider the following record: $[l_1 \mapsto 2 + l_2, l_2 \mapsto l_1 + 1]$. Note that both labels are in evaluation contexts in both components. Without a black hole the substitution into the first component would yield $[l_1 \mapsto 2 + l_1 + 1, l_2 \mapsto l_1 + 1]$, substitution into the second component gives $[l_1 \mapsto 2 + l_2, l_2 \mapsto 2 + l_2 + 1]$. In the first resulting record both components reference l_1, in the second one they both reference l_2, and any subsequent substitutions preserve these properties. This is a variation of a famous non-confluence example introduced in [3].

However, a black hole allows us to bring these two records together by a sequence of evaluation steps since both labels appear in an evaluation context, and thus represent an infinite cycle of substitutions:

$$\begin{array}{ll}
[l_1 \mapsto 2 + l_1 + 1, l_2 \mapsto l_1 + 1] & \Rightarrow \\
[l_1 \mapsto \bullet, l_2 \mapsto l_1 + 1] & \Rightarrow \\
[l_1 \mapsto \bullet, l_2 \mapsto \bullet + 1] & \Rightarrow \\
[l_1 \mapsto \bullet, l_2 \mapsto \bullet] &
\end{array}$$

The record $[l_1 \mapsto 2 + l_2, l_2 \mapsto 2 + l_2 + 1]$ also evaluates to $[l_1 \mapsto \bullet, l_2 \mapsto \bullet]$.

Confluence of evaluation guarantees uniqueness of a normal form w.r.t. $\Rightarrow$ if a term has one. We also prove that a record may not have a normal form and diverge at the same time (this property, known as *uniform normalization*, is not automatically implied by confluence).

Lemma 4.2 *If $D \Rightarrow^* D'$, $nf_{\Rightarrow}(D')$, and no component in D' is bound to $\bullet$, then there is no infinite sequence $D \Rightarrow D_1 \Rightarrow D_2 \Rightarrow^* \dots$.*

4.1 An Efficient Evaluation Strategy

Confluence of evaluation guarantees that no matter what path an evaluation of a record takes, all of the resulting records can be evaluated to the same record. We do not want to fix the order of evaluating components since we would like to have the flexibility of modeling systems where progress can be made on evaluating a record before all of its components become available or where components may be evaluated in parallel.

However, for proving properties of our calculus it is convenient to impose a particular order of evaluation that we call *efficient evaluation strategy.* Intuitively, this strategy requires that if a component bound to l_1 needs a component bound to l_2 (i.e. the component bound to l_1 is of the form $\mathbb{E}\{l_2\}$) then the term bound to l_2 must be completely evaluated (i.e. it has neither term redexes nor substitution

[4] The black hole rules in the system in [9] differ slightly from the rules presented here. However, the difference does not affect the essence of the proof.

redexes) before the substitution into the component bound to l_1 is made. This strategy imposes a partial order on components. The process stops if it discovers a cycle of mutual evaluation dependencies.

The formal definition depends on the partial function $\mathrm{next}(D, l)$ that defines the label of the component in which the next evaluation step takes place in order to make progress on evaluation of the component bound to l in D.

Definition 4.3 (Next Component To Be Evaluated) *Let $l \mapsto M \in D$. A function $next(D, l) : RTerm \times L(D) \to L(D) \cup \{\bullet\}$ is defined as follows:*

(i) *If $M = \mathbb{E}\{R\}$ then $next(D, l) = l$,*

(ii) *If $M = \mathbb{E}\{\bullet\}$ or $M = \mathbb{E}\{l\}$ then $next(D, l) = \bullet$,*

(iii) *If $M = \mathbb{E}\{l'\}$ then:*

- (a) *If $next(D, l')$ is undefined, $next(D, l) = l$,*
- (b) *If $next(D, l') = \bullet$ or l' is bound to $\mathbb{E}'\{l\}$ or there is a sequence of labels $l_1, \ldots, l_n \in L(D)$, $n \geq 1$, such that D is of the form*

$$[l \mapsto \mathbb{E}\{l'\}, l' \mapsto \mathbb{E}_1\{l_1\}, \ldots, l_i \mapsto \mathbb{E}_i\{l_{i+1}\}, \ldots, l_n \mapsto \mathbb{E}_n\{l\}, \ldots]$$

  then $next(D, l) = \bullet$,
- (c) *Otherwise $next(D, l) = next(D, l')$.*

(iv) *Otherwise $next(D, l)$ is undefined.*

If $\mathrm{next}(D, l)$ is undefined then the component bound to l is fully evaluated.

Let $\mathcal{L}$ denote an ordered sequence of distinct labels; $\mathcal{L}_1 \preceq \mathcal{L}_2$ means that $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$ or $\mathcal{L}_1 = \mathcal{L}_2$. An efficient evaluation strategy follows the sequence of labels in $\mathcal{L}$ as a sequence of "goals".

Definition 4.4 (Efficient Evaluation Strategy) *Given a record D and a label l, an efficient evaluation strategy starting at l is a sequence of evaluation steps $D_1 \Rightarrow D_2 \Rightarrow \ldots \Rightarrow D_n$ s.t. for all $i < n$ $next(D_i, l)$ is defined and not equal to $\bullet$ and an evaluation step $D_i \Rightarrow D_{i+1}$ evaluates the component bound to $next(D_i, l)$. We denote this sequence as $D \overset{l}{\underset{e}{\Rightarrow}}{}^* D_n$.*

Given a sequence $\mathcal{L} = l_1, l_2, \ldots l_n$ s.t. $l_i \in L(D)$ for all i, an efficient strategy w.r.t. $\mathcal{L}$ is a sequence $D \overset{l_1}{\underset{e}{\Rightarrow}}{}^* D_1 \overset{l_2}{\underset{e}{\Rightarrow}}{}^* \ldots \overset{l_n}{\underset{e}{\Rightarrow}}{}^* D_n$ s.t. $next(D_i, l_j)$ is undefined for all $j < i$ for $1 \leq i \leq n$ (i.e. each component l_j in $\mathcal{L}$ is fully evaluated before evaluation of l_i starts). Note that it is possible that $next(D_n, l_n)$ is not undefined. An efficient strategy w.r.t. $\mathcal{L}$ is denoted $\overset{\mathcal{L}}{\underset{e}{\Rightarrow}}{}^*$.

The efficient evaluation strategy stops if it discovers that a record component evaluates to a black hole since such records represent divergence (see Definition 5.2). Thus, if $\mathrm{next}(D, l) = \bullet$, no evaluation takes place.

The strategy is called "efficient" because it evaluates a component only once - the first time it is needed. Since no unevaluated components are copied, no computation is duplicated. This is similar to a call-by-value or a call-by-need strategy. However, unlike the call-by-value strategy, it does not require that a component evaluates to a value before it can be substituted (traditionally only constants, variables, and λ-abstractions are considered values), only to a substitution-free normal form which includes errors. If a record D evaluates to a normal form D' then efficient evaluation strategy with any choice of $\mathcal{L}$ that includes all labels in $L(D)$ reaches D'. The strategy detects cycles of substitution as early as possible since the evaluation follows component dependencies as far as possible before evaluating any of them.

Below is an evaluation sequence that follows the efficient strategy w.r.t. l_1. On the left on line i we show the value of $\text{next}(D_i, l_1)$. For simplicity we write just $\text{next}(l)$ instead of $\text{next}(D, l)$ since the record on each line is obvious.

$$\begin{array}{lll}
\text{next}(l_1) = l_3 & [l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] & \Rightarrow \\
\text{next}(l_1) = l_2 & [l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) = l_2 & [l_1 \mapsto l_2, l_2 \mapsto 4 + 2, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) = l_1 & [l_1 \mapsto l_2, l_2 \mapsto 6, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) \text{ is undefined} & [l_1 \mapsto 6, l_2 \mapsto 6, l_3 \mapsto 4] &
\end{array}$$

In contrast the step below does not follow the efficient strategy: the redex $l_3 + 2$ is duplicated so it will have to be evaluated twice, possibly duplicating evaluation of $1 + 3$ as well. Note, however, that the record eventually evaluates to the same one as in the efficient evaluation sequence above.

$$\begin{array}{ll}
[l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] & \Rightarrow \\
[l_1 \mapsto l_3 + 2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] & \Rightarrow \dots
\end{array}$$

If a record D has a normal form in which no component is bound to a black hole, it is possible to reach the normal form using an efficient evaluation strategy with any sequence $\mathcal{L}$ that includes all labels in $L(D)$ (see [8] for the proof).

5 Computational Soundness of the Calculus

5.1 *Definition of Computational Soundness*

Computational soundness states that rewriting relation in the calculus preserves the meaning of terms. A term's meaning is given by its normal form w.r.t. evaluation relation if such a normal form exists, otherwise the "meaning" is divergence. The notion of *outcome* in Definition 5.2 formalizes this idea. Some normal forms may be syntactically different, but have the same "meaning". The classification function (Definition 5.1) groups terms based on their "meaning".

5.1.1 *Classification function.*

In order to define a term's meaning, we partition all terms into *equivalence classes*. The function that assigns a class to a term is called a *classification* (the term first introduced in [7]). Two elements of the same class have the same meaning (however, they may be further distinguished by supplying a context that uses them). For instance, at the term level it is reasonable to make constants 2 and 3 be in different classes since their meaning is clearly different. However, it is common to group all lambda abstractions in the same class since a function by itself is not distinguishable from any other function until it is applied.

It is possible to define different classification functions for the same calculus. For instance, one may wish to distinguish between different types of errors by further subdividing the **error** class. On the other hand, if one is only concerned with proving termination equivalence then all normal forms may be placed into one class[5]. A calculus may be computationally sound for one choice of classification and unsound for another.

The classification function used in this work is defined in Definition 5.1. For simplicity we use the same notation Cl for the classification function at both levels of the calculus. This function is very similar to the one used in [7], except for the inclusion of a black hole. Since record components contain only closed terms, we do not have term-level classes for variables: a label bound to a variable would be considered an error. The function is well-defined on α-equivalence classes both at the term and at the record level.

Definition 5.1 (Classification) *The classification function $Cl : TTerm \cup RTerm \to S$, where S is a set of equivalence classes, is defined as follows:*

- $Cl(M) = \mathbf{eval}$ *if* $M = \mathbb{E}\{R\}$, *R is a redex. Such terms are called evaluatable.*
- $Cl(c) = \mathbf{const}(c)$, *where* $\mathbf{const}(c_1) = \mathbf{const}(c_2)$ *if and only if* $c_1 = c_2$
- $Cl(\bullet) = \bullet$
- $Cl(\lambda x.N) = \mathbf{abs}$
- $Cl(\mathbb{E}\{l\}) = \mathbf{stuck}(l)$, *where* $\mathbf{stuck}(l_1) = \mathbf{stuck}(l_2)$ *if and only if* $l_1 = l_2$
- $Cl(M) = \mathbf{error}$ *if M does not belong to any of the above categories*
- $Cl([l_1 \mapsto M_1, \ldots l_n \mapsto M_n]) = [l_1 \mapsto Cl(M_1), \ldots l_n \mapsto Cl(M_n)]$ *if* $Cl(M_i) \neq \bullet$ *for all i s.t.* $1 \leq i \leq n$
- $Cl([\ldots, l_i \mapsto \bullet, \ldots]) = \perp$

An equivalence class of a record D with no label bound to a black hole is an unordered collection of labeled term-level classes corresponding to components of D. For instance, $Cl([l_1 \mapsto \lambda x.x, l_2 \mapsto l_1 \mathbin{@} 1]) = [l_1 \mapsto \mathbf{abs}, l_2 \mapsto \mathbf{stuck}(l_1)]$.

Since a black hole represents an infinite substitution, the class of a record with a black-hole-bound component is $\perp$. Note that a record with a black hole in a non-evaluation context does not necessarily diverge, and thus is not classified as $\perp$: consider $[l \mapsto (\lambda x.1) \mathbin{@} \bullet] \Rightarrow [l \mapsto 1]$, the latter record is a normal form.

The following property, called *class preservation*, is important for proving computational soundness: if $D_1 \hookrightarrow D_2$ (recall Definition 3.6) then $Cl(D_1) = Cl(D_2)$.

The above classification groups all abstractions in one class. However, this does

[5] Records with at least one component bound to a black hole should be classified as diverging

not mean that replacing an abstraction by any other one may be considered meaning preserving. One can always distinguish two semantically different abstractions by considering them in a record with a term that applies the abstraction to an argument. A transformation is provably meaning preserving if its results are the same no matter what other components appear in a record. Since we can assume that any abstraction bound to a label is applied to arbitrary terms in other components, transformations must preserve the actual behavior of abstractions. [7] formalizes this notion via record contexts which we do not present here due to lack of space.

5.1.2 Outcome and Computational Soundness.

Classification characterizes a record at a given moment, while *outcome* characterizes its "ultimate fate" - what happens to it if it gets evaluated as far as possible.

Definition 5.2 (Outcome) *The outcome of a record D, denoted $Outcome(D)$, is $Cl(D')$ where D' is the normal form of D w.r.t. $\Rightarrow$ if D has a normal form or a symbol $\perp$ if evaluation of D diverges.*

Lemmas 4.1 and 4.2 guarantee that the outcome is well-defined since every record either has a unique normal form or diverges on all evaluation paths (we identify a label bound to a black hole with divergence). The outcome formalizes the notion that the meaning of a term is the result of its evaluation.

Definition 5.3 (Meaning Preservation and Computational Soundness) *A relation R is meaning preserving if MRN implies that $Outcome(M) = Outcome(N)$. A calculus is computationally sound if $\leftrightarrow$ is meaning preserving.*

By confluence and uniform normalization (Lemma 4.1, 4.2) $\Rightarrow$ is meaning preserving.

5.2 Proof Methods and Their Applicability

Historically various methods have been used for proving computational soundness. Plotkin's method in [11] requires *confluence of the rewriting relation* in the calculus. However, many recently developed calculi model such inherently non-confluent features of programming languages as mutually dependent components. The repertoire of proof methods has been expanded to relax requirements on the calculus. In this section we review some of these proof methods and discuss why they are not applicable to our calculus.

Failure of Confluence and Standardization Method. The traditional method for computational soundness proofs has three requirements: confluence of the rewriting relation, standardization (a property that relates the rewriting relation and the evaluation relation), and the class preservation property defined in Section 5.1.1 (see [7] for detailed discussion). However, in our system $\rightarrow$ is non-confluent. The non-confluence example below is based on that in [3]. It also appears in the call-by-value version of our calculus described in [7,10]. Recall that confluence is preserved when both reductions are evaluation steps, see section 4.

Example 5.4 Consider a record $[l_1 \mapsto \lambda x.l_2, l_2 \mapsto \lambda y.l_1]$. By reducing each of the two redexes we obtain these two records: $[l_1 \mapsto \lambda x.\lambda y.l_1, l_2 \mapsto \lambda y.l_1]$ and $[l_1 \mapsto$

$$\begin{tikzcd}
D_1 \arrow[r, Rightarrow, dashed, "*"] \arrow[d, hook] & D_4 \arrow[d, hook, dashed, "*"] \\
D_2 \arrow[r, Rightarrow, "*"] & D_3
\end{tikzcd}
\qquad
\begin{tikzcd}
D_1 \arrow[r, Rightarrow, "*"] \arrow[d, hook] & D_2 \arrow[r, Rightarrow, dashed, "*"] & D_4 \arrow[d, hook, dashed, "*"] \\
D_3 \arrow[rr, Rightarrow, dashed, "*"] & & D_5
\end{tikzcd}$$

Fig. 1. Lift and Project properties.

$\lambda x.l_2, l_2 \mapsto \lambda y.\lambda x.l_2]$. The only reductions that can originate from these records are substitutions. No matter what substitutions we perform on the records, they cannot be reduced to a common one since in the first one both components always reference l_1, and in the second record they reference l_2.

Failure of Lift and Project Method. In [7,10] we use an approach based on three properties of the calculus: the *lift*, and *project* properties defined in Definition 5.5, and the class preservation property in Section 5.1.1 to prove computational soundness of a call-by-value calculus of recursively-scoped records. The project property "projects" a given evaluation sequence down, and the lift property "lifts" a given sequence up, according to the diagram layout in Figure 1[6].

Definition 5.5 (Lift and Project) *A calculus has the lift property if, given $D_1 \hookrightarrow D_2 \Rightarrow^* D_3$, there exists D_4 s.t. $D_1 \Rightarrow^* D_4 \hookrightarrow^* D_3$. A calculus has the project property if, given $D_1 \Rightarrow^* D_2$ and $D_1 \hookrightarrow D_3$, there exist D_4, D_5 s.t. $D_2 \Rightarrow^* D_4 \hookrightarrow^* D_5$ and $D_3 \Rightarrow^* D_5$.*

Even though the current system is very similar to the one considered in [7,10], the call-by-name nature of substitution breaks the lift and the project properties, as shown by the following counterexample. The right hand side non-evaluation arrow pointing up contradicts the properties.

$$\begin{tikzcd}
{[l_1 \mapsto 2+3, l_2 \mapsto \lambda x.l_1]} \arrow[r, Rightarrow] \arrow[dd, hook] & {[l_1 \mapsto 5, l_2 \mapsto \lambda x.l_1]} \arrow[d, hook] \\
 & {[l_1 \mapsto 5, l_2 \mapsto \lambda x.5]} \\
{[l_1 \mapsto 2+3, l_2 \mapsto \lambda x.2+3]} \arrow[r, Rightarrow] & {[l_1 \mapsto 5, l_2 \mapsto \lambda x.2+3]} \arrow[u, hook]
\end{tikzcd}$$

Applicability of Other Diagram-Based Methods. The lift and project method has been extended and generalized in [14]. While it is possible that a form of the approach presented there, known as *lift/project when terminating* (or LPT), is applicable, we have not been able to construct such a proof.

A black hole, which is technically a normal form, may require a modification of the LPT approach. In our system a non-evaluation step may convert a record with a component evaluating to black hole to a diverging record, as shown below. Diagram-based methods generally do not equate diverging terms with normal forms.

[6] In diagrams double arrows represent $\Rightarrow$, single arrows $\rightarrow$, arrows with a hook are $\hookrightarrow$. Solid arrows are the given relations, dashed arrows are the ones claimed to exist. See Definition 3.6 for closure notations.

Note that the outcome of both records is $\bot$ so the meaning is preserved.

$$\begin{array}{lll}
[l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_1] & \Rightarrow & [l_1 \mapsto (\lambda x.l_1) \mathbin{@} 2, l_2 \mapsto \lambda x.l_1] \quad \Rightarrow \\
[l_1 \mapsto l_1, l_2 \mapsto \lambda x.l_1] & \Rightarrow^* & [l_1 \mapsto \bullet, l_2 \mapsto \lambda x.l_1] \\
[l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] & \Rightarrow & [l_1 \mapsto (\lambda x.l_2 \mathbin{@} 2) \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] \Rightarrow \\
[l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] & \Rightarrow & \ldots
\end{array}$$

5.3 Context-Based Proof of Computational Soundness

Meaning Preservation of the Term Reduction. The meaning preservation property of a term reduction can be proven using the lift and project approach with the machinery of marked redexes and residuals. The proof is similar to that for the call-by-value calculus in [7]. See [8] for details.

Meaning Preservation of Substitution. We show that substitution preserves the outcome of a record. A key idea of the proof is to use the efficient evaluation strategy (see Definition 4.4) to guarantee that each component is only evaluated only once, the first time it is needed.

Definition 5.6 (Multi-hole contexts) *A multi-hole context* $\mathbb{M}$ *is defined as*

$$\mathbb{M} ::= \Box \mid M \mid \lambda x.\mathbb{M} \mid \mathbb{M} + \mathbb{M} \mid \mathbb{M} \mathbin{@} \mathbb{M}$$

Contexts $\mathbb{M}$ are filled with terms in the same manner as single-hole contexts.

Multi-hole contexts allow us to formalize the notion that two records differ only by replacing some occurrences of a term M_1 by M_2.

Definition 5.7 *A record* D_1 *is called* (M_1, M_2)*-similar to a record* D_2 *(denoted* $D_1 \sim^{M_1}_{M_2} D_2$*) if there exist multi-hole contexts* $\mathbb{M}_1, \ldots, \mathbb{M}_n$ *s.t.*

$$D_1 = [l_1 \mapsto \mathbb{M}_1\{M_1, \ldots, M_1\}, \ldots, l_n \mapsto \mathbb{M}_n\{M_1, \ldots, M_1\}],$$
$$D_2 = [l_1 \mapsto \mathbb{M}_1\{M_2, \ldots, M_2\}, \ldots, l_n \mapsto \mathbb{M}_n\{M_2, \ldots, M_2\}].$$

Lemma 5.8 *(see Figure 2) Let* $D_1 = [l \mapsto M, l' \mapsto \mathbb{N}\{l\}, \ldots] \overset{S}{\hookrightarrow} [l \mapsto M, l' \mapsto \mathbb{N}\{M\}, \ldots] = D_2$ *and* $D_1 \Rightarrow^* D'_1$ *(recall that* $\mathbb{N}$ *is a non-evaluation context) and let* $\mathcal{L} \preceq l, l', l_1, \ldots, l_n$*, where* $l_1 \ldots l_n$ *is a sequence of labels in* $L(D_1)$*,* $n \geq 0$*, and* $l \neq l_i$*,* $l' \neq l_i$ *for all* $1 \leq i \leq n$*. It is possible that* $l = l'$*. Then*

- *If* $D_1 \overset{\mathcal{L}}{\underset{e}{\Rightarrow}}{}^* D'_1$ *then there exists* D'_2 *s.t.* $D_2 \overset{\mathcal{L}}{\underset{e}{\Rightarrow}}{}^* D'_2$*,* $D'_1 \sim^{l}_{M} D'_2$*, and* *Outcome*$(D'_1) = \bot$ *if and only if Outcome*$(D'_2) = \bot$*.*
- *If* $D_2 \overset{\mathcal{L}}{\underset{e}{\Rightarrow}}{}^* D'_2$ *then there exist* D'_1, D''_2 *s.t.* $D_1 \overset{\mathcal{L}}{\underset{e}{\Rightarrow}}{}^* D'_1$ *and* $D'_2 \overset{l_n}{\underset{e}{\Rightarrow}}{}^* D''_2$*,* $D''_2 \sim^{l}_{M} D'_1$*, and Outcome*$(D'_1) = \bot$ *if and only if Outcome*$(D'_2) = \bot$*.*

$$\begin{tikzcd}
D_1 \arrow[r, Rightarrow, "*"] \arrow[d, hook] & D_1' \arrow[d, squiggly, dash, "\sim^l_M"] \\
D_2 \arrow[r, Rightarrow, dashed, "*"] & D_2'
\end{tikzcd}
\qquad
\begin{tikzcd}
D_1 \arrow[rr, Rightarrow, dashed, "*"] \arrow[d, hook] & & D_1' \arrow[d, squiggly, dash, "\sim^l_M"] \\
D_2 \arrow[r, Rightarrow, "*"] & D_2' \arrow[r, Rightarrow, dashed, "*"] & D_2''
\end{tikzcd}$$

Fig. 2. Lemma 5.8: (l, M)-similarity (denoted by a wave-like line) preserved by $\Rightarrow$

Lemma 5.8 states the key property for meaning preservation of a substitution step: the original and the transformed records remain (l, M)-similar after any number of steps in the evaluation sequence that follows the efficient strategy with the sequence of labels given in the lemma.

The efficient evaluation strategy guarantees that every component gets evaluated to a normal form *before* it gets substituted into any other component. The strategy attempts to evaluate the term M (bound to l) in both records. By Lemma 5.8 if such an evaluation terminates with a black-hole-free term in one record, it does so in the other. In this case all components needed for evaluating M have been evaluated as well, so future evaluation of M gives the same result. In an example below the initial substitution occurs in the second component, and evaluating l requires evaluating l''; the corresponding records in the two sequences are $(l, l'' + 2)$-similar:

$$\begin{aligned}
&[l \mapsto l'' + 2, l' \mapsto (\lambda x.l) \mathbin{@} 1, l'' \mapsto 3 + 1] &&\Rightarrow^* [l \mapsto 6, l' \mapsto (\lambda x.l) \mathbin{@} 1, l'' \mapsto 4]\\
&[l \mapsto l'' + 2, l' \mapsto (\lambda x.l'' + 2) \mathbin{@} 1, l'' \mapsto 3 + 1] &&\Rightarrow^* [l \mapsto 6, l' \mapsto (\lambda x.l'' + 2) \mathbin{@} 1, l'' \mapsto 4]
\end{aligned}$$

The sequences continue with $(l, l''+2)$-similar records until both arrive at an identical result $[l \mapsto 6, l' \mapsto 6, l'' \mapsto 4]$. See [8] for other cases of component dependencies.

We show that if two normal forms D_1, D_2 are (l, M)-similar then $Cl(D_1) = Cl(D_2)$. Thus non-evaluation substitution preserves the outcome.

Evaluation steps preserve the outcome since $\Rightarrow$ is confluent. We have shown that both a term reduction non-evaluation step and a non-evaluation substitution preserve the outcome. Thus we have the desired computational soundness result:

Theorem 5.9 *If $D_1 \leftrightarrow D_2$ then $Outcome(D_1) = Outcome(D_2)$.*

6 Conclusions and Future Work

We have proven that the call-by-name calculus of recursively-scoped records is computationally sound. Our system captures the essential features of mutually recursive components. We plan to study applicability of our proof method to more complex systems with possible cyclic dependencies, such as `letrec` calculi and more sophisticated systems that model modules and linking. We will also investigate how the context method compares to other methods of proving computational soundness.

Acknowledgement

Many thanks to Emily Christiansen who participated in the early stages of this research, to Dr. Manfred Schmidt-Schauß for helpful discussions, to Dr. Franklyn Turbak and Dr. Nicholas McPhee for detailed feedback on drafts of this paper, and to anonymous WRS reviewers for many helpful comments.

References

[1] Davide Ancona and Elena Zucca: A calculus of module systems. Vol. 12, 2002, pp. 91-132.

[2] Z. M. Ariola, Stefan Blom: Skew confluence and the lambda calculus with letrec. Annals of pure and applied logic 117/1-3, 97-170, 2002

[3] Z. M. Ariola and J. W. Klop: Equational Term Graph Rewriting. Fundamentae Informaticae, Vol. 26, Nrs. 3,4, June 1996. p. 207-240.

[4] Z. M. Ariola, J. W. Klop: Lambda calculus with explicit recursion. Journal of Information and Computation, Vol. 139 (2): 154-233, 1997.

[5] H. P. Barendregt: The Lambda Calculus, its Syntax and semantics. Studies in Logic, volume 103, Elsevier Science Publishers, 1984.

[6] Sonia Fagorzi and Elena Zucca: A Calculus for Reconfiguration: (Extended abstract). Electr. Notes Theor. Comput. Sci., Vol. 135, N. 3, 2006, pp. 49-59.

[7] E. Machkasova: Computational Soundness of Non-Confluent Calculi with Applications to Modules and Linking, Ph.D. dissertation, April 2002, Boston University

[8] E. Machkasova: Computational Soundness of a Call by Name Calculus of Recursively-scoped Records. Working Papers Series, University of Minnesota, Morris, Volume 2 Number 3, 2007. Available at http://cda.morris.umn.edu/~elenam/

[9] E. Machkasova, E. Christiansen: Call-by-name Calculus of Records and its Basic Properties. Working Papers Series, University of Minnesota, Morris, Volume 2 Number 2, 2006 (updated 2007). Available at http://cda.morris.umn.edu/~elenam/

[10] E. Machkasova, F. Turbak: A calculus for link-time compilation. In Programming Languages & Systems, 9th European Symp. Programming, volume 1782 of LNCS, pages 260-274 Springer-Verlag, 2000

[11] G. D. Plotkin: Call-by-name, call-by-value and the lambda calculus. Theoret. Comput. Sci., 1, 1975.

[12] M. Schmidt-Schauß: Correctness of copy in calculi with letrec, case and constructors. Frank report 28, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, February 2007.

[13] Manfred Schmidt-Schauß and Michael Huber: A lambda-calculus with letrec, case, constructors and non-determinism. In First International Workshop on Rule-Based Programming, 2000.

[14] J. B. Wells, Detlef Plump, and Fairouz Kamareddine: Diagrams for meaning preservation. In Rewriting Techniques & Applications, 14th Int'l Conf., RTA 2003, volume 2706 of LNCS, pp. 88-106. Springer-Verlag, 2003

[15] J. B. Wells and René Vestergaard: Equational reasoning for linking with first-class primitive modules. In Programming Languages & Systems, 9th European Symp. Programming, volume 1782 of LNCS, pages 412-428. Springer-Verlag, 2000.