Correctness of Copy in Calculi with Letrec, Case and Constructors

Manfred Schmidt-Schauß

FB Informatik und Mathematik, Institut für Informatik, J.W.Goethe-Universität,
Robert-Mayer-Str 11-15, Postfach 11 19 32,
D-60054 Frankfurt, Germany,
schauss@ki.informatik.uni-frankfurt.de

Technical Report Frank-28

1. February 2007

Abstract. Call-by-need lambda calculi with letrec provide a rewriting-based operational semantics for (lazy) call-by-name functional languages. These calculi model the sharing behavior during evaluation more closely than let-based calculi that use a fixpoint combinator. In a previous paper we showed that the copy-transformation is correct for the small calculus LRλ. In this paper we demonstrate that the proof method based on a calculus on infinite trees for showing correctness of instantiation operations can be extended to the calculus LRCCλ with case and constructors, and show that copying at compile-time can be done without restrictions. We also show that the call-by-need and call-by-name strategies are equivalent w.r.t. contextual equivalence. A consequence is correctness of all the transformations like instantiation, inlining, specialization and common subexpression elimination in LRCCλ.
We are confident that the method scales up for proving correctness of copy-related transformations in non-deterministic lambda calculi if restricted to "deterministic" subterms.

1 Introduction

A good semantics that supports all phases from programming, compiling, verification, and optimization to execution is indispensable for the reliable application of a programming language. Extended lambda calculi are widely used to provide operational semantics for programming languages, e.g. the semantics of non-strict functional programming languages like Haskell [18] and Clean [19] can be defined by a small-step (rewriting) reduction in a lambda calculus. An efficient evaluation of programs in these languages is based on call-by-need evaluation that implements call-by-name evaluation by exploiting sharing of subexpressions in order to avoid multiple evaluation of the same subexpression. Hence it is important to investigate lambda calculi having a possibility to represent

sharing of subexpressions, which is usually made explicit by let-expressions or by recursive let-expressions [5,1,3,4,14].
Reasoning about the semantics requires a notion of equality. Conversion equality is the classic variant, which is known to be inadequate, since not all useful equations can be justified. Defining equality as observational equality, also known as contextual equality, regards expressions as equal, if they cannot be distinguished by all permitted observations, where contextual equality defines as observation the convergence of $C[s]$ for any context C, i.e. s, t are observationally equivalent, if for all contexts C: $C[s]$ converges iff $C[t]$ converges. This is also the coarsest equality for this observation, and justifies correctness of a maximal number of program transformations.

1.1 Call-by-Name, Call-by-Need and Lambda-Calculi with Let or Letrec

An early and influential comparison between different implementations of lambda-calculi was Plotkin's [20] treatment of call-by-value, call-by-name strategies and different abstract machines as implementations, where Plotkin used, besides conversion, also contextual equivalence for comparing expressions. One result in [20] is that call-by-value and call-by-name are essentially different in the considered lambda calculi. Comparing these strategies with call-by-need leads as a natural approach to extending the lambda-calculus syntax by `let` or `letrec`. It is well-known that non-recursive let-expressions can be simulated by an application (see e.g. [5]). It is also well-known that `letrec` has improved sharing properties during reductions (see e.g. [1,12]), even better than an encoding using the fixpoint combinator Y, and also allows in several cases to syntactically detect non-termination during evaluation.
In calculi with sharing an important issue is in which cases an improvement of sharing is permitted, or the contrary, which kind of unsharing is permitted, perhaps to enable other program transformations. Note that in non-deterministic calculi, arbitrarily modifying sharing is in general not correct, but correct in special cases (see e.g. [15,16]). There are also undecided cases, for which the issue of correctness is unsolved, see the letrec-calculi in [16,8]. In the deterministic `letrec`-calculus treated in [23], the copy reduction is proved only correct if the expression is not copied into an abstraction. The technical problem of showing correctness of the general copy-transformation is that proofs based on diagrams or rearranging the reduction do not work. Even the proofs that the restricted copy-reduction, where only abstractions or variables are allowed to be copied, is intricate and requires splitting the reduction and a complex measure on reduction sequences [23,21].
There are several papers investigating the relationship between call-by-name and call-by-need calculi (e.g. [5,4,14]). Other work on lambda-calculi extended with letrec is centered around confluence, non-confluence or variants of confluence of the reduction relation of the calculi ([3,1,2]). A proof of the observational equivalence of a call-by-name and a call-by-need calculus with non-recursive let is in [14], which also mentions at the very end an open question, which can be

reformulated as the question, whether a letrec-calculus with a reduction that allows only to copy values is strong enough to show also that the equation that allows copying arbitrary expressions holds. A similar question is also implicitly mentioned as unresolved in [4]. As far as we know, there is no proof for this equality w.r.t. contextual equivalence for a calculus using recursive let and call-by-need reductions.

The paper [10] provides a fully abstract denotational semantics for a deterministic extended lambda-calculus with letrec, and a "referential transparency" property is proved. This means for a calculus similar to LRλ, which is LRCCλ without case, constructors and seq, that the copy-transformation is correct w.r.t. contextual equality. The correctness of the copy transformation for LRλ could be derived from this result, however, the term representation in [10] presupposes the correctness of the transformation $C[s] \to (\texttt{letrec}\ x = s\ \texttt{in}\ C[x])$, which would require a correctness proof in LRλ, and moreover, the denotational method does not support the comparison of the different evaluation strategies. Also, it is not possible via using the result in [10], to resolve the open problem in [14,4], since these concern evaluation strategies. It is not possible to derive our result for LRCCλ from [10]. It is unclear whether Jeffrey's denotational proofs can be used for a calculus with case and constructors, since confluence does no longer hold in LRCCλ (see [3]), but his method is based on confluence properties. It is also not clear whether his methods can also be adapted to non-deterministic call-by-need calculi with letrec.

The work on letrec-calculi in [1] proves an equivalence of call-by-name and call-by-need, however, for a non-maximal equivalence, i.e. one that distinguishes more expressions than contextual equivalence, a corresponding example can be found in [9]: there are two contextual equivalent lambda terms, $\lambda x.(x\ x) \sim_c \lambda x.(\lambda y.x\ y)$, which have different Böhm-trees, and also different Levy-Longo-trees. However, these terms are contextually equivalent in our calculus.

1.2 Structure and Result of this Paper

This paper extends the result in [22], where a tiny letrec-calculus LRλ is treated, to the calculus LRCCλ that also permits case, constructors and seq, and that is equipped with a normal order reduction and a contextual semantics as definition of equality of expressions. It reworks the report [22] showing that there is no problem in extending it to a language with case and constructors and `seq`. First it defines the infinite trees corresponding to the unrolling of expressions as in the 111-calculus of [11]. Then reduction on the infinite trees is defined, where the basic rules are beta-rule (betaTr), the case-rule (caseTr), and the seq-rule (seqTr), and the other rule $\xrightarrow{\infty}$ is a generalization of the (parallel) 1-reduction (see [7]); it can also be seen as an infinite development (see also [11]), however, the tree structure is a bit more general for LRCCλ. It is shown that convergence of expressions in the call-by-need lambda-calculus, as well as for the call-by-name calculus is equivalent to convergence of a normal-order variant of (Tr)-reduction, i.e. (betaTr), (caseTr) and (seqTr), on the corresponding infinite trees.

An essential step is the standardization lemma for $\xrightarrow{\infty,*}$-reductions, shown in the appendix. Finally, as a corollary we obtain the correctness of a general copy-rule in LRCCλ (see Theorem 4.10). The equivalence of the call-by-need letrec-calculus LRCCλ, and its call-by-name variant is proved in Theorem 5.8.
It is also shown as a spin-off that (cp) and (lll) are correct (see Theorems 4.10, 4.11); the proof of correctness of further rules like (lbeta), (case), (seq) is omitted, but can be done by copying the proofs in [23].
Our results imply that our calculus LRCCλ together with its contextual equivalence is equivalent to the theory of the extended lambda-calculi with case and constructors, but without letrec, where also call-by-name may be used.
As a summary, we have demonstrated that going via a calculus on infinite trees is a successful and extendible method to resolve questions concerning correctness of copy-related transformations in call-by-need letrec-calculi. We are confident that our purely operational method can be adapted to extensions of the calculi by non-deterministic operators to prove correctness of copy-related transformations, for which currently there is no other proof method.

2 Syntax and Reductions of the Functional Core Language LRCCλ

2.1 The Language and the Reduction Rules

We define the calculus LRCCλ consisting of a language $\mathcal{L}$(LRCCλ) and its reduction rules, presented in this section, the normal order reduction strategy and contextual equivalence. There is a set K of constructors that have a built-in arity. The syntax for expressions E is as follows:

$$\begin{array}{l} E ::= V \ \mid (E_1\ E_2) \mid (\lambda\, V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \\ \quad (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\texttt{case}\ E\ \texttt{of}(c\ V_1 \ldots V_{\mathrm{ar}(c)}) \to E) \ldots \\ \quad (\texttt{seq}\ E\ E) \end{array}$$

where E, E_i are expressions and V, V_i are variables, and c means a constructor. The expressions $(E_1\ E_2)$, $(\lambda V.E)$, $(\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$, $(c\ s_1 \ldots, s_{\mathrm{ar}(c)})$, $(\texttt{case} \ldots)$ and $(\texttt{seq}\ s\ t)$ are called *application*, *abstraction*, `letrec`-*expression*, constructor application, case-expression and seq-expression, respectively.
All `letrec`-expressions obey the following conditions: The variables V_i in the bindings are all distinct. We also assume that the bindings in `letrec` are commutative, i.e. `letrec`s with bindings interchanged are considered to be syntactically equivalent. The bindings by a `letrec` may be recursive: I.e., the scope of x_j in $(\texttt{letrec}\ x_1 = E_1, \ldots, x_j = E_j, \ldots, x_n = t_n\ \texttt{in}\ E)$ is E and all expressions E_i for $i = 1, \ldots, n$. We also assume that the variables in a case-pattern are disjoint and that their scope is within the continuation expression. This fixes the notions of closed, open expressions and α-renamings. Free and bound variables in expressions are defined using the usual conventions. Variable binding primitives are λ and `letrec`. The set of free variables in an expression t is denoted as $FV(t)$. For

simplicity we use the distinct variable convention: I.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly rename bound variables in the result by α-renaming if necessary to obey this convention. Note that this is only necessary for the copy and the case-rules (cp) and (case) (see below). We omit parentheses in nested applications: $(s_1 \dots s_n)$ denotes $(\dots(s_1\ s_2)\dots s_n)$. The set of closed LRCC$\lambda$-expressions is denoted as LRCCλ^0.
Sometimes we abbreviate the notation of `letrec`-expression (`letrec` $x_1 = E_1, \dots, x_n = E_n$ `in` E), as (`letrec` *Env* `in` E), where $Env \equiv \{x_1 = E_1, \dots, x_n = E_n\}$. This will also be used freely for parts of the bindings. The set of variables bound in an environment *Env* is denoted as $LV(Env)$.
In the following we define different context classes and contexts. To visually distinguish context classes from individual contexts, we use different text styles. The class $\mathcal{C}$ of all *contexts* is the set of all expressions C from LRCCλ, where the symbol $[\cdot]$, the *hole*, is a predefined context that is syntactically treated as an atomic expression, such that $[\cdot]$ occurs exactly once in C. Given a term t and a context C, we will write $C[t]$ for the expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 2.1. *A* value *is an abstraction or a constructor-application. We denote values by the letters v, w. A* weak head normal form (WHNF) *is either a value, or an expression* (`letrec` *Env* `in` v)*, where v is a value.*

The reduction rules in figure 1 are defined more liberally than necessary for the normal order reduction, in order to permit an easy use as transformations.

Definition 2.2 (Reduction Rules of the Calculus LRCCλ**).** *The (base) reduction rules for the calculus and language* LRCCλ *are defined in figure 1. The union of (llet-in) and (llet-e) is called (llet), the union of (cp-in) and (cp-e) is called (cp), and the union of (llet), (lapp), (lseq), and (lcase) is called (lll). Reductions (and transformations) are denoted using an arrow with super and/or subscripts: e.g. $\xrightarrow{llet}$. To explicitly state the context in which a particular reduction is executed we annotate the reduction arrow with the context in which the reduction takes place. If no confusion arises, we omit the context at the arrow. The* redex *of a reduction is the term as given on the left side of a reduction rule. Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\rightarrow$.*

A cv-expression is a constructor-application of the form $(c\ x_1 \dots x_n)$, where all x_i are variables.

2.2 The Unwind Algorithm

The following labeling algorithm (unwind) will detect the position to which a reduction rule will be applied according to normal order. It uses three labels:

(lbeta)	$((\lambda x.s)\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s])$ where s is an abstraction or a variable or a cv-expression
(cp-e)	$(\texttt{letrec}\ x = s, Env, y = C[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r)$ where s is an abstraction or a variable
(case)	$(\texttt{case}\ (c\ s_1 \dots s_n)\ \texttt{of} \dots (c\ x_1 \dots x_n) \to t \dots)$ $\to\ (\texttt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \texttt{in}\ t)$
(abs)	$(\texttt{letrec}\ x = (c\ s_1 \dots s_n), Env\ \texttt{in}\ t) \ \to$ $(\texttt{letrec}\ x = (c\ x_1 \dots x_n), x_1 = s_1, \dots, x_n = s_n, Env\ \texttt{in}\ t)$ if $(c\ s_1 \dots s_n)$ is not a cv-expression
(seq)	$(\texttt{seq}\ s\ t) \ \to\ t$ if s is a value
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r))$ $\to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)\ \texttt{in}\ r)$ $\to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \ \to\ (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \ \to\ (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$
(lcase)	$(\texttt{case}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ \texttt{of}\ \text{alts}) \ \to\ (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}\ s\ \texttt{of}\ \text{alts}))$

Fig. 1. Reduction Rules for Call-By-Need

S, T, V, where T means reduction of the top term, S means reduction of a subterm, and V labels already visited subexpressions, and $S \vee T$ matches T as well as S. The algorithm does not look into S-labeled letrec-expressions. We also denote the fresh V only in the result of the unwind-steps, and do not indicate the already existing V-labels. For a term s the labeling algorithm starts with s^T, where no subexpression in s is labeled. The rules of the labeling algorithm are:

$(\texttt{letrec}\ Env\ \texttt{in}\ t)^T$	$\to (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V$
$(s\ t)^{S \vee T}$	$\to (s^S\ t)^V$
$(\texttt{seq}\ s\ t)^{S \vee T}$	$\to (\texttt{seq}\ s^S\ t)^V$
$(\texttt{case}\ s\ \texttt{of}\text{alts})^{S \vee T}$	$\to (\texttt{case}\ s^S\ \texttt{of}\text{alts})^V$
$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S])$	$\to (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V])$ if s was not labeled
$(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ t)$	$\to (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ t)$ if s was not labeled and if $C[x] \neq x$

We assume that the term that gets a new label was not labelled before. Then this algorithm terminates. For example for $(\texttt{letrec}\ x = x\ \texttt{in}\ x)^T$ it will stop with $(\texttt{letrec}\ x = x^S\ \texttt{in}\ x^V)^V$.

Definition 2.3 (Normal Order Reduction). *A normal order reduction is defined as the reduction at the position of the final label S, or one position higher up, or copying the term from the final position to the position before, as indicated in figure 2. A normal-order reduction step is denoted as $\xrightarrow{n}$. Note that normal order reduction is unique.*

(lbeta)	$C[((\lambda x.s)^S\ r)] \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$
(cp-in)	$(\texttt{letrec}\ x = s^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s])$ where s is an abstraction or a variable or a cv-expression
(cp-e)	$(\texttt{letrec}\ x = s^S, Env, y = C[x^V]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r)$ where s is an abstraction or a variable or a cv-expression
(case)	$C[(\texttt{case}\ (c\ s_1 \dots s_n)^S\ \texttt{of} \dots (c\ x_1 \dots x_n) \to t) \dots] \to C[(\texttt{letrec}\ x_1 = s_1, \dots, x_n = s_n\ \texttt{in}\ t)]$
(abs)	$(\texttt{letrec}\ x = (c\ s_1 \dots s_n)^S, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ x = (c\ x_1 \dots x_n), x_1 = s_1, \dots, x_n = s_n, Env\ \texttt{in}\ t)$
(seq)	$C[(\texttt{seq}\ s^S\ t)] \to C[t]$ if s is a value
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^S) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r) \to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))]$
(lseq)	$C[(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))]$
(lcase)	$C[(\texttt{case}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ \texttt{of}\ \text{alts})] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}\ s\ \texttt{of}\ \text{alts}))]$

Fig. 2. Normal-Order Reduction Rules

Definition 2.4. *A normal order reduction sequence is called an* (normal-order) evaluation *if the last term is a WHNF. Otherwise, i.e. if the normal order reduction sequence is non-terminating, or if the last term is not a WHNF, but has no further normal order reduction, then we say that it is a* failing *normal order reduction sequence.*
For a term t, we write $t{\Downarrow}$ iff there is an evaluation starting from t. We call this the evaluation of t. *If $t{\Downarrow}$, we also say that t is* terminating. *Otherwise, if there is no evaluation of t, we write $t{\Uparrow}$.*

Definition 2.5 (contextual preorder and equivalence). *Let s, t be terms. Then:*

$$s \leq_c t \text{ iff } \forall C[\cdot]: \ C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}$$
$$s \sim_c t \text{ iff } s \leq_c t \wedge t \leq_c s$$

3 Reductions on Trees

In the following we use "expression" for finite expressions including `letrec`, and "tree" for the finite or infinite trees, which can co-inductively be defined like LRCCλ-expressions, but do not contain `letrec`-expressions.
The infinite tree corresponding to an expression is intended to be the letrec-unfolding of the expression with the extra condition that cyclic variable chains lead to local nontermination, represented by the symbol $\bot$. This corresponds to the infinite trees in the 111-variant of the calculus in [11]. A rigorous definition is as follows, where we use the explicit binary application operator @, since it is easier to explain, but stick to the common notation in examples.

$$
\begin{array}{lll}
C[(@\ s\ t)|_\varepsilon] & \mapsto @ & \\
C[(\texttt{case}\ \ldots)|_\varepsilon] & \mapsto \texttt{case} & \\
C[(\texttt{seq}\ s\ t)|_\varepsilon] & \mapsto \texttt{seq} & \\
C[(c\ s_1 \ldots s_n)|_\varepsilon] & \mapsto c & \\
C[(\lambda x.s)|_\varepsilon] & \mapsto \lambda x & \\
C[x|_\varepsilon] & \mapsto x & \text{if } x \text{ is a free or a lambda-bound variable}
\end{array}
$$

If the position ε hits the same (let-bound) variable twice, using the rules below, then the result is $\bot$. The general case:

$$
\begin{array}{ll}
C[(\lambda x.s)|_{1.p}] & \to C[\lambda x.(s|_p)] \\
C[(@\ s\ t)|_{1.p}] & \to C[(@\ s|_p\ t)] \\
C[(@\ s\ t)|_{2.p}] & \to C[(@\ s\ t|_p)] \\
C[(\texttt{seq}\ s\ t)|_{1.p}] & \to C[(\texttt{seq}\ s|_p\ t)] \\
C[(\texttt{seq}\ s\ t)|_{2.p}] & \to C[(\texttt{seq}\ s\ t|_p)] \\
C[(\texttt{case}\ s\ \texttt{of}\ alt_1 \ldots alt_n)|_{1.p}] & \to C[(\texttt{case}\ s|_p\ \texttt{of}\ alt_1 \ldots alt_n)] \\
C[(\texttt{case}\ s\ \texttt{of}\ alt_1 \ldots alt_n)|_{i.p}] & \to C[(\texttt{case}\ s\ \texttt{of} alt_1 \ldots alt_i|_p \ldots alt_n)] \\
C[\ldots(c\ldots) \to s)|_{1.p}\ldots] & \to C[\ldots(c\ldots) \to s|_{1.p})\ldots] \\
C[(c\ s_1 \ldots s_n))|_{i.p}] & \to C[(c\ s_1 \ldots s_i|_p \ldots s_n)] \\
C[(\texttt{letrec}\ Env\ \texttt{in}\ r)|_p] & \to C[(\texttt{letrec}\ Env\ \texttt{in}\ r|_p)] \\
C_1[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x|_p])] & \to C_1[(\texttt{letrec}\ x = s|_p, Env\ \texttt{in}\ C_2[x])] \\
C_1[(\texttt{letrec}\ x = s, y = C_2[x|_p], Env\ \texttt{in}\ r)] & \\
\qquad \to C_1[(\texttt{letrec}\ x = s|_p, y = C_2[x], Env\ \texttt{in}\ r)] &
\end{array}
$$

Fig. 3. Infinite tree construction from positions

Definition 3.1. *Given an expression t, the infinite tree $IT(t)$ of t is defined by giving an algorithm to compute for every position p the label of the infinite tree at position p. The computed label for the position ε is defined in figure 3.*
In all cases not mentioned in figure 3, the result is undefined (and also not necessary). The equivalence of trees is syntactic, where α-equal trees are assumed to be equivalent. A tree of the form $\lambda x.s$ or $(c\ s_1 \ldots s_n)$ is called a value.

Example 3.2. The expression $\texttt{letrec}\ x = x, y = (\lambda z.z)\ x\ y\ \texttt{in}\ y$ has the corresponding tree $((\lambda z.z) \bot ((\lambda z.z) \bot ((\lambda z.z) \bot \ldots)))$.

Definition 3.3. *Reduction contexts $\mathcal{R}$ for (infinite) trees are defined by $\mathcal{R} ::= [\cdot] \mid (@\ \mathcal{R}\ E) \mid (\texttt{case}\ \mathcal{R}\ alts) \mid (\texttt{seq}\ \mathcal{R}\ E)$, where E means a tree.*

Lemma 3.4. *Let s, t be expressions and C be a context. Then $IT(s) = IT(t) \Rightarrow IT(C[s]) = IT(C[t])$.*

Lemma 3.5. *Let s, t be expressions and $s \to t$ by a rule (cp) or (lll). Then $IT(s) = IT(t)$.*

Definition 3.6. *The reduction rules on trees are allowed in any tree context and are as follows:*

(betaTr) $((\lambda x.s)\ r) \to s[r/x]$
(seqTr) $(\texttt{seq}\ s\ t) \to t$ *if s is a value*
(caseTr) $(\texttt{case}\ (c\ s_1 \ldots s_n)\ \texttt{of}\ \ldots (c\ x_1 \ldots x_n) \to s \to s[s_1/x_1 m \ldots, s_n/x_n]$

If a tree-reduction rule is applied within an $\mathcal{R}$-context, we call it an $\mathcal{R}$-reduction on trees. A sequence of $\mathcal{R}$-reductions of T that terminates with a value tree is called evaluation. *If T has an evaluation, then we also say T* converges *and denote this as $T{\Downarrow}$.*

Note that (betaTr) as a reduction may modify infinitely many positions, since there may be infinitely many positions of the variable x. E.g. a top-level (betaTr) of $IT((\lambda x.(\texttt{letrec}\ z = (z\ x)\ \texttt{in}\ z))\ r) = (\lambda x.((\ldots\ (\ldots\ x)\ x)\ x))\ r$ modifies the infinite number of occurrences of x. Further note that (betaTr) does not overlap with itself, where we ignore overlaps within the meta-variables s, r.

Lemma 3.7. *Let s be an expression and let $IT(s)$ be a value tree. Then $s{\Downarrow}$.*

We will use a variant of infinite outside-in developments [7,11] as a reduction on trees that may reduce infinitely many redexes in one step. For a more detailed definition, in particular concerning the labeling, see [22].

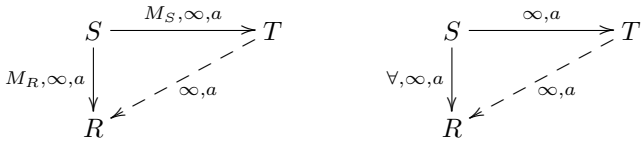
Definition 3.8. *For trees S, T, we define the reduction $S \xrightarrow{\infty,a} T$ as follows. We mark a possibly infinite subset of all a-redexes in S for one reduction type $a \in \{$(betaTr), (caseTr), (seqTr)$\}$, say with a $\dagger$. The reduction constructs a new infinite tree top-down by iteratedly using labelled reduction, where the label of the redex is removed before the reduction. If the reduction does not terminate for a subtree at the top level of the subtree, then this subtree is the constant $\bot$ in the result. This recursively defines the result tree top-down.*
Sometimes we omit the reduction rule type a, if it is not important, and write only $\xrightarrow{\infty}$. We write $T{\Downarrow}(\infty)$ if $T \xrightarrow{\infty,*} T'$, where T' is a value tree.
The reduction $S \xrightarrow{\forall,\infty,a} T$ is defined as the specific $S \xrightarrow{\infty} T$-reduction, if all a-redexes in S are labeled.

Note that even for a tree with only two marked redexes, it is possible that after the first reduction, infinitely many redexes are labeled.

Example 3.9. We give two examples for a $\xrightarrow{\infty}$-reduction:

- $t = (\lambda z.\texttt{letrec}\ y = \lambda u.u, x = (z\ (y\ y)\ x)\ \texttt{in}\ x)$. The infinite tree $IT(t)$ is like an infinite list, descending to the right, with elements $((\lambda u.u)\ \lambda u.u)$. The ∞-reduction may label any subset of these redexes, even infinitely many, and then reduce them by (betaTr).
- $t = (\texttt{letrec}\ x = \lambda y.x\ (\lambda u.u)\ \texttt{in}\ x)$ has the infinite tree $(\lambda y.(\lambda y.(\lambda y.\ldots)\ (\lambda u.u)\ (\lambda u.u))\ (\lambda u.u))$ which, depending on the labeling, may reduce to itself, or, if all redexes are labeled, it will reduce to $\bot$, i.e., $t \xrightarrow{\forall,\infty} \bot$.

Lemma 3.10. *For all trees S, R, T and reduction types a: if $S \xrightarrow{\infty,a} R$ where the set of a-redex positions is M_R, and $S \xrightarrow{\infty,a} T$, where the the set of a-redex positions is M_S, and $M_S \subseteq M_R$, then also $T \xrightarrow{\infty,a} R$. A special case is that $S \xrightarrow{\forall,\infty,a} R$, and $S \xrightarrow{\infty,a} T$ imply that $T \xrightarrow{\infty,a} R$.*



Proof. The argument is that we can mark the a-redexes in S that are not reduced in $S \xrightarrow{M_S,\infty,a} T$. Reduce all M_R-labeled redexes in the reduction $T \xrightarrow{\infty,a} R$.

In the appendix it is shown:

Theorem 3.11 (Standardization for tree-reduction). *Let S be a tree. Then $S{\Downarrow}(\infty)$ implies $S{\Downarrow}$.*

4 Properties of Call-by-Need Convergence

4.1 Call-by-Need Convergence Implies Infinite Tree Convergence

Lemma 4.1. *If $s \xrightarrow{a} t$ for two expressions s, t and $a \in \{(lbeta), (seq), (case)\}$ then $IT(s) \xrightarrow{\infty,a'} IT(t)$ for the tree reduction type a' corresponding to a.*

Proof. We label every redex of $IT(t)$ that is derived from the redex of $s \xrightarrow{a} t$. As a specific case, let $a = (lbeta)$. If the redex is $((\lambda x.s')\ r')$ and s' is not a variable, then the lemma is obvious. The only nontrivial case is that the subexpression is e.g. of the form ($\texttt{letrec}\ Env, y_2 = y_1, y_1 = ((\lambda x.y_2)\ r')\ \texttt{in}\ s'$), and after the (lbeta)-reduction, and perhaps some (lll)-reductions, y_2 is in a cyclic chain of variables like ($\texttt{letrec}\ Env, y_2 = y_1, y_1 = y_2, x = r'\ \texttt{in}\ s'$). In this case the tree-reduction of the redex corresponding to y_1 does not terminate during computing the development, and hence the result will be $\bot$.
The other cases for a are similar.

Proposition 4.2. *Let t be an expression. Then $t{\Downarrow} \Rightarrow IT(t){\Downarrow}$.*

Proof. That $IT(t){\Downarrow}(\infty)$ holds follows from Lemma 4.1 by induction on the length of evaluation of t, from Lemma 3.5 and from the fact that a WHNF has a value tree as corresponding infinite tree. Then Theorem 3.11 shows that $T{\Downarrow}(\infty)$ implies also $T{\Downarrow}$.

4.2 Infinite Tree Convergence Implies Call-by-Need Convergence

Now we show the harder part of the desired equivalence in a series of lemmas.

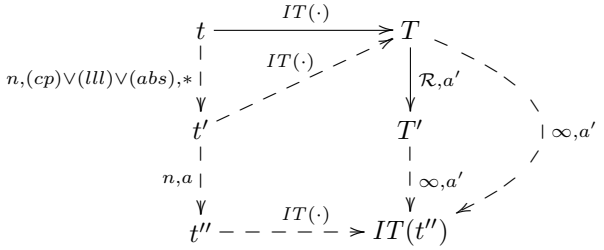
Lemma 4.3. *For every reduction possibility $S_1 \xleftarrow{\mathcal{R}} T \xrightarrow{\infty} S_2$, either $S_1 \xrightarrow{\infty} S_2$ or there is some T' with $S_1 \xrightarrow{\infty} T' \xleftarrow{\mathcal{R}} S_2$. I.e. we have the following forking diagrams for trees between an $\mathcal{R}$-reduction and an $\xrightarrow{\infty}$-reduction:*

Proof. This follows by checking the overlaps of $\xrightarrow{\infty}$ with $\mathcal{R}$-reductions. Note that if the type of the $\xrightarrow{\infty}$ and $\xrightarrow{\mathcal{R}}$ reductions are different, then the first diagram applies.

Lemma 4.4. *Let T be a tree such that there is an $\mathcal{R}$-evaluation of length n, and let S be a tree with $T \xrightarrow{\infty} S$. Then S has an $\mathcal{R}$-evaluation of length $\leq n$.*

Proof. Follows from Lemma 4.3 by induction.

Lemma 4.5. *Let t be a term and let $T := IT(t) \xrightarrow{a'} T'$ be an $\mathcal{R}$-reduction with $a' \in \{(betaTr), (seqTr), (caseTr)\}$ Then there is an expression t', a reduction $t \xrightarrow{n,*} t'$ using (lll), (cp) and (abs)-reductions, an expression t'' with $t' \xrightarrow{n,a} t''$, where a is the expression reduction corresponding to a, such that there is a reduction $T' \xrightarrow{\infty,a'} IT(t'')$.*



Proof. The expressions t', t'' are constructed as follows: t' is the resulting term from a maximal normal-order reduction of t consisting only of (cp), (lll) and (abs)-reductions. It is clear that such a sequence of $\xrightarrow{(cp)\vee(lll)\vee(abs),n}$-reductions is terminating. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order (a)-redex in t' must correspond to $T \xrightarrow{\mathcal{R},a'} T'$ and is used for the reduction $t' \xrightarrow{n,a} t''$. Note that the (a)-redex in t' may correspond to infinitely many redexes in T. Lemma 4.1 shows that there is a reduction $T \xrightarrow{\infty,a'} IT(t'')$, and Lemma 3.10 shows that also $T' \xrightarrow{\infty,a'} IT(t'')$.

Proposition 4.6. *Let t be an expression such that $IT(t)\Downarrow$. Then $t\Downarrow$.*

Proof. The precondition $IT(t)\Downarrow$ and the Standardization Theorem 3.11 imply that there is an $\mathcal{R}$-evaluation of T to a value tree. The base case, where no $\mathcal{R}$-reductions are necessary is treated in Lemma 3.7. In the general case, let $T \xrightarrow{a'} T'$ be the unique first $\mathcal{R}$-reduction of a single redex. Lemma 4.5 shows that there are expressions t', t'' with $t \xrightarrow{n,(cp)\vee(lll)\vee(abs),*} t' \xrightarrow{n,lbeta} t''$, and $T' \xrightarrow{\infty} IT(t'')$. Lemma 4.4 shows that the number of $\mathcal{R}$-reductions of $IT(t'')$ to a value tree is strictly smaller than the number of $\mathcal{R}$-reductions of T to a value. Hence we can use induction on this length and obtain a normal-order reduction of t to a WHNF.

Convergence is equivalent for a term and its corresponding infinite tree:

Theorem 4.7. *Let t be an expression. Then $t\Downarrow$ iff $IT(t)\Downarrow$.*

Proof. This follows from Propositions 4.2 and 4.6.

Definition 4.8. *Let the generalized copy rule be:*
(gcp) $C_1[\texttt{letrec}\ x = r \ldots C_2[x] \ldots] \to C_1[\texttt{letrec}\ x = r \ldots C_2[r] \ldots]$

This is just like the rule (cp), but all kinds of terms r can be copied, not only abstractions. Obviously the following holds:

Lemma 4.9. *If $s \xrightarrow{gcp} t$, then $IT(s) = IT(t)$*

Theorem 4.10. *Let s, t be expressions with $s \xrightarrow{gcp} t$ Then $s \sim_c t$.*

Proof. Lemma 3.4 shows that it is sufficient to show equivalence of termination of s, t. Lemma 4.9 implies $IT(s) = IT(t)$. Hence equivalence of termination follows from Theorem 4.7.

Theorem 4.11. *Let s, t be expressions with $s \xrightarrow{lll} t$ or $s \xrightarrow{abs} t$ Then $s \sim_c t$.*

Proof. Follows in the same way as in the proof of Theorem 4.10 using Lemma 3.5.

5 Relation Between Call-By-Name and Call-By-Need

For the same language we now treat a call-by-name variant of the reduction strategy using beta-reduction instead of the rule (lbeta) that respects sharing, and also a substituting case as well as a different (cp) and omitting the (abs)-rule.

Definition 5.1. *The* call-by-name *normal-order reduction is defined by using the (lll)-rules and the (seq)-rule and the following modified rules in the call-by-need normal-order reduction as follows:*

$$
\begin{array}{ll}
\textit{(beta)} & ((\lambda x.s)^S\ r) \to s[r/x] \\
\textit{(cpn-in)} & (\texttt{letrec}\ x = s^S, Env\ \ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s]) \\
 & \textit{where } s \textit{ is an abstraction or a variable} \\
 & \textit{or a constructor-application} \\
\textit{(cpn-e)} & (\texttt{letrec}\ x = s^S, Env, y = C[x^V]\ \texttt{in}\ r) \\
 & \quad \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r) \\
 & \textit{where } s \textit{ is an abstraction or a variable} \\
 & \textit{or a constructor-application} \\
\textit{(casen)} & (\texttt{case}\ (c\ s_1 \ldots s_n)^S\ \texttt{of} \ldots (c\ x_1 \ldots x_n) \to t) \ldots \\
 & \quad \to \quad s[s_1/x_1, \ldots, s_n/x_n]
\end{array}
$$

where the same labelling and redex is used. Let (cpn) be the union of (cpn-e) and (cpn-in). We denote the reduction as $\xrightarrow{name}$*, the corresponding call-by-name convergence of a term* t *as* $t{\Downarrow}(name)$*, and the corresponding contextual preorder and equivalence as* $\leq_{c,name}$ *and* $\sim_{c,name}$*, respectively.*

Note that (abs) does not occur in $\xrightarrow{name}$-reductions;
We give an example showing that the call-by-name evaluation and the call-by-need evaluation may have essentially different infinite tree evaluations.

Example 5.2. We start with the term $(\texttt{letrec}\ z = (\lambda x.(\lambda y.x))\ (z\ z)\ \texttt{in}\ z\ z)$. The call-by-need normal order reduction is as follows:

$$
\begin{array}{ll}
\xrightarrow{lbeta} & (\texttt{letrec}\ z = (\texttt{letrec}\ x = z\ z\ \texttt{in}\ \lambda y.x)\ \texttt{in}\ z\ z) \\
\xrightarrow{lll} & (\texttt{letrec}\ z = \lambda y.x, x = z\ z\ \texttt{in}\ z\ z) \\
\xrightarrow{cp} & (\texttt{letrec}\ z = \lambda y.x, x = z\ z\ \texttt{in}\ (\lambda y.x)\ z) \\
\xrightarrow{lbeta} & (\texttt{letrec}\ z = \lambda y.x, x = z\ z\ \texttt{in}\ (\texttt{letrec}\ y = z\ \texttt{in}\ x)) \\
\xrightarrow{lll} & (\texttt{letrec}\ z = \lambda y.x, x = z\ z, y = z\ \texttt{in}\ x) \\
\xrightarrow{cp} & (\texttt{letrec}\ z = \lambda y.x, x = (\lambda y'.x)\ z, y = z\ \texttt{in}\ x) \\
\xrightarrow{lbeta} & (\texttt{letrec}\ z = \lambda y.x, x = (\texttt{letrec}\ y' = z\ \texttt{in}\ x), y = z\ \texttt{in}\ x) \\
\xrightarrow{lll} & (\texttt{letrec}\ z = \lambda y.x, x = x, y' = z, y = z\ \texttt{in}\ x)
\end{array}
$$

Thus it fails. The call-by-name normal order reduction loops, where the first reduction gives $(\texttt{letrec}\ z = (\lambda y.(z\ z))\ \texttt{in}\ z\ z)$, which immediately starts a loop using (beta) and (cp)-reductions.
Thus the call-by-name and call-by-need reductions have a different trace of infinite trees, hence an easy correspondence proof of the reductions is not possible. Witnesses are the expressions $s_1 = (\texttt{letrec}\ z = (\lambda y.(z\ z))\ \texttt{in}\ z\ z)$ and $s_2 = (\texttt{letrec}\ z = \lambda y.x, x = (\lambda y'.x)\ z, y = z\ \texttt{in}\ x)$ that have the same infinite tree, and the call-by-name reduction of s_1 gives an expression with the same infinite tree, whereas the call-by-need reduction of s_2 results in the tree $\bot$.

5.1 Call-by-Name Convergence Implies Infinite Tree Convergence

Lemma 5.3. *Let $a \in \{(beta), (casen)\}$, and $a' \in \{betaTr), (caseTr)\}$ be the corresponding tree-reduction. If $s \xrightarrow{a} t$ for two expressions s, t, then $IT(s) \xrightarrow{\infty,a'} IT(t)$.*

Proof. This is easy by computing the positions in the infinite tree.

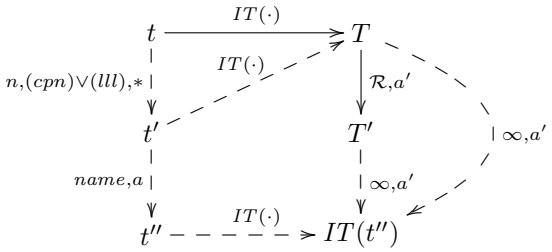
Proposition 5.4. *Let t be an expression. Then $t{\Downarrow}(name) \Rightarrow IT(t){\Downarrow}$.*

Proof. This follows from Lemma 5.3 by induction on the length of the call-by-name evaluation of t, from Lemma 3.5 using the standardization theorem 3.11 and from the fact that a WHNF has a value tree as corresponding infinite tree.

5.2 Infinite Tree Convergence Implies Call-by-Name Convergence

Now we show the desired implication also for call-by-name.

Lemma 5.5. *Let t be a term, $a \in \{(beta), (casen), (seq)\}$, and let $a' \in \{(betaTr), (caseTr), (seqTr)\}$ be the corresponding tree reduction. Let $T := IT(t) \xrightarrow{(a')} T'$ be an $\mathcal{R}$-reduction. Then there is an expression t', a reduction $t \xrightarrow{n,*} t'$ using (lll) and (cpn)-reductions, an expression t'' with $t' \xrightarrow{name,a} t''$, such that there is a reduction $T' \xrightarrow{\infty,a'} IT(t'')$.*



Proof. The expressions t', t'' are constructed as follows: t' is the resulting term from a maximal normal-order reduction consisting only of (cpn) and (lll)-reductions. It is clear that such a sequence of $\xrightarrow{(cpn)\vee(lll),n}$-reductions is terminating. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order (a)-redex in t' corresponding to $T \xrightarrow{a'} T'$ is used for the reduction $t' \xrightarrow{name,a} t''$. Note that the normal-order a-redex in t' may correspond to infinitely many a'-redexes in T. Lemma 5.3 shows that there is a reduction $T \xrightarrow{\infty,a'} IT(t'')$, and Lemma 3.10 shows that also $T' \xrightarrow{\infty,a'} IT(t'')$.

Proposition 5.6. *Let t be an expression such that $IT(t){\Downarrow}$. Then $t{\Downarrow}(name)$.*

Proof. The precondition $IT(t)\Downarrow$ means that there is an $\mathcal{R}$-evaluation of $T := IT(t)$ to a value tree. The base case, where no $\mathcal{R}$-reductions are necessary is treated in Lemma 3.7. In the general case, let $T \xrightarrow{a'} T'$ with $a' \in \{(\text{betaTr}), (\text{caseTr}), (\text{seqTr})\}$ be the unique first $\mathcal{R}$-reduction of a single redex. Lemma 5.5 shows that there are expressions t', t'' with $t \xrightarrow{n,(cpn)\vee(lll),*} t' \xrightarrow{name,a} t''$, and $T' \xrightarrow{\infty,a'} IT(t'')$. Lemma 4.4 shows that the number of $\mathcal{R}$-reductions of $IT(t'')$ to a value tree is strictly smaller than the number of $\mathcal{R}$-reductions of T to a value. Hence we can use induction on this length and obtain a call-by-name normal-order reduction of t to a WHNF.

Now we can show that call-by-name convergence for a term is equivalent to convergence of its corresponding infinite tree.

Theorem 5.7. *Let t be an expression. Then $t\Downarrow(name)$ iff $IT(t)\Downarrow$.*

Proof. Follows from Propositions 5.4 and 5.6.

The strategies call-by-need and call-by-name are equivalent:

Theorem 5.8. *The contextual preorders for call-by-need and call-by-name are equivalent.*

Proof. This follows from Theorems 4.7 and 5.7.

6 Conclusion

We demonstrated the proof method via infinite trees by showing correctness of unrestricted copy-reductions and the equivalence of call-by-name and call-by-need for a deterministic letrec-calculus LRCCλ with case and constructors. We are sure that the method also applies to the letrec-calculi from [5,1,3,4,14], if the contextual equivalence as equality is adopted, which appears to cover all the desired equalities in these calculi. It could also be applied to the record calculus in [13] after specializing meaning-preservation to contextual equivalence. This shows that the proof method using infinite trees that we have developed and also successfully applied has a great potential in exhibiting correctness of variants of copy-transformations in different kinds of calculi with cyclic sharing mechanisms. For non-deterministic calculi like [15,17,21] we plan to extend the method to show correctness of the copy-reduction for deterministic subexpressions, which appears to be a hard obstacle for other methods.

Acknowledgements

I thank David Sabel for reading and correcting drafts of this paper. I also acknowledge discussions with Elena Machkasova.

References

1. Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *TACS*, pages 77–106, 1997. Sendai, Japan.
2. Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117:95–168, 2002.
3. Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997.
4. Z.M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. functional programming*, 7(3):265–301, 1997.
5. Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Principles of Programming Languages*, pages 233–246, San Francisco, California, 1995. ACM Press.
6. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
7. H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.
8. K. Claessen and D. Sands. Observable sharing for functional circuit description. In P.S. Thiagarajan and R. Yap, editors, *Advances in Computing Science ASIAN'99*, LNCS 1742, pages 62–73. Springer-Verlag, 1999.
9. M. Dezani-Ciancaglini, J. Tiuryn, and P. Urzyczyn. Discrimination by parallel observers: The algorithm. *Information and Computation*, 150(2):153–186, 1999.
10. A. Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proc. LICS*, pages 82–91, 1994.
11. R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theor. Comput. Sci*, 175(1):93–125, 1997.
12. J Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th Principles of Programming Languages*, 1993.
13. E. Machkasova and F. A. Turbak. A calculus for link-time compilation. In *ESOP'2000*, 2000.
14. J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Functional programming*, 8:275–317, 1998.
15. A. K.D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *LNCS*, pages 85–102. Springer-Verlag, 1999.
16. A. K.D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.
17. A.K.D. Moran. *Call-by-name, call-by-need, and McCarthys Amb.* PhD thesis, Dept. of Comp. Science, Chalmers University, Sweden, 1998.
18. S. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003. `www.haskell.org`.
19. R. Plasmeijer and M. van Eekelen. The concurrent Clean language report: Version 1.3 and 2.0. Technical report, Dept. of Computer Science, University of Nijmegen, 2003. `http://www.cs.kun.nl/~clean/`.
20. G. D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
21. D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank report 24, Inst. Informatik. J.W.G.-university Frankfurt, 2006.

22. M. Schmidt-Schauß. Equivalence of call-by-name and call-by-need for lambda-calculi with letrec. Frank report 25, Inst. Informatik. J.W.G.-university Frankfurt, 2006.
23. M. Schmidt-Schauß, M. Schütz, and D. Sabel. A complete proof of the safety of Nöcker's strictness analysis. Frank report 20, Inst. Informatik. J.W.G.-University Frankfurt, 2005.

A Labeled Reduction

We define two variants of the notion of *labeled reduction* for trees. Labelled reduction is used to identify correspondences of positions during a reduction step. It will be used in two variants, the joining variant for the inheritance of positions during reductions and the consuming one for a reduction that is similar to a development: Some redexes are marked at the start of the reduction process, and all the labeled redexes have to be reduced.

Definition A.1 (labeled reduction of trees). *First we define* joining labeled reduction *for sets of labels.*
Let S be a tree and assume there are sets of labels at certain (subexpression-) positions of S. We can assume that every position is labeled, perhaps with an empty set. Let T be a tree with $S \xrightarrow{(betaTr)} T$, and assume that the reduction is $S = C[(\lambda x.r)\ s] \to C[r[s/x]]$. Then the labels in the result are as follows:

- *label sets within C are unchanged.*
- *label sets properly within s are copied to all occurrences of s in the result.*
- *If $r = x$, then $((\lambda x.x^A)^B\ s^C)^D \to r[s/x]^{A \cup B \cup C \cup D}$.*
- *If $r \neq x$, label sets of positions properly within r and not at x remain unchanged.*
- *If $r \neq x$, the label sets of the new occurrences of s^B in $C[r[s/x]]$ are as follows: For every occurrence of x^A in r the new label set of the new occurrence of s is $A \cup B$.*
- *If $r \neq x$, the new label set of $r[s/x]$ is computed as follows: $((\lambda x.r^A)^B\ s)^C \to r[s/x]^{A \cup B \cup C}$.*

In an analogous way the inheritance for the case and seq-rule are defined; we make only the redex-case explicit:
$(\texttt{case}\ (c\ s_1\ \ldots\ s_n)^A\ \texttt{of}\ \ldots\ ((c\ x_1\ \ldots\ x_n) \to t^B)\ \ldots)^C \to t[s_1/x_1, \ldots, s_n/x_n]^{A \cup B \cup C}$, *where in the case that t is a variable x_i, the label of respective s_i is also joined.*
$(\texttt{seq}\ s^A\ t^B)^C \to t^{B \cup C}$ *if s is a value. Here the label of s is not inherited, since s is discarded after evaluation.*

The consuming labeled reduction *is like the joining variant, and can be derived by removing the label of the redex before the reduction and then using the joining variant.*
We will use the consuming labeled reduction below for one label $\dagger$ in the developments. In the case of only one label-value it is usually assumed that empty sets mean no label, and a non-empty set, which must be a singleton in this case, means that the position is labeled.

B Standardization of Tree Reduction

Definition B.1. *We call a set M of positions* prefix-closed, *iff for every $p \in M$, and prefix q of p, also $q \in M$. If M is a finite prefix-closed set of positions of the*

tree T, and for every $p \in M$, we have $T_{|_p} \neq \bot$, then we say M is admissible*, and call this set an* FAPC-set *(finite admissible prefixed-closed) of positions of T.*

In the following we use sets of positions in terms.

Lemma B.2. *Let S, T be trees with $S \xrightarrow{\infty} T$, and let M_T be an FAPC-set of positions of T. Then the set of positions M_S which are mapped by joining labeled reduction to positions in M_T is also an FAPC-set of positions of S.*

Proof. First we analyze the transport of positions by the reduction $S \xrightarrow{\infty} T$ using joining labeled reduction. For the reduction $S \xrightarrow{\infty} T$, there are some †-labeled positions in S, which are exactly the redexes that are to be reduced. We determine the new position(s) in T of every position from S. This has to be done by looking at the construction of the results of the reduction as explained in Definition 3.8. We will use joining labeled reduction to trace the positions. At the start of the construction, we assume that all S-positions are labeled with a singleton set, containing their position. The definition implies that after a single (betaTr), (seqTr) or (caseTr)-reduction, the set of labels at every position remains a finite set. If for a subtree A the top-reduction sequence does not terminate, then this subtree will be $\bot$ in the resulting tree, hence only finitely many reductions for A have to be considered. The construction will then proceed with the direct subtrees of A, which guarantees that every position in T either has finitely many ancestors in S, or is $\bot$. It is obvious that there are no positions in M_S pointing to $\bot$.
Now we can simply reverse the mapping. For the set M_T, we define the set M_S as the set of all positions of S that are in the label set of any position in M_T. This set is finite, since there are no positions of $\bot$ in M_T, there is also no position of $\bot$ in M_S. The set M_S is prefix-closed, since the mapping behaves monotone, i.e. if p is a position in S, q is a prefix of p, then for every position p' in T that is derived from p, there is a position q' in T derived from q such that q' is a prefix of p'.

Lemma B.3. *Let S be a tree, and let* RED *be the reduction sequence $S = S_0 \xrightarrow{\infty} S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$, where S' is a value. Then the set M_0 of all positions of S that are mapped by* RED *to the top position of S' is an FAPC-set of S.*

Proof. We perform induction on the number of $\xrightarrow{\infty}$-reductions in the sequence $S = S_0 \xrightarrow{\infty} S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$. If the sequence has no reductions, then the lemma holds, since $M = \{\varepsilon\}$ consists only of the top position of S'. In the induction step, we can assume that the lemma holds already for the reduction sequence $S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$, and then we can apply Lemma B.2 to the reduction step $S_0 \xrightarrow{\infty} S_1$, which shows the Lemma.

Corollary B.4. *Let S be a tree, and let* RED *be the reduction sequence $S = S_0 \xrightarrow{\infty} S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$, where S' is a value. If a position p from S is not mapped by* RED *to the top position of S', then all positions q of S such that p is a prefix of q, are also not mapped to the top position of S'.*

We distinguish relevant and irrelevant positions for a reduction sequence to a value:

Definition B.5. *Let* RED $\equiv S \xrightarrow{\infty,*} S'$ *be a reduction sequence, where* S' *is a value. Let* M_0 *be the set of positions of* S *that are mapped using joining labeled reduction to the top position of* S'*. Then the positions* $p \in M_0$ *in* S *are called* relevant *for* RED*, and the positions of* S *that are not in* M_0 *a are called* irrelevant *for* RED*. We omit* RED *in the notation if it is clear from the context,*

Note that the set of relevant positions for some reduction sequence RED $\equiv S \xrightarrow{\infty,*} S'$ is always an FAPC-set in S.
Let $\xrightarrow{(Tr),*}$ be the reduction $\xrightarrow{(\text{betaTr}),*} \cup \xrightarrow{(\text{seqTr}),*} \cup \xrightarrow{(\text{caseTr}),*}$.

Lemma B.6. *Let* RED $\equiv S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty,*} S'$ *be a reduction sequence to the value* S'*. Let* M_1 *be the set of all relevant positions in* S_1*. Then the reduction* $S_1 \xrightarrow{\infty} S_2$ *can be splitted into* $S_1 \xrightarrow{(Tr),*} S_1' \xrightarrow{\infty,M'} S_2$*, where* M' *is a set of irrelevant positions. Moreover, if the first reduction is* $S_1 \xrightarrow{a,\infty} S_2$*, then we can split as follows:* $S_1 \xrightarrow{a,*} S_1' \xrightarrow{a,\infty,M'} S_2$*.*

Proof. We split the reduction $S_1 \xrightarrow{\infty} S_2$ into $S_1 \xrightarrow{(Tr),*} S_{0,1} \xrightarrow{\infty} S_2$, such that $S_1 \xrightarrow{(Tr),*} S_{0,1}$ is the maximal prefix of the (Tr)-reduction sequence, which defines the ∞-reduction, consisting only of top level reductions, and the first reduction in the definition of $S_{0,1} \xrightarrow{\infty} S_2$ is not at top level.

$$\begin{array}{ccc} S_1 & \xrightarrow{\infty} & S_2 \\ & & \downarrow{\scriptstyle (Tr),*} \\ & & S' \end{array} \qquad\qquad S_1 \xrightarrow{(Tr),*} S_{0,1} \xrightarrow{(Tr),*} S_{1,1} \xrightarrow{\infty} \begin{array}{c} S_2 \\ \downarrow{\scriptstyle (Tr),*} \\ S' \end{array}$$

Let $M_{0,1}$ be the FAPC-set of all positions in $S_{0,1}$ that are mapped by $S_{0,1} \xrightarrow{\infty} S_2$ to M_2, the set of relevant positions in S_2. By induction on the depth of positions in $M_{0,1}$, and since we can split into reduction sequences at independent positions, it is easy to see that there is a reduction sequence $S_{0,1} \xrightarrow{(Tr),*} S_{1,1} \xrightarrow{\infty} S_1$, such that the reduction $S_{1,1} \xrightarrow{\infty} S_2$ is w.r.t. a set $M_{1,1}$ of irrelevant positions.

Lemma B.7. *Let* RED $= S_0 \xrightarrow{M_0,\infty} S_1$ *be a reduction of trees to a value* S_1*, such that* M_0 *is the set of irrelevant positions in* S_0*. Then* S_0 *is a value*

Proof. This is obvious.

Lemma B.8. *Let* RED $= S_0 \xrightarrow{M_0,\infty} S_1 \xrightarrow{a} S_2 \cdot$ RED$'$ *for* $a \in \{(betaTr), (seqTr), (caseTr)\}$ *be a reduction sequence of trees to a value, such that* M_0 *is the set of irrelevant positions in* S_0 *and let* $S_1 \xrightarrow{a} S_2$ *be a reduction at the relevant position* p_1*.*

Then there is some S_0', a set M_0' of positions of S_0' with $S_0 \xrightarrow{a} S_0' \xrightarrow{M_0',\infty} S_2$, such w.r.t the reduction sequence $\text{RED}'' \equiv S_0' \xrightarrow{M_0',\infty} S_2 \cdot \text{RED}'$*, the set of positions M_0' is irrelevant. Moreover, the reduction $S_0 \xrightarrow{a} S_0'$ is also at the relevant position p_1, and the constructed reduction sequence has the same length and reduces at the same positions.*

$$\begin{array}{ccc} S_0 & \xrightarrow{M_0,\infty} & S_1 \\ {\scriptstyle a}\,\vdots & & \Big\downarrow{\scriptstyle a} \\ S_0' & \overset{M_0',\infty}{- - \succ} & S_2 \\ & & \Big\downarrow{\scriptstyle \text{RED}'} \\ & & \cdot \end{array}$$

Proof. Let C be a multicontext that has holes at p_1, the position of the redex of the $S_1 \xrightarrow{a} S_2$-reduction, and additionally finitely many holes, such that all positions of M_0 are below a hole of C. Then the following diagrams shows the given and the derived reductions for every type of reduction:

$$\begin{array}{ccc} C[s_1,\ldots,s_n,((\lambda x.s)\ r)] & \xrightarrow{M_0,\infty} & C[s_1',\ldots,s_n',((\lambda x.s')\ r')] \\ {\scriptstyle (betaTr)}\,\vdots & & \Big\downarrow{\scriptstyle (betaTr)} \\ C[s_1,\ldots,s_n,s[r/x]] & \overset{M_0',\infty}{- - - \succ} & C[s_1',\ldots,s_n',s'[r'/x]] \end{array}$$

For case a similar diagram can be drawn. For `seq` the diagram is as follows:

$$\begin{array}{ccc} C[s_1,\ldots,s_n,(\texttt{seq}\ v\ t)] & \xrightarrow{M_0,\infty} & C[s_1',\ldots,s_n',(\texttt{seq}\ v'\ t')] \\ {\scriptstyle (seqTr)}\,\vdots & & \Big\downarrow{\scriptstyle (seqTr)} \\ C[s_1,\ldots,s_n,t] & \overset{M_0',\infty}{- - - - \succ} & C[s_1',\ldots,s_n',t'] \end{array}$$

The diagram shows how to construct the required reduction.

Lemma B.9. *Let S be a tree with $S{\Downarrow}(\infty)$. Then there is a finite (perhaps non-$\mathcal{R}$-) reduction sequence $S \xrightarrow{(Tr),*} T$, such that T is a value tree.*

Proof. Let RED be the ∞-reduction sequence from S to a value tree S'. Lemma B.3 shows that the set of all positions in S that are mapped by RED to the top position of S' is an FAPC-set. Using Lemma B.6, the reduction sequence can be splitted into $S \xrightarrow{(Tr),*} S_{0,1} \xrightarrow{(Tr),*} S_{1,1} \xrightarrow{\infty} S_1$, such that the reduction $S_{1,1} \xrightarrow{\infty} S_1$ is w.r.t. a set $M_{1,1}$ of irrelevant positions.

Now induction on the length of the reduction sequence $S_1 \xrightarrow{(Tr),*} S'$ and using Lemma B.8 shows that the reduction $S_{1,1} \xrightarrow{\infty} S_1$ can be shifted to the end of the reduction sequence until we obtain a reduction sequence $S \xrightarrow{(betaTr),*} S''$, where S'' is a value. □

The remaining step for the standardization theorem is to remove the non-$\mathcal{R}$-reduction either by shifting them to the right in the reduction sequence until they are no longer necessary, or until they are also $\mathcal{R}$-reductions. This shifting may increase the number of single reductions. Note, that the diagram for the overlapping case of two (betaTr)-reductions

$$\begin{array}{ccc} C[(\lambda x.r)s] & \xrightarrow{(betaTr)} & C[(\lambda x.r)s'] \\ \mathcal{R}\,\downarrow & & \downarrow\,\mathcal{R} \\ C[r[s/x]] & \xrightarrow{(betaTr),*} & C[r[s'/x]] \end{array}$$

is only valid, if the number of occurrences of the variable x in r is finite. Hence a further analysis is required, which is possible due to the distinction between relevant and irrelevant positions.
Now we show that a reduction sequence of a tree to a value can be done by reducing finitely many redexes in reduction position, i.e. by an $\mathcal{R}$-reduction.

Lemma B.10. *Let $S_1 \xrightarrow{a} S_2 \xrightarrow{\mathcal{R},(Tr),*} S'$ for $a \in \{(betaTr), (seqTr), (caseTr)\}$, where S' is a value. Then $S_1\Downarrow$, i.e. there is also an evaluation $S_0 \xrightarrow{\mathcal{R},(Tr),*} S''$, where S'' is a value.*

Proof. The proof is by analyzing the traces of the relevant positions using joining labeled reduction. Let RED $\equiv S_1 \xrightarrow{a} S_2 \xrightarrow{(Tr),\mathcal{R}} S_3 \xrightarrow{(Tr),*} S'$ be a reduction sequence from S_1 to the value S'. Let the FAPC-set M be the set of relevant positions in S_1. We analyze the possibilities to commute a non-$\mathcal{R}$-reduction with the following $\mathcal{R}$-reduction: There are two possibilities:

$$\begin{array}{ccc} S_1 & \xrightarrow{\neg\mathcal{R},a} & S_2 \\ \mathcal{R}\,\downarrow & & \downarrow\,\mathcal{R} \\ S_1' & \xrightarrow{a} & S_3 \end{array} \qquad \begin{array}{ccc} S_1 & \xrightarrow{\neg\mathcal{R},a} & S_2 \\ \mathcal{R}\,\downarrow & & \downarrow\,\mathcal{R} \\ s_1' & \xrightarrow{a,\infty} & S_3 \end{array}$$

where the first diagram covers the case of independent positions of the reductions, the case where the $\mathcal{R}$-reduction is a `seq`-reduction, and the cases that the a-reduction is within $(\lambda x.r)$ for a (betaTr)-$\mathcal{R}$-reduction with redex $((\lambda x.r)\ s)$, or within the alternatives for a (caseTr)-$\mathcal{R}$-reduction with redex $((\lambda x.r)\ s)$; and the second diagram covers the overlapping cases, where the a-redex may be copied several times by the $\mathcal{R}$-reduction. Lemma B.6 shows that the second diagram can be further modified as

$$\begin{array}{ccccc} S_1 & & \xrightarrow{\neg\mathcal{R},a} & & S_2 \\ \mathcal{R}\,\downarrow & & & & \downarrow\,\mathcal{R} \\ S_1' & \xrightarrow{a,*} & S_4 & \xrightarrow{a,\infty,M'} & S_3 \end{array}$$

where M' is a set of irrelevant positions. Lemma B.8 and Lemma B.7 show that in the case of the second diagram, the reduction w.r.t. the irrelevant set of positions can be shifted to the right end of the reduction sequence RED.

We consider reduction sequences that are mixtures of $\xrightarrow{\neg\mathcal{R}}$ and $\xrightarrow{\mathcal{R}}$-reductions, where the goal is to construct a $\xrightarrow{\mathcal{R},(Tr),*}$-reduction sequence to a value. The start is the reduction sequence RED $\equiv S_0 \xrightarrow{\mathcal{R},a} S_1 \xrightarrow{\mathcal{R},(Tr),*} S'$ where S' is a value. The operation on the reduction sequences is to focus the rightmost subsequence $T_1 \xrightarrow{\neg\mathcal{R},a} T_2 \xrightarrow{\mathcal{R},(Tr)} T_3$ for $a \in \{(\text{betaTr}), (\text{seqTr}), (\text{caseTr})\}$, and to apply one of the following:

1. If $T_1 \xrightarrow{\mathcal{R},a} T_2$ is at an irrelevant position, then shift $\xrightarrow{\mathcal{R},a}$ to the end of the reduction sequence. This is considered as one step of the operation.
2. If $T_1 \xrightarrow{\mathcal{R},a} T_2$ is at a relevant position, but the two redexes are at independent positions, then apply the first diagram, if the mentioned conditions are satisfied.
3. If $T_1 \xrightarrow{\mathcal{R},a} T_2$ is at a relevant position, and the redexes overlap, then apply the second diagram; in this case a shifting-away of the irrelevant reduction immediately follows and is counted as one step.

We have to show that finitely many such operations of modifying the reduction sequence are sufficient to reach the desired $\mathcal{R}$-reduction sequence.

Now we construct a measure for mixed reduction sequences. Let RED $\equiv S_0 \to S_1 \to \ldots \to S_n$, let RED' be the reduction sequence to a value after the modification, and let *Trace*(RED) be defined as follows: It contains all sequences $p_0, p_1, \ldots, p_n$, called *traces*, where p_0 is a RED-relevant position of S_0, and for all i: p_i is a relevant position in S_i, and p_{i+1} is a successor of p_i. The trace stops either at the last term, or at p_i, if p_i is the position of the $\mathcal{R}$-redex that is reduced in this step, or the position of the term $(c\ s_1 \ldots s_n)$ in the $\mathcal{R}$-(caseTr)-redex $(\texttt{case}\ (c\ s_1 \ldots s_n)\ \ldots)$. An *annotated trace* is a trace, where the form of inheritance is also annotated: $p \xrightarrow{a_1} p_1 \xrightarrow{a_2} \ldots, \xrightarrow{a_n} p_n$, where $a_i \in \{inst, red, trans\}$, where $a_i = red$ means that p_{i-1} is exactly the redex-position of a non-$\mathcal{R}$-a-reduction, and *inst* means that P_{i-1} is in the argument of the redex of the $\mathcal{R}$ or non-$\mathcal{R}$-(betaTr)-reduction, or within a term s_i in a (caseTr)-redex, $\mathcal{R}$ or non-$\mathcal{R}$, of the form $(\texttt{case}\ (c\ s_1 \ldots s_n)\ alts)$. The $\mathcal{R}$-redex does not occur, and the other possibilities are annotated with $a_i = trans$.

We only use the *fingerprint* of traces, which is the sequence of *inst* and *red* occurring in a trace. Two fingerprints are compared first by length, and then lexicographically as strings, where $inst < red$. The whole reduction sequence is measured by a triple $\mu = (\mu_1, \mu_2, \mu_3)$, where μ_1 is the multiset of all fingerprints of (relevant) traces, where we use the multiset-ordering for comparing multisets, μ_2 is the number of non-$\mathcal{R}$-reductions of the reduction sequence to a value, μ_3 is the number of $\mathcal{R}$-reductions after the rightmost non-$\mathcal{R}$-reduction in the reduction sequence, and we use the lexicographic ordering on μ.

This is a well-founded measure, see e.g. [6] for the multiset-part. We have to show that every diagram application strictly reduces this measure. The trivial

commuting diagram leaves the fingerprints as they are, since the positions of reductions are independent, and the *trans*-reductions are ignored in the fingerprints of traces, and strictly decreases μ_2, or leaves μ_2 invariant and strictly decreases μ_3. The hard part is to treat the application of the overlapping diagram.

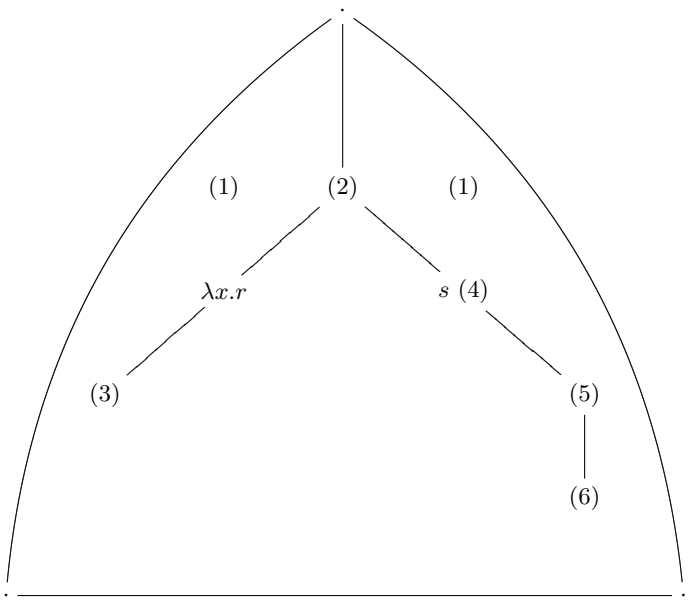


Fig. 4. Cases for the position p in traces for a (betaTr)-$\mathcal{R}$-redex

For the following case distinction we treat the two possibilities that the $\mathcal{R}$-redex is a (betaTr)-redex of the form $((\lambda x.r)\ s)$, or a (caseTr)-redex of the form $(\texttt{case}\ (c\ s_1 \dots s_n)\ \dots); \dots; cx_1 \dots c_n \to t_c; \dots)$. For a redex $((\lambda x.r)\ s)$, the picture in figure 4 illustrates the possibilities. There are several cases for a relevant position p:

1. p is independent of the position of the redexes, or a proper prefix of the position of the $\mathcal{R}$-redex. Then the trace remains unchanged by the diagram application.
2. p is the position of the $\mathcal{R}$-redex, or p is the position of the $(c\ s_1 \dots s_n)$ in the $\mathcal{R}$-redex for a (caseTr)-reduction. Then the fingerprint of the trace stops for both reduction sequences.
3. p is within $\lambda x.r$ if the $\mathcal{R}$-redex is a (betaTr), or within t in the $\mathcal{R}$-redex if it is a (caseTr). Then the fingerprint of the trace is unchanged.
4. p is within s, but not within the redex in s for a (betaTr)-$\mathcal{R}$-reduction, or p is within some s_i, but not within the redex in s_i for a (caseTr)-$\mathcal{R}$-reduction.

Then the fingerprint part is *inst* for all traces and unchanged. Also the number of traces remains the same.

5. p is the redex position within s for (betaTr) or within s_i for a (caseTr). Then the fingerprints before are $\langle \ldots red, inst \ldots \rangle$. They are changed into $\langle \ldots inst, red \ldots \rangle$, if the corresponding reduction in the bottom arrow of the second diagram is not turned into an $\mathcal{R}$-reduction. Otherwise the trace is stopped before the *red*. At least one fingerprint of some trace will be replaced by a strictly smaller one.
6. p is properly within the redex in s for a (betaTr)-reduction or in s_i for a (caseTr)-reduction. Then the fingerprints $\langle inst, inst \rangle$ remains the same, though the middle position is modified. The number of traces is the same.

Now we have to argue that the measure is indeed strictly reduced. There are several cases:

1. If the first diagram is applied, then μ_2 is strictly reduced, or μ_2 is the same, and μ_3 is strictly reduced.
2. If the second diagram is applied, and the position of the redex in s is relevant, then μ_1, the multiset of traces is strictly reduced, since according to case (5), at least one trace is strictly decreased.
3. If the second diagram is applied, and the position of the redex in s is irrelevant, then the lower part of the second diagram consists of an irrelevant reduction, that can be moved to the end of the reduction sequence without changing the traces, and a non-$\mathcal{R}$-reduction is removed, hence μ_3 is strictly decreased.

Since the measure is well-founded and strictly decreased in every step, the diagram-application is able to shift the non-$\mathcal{R}$-reductions to the right, until an evaluation is reached.

Remark B.11. For other kinds of orderings on traces, the case $((\lambda x.x)\ s) \to s$ may be exceptional. The problem is solved in our treatment by stopping the traces after an $\mathcal{R}$-reduction.

Lemma B.12. *Let S be a tree, such that $S \xrightarrow{(Tr),*} S'$, where S' is a value tree. Then there is also an $\mathcal{R}$-(tr)-reduction sequence to a value tree, i.e., $S{\Downarrow}$.*

Proof. This follows by induction on the length of the reduction sequence using Lemma B.10.

The lemmas in this appendix imply now Theorem 3.11.