Equivalence of Call-By-Name and Call-By-Need for Lambda-Calculi with Letrec

Manfred Schmidt-Schauß

Fachbereich Informatik und Mathematik,
Institut für Informatik, Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany,
schauss@ki.informatik.uni-frankfurt.de

Technical Report Frank-25

29. September 2006

Abstract. We develop a proof method to show that in a (deterministic) lambda calculus with letrec and equipped with contextual equivalence the call-by-name and the call-by-need evaluation are equivalent, and also that the unrestricted copy-operation is correct. Given a let-binding x = t, the copy-operation replaces an occurrence of the variable x by the expression t, regardless of the form of t. This gives an answer to unresolved problems in several papers, it adds a strong method to the tool set for reasoning about contextual equivalence in higher-order calculi with letrec, and it enables a class of transformations that can be used as optimizations. The method can be used in different kind of lambda calculi with cyclic sharing. Probably it can also be used in non-deterministic lambda calculi if the variable x is "deterministic", i.e., has no interference with non-deterministic executions. The main technical idea is to use a restricted variant of the infinitary lambda-calculus, whose objects are the expressions that are unrolled w.r.t. let, to define the infinite developments as a reduction calculus on the infinite trees and showing a standardization theorem.

1 Introduction

A good semantics that supports all phases from programming, compiling, verification, optimization to execution is indispensable for the reliable use of a programming language. Extended lambda calculi are widely used to provide operational semantics for programming languages. In this paper we will make a contribution to the semantics of non-strict functional programming languages like Haskell [16] and Clean [17]. An efficient evaluation of programs in these languages is based on call-by-need evaluation that implements call-by-name evaluation avoiding multiple evaluation of the same expression, for example Haskell has a call-by-name semantics, but the implementation of evaluation is call-by-need.

Hence it is important to investigate lambda calculi having a possibility to represent sharing of subexpressions, which is usually made explicit by let-expressions and also by recursive let-expressions [6,1,4,5,12]. The calculus in general determines an evaluation relation and an equivalence of expression. The first is used for correct and efficient execution, the second for correctness of transformations and optimizations. There are different technical methods to define equivalence of expressions for lambda calculi. Equality may be derived directly from the reduction by the compatible equivalence closure, which is also called conversion equality. Other possibilities are by predefining the equality axioms, or by defining equality via equality of (finite or infinite) normal forms, or by defining equality using a notion of information content [2,3], or by defining equality as observational equality, where expressions are regarded as equal, if they cannot be distinguished by all permitted observations. The coarsest equality that justifies correctness of a maximal number of program transformations is in general provided by contextual equality.

1.1 A Framework for Lambda Calculi

Our view of (deterministic) lambda-calculi, their reduction relation and their equational theory is as follows:
We assume given the language L of terms, a reduction $\rightarrow$ of terms (the abstract reduction machine), the successful results (or values) V as a subset of terms, and the one-hole contexts $\mathcal{C}$ valid in the language L. Convergence (or termination) of a term t, denoted $t{\Downarrow}$ is defined on this basis as $t{\Downarrow}$ iff there exists a reduction sequence from t to some value v: $t \xrightarrow{*} v$. Then observational equality is defined as contextual equivalence:
$s \leq_c t$ iff for all contexts $C : C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}$, and
$s \sim_c t$ iff $s \leq_c t$ and $t \leq_c s$.
The equality provides us with rigorous criteria for the correctness of transformations, which can only transform $\sim_c$-equal expressions. In contrast with explicitly given equality axioms of a lambda calculus, our presentation implicitly defines the equations through $\sim_c$, and it may be that there is no finite set of equality axioms for the equality theory. An advantage is that contextual equality is always the largest possible one. It is also clear that the equivalence $\sim_c$ is a congruence, i.e. $s \sim_c t$ implies $C[s] \sim_c C[t]$, and that the given reduction has a weak standardization property for $\sim_c$ in the following sense: $s \sim_c t \Rightarrow s{\Downarrow} \Leftrightarrow t{\Downarrow}$. In concrete calculi, usually values are abstractions and reduction is normal-order reduction. Since the equality is implicitly given, there are several proof obligations: One is to show that the given (deterministic) calculus is "internally consistent", i.e. that $s \rightarrow t \Rightarrow s \sim_c t$. If this is shown, then the following is immediate: the reduction relation $\rightarrow$ can be extended to a general reduction relation $\xrightarrow{\mathcal{C}}$ by $C[s] \xrightarrow{\mathcal{C}} C[t]$ if $s \rightarrow t$ for any context C, where $\xrightarrow{\mathcal{C}} \subseteq \sim_c$ and hence that the reduction relation $\xrightarrow{\mathcal{C}}$ can be used as a partial evaluation. Also full standardization in the following sense holds: If $s \xrightarrow{\mathcal{C},*} t$ where t is a value, then $s \sim_c t$, and hence $s{\Downarrow}$. This property

of internal consistency is also the replacement for the Church-Rosser-property: If $s \xrightarrow{\mathcal{C},*} s_1$ and $s \xrightarrow{\mathcal{C},*} s_2$, then $s_1 \sim_c s_2$.
The other proof obligation is to exhibit a large set of equalities and correct transformations that hold in the given calculus, i.e. that are correct w.r.t. contextual equality, and also to provide proof tools for proving these equalities.
The reduction $\rightarrow$ can be restricted by a minimality principle: Defining a small number of reduction rules, and also restricting their applicability, such that reduction is deterministic for a deterministic lambda calculus. This may support easy reasoning about the calculus and its equality. It is important to note that this framework also permits a switch to another reduction relation, perhaps more efficient or otherwise more convenient, as long as convergence and contextual equality remains unchanged.
This framework has the potential of unifying the view of lambda-calculi reduction strategies insofar as all the variants of strategies in call-by-name and call-by-need calculi can be compared for their equational theory. Though not explicitly proved, it appears that in the pure untyped lambda calculus, lots of added equations or rules like fully lazy reduction or optimal reduction do not change the equational theory, and can thus be seen as different reduction strategies for essentially the same calculus. We conjecture that the same is true for call-by-name-based let- or letrec-calculi, which are often extended by different transforming and optimizing rules like garbage collection, fully-lazy reduction, or variants in the treatment of let-shuffling, instantiation or copy-reduction. E.g the strict inclusion of the theory of call-by-need in the theory of call-by-name for a non-recursive `let` mentioned in [5] will turn into an equivalence using contextual equivalence.

1.2 Call-by-Name, call-by-Need and Let-Lambda-Calculi

An early and influential comparison between different implementations of lambda-calculi was Plotkin's [18] treatment of call-by-value, call-by-name strategies and different abstract machines as implementations, where Plotkin used contextual equivalence as one of its criteria. It becomes clear that call-by-value and call-by-name are essentially different. 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. [6]). However, `letrec` cannot easily be translated or simulated in `letrec`-free calculi. It is also well-known that `letrec` has improved sharing properties during reductions (see e.g. [1]), and also allows in several cases to syntactically detect non-termination, which otherwise would require a loop-checker for a reduction.
In calculi with sharing an important issue is in which cases an improvement of sharing is permitted, or the counterpart, which kind of unsharing is permitted, perhaps to enable other program transformations, for example inlining and common subexpression elimination. Note that in non-deterministic calculi, modifying sharing is in general not correct, however, there are also several special

cases where it is explicitly required or permitted, but also several cases, where it is completely unclear to which extent modifying sharing is correct (see e.g. [14,15]). These are usually very hard questions, for example in the letrec-calculi in [15,9] proofs of correctness of corresponding program transformations are missing. In the deterministic `letrec`-calculus treated in [20], 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 diagram-based methods do not work. Even the proofs that the special 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. A reason for the failure of diagram-based methods is that the copy-reduction and the let-based reductions interfere, and that the duplication of subterms prevents to construct well-founded measures on terms and reduction sequences.
There are several paper investigating the relationship between call-by-name and call-by-need calculi (e.g. [6,5,12]). Other work on lambda-calculi extended with letrec is centered around confluence or non-confluence properties of the reduction relation of the calculi ([4,1]). A proof of the observational equivalence of call-by-name and a call-by-need calculus with non-recursive let is in [12], which also mentions at the very end an open question concerning instantiating of non-values, 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. 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, which is also implicitly mentioned as unresolved in [5].

1.3 Structure and Result of this Paper

This paper treats a tiny letrec-calculus which is equipped with a normal order reduction and a contextual semantics as definition of equality of expressions. First it defines the infinite trees corresponding to the unrolling of expressions as in the 111-calculus of [10]. Then reduction rules on the infinite trees are defined, where the basic rule is the beta-rule, and the other rule $\xrightarrow{\infty}$ corresponds to an infinite development (see also [10]). Then it is shown that termination of expressions in the call-by-need lambda-calculus, as well as for the call-by-name calculus is equivalent to termination of beta-reduction on the corresponding infinite trees. An essential step is the standardization lemma for $\xrightarrow{\infty,*}$-reductions on the infinite trees. Finally, as a corollary we obtain the correctness of the copy-rule, which holds in both calculi, since they are equivalent. Thus we have solved an open problem mentioned in [5], the equivalence of a call-by-need letrec-calculus and a call-by-name letrec-calculus.
Our method could be applied to the letrec-calculi from [6,1,4,5,12], if they adopt the contextual equivalence as equality, which appears to cover all the desired

equalities in these calculi. It could also be applied to the record calculus of Machkasova and Turbak [11], though there may be other specialized methods in this calculus, since in this calculus letrec is only allowed at the top level of expressions. We believe that our method can be extended also to calculi with constructors, which we leave as future work.

2 Syntax and Reductions of the Functional Core Language LRλ

2.1 The Language and the Reduction Rules

We define the calculus LRλ consisting of a language $\mathcal{L}(\text{LR}\lambda)$ and its reduction rules, presented in this section, the normal order reduction strategy and contextual equivalence. The syntax for expressions E is as follows:

$$E ::= V \mid (E_1\ E_2) \mid (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$$

where E, E_i are expressions and V, V_i are variables. The expressions $(E_1\ E_2)$, $(\lambda V.E)$, $(\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$ are called *application*, *abstraction*, or `letrec`-*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. `letrec` is 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 $i = 1, \ldots, n$. 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 rule (cp) (see below). We omit parentheses in nested applications: $(s_1 \ldots s_n)$ denotes $(\ldots(s_1\ s_2)\ldots s_n)$ provided s_1 is an expression. The set of closed LRλ-expressions is denoted as $\text{LR}\lambda^0$.

Sometimes we abbreviate the notation of `letrec`-expression $(\texttt{letrec}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ E)$, as $(\texttt{letrec}\ Env\ \texttt{in}\ E)$, where $Env \equiv \{x_1 = E_1, \ldots, x_n = E_n\}$. This will also be used freely for parts of the bindings. The set of bound variables 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.

(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
(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
(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)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \to (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$

Fig. 1. Reduction rules for call-by-need

Definition 2.1. *The class $\mathcal{C}$ of all* contexts *is defined as the set of expressions C from* LRλ*, 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 2.2. *A* value *is an abstraction. We denote values by the letters v, w. A* weak head normal form (WHNF) *is either a value, or an expression* $(\texttt{letrec}\ Env\ \texttt{in}\ v)$*, where v is a value.*

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

Definition 2.3 (Reduction Rules of the Calculus LRλ). *The (base) reduction rules for the calculus and language* LRλ *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) and (lapp) 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. We will also speak of the* inner redex*, which is the variable position which is replaced by a (cp). Otherwise it is the same as the redex. Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $*$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of $\to$. If necessary, we attach more information to the arrow.*

(lbeta)	$((\lambda x.s)^S\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(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
(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
(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)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ s) \to (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$

Fig. 2. Normal Order Reduction rules

2.2 The Unwind Algorithm

Searching for a maximal reduction context can be seen as an algorithm walking over the term structure. In implementations of functional programming this is usually called "unwind".
The following labeling algorithm will detect the position to which a reduction rule will be applied according to normal order. It uses four labels: $S, T, S \vee T, V$, where T means reduction of the top term, S means reduction of a subterm, $S \vee T$ matches T as well as S, and V labels already visited subexpressions. 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:

$$
\begin{array}{ll}
(\texttt{letrec}\ Env\ \texttt{in}\ t)^T & \to (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V \\
(s\ t)^{S\vee T} & \to (s^S\ t)^V \\
(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) & \to (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V]) \\
 & \quad \text{if } s \text{ was not labeled } V \\
(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ t) & \to (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ t) \\
 & \quad \text{if } s \text{ was not labeled } V \text{ and if } C[x] \neq x
\end{array}
$$

This algorithm terminates. For example for $(\texttt{letrec}\ x = x\ \texttt{in}\ x)^T$ it will stop with $(\texttt{letrec}\ x = x^S\ \texttt{in}\ x)$.

Definition 2.4 (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 $\xrightarrow{n}$.*

Lemma 2.5. *Normal order reduction is unique.*

Definition 2.6. *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 *and denote it as $nor(t)$. If $t{\Downarrow}$, we also say that t is* terminating. *Otherwise, if there is no evaluation of t, we write $t{\Uparrow}$.*

Definition 2.7 (contextual preorder and equivalence). *Let s, t be terms. Then:*

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

3 Reductions on Trees

In the following we use "expression" for finite expressions including `letrec`, and "tree" for the finite or infinite trees, which are only built from applications, abstractions and variables.
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 nontermination, represented by the symbol $\bot$. This corresponds to the infinite trees in the 111-variant of the calculus in [10]. 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.

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 as follows:

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

If the position ε hits the same (let-bound) variable twice, then the result is $\bot$.
For the other cases, we proceed as follows:

$$\begin{array}{ll} C[(\lambda x.s)|_{1.p}] & \to C[\lambda x.(s|_p)] \\ C[(@\ s\ t)|_{1.p}] & \to C[(@\ s|_p\ t)] \\ C[(@\ s\ t)|_{2.p}] & \to C[(@\ s\ t|_p)] \\ C[(\texttt{letrec}\ Env\ \texttt{in}\ r)|_p] & \to C[(\texttt{letrec}\ Env\ \texttt{in}\ r|_p)] \\ C_1[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x|_p])] & \to C_1[(\texttt{letrec}\ x = s|_p, Env\ \texttt{in}\ C_2[x])] \\ C_1[(\texttt{letrec}\ x = s, y = C_2[x|_p], Env\ \texttt{in}\ r)] & \to C_1[(\texttt{letrec}\ x = s|_p, y = C_2[x], Env\ \texttt{in}\ r)] \end{array}$$

In all cases not mentioned above, the result is undefined (and also not necessary).
The equivalence of trees is syntactic, where α-equal trees are assumed to be equivalent. If $IT(s) = IT(t)$ for two expressions s, t, then we write $s =_\infty t$.

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* $\mathcal{R} ::= [\cdot] \mid (@\ \mathcal{R}\ E)$.

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

Proof. For closed s, t this is obvious. In the general case, there may be occurrences of free variables in s, t. Consider the computation of the labels of the infinite trees in the case that there is a free variable x in s, t that is bound in C. Also in this case the label will be the same for $C[s]$ and $C[t]$, since the computation within s, t results in the same tree labels.

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

Proof. This is obvious, since the (cp)-rule and all (lll)-rules commute with the label computation.

Definition 3.6. *(betaTr) is the only reduction rule on trees. It is also allowed in any context.*

$$\boxed{\textit{(betaTr)}\ ((\lambda x.s)\ r) \to s[r/x]}$$

If the 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 positions labeled with x. Further note that (betaTr) does not overlap with itself, where we usually ignore overlaps within the meta-variables s, r.

Definition 3.7. *A tree of the form $\lambda x.s$ is called a* value.

Lemma 3.8. *Let s be an expression and let $IT(s)$ be a value tree. Then $s{\Downarrow}$.*

Proof. A normal order reduction of s can only be a (cp) or an (lll) reduction, which follows by induction on the structure of s and a case analysis. The same argument shows that if no more normal reductions are possible, then s is a WHNF. Since neither (cp) nor (lll)-reductions modify the corresponding infinite tree, all reductions of the normal order reduction can only be (cp) or (lll)-reductions. This will terminate, since (cp) and (lll)-reductions strictly reduce the number of steps of the unwind-algorithm.

We will use a variant of infinite developments [8,10] as a reduction on trees that may reduce infinitely many redexes in one step. In order to define it, we require 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 3.9 (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 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.*
- *the label sets of the new occurrences of s in $C[r[s/x]]$ are as follows: For every occurrence of x in r if $r \neq x$ it can be written as follows: $C[(\lambda x.D[x^A])\ s^B] \to C[(D\rho)[s^{A \cup B}]$ with $\rho := [s/x]$.*
- *The label set of $r[s/x]$ for $r \neq x$ is computed as follows: $((\lambda x.r^A)^B\ s^C)^D \to r[s/x]^{A \cup B \cup D}$. If $r = x$, then $((\lambda x.x^A)^B\ s^C)^D \to r[s/x]^{A \cup B \cup C \cup D}$.*

The consuming labeled reduction *is similar to the joining variant, but the labels at the reduced redex are consumed, and thus removed. The inheritance is the same as for the trivial cases, however, the label of the reduct of the redex is computed differently: $((\lambda x.r^A)^B\ s^C)^D \to r[s/x]^{A \cup B}$ if $r \neq x$, and $((\lambda x.x^A)^B\ s^C)^D \to r[s/x]^{A \cup B \cup C}$, if $r = x$.*
We will use the consuming labeled reduction below for one label †. 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.

We define a reduction relation between two infinite trees, which corresponds to the relation defined by outside-in infinitary developments in [10] where we use the variant, that if a subtree has an infinite sequence of reductions at top level, then the reduction of this subtree results in $\bot$. Note that confluence of reduction within the 111-infinitary calculus (see also Theorem 5.2 of [4]) cannot be used here, since the calculi are different, and since the standardization cannot be derived from confluence.

Definition 3.10. *For trees S, T, we define a reduction $S \xrightarrow{\infty} T$ as follows. First, we mark a possibly infinite subset of all $(betaTr)$-redexes in S, say with a †, where we use consuming labeled reduction for the inheritance of labels. The reduction constructs a new infinite tree top-down as follows, where A is the currently considered subtree.*

- *If A is not marked with a †, then the reduction proceeds with the direct subtrees of A.*
- *If A is marked with a †, then the $(betaTr)$-reduction is applied to A. After the reduction, the label of A will again be inspected and the same procedure is applied. If for the subtree A the reduction does not terminate at the top level of A, then the result of the reduction of subtree A is $\bot$.*

The result is the recursively defined (infinite) tree.
We write $T{\Downarrow}(\infty)$ if $T \xrightarrow{\infty,*} T'$, where T' is a value tree.

The reduction $S \xrightarrow{\forall,\infty} T$ is defined as the $S \xrightarrow{\infty} T$-reduction, if all $(betaTr)$-redexes in S are labeled with †.

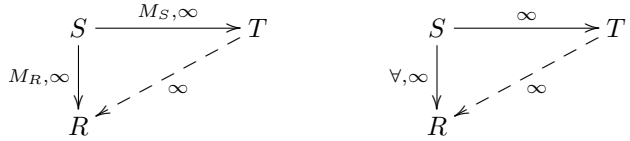
Note that even for only two marked redexes, it is possible that after the first reduction, infinitely many redexes are labeled.

Example 3.11. 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 infinite 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., $t \xrightarrow{\forall,\infty} \bot$.

3.1 Standardization of Tree Reduction

Lemma 3.12. *For all trees S, R, T: if $S \xrightarrow{\infty} R$ where the set of redex positions is M_R, and $S \xrightarrow{\infty} T$, where the the set of redex positions is M_S, and $M_S \subseteq M_R$, then also $T \xrightarrow{\infty} R$. A special case is that $S \xrightarrow{\forall,\infty} R$, and $S \xrightarrow{\infty} T$ imply that $T \xrightarrow{\infty} R$.*



Proof. The argument is that we can mark the $(betaTr)$-redexes in S that are not reduced in $S \xrightarrow{M_S,\infty} T$. This can be detected by a consuming labeled reduction, where the M_R-redexes are labeled. Then reduce all these labeled redexes in the reduction $T \xrightarrow{\infty} R$.

Definition 3.13. *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 also call this set* FAPC-set *of positions of T.*

In the following we use sets of positions in terms.

Lemma 3.14. *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 3.10. 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)-reduction, the set of labels at every position remains a finite set. If for a subtree A the top-reduction 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 3.15. *Let S be a tree, and let RED be the reduction $S = S_0 \xrightarrow{\infty} S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$, 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 3.14 to the reduction step $S_0 \xrightarrow{\infty} S_1$, which shows the Lemma.

Corollary 3.16. *Let S be a tree, and let RED be the reduction $S = S_0 \xrightarrow{\infty} S_1 \xrightarrow{\infty} S_2 \xrightarrow{\infty} \ldots \xrightarrow{\infty} S_n = S'$, 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'.*

Lemma 3.17. *Let S_0, S_1, S_2 be trees, such that M_2 is an FAPC-set of S_2, let $S_1 \xrightarrow{(betaTr)} S_2$ be a reduction at position p_1, let M_1 be all positions of S_1 that are mapped to positions in M_2, let $p_1 \in M_1$, and let $S_0 \xrightarrow{M_0,\infty} S_1$, where no position*

of M_0 is mapped to some position in M_1. Then $S_0 \xrightarrow{(betaTr)} S'_0 \xrightarrow{M'_0,\infty} S_2$, and there is no position in M'_0 that is mapped to some position in M_2.

$$\begin{array}{ccc} S_0 & \xrightarrow{M_0,\infty} & S_1 \\ \Big| & & \Big| \\ (betaTr) & & (betaTr) \\ \Big| & & \Big| \\ \downarrow & & \downarrow \\ S'_0 & \overset{M'_0,\infty}{- - \succ} & S_2 \end{array}$$

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

$$\begin{array}{ccc} C[s_1,\ldots,s_n,((\lambda x.s)\ r)] & \xrightarrow{M_0,\infty} & C[s'_1,\ldots,s'_n,((\lambda x.s')\ r')] \\ \Big| & & \Big| \\ (betaTr) & & (betaTr) \\ \Big| & & \Big| \\ \downarrow & & \downarrow \\ C[s_1,\ldots,s_n,s[r/x]] & \overset{M'_0,\infty}{- - \succ} & C[s'_1,\ldots,s'_n,s'[r'/x]] \end{array}$$

The diagram shows how to construct the required reductions.

Lemma 3.18. *Let S be a tree with $S{\Downarrow}(\infty)$. Then there is a finite reduction $S \xrightarrow{(betaTr),*} S'$, such that S' is a value tree.*

Proof. Let RED be the reduction from S to a value tree S'. Lemma 3.15 shows that the set of all positions in S that are mapped by RED to the top position of S' is an FAPC-set.

We use induction on the length of the reduction RED to show the claim. If the reduction has length 0, then it is obvious. Otherwise, in the induction case, we have $S \xrightarrow{\infty} S_1 \xrightarrow{(betaTr),*} S''$, where S'' is a value. Lemma 3.15 shows that the set M_0 of positions from S that are mapped to the top level of S'' is an FAPC-set. We split the reduction $S \xrightarrow{\infty} S_1$ into $S \xrightarrow{(betaTr),*} S_{0,1} \xrightarrow{\infty} S_1$, such that $S \xrightarrow{(betaTr),*} S_{0,1}$ are all the top level reductions, and the first reduction in the definition of $S_{0,1} \xrightarrow{\infty} S_1$ is not at top level. Let $M_{0,1}$ be the FAPC-set of all positions that are mapped by $S_{0,1} \xrightarrow{\infty} S_1$ to M_1. 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 show that there is a reduction sequence $S_{0,1} \xrightarrow{(betaTr),*} S_{1,1} \xrightarrow{\infty} S_1$, such that no †-labeled position in $S_{1,1}$ is mapped by $S_{1,1} \xrightarrow{\infty} S_1$ to M_1.

Now induction on the length of the reduction $S_1 \xrightarrow{(betaTr),*} S''$ and using Lemma 3.17 shows that the reduction $S_{1,1} \xrightarrow{\infty} S_1$ can be shifted to the end of the reduction until we obtain a reduction $S \xrightarrow{(betaTr),*} S'''$, where S''' is a value.

Now we show that a reduction of a tree to a value can be done by reducing finitely many redexes in reduction position. Note that the following proof and the method can be transferred to the untyped lambda calculus (on finite expressions).

Lemma 3.19. *Let* $S_0 \xrightarrow{(betaTr)} S_1 \xrightarrow{\mathcal{R},(betaTr),*} S'$*, where* S' *is a value. Then* $S_0\Downarrow$*, i.e. there is also an evaluation* $S_0 \xrightarrow{\mathcal{R},(betaTr),*} S''$*, where* S'' *is a value.*

Proof. The proof is by analyzing the trace of the positions using joining labeled reduction. The problem is that the diagram for the overlapping case

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

is only valid, if the number of occurrences of the variable x in r is finite. Assume that there is an FAPC-set M of $C[r[s'/x]]$ of positions that are mapped to the top level position of a value by an evaluation starting from $C[r[s'/x]]$. Using Lemma 3.14, we see that it is sufficient to reduce only a finite number of occurrences of s to s' within $r[s/x]$ in order to reach some value tree. Hence the diagram w.r.t. M is

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

where the positions in M' will not be mapped to the final top position of a value. Similar as in the proof of Lemma 3.18, the reduction $\cdot \xrightarrow{\infty,M'} C[r[s'/x]]$ can be shifted along the evaluation to the end. This will motivate the following notion. We will call the positions that will be mapped by a reduction to the top of the final value tree as *relevant*, the other positions as *irrelevant*, where we assume that the reduction is clear from the context. An application of the diagram is always accompanied by a shift of the irrelevant reduction to the end, which means to modify the whole evaluation $S_0 \xrightarrow{(betaTr)} S_1 \xrightarrow{\mathcal{R},(betaTr),*} S'$, which is only different from the former one at irrelevant positions in the trees There is another (trivial) diagram

$$\begin{array}{ccc} C[((\lambda x.r)s),t] & \longrightarrow & C[((\lambda x.r)s),t'] \\ \mathcal{R}\,\downarrow & & \downarrow\,\mathcal{R} \\ C[r[s/x],t] & \longrightarrow & C[r[s/x],t'] \end{array}$$

which means simply a commutation of reductions.

Now we construct a measure for reduction sequences that is derived by starting with the reduction $S_0 \xrightarrow{(betaTr)} S_1 \xrightarrow{\mathcal{R},(betaTr),*} S'$, applying the diagrams, which will result in reduction sequences that are a mixture of $\mathcal{R}-$ and non-$\mathcal{R}$-(betaTr)-reductions.
For every position p of S_0 that is relevant w.r.t. the reduction above, the set $Trace(p, RED)$ contains all relevant traces of p, where a trace is a sequence $p, p_1, \ldots, p_n$, such that p_i is a relevant position in S_i, and p_{i+1} is a successor of p_i. An annotated trace is a trace, where the form of inheritance is also annotated: $p \xrightarrow{a_1} p_1, \ldots, \xrightarrow{a_n} p_n$, where $a_i \in \{inst, red, trans\}$, where *trans* means that the position is the same and not influenced by the (betaTr)-reduction, *red* means that it is exactly the position of the (betaTr)-reduction, and *inst* means that the position was in the argument of the redex of the (betaTr)-reduction. We only use the fingerprint of traces, which is the sequence of *inst*, *red* occurring in a trace. Two fingerprints are compared lexicographically as strings, where $inst < red$. The whole reduction is measured by a pair $\mu = (\mu_1, \mu_2)$, where μ_1 is the multiset of all fingerprints of (relevant) traces, and μ_2 is the number of $\mathcal{R}$-reductions after the rightmost non-$\mathcal{R}$-reduction in the reduction, and we use the lexicographic ordering on μ.
First of all, this is a well-founded measure, see [7] for the multiset-part. We have to show that the diagram application strictly reduces this measure. The trivial commuting diagram leaves the fingerprints as they are, since the positions of reduction are independent, and the *trans*-reductions are ignored in the fingerprints of traces. The hard part is to treat the overlapping diagram. There are several cases for a position p w.r.t. the redex $((\lambda x.r)\ s)$.

- p is independent of the position of the redexes, or a prefix of the position of the redex. Then the trace remains unchanged by the diagram application.
- p is within r. Then the fingerprint of the trace is unchanged.
- p is within s, but not within the redex in s. Then the fingerprint part is *inst* for all traces and unchanged. Also the number of traces is the same.
- p is within the redex in s. Then the fingerprints $\langle\ red, inst, \text{rest}\ \rangle$ are changed into $\langle\ inst, red, \text{rest}\ \rangle$. and there is a one-to-one correspondence, since exactly the relevant positions are reduced, and the traces are only considered for relevant positions. At least one fingerprint will be replaced by a strictly smaller one, since only trace with relevant positions are considered.

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.

Lemma 3.20. *Let S be a tree, such that $S \xrightarrow{(betaTr),*} S'$, where S' is a value tree. Then there is also an $\mathcal{R}$-reduction to a value tree, i.e., $S{\Downarrow}$.*

Proof. This follows by induction on the length of the reduction using Lemma 3.19.

The lemmas in this subsection imply now:

Theorem 3.21 (Standardization for tree-reduction). *Let S be a tree. Then $S{\Downarrow}(\infty)$ implies $S{\Downarrow}$.*

3.2 Call-by-Need Convergence Implies Infinite Tree Convergence

Lemma 3.22. *If $s \xrightarrow{lbeta} t$ for two expressions s, t, then $IT(s) \xrightarrow{\infty} IT(t)$.*

Proof. We label every redex of $IT(t)$ that is derived from the redex of $s \xrightarrow{lbeta} t$. This is easy by comparing the positions in the infinite tree before and after the reductions. 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 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 y_1 does not terminate, and hence the result will be $\bot$.

Proposition 3.23. *Let t be an expression. Then $t{\Downarrow} \Rightarrow IT(t){\Downarrow}$.*

Proof. That $IT(t){\Downarrow}(\infty)$ holds follows from Lemma 3.22 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 3.21 shows that $T{\Downarrow}(\infty)$ implies also $T{\Downarrow}$.

3.3 Infinite Tree Convergence Implies Call-by-Need Convergence

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

Lemma 3.24. *The forking diagrams for trees between an $\mathcal{R}$-reduction and an $\xrightarrow{\infty}$-reduction are as follows:*

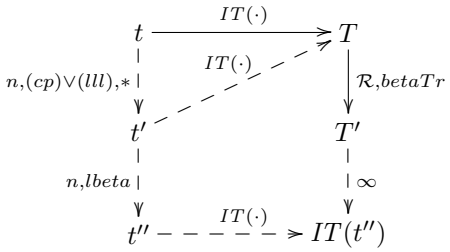
Proof. This follows by checking the overlaps with $\mathcal{R}$-reductions.

Lemma 3.25. *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 3.24 by induction.

Lemma 3.26. *Let t be a term and let $T := IT(t) \xrightarrow{(betaTr)} T'$ be an $\mathcal{R}$-reduction. Then there is an expression t', a reduction $t \xrightarrow{n,*} t'$ using (lll) and*

(cp)-reductions, an expression t'' with $t' \xrightarrow{n,lbeta} t''$, such that there is a reduction $T' \xrightarrow{\infty} IT(t'')$.



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) and (lll)-reductions. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order ($lbeta$)-redex in t' must correspond to $T \xrightarrow{\mathcal{R},(betaTr)} T'$ and is used for the reduction $t' \xrightarrow{n,lbeta} t''$. Note that the ($lbeta$)-redex in t' may correspond to infinitely many redexes in T. Lemma 3.22 shows that there is a reduction $T \xrightarrow{\infty} IT(t'')$, and Lemma 3.12 shows that also $T' \xrightarrow{\infty} IT(t'')$.

Proposition 3.27. *Let t be an expression such that $IT(t)\Downarrow$. Then $t\Downarrow$.*

Proof. The precondition $IT(t)\Downarrow$ and Theorem 3.21 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 3.8. In the general case, let $T \xrightarrow{(betaTr)} T'$ be the unique first $\mathcal{R}$-reduction of a single redex. Lemma 3.26 shows that there are expressions t', t'' with $t \xrightarrow{n,(cp)\vee(lll),*} t' \xrightarrow{n,lbeta} t''$, and $T' \xrightarrow{\infty} IT(s)$. Lemma 3.25 shows that the number of $\mathcal{R}$-reductions of $IT(s)$ 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.

The main theorem is that termination is equivalent for a term and its corresponding infinite tree.

Theorem 3.28. *Let t be an expression. Then $t\Downarrow$ iff $IT(t)\Downarrow$.*

Proof. One direction follows follows from Proposition 3.23. The other direction follows from Proposition 3.27.

Definition 3.29. *Let the generalized copy rule be:*
(gcp) $\quad C_1[\texttt{letrec}\ x = r \ldots C_2[x] \ldots] \to C_1[\texttt{letrec}\ x = r \ldots C_2[r] \ldots]$

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

Lemma 3.30. *If $s \xrightarrow{gcp} t$, then $IT(s) = IT(t)$*

Theorem 3.31. *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 3.30 implies $IT(s) = IT(t)$. Hence equivalence of termination follows from Theorem 3.28.

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

Definition 4.1. *The* call-by-name reduction *is defined by changing the call-by-need reduction: Instead of applying a normal-order (lbeta)-reduction, apply the rule*

$$(beta)\quad ((\lambda x.s)\ r) \to s[r/x]$$

to the same redex.
Accordingly, we write the normal-order call-by-name reduction as $\xrightarrow{name}$*, and denote convergence of a term* t *as* $t{\Downarrow}(name)$.

Note that $\xrightarrow{n,a}$ is the same as $\xrightarrow{name,a}$ only for $a \in \{(lll),(cp)\}$.
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 4.2. 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 is as follows, and loops.

$$\begin{array}{ll}
\xrightarrow{beta} & (\texttt{letrec}\ z = (\lambda y.(z\ z))\ \texttt{in}\ z\ z)\\
\xrightarrow{cp} & (\texttt{letrec}\ z = (\lambda y.(z\ z))\ \texttt{in}\ (\lambda y.(z\ z))\ z)\\
\xrightarrow{beta} & (\texttt{letrec}\ z = (\lambda y.(z\ z))\ \texttt{in}\ (z\ z))\\
\ldots &
\end{array}$$

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.

4.1 Call-by-Name Convergence Implies Infinite Tree Convergence

Lemma 4.3. *If* $s \xrightarrow{beta} t$ *for two expressions* s, t*, then* $IT(s) \xrightarrow{\infty} IT(t)$.

Proof. This is easy by comparing the positions in the infinite tree before and after the reductions.

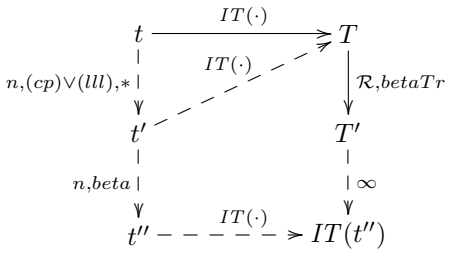
Proposition 4.4. *Let* t *be an expression. Then* $t{\Downarrow}(name) \Rightarrow IT(t){\Downarrow}$.

Proof. This follows from Lemma 4.3 by induction on the length of the call-by-name evaluation of t, from Lemma 3.5 using the standardization theorem 3.21 and from the fact that a WHNF has a value tree as corresponding infinite tree.

4.2 Infinite Tree Convergence Implies Call-by-Name Convergence

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

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



Proof. The expressions t', t'' are constructed as follows: t' is the resulting term from a maximal normal-order reduction consisting only of (cp) and (lll)-reductions. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order ($beta$)-redex in t' corresponding to $T \xrightarrow{(betaTr)} T'$ is used for the reduction $t' \xrightarrow{name,beta} t''$. Note that the ($beta$)-redex in t' may correspond to infinitely many redexes in T. Lemma 4.3 shows that there is a reduction $T \xrightarrow{\infty} IT(t'')$, and Lemma 3.12 shows that also $T' \xrightarrow{\infty} IT(t'')$.

Proposition 4.6. *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 to a value tree. The base case, where no $\mathcal{R}$-reductions are necessary is treated in Lemma 3.8. In the general case, let $T \xrightarrow{(betaTr)} T'$ be the unique first $\mathcal{R}$-reduction of a single redex. Lemma 4.5 shows that there are expressions t', t'' with $t \xrightarrow{n,(cp)\vee(lll),*} t' \xrightarrow{name,beta} t''$, and $T' \xrightarrow{\infty} IT(t'')$. Lemma 3.25 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.

The main theorem is that call-by-name termination is equivalent for a term and its corresponding infinite tree.

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

Proof. One direction follows follows from Proposition 4.4. the other direction from Proposition 4.6.

Theorem 4.8. *The contextual preorders for call-by-need and call-by-name are equivalent in a letrec-calculus.*

Proof. This follows from Theorems 3.28 and 4.7.

5 Conclusion

We showed equivalence of call-by-name and call-by-need for a tiny deterministic letrec-calculus and also the correctness of an unrestricted copy-reduction in both calculi. The method is by defining a calculus on the unrolled infinite tree, and carefully analyzing the reduction, and showing that only finitely many normal-order beta-reductions are necessary to reduce an infinite tree to a value. We expect that the method scales up to extended letrec-calculi, for example extended by constructors and case-expressions. For non-deterministic calculi like [14,13,19] the method can perhaps also be used to show correctness of the copy-reduction for deterministic subexpressions, which appears to be a hard obstacle for other methods, however, the method has to be adapted to the needs of non-deterministic calculi, in particular, the infinite trees will be significantly different, since in this case they also include letrec-expressions.

Acknowledgement

I thank David Sabel for reading and correcting drafts of this paper.

References

1. Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In *TACS*, pages 77–106, 1997. Sendai, Japan.
2. Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117:95–168, 2002.
3. Zena M. Ariola and Stefan Blom. Skew and *omega*-skew confluence and abstract Böhm semantics. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer, editors, *Processes, Terms and Cycles*, volume 3838 of *Lecture Notes in Computer Science*, pages 368–403. Springer, 2005.
4. Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997.
5. Z.M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. functional programming*, 7(3):265–301, 1997.
6. 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.
7. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

8. H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.
9. 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; 5th Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, 1999.
10. Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theor. Comput. Sci*, 175(1):93–125, 1997.
11. Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *ESOP'2000*, 2000.
12. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. of Functional programming*, 8:275–317, 1998.
13. A.K.D. Moran. *Call-by-name, call-by-need, and McCarthys Amb*. PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.
14. Andrew K.D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Computer Science*, pages 85–102. Springer-Verlag, 1999.
15. Andrew K.D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.
16. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. `www.haskell.org`.
17. 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/`.
18. Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
19. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank report 24, Institut für Informatik. J.W.Goethe-Universität Frankfurt am Main, January 2006.
20. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. A complete proof of the safety of Nöcker's strictness analysis. Technical Report Frank-20, Institut für Informatik. J.W.Goethe-University, 2005.