Towards Correctness of Program Transformations Through Unification and Critical Pair Computation

Conrad Rau* and Manfred Schmidt-Schauß

Institut für Informatik
Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
{rau,schauss}@ki.informatik.uni-frankfurt.de

Technical Report Frank-41

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

January 12, 2011

Abstract. Correctness of program transformations in extended lambda calculi with a contextual semantics is usually based on reasoning about the operational semantics which is a rewrite semantics. A successful approach to proving correctness is the combination of a context lemma with the computation of overlaps between program transformations and the reduction rules, and then of so-called complete sets of diagrams. The method is similar to the computation of critical pairs for the completion of term rewriting systems. We explore cases where the computation of these overlaps can be done in a first order way by variants of critical pair computation that use unification algorithms. As a case study we apply the method to a lambda calculus with recursive let-expressions and describe an effective unification algorithm to determine all overlaps of a set of transformations with all reduction rules. The unification algorithm employs many-sorted terms, the equational theory of left-commutativity modelling multi-sets, context variables of different kinds and a mechanism for compactly representing binding chains in recursive let-expressions.

1 Introduction and Motivation

Programming languages are often described by their syntax and their operational semantics, which in principle enables the implementation of an interpreter and a compiler in order to put the language into use. Of course, also optimizations and transformations into low-level constructs are part of the implementation. The justification of correctness is in many cases either omitted, informal or by intuitive reasoning. Inherent obstacles are that programming languages are usually complex, use operational features that are not deterministic like parallel execution, concurrent threads, and effects like input and output, and may even be modified or extended in later releases.

* This author is supported by the DFG under grant SCHM 986/9-1.

Here we want to pursue the approach using contextual semantics for justifying the correctness of optimizations and compilation and to look for methods for automating the correctness proofs of transformations and optimizations.

We assume given the syntax of programs $\mathcal{P}$, a deterministic reduction relation $\to \subseteq \mathcal{P} \times \mathcal{P}$ that represents a single execution step on programs and values that represent the successful end of program execution. The reduction of a program may be non-terminating due to language constructs that allow iteration or recursive definitions. For a program $P \in \mathcal{P}$ we write $P{\Downarrow}$ if there is a sequence of reductions to a value, and say P *converges* (or terminates successfully) in this case. Then equivalence of programs can be defined by $P_1 \sim P_2 \iff \big(\text{for all } C : C[P_1]{\Downarrow} \iff C[P_2]{\Downarrow}\big)$, where C is a context, i.e. a program with a hole $[\cdot]$ at a single position. Justifying the correctness of a program transformation $P \rightsquigarrow P'$ means to provide a proof that $P \sim P'$. Unfortunately, the quantification is over an infinite set: the set of all contexts, and the criterion is termination, which is undecidable in general. Well-known tools to ease the proofs are context lemmas [10], ciu-lemmas [7] and bisimulation, see e.g. [8].

The reduction relation $\to$ is often given as a set of rules $l_i \to r_i$ similarly to rewriting rules, but extended with different kinds of meta-variables and some other constructs, together with a strategy determining when to use which rule and at which position. In order to prove correctness of a program transformation that is also given in a rule form $s_1 \to s_2$, we have to show that $\sigma(s_1) \sim \sigma(s_2)$ for all possible rule instantiations σ i.e. $C[\sigma(s_1)]{\Downarrow} \iff C[\sigma(s_2)]{\Downarrow}$ for all contexts C. Using the details of the reduction steps and induction on the length of reductions, the hard part is to look for conflicts between instantiations of s_1 and some l_i, i.e. to compute all the overlaps of l_i and s_1, and the possible completions under reduction and transformation. This method is reminiscent of the critical pair criterion of Knuth-Bendix method [9] but has to be adapted to an asymmetric situation, to extended instantiations and to higher-order terms.

In this paper we develop a unification method to compute all overlaps of left hand sides of a set of transformations rules and the reduction rules of the calculus L_{need} which is a call-by-need lambda calculus with a letrec-construct (see [12]). We show that a custom-tailored unification algorithm can be developed that is decidable and produces a complete and finite set of unifiers for the required equations. The following expressiveness is required: *Many-sorted terms* in order to avoid most of the junk solutions; *context variables* which model the context meta-variables in the rule descriptions; *context classes* allow the unification algorithm to treat different kinds of context meta-variables in the rules; the *equational theory of multi-sets* models the letrec-environment of bindings; *Empty sorts* are used to approximate scoping rules of higher-order terms, where, however, only the renaming can be modeled. Since the reduction rules are linear in the meta-variables, we finally only have to check whether the solutions produce expressions that satisfy the distinct variable convention. *Binding Chains* in letrec-expressions are a syntactic extension that models binding sequences of unknown length in the rules. This also permits to finitely represent infinitely many unifiers, and thus is indispensable for effectively computing all solutions.

The required complete sets of diagrams can be computed from the overlaps by applying directed transformations and reduction rules. These can be used to prove correctness of program transformations by inductive methods.

Since our case study is done for a small calculus, the demand for extending the method to other calculi like the extended lambda calculus in [15] would justify further research.

In Section 2 we present the syntax and operational semantics of a small call-by-need lambda calculus with a cyclic let. The normal order reduction rules and transformations are defined. In Section 3, the translation into extended first-order terms is explained. Section 4 contains a description of the unification algorithm that computes overlaps of left hand sides of rules and transformations in a finite representation. Finally, in Section 6, we illustrate a run of the unification algorithm by an example.

2 A Small Extended Lambda Calculus with letrec

In this section we introduce the syntax and semantics of a small call-by-need lambda calculus and use it as a case-study. Based on the definition of the small-step reduction semantics of the calculus we define our central semantic notion of *contextual equivalence* of calculi expressions and correctness of program transformations. We illustrate a method to prove the correctness of program transformations which uses a *context lemma* and *complete sets of reduction diagrams*.

2.1 The Call-by-Need Calculus L_{need}

We define a simple call-by-need lambda calculus L_{need} which is exactly the call-by-need calculus of [12]. Calculi that are related are in [14], and [1].

The set $\mathcal{E}$ of L_{need}-expressions is as follows where x, x_i are variables:

$$s_i, s, t \in \mathcal{E} ::= x \mid (s\ t) \mid (\lambda x.s) \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$$

We assign the names *application*, *abstraction*, or *letrec-expression* to the expressions $(s\ t)$, $(\lambda x.s)$, $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$, respectively. A group of letrec-bindings, also called *environment*, is abbreviated as Env.

We assume that variables x_i in letrec-bindings are all distinct, that letrec-expressions are identified up to reordering of binding-components (i.e. the binding-components can be interchanged), and that, for convenience, there is at least one binding. Letrec-bindings are recursive, i.e., the scope of x_j in $(\texttt{letrec}\ x_1 = s_1, \ldots, x_{n-1} = s_{n-1}\ \texttt{in}\ s_n)$ are all expressions s_i with $1 \leq i \leq n$. Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in t is denoted as $FV(t)$. We use the distinct variable convention (DVC), 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 α-rename bound variables in the result if necessary.

A *context* C is an expression from L_{need} extended by a symbol $[\cdot]$, the *hole*, such that $[\cdot]$ occurs exactly once (as sub-expression) in C. A formal definition is:

Definition 2.1. Contexts $\mathcal{C}$ *are defined by the following grammar:*

$$C \in \mathcal{C} ::= [\cdot] \mid (C\ s) \mid (s\ C) \mid (\lambda x.C) \mid\ (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ C) \mid\ (\texttt{letrec}\ Env, x = C\ \texttt{in}\ s)$$

Given a term t and a context C, we write $C[t]$ for the L_{need}-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. Note that α-renaming of contexts is restricted.

Definition 2.2. *The* unrestricted reduction rules *for the calculus L_{need} are defined in Figure 1. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet), the union of (cp-e) and (cp-in) is called (cp), the union of (llet) and (lapp) is called (lll).*

The reduction rules of L_{need} contain different kinds of meta-variables. The meta-variables r, s, s_x, t denote arbitrary L_{need}-expressions. Env, Env_1, Env_2 represent letrec-environments and x, y denote bound variables. All meta-variables can be instantiated by an L_{need}-expression of the appropriate syntactical form. A reduction rule $\rho = l \to r$ is applicable to an expression e if l can be matched to e. Note that an expression may contain several sub-expressions that can be reduced according to the reduction rules of Figure 1.

A standardizing order of reduction is the *normal order reduction* (see definitions below) where reduction takes place only inside *reduction contexts*.

(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. Unrestricted reduction rules of L_{need} (also used as transformations)

Definition 2.3. Reduction contexts $\mathcal{R}$, application contexts $\mathcal{A}$ *and* surface contexts $\mathcal{S}$ *are defined by the following grammars:*

$$
\begin{array}{l}
A \in \mathcal{A} := [\cdot] \mid (A\ s) \qquad \textit{where } s \textit{ is an expression.} \\
R \in \mathcal{R} := A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \mid \texttt{letrec}\ y_1 = A_1, Env\ \texttt{in}\ A[y_1] \\
\qquad\qquad \mid \texttt{letrec}\ y_1 = A_1, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env\ \texttt{in}\ A[y_n] \\
S \in \mathcal{S}\ := [\cdot] \mid (S\ s) \mid (s\ S) \mid\ (\texttt{letrec}\ y_1 = s_1, \ldots, y_n = s_n\ \texttt{in}\ S) \mid\ (\texttt{letrec}\ Env, y = S\ \texttt{in}\ s)
\end{array}
$$

A sequence of bindings of the form $y_{m+1} = A_{m+1}[y_m], y_{m+2} = A_{m+2}[y_{m+1}], \ldots, y_n = A_n[y_{n-1}]$ where the y_i are distinct variables, the A_i are not the empty context and $m < n$ is called a *binding chain* and abbreviated by $\{y_{i+1} = A_{i+1}[y_i]\}_{i=m}^{n}$.

Definition 2.4. *Normal order reduction* $\xrightarrow{no}$ *(called no-reduction for short) is defined by the reduction rules in Figure 2.*

(lbeta)	$R[(\lambda x.s)\ r] \to R[\texttt{letrec}\ x = r\ \texttt{in}\ s]$
(cp-in)	$\texttt{letrec}\ y = s, Env\ \texttt{in}\ A[y] \to \texttt{letrec}\ y = s, Env\ \texttt{in}\ A[s]$ where s is an abstraction or a variable.
(cp-e)	$\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], Env\ \texttt{in}\ A[y_2] \to \texttt{letrec}\ y_1 = s, y_2 = A_2[s], Env\ \texttt{in}\ A[y_2]$
(cp-e-c)	$\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], \{y_{i+1} = A_{i+1}[y_i]\}_{i=2}^{n}, Env\ \texttt{in}\ A[y_n]$ $\to \texttt{letrec}\ y_1 = s, y_2 = A_2[s], \{y_{i+1} = A_{i+1}[y_i]\}_{i=2}^{n}, Env\ \texttt{in}\ A[y_n]$ in the cp-e rules s is an abstraction or a variable and A_2 is a non-empty context.
(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)	$\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_1\ \texttt{in}\ r), Env_2\ \texttt{in}\ A[y_1] \to \texttt{letrec}\ y_1 = r, Env_1, Env_2\ \texttt{in}\ A[y_1]$
(llet-e-c)	$\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_1\ \texttt{in}\ r), \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env_2\ \texttt{in}\ A[y_n]$ $\to \texttt{letrec}\ y_1 = r, Env_1, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env_2\ \texttt{in}\ A[y_n]$
(lapp)	$R[((\texttt{letrec}\ Env\ \texttt{in}\ r)\ t)] \to R[(\texttt{letrec}\ Env\ \texttt{in}\ (r\ t))]$

Fig. 2. Normal order reduction rules of L_{need}

Note that the normal order reduction is unique. A *weak head normal form in* L_{need} *(WHNF)* is defined as either an abstraction $\lambda x.s$, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ \lambda x.s)$.

The *transitive closure* of the reduction relation $\to$ is denoted as $\xrightarrow{+}$ and the *transitive and reflexive closure* of $\to$ is denoted as $\xrightarrow{*}$. Respectively we use $\xrightarrow{no,+}$ for the transitive closure of the normal order

reduction relation, $\xrightarrow{no,*}$ for its reflexive-transitive closure, and $\xrightarrow{no,k}$ to indicate k normal order reduction steps. If for an expression t there exists a (finite) sequence of normal order reductions $t \xrightarrow{no,*} t'$ to a WHNF t', we say that the reduction *converges* and denote this as $t \Downarrow t'$ or as $t\Downarrow$ if t' is not important. Otherwise the reduction is called *divergent* and we write $t\Uparrow$.

The semantic foundation of our calculus L_{need} is the equality of expressions defined by contextual equivalence.

Definition 2.5 (Contextual Preorder and Equivalence). *Let s, t be L_{need}-expressions. Then:*

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

Definition 2.6. *A* program transformation *$T \subseteq L_{need} \times L_{need}$ is a binary relation on L_{need}-expressions. A program transformation is called* correct *iff $T \subseteq \sim_c$.*

Program transformations are usually given in a format similarly to reduction rules (as in Figure 1 and Figure 2). A program transformation T is written as $s \xrightarrow{T} t$ where s, t are meta-expressions i.e. expression that contain meta-variables. Here we restrict our attention for the sake of simplicity to the program transformations that are given by the reduction rules in Figure 1.

Proving that a program transformation $s \xrightarrow{T} t$ is not correct is often an easy task: It is sufficient to give a context C that distinguishes the termination behavior of s and t.

Example 2.7. A simple example for an incorrect transformation in L_{need} is η-reduction: $(\lambda x.(s\ x)) \xrightarrow{\eta} s$ where x is not a free variable in s and s is a L_{need}-expression. We define an expression Ω which is divergent as $\Omega := (\lambda y.(y\ y))\ (\lambda z.(z\ z))$ and we instantiate s in the η-reduction as Ω. Then we have $(\lambda x.(\Omega\ x)) \xrightarrow{\eta} \Omega$ where the two expressions are distinguished by the empty context $[\cdot]$, because the expression $(\lambda x.(\Omega\ x))$ is a WHNF, which is convergent by definition and Ω is a divergent expression.

An important tool to prove contextual equivalence is a *context lemma* (see for example [10], [13],[15]), which allows to restrict the class of contexts that have to be considered in the definition of the contextual equivalence from general $\mathcal{C}$ to $\mathcal{R}$ contexts.

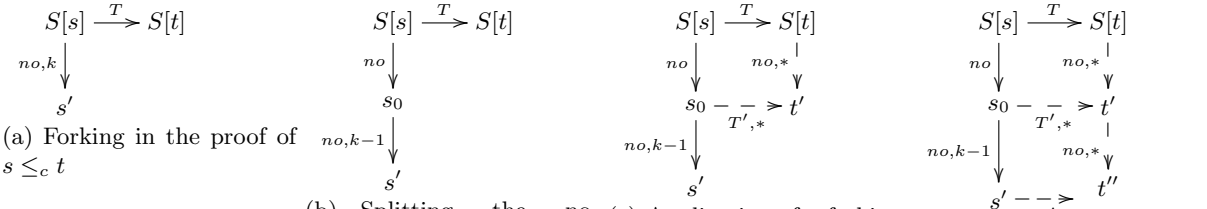
However, often $\mathcal{S}$-contexts are more appropriate for computing overlaps and closing the diagrams, so we will use $\mathcal{S}$-contexts instead of $\mathcal{R}$-contexts.

Lemma 2.8. *Let s, t be L_{need}-expressions and S a context of class $\mathcal{S}$. $(S[s]\Downarrow \Rightarrow S[t]\Downarrow)$ iff $\forall C: (C[s]\Downarrow \Rightarrow C[t]\Downarrow)$; i.e. $s \leq_c t$.*

Proof. A proof of this lemma when the contexts are in class $\mathcal{R}$ is in [13]. Since every $\mathcal{R}$-context is also an $\mathcal{S}$-context, the lemma holds.

To prove the correctness of a transformation $s \xrightarrow{T} t$ we have to prove that $s \sim_c t \Leftrightarrow s \leq_c t \ \wedge \ t \leq_c s$ which by Definition 2.5 amounts to showing $\forall C: C[s]\Downarrow \Rightarrow C[t]\Downarrow \ \wedge \ C[t]\Downarrow \Rightarrow C[s]\Downarrow$. The context lemma yields that it is sufficient to show $\forall S: S[s]\Downarrow \Rightarrow S[t]\Downarrow \ \wedge \ S[t]\Downarrow \Rightarrow S[s]\Downarrow$. We restrict our attention here to $S[s]\Downarrow \Rightarrow S[t]\Downarrow$ because $S[t]\Downarrow \Rightarrow S[s]\Downarrow$ could be treated in a similar way. To prove $s \sim_c t$ we assume that $s \xrightarrow{T} t$ and $S[s]\Downarrow$ holds, i.e. there is a WHNF s', such that $S[s] \xrightarrow{no,k} s'$ (see Figure 3(a)). It remains to show that there also exists a sequence of normal order reductions from $S[t]$ to a WHNF. This can often be done by induction on the length k of the given normal order reduction $S[s] \xrightarrow{no,k} s'$ using *complete sets of reduction diagrams*. Therefore we split $S[s] \xrightarrow{no,k} s'$ into $S[s] \xrightarrow{no} s_o \xrightarrow{no,k-1} s'$ (see Figure 3(b)).

Then an applicable *forking diagram* defines how the fork $s_0 \xleftarrow{no} S[s] \xrightarrow{T} S[t]$ can be closed specifying two sequences of transformations such that a common expression t' is eventually reached: one starting from $S[t]$ consisting only of no-reductions and one starting from s_0 consisting of some other reductions (that are not normal order) denoted by T' in Figure 3(c).



(a) Forking in the proof of $s \leq_c t$

(b) Splitting the no-sequence

(c) Application of a forking diagram

(d) Inductive proof of $s \leq_c t$

Fig. 3. Sketch of the correctness proof for $s \xrightarrow{T} t$

A set of forking diagrams for a transformation T is *complete* if the set comprises an applicable diagram for every forking situation. If we have a complete set of forking diagrams we often can inductively construct a terminating reduction sequence for $S[t]$ if $S[s]{\Downarrow}$ (as indicated in Figure 3(d)). To prove $S[t]{\Downarrow} \Rightarrow S[s]{\Downarrow}$ another complete set of diagrams called *commuting diagrams* is required which usually can be deduced from a set of forking diagrams (see [15]). We restrict our attention to complete sets of forking diagrams.

2.2 Complete Sets of Forking and Commuting Diagrams

Reduction diagrams describe transformations on reduction sequences. They are used to prove the correctness of program transformations.

Non-normal order reduction steps for the language L_{need} are called *internal* and denoted by a label i. An internal reduction in a reduction context is marked by $i\mathcal{R}$, and an internal reduction in a surface context by $i\mathcal{S}$.

A *reduction sequence* is of the form $t_1 \to \ldots \to t_n$, where t_i are L_{need}-expressions and $t_i \to t_{i+1}$ is a reduction as defined in Definition 2.2. In the following definition we describe transformations on reduction sequences. Therefore we use the notation

$$\xrightarrow{iX,T} . \xrightarrow{no,a_1} \ldots \xrightarrow{no,a_k} \quad \rightsquigarrow \quad \xrightarrow{no,b_1} \ldots \xrightarrow{no,b_m} . \xrightarrow{iX,T_1} \ldots \xrightarrow{iX,T_h}$$

for transformations on reduction sequences. Here the notation $\xrightarrow{iX,T}$ means a reduction with $iX \in \{i\mathcal{C}, i\mathcal{R}, i\mathcal{S}\}$, and T is a reduction from L_{need}.

In order for the above transformation rule to be applied to the prefix of the reduction sequence RED, the prefix has to be $s \xrightarrow{iX,T} t_1 \xrightarrow{no,a_1} \ldots t_k \xrightarrow{no,a_k} t$. Since we will use sets of transformation rules, it may be the case that there is a transformation rule in the set, where the pattern matches a prefix, but it is not applicable, since the right hand side cannot be constructed.

We will say the transformation rule

$$\xrightarrow{iX,T} . \xrightarrow{no,a_1} \ldots \xrightarrow{no,a_k} \quad \rightsquigarrow \quad \xrightarrow{no,b_1} \ldots \xrightarrow{no,b_m} . \xrightarrow{iX,T_1} \ldots \xrightarrow{iX,T_h}$$

is *applicable* to the prefix $s \xrightarrow{iX,T} t_1 \xrightarrow{no,a_1} \dots t_k \xrightarrow{no,a_k,} t$ of the reduction sequence RED iff the following holds:

$$\exists y_1, \dots, y_m, z_1, \dots, z_{h-1} : s \xrightarrow{no,b_1} y_1 \dots \xrightarrow{no,b_m} y_m \xrightarrow{iX,T_1} z_1 \dots z_{h-1} \xrightarrow{iX,T_h} t$$

The transformation consists in replacing this prefix with the result:

$$s \xrightarrow{no,b_1} t'_1 \dots t'_{m-1} \xrightarrow{no,b_m} t'_m \xrightarrow{iX,T_1} t''_1 \dots t''_{h-1} \xrightarrow{iX,T_h} t$$

where the terms in between are appropriately constructed.

Definition 2.9.
• *A* complete set of forking diagrams for the reduction $\xrightarrow{iX,T}$ *is a set of transformation rules on reduction sequences of the form*

$$\xleftarrow{no,a_1} \dots \xleftarrow{no,a_k} \cdot \xrightarrow{iX,T} \quad \rightsquigarrow \quad \xrightarrow{iX,T_1} \dots \xrightarrow{iX,T_{k'}} \cdot \xleftarrow{no,b_1} \dots \xleftarrow{no,b_m},$$

where $k, k' \geq 0, m \geq 1, h > 1$*, such that for every reduction sequence* $t_h \xleftarrow{no} \dots t_2 \xleftarrow{no} t_1 \xrightarrow{iX,T} t_0$*, where* t_h *is a WHNF, at least one of the transformation rules from the set is applicable to a suffix of the sequence.*

The case $h = 1$ *must be treated separately in the induction base.*

• *A* complete set of commuting diagrams for the reduction $\xrightarrow{iX,T}$ *is a set of transformation rules on reduction sequences of the form*

$$\xrightarrow{iX,red} \cdot \xrightarrow{no,a_1} \dots \xrightarrow{no,a_k} \quad \rightsquigarrow \quad \xrightarrow{no,b_1} \dots \xrightarrow{no,b_m} \cdot \xrightarrow{iX,red_1} \dots \xrightarrow{iX,red_{k'}},$$

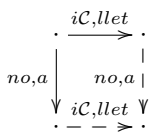
where $k, k' \geq 0, m \geq 1, h > 1$*, such that for every reduction sequence* $t_0 \xrightarrow{iX,T} t_1 \xrightarrow{no} \dots \xrightarrow{no} t_h$*, where* t_h *is a WHNF, at least one of the transformation rules is applicable to a prefix of the sequence.*

In the proofs below using the complete sets of commuting diagrams, the case $h = 1$ *must be treated separately in the induction base.*

The two different kinds of diagrams are required for two different parts of the proof of the contextual equivalence of two terms.

In most of the cases, the same diagrams can be drawn for a complete set of commuting and a complete set of forking diagrams, though the interpretation is different for the two kinds of diagrams. The starting term is in the northwestern corner, and the normal order reduction sequences are always downwards. where the deviating reduction is pointing to the east. There are rare exceptions for degenerate diagrams, which are self explaining.

For example, the forking diagram $\xleftarrow{no,a} \cdot \xrightarrow{iC,llet} \quad \rightsquigarrow \quad \xrightarrow{iC,llet} \cdot \xleftarrow{no,a}$ is represented as

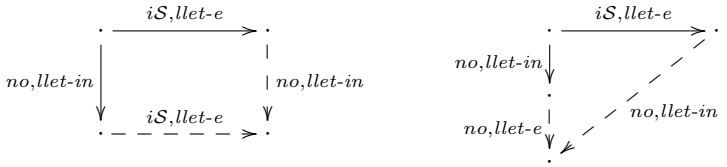


The solid arrows represent given reductions and dashed arrows represent existential reductions. A common representation is without the dashed arrows, where the interpretation depends on whether the

diagram is interpreted as a forking or a commuting diagram. We may also use the * and +-notation of regular expressions for the diagrams. The interpretation is obvious and is intended to stand for an infinite set accordingly constructed.

Note that the selection of the reduction label is considered to occur outside the transformation rule, i.e. if $\xrightarrow{no,a}$ occurs on both sides of the transformation rule the label a is considered to be the same on both sides.

Example 2.10. Example forking diagrams are



where the dashed lines indicate existentially quantified reductions and the prefix $i\mathcal{S}$ marks that the transformation is not a normal order reduction (but a so called *internal reduction* which we also call transformation), and occurs within a surface context. By application of the diagram a fork between a *(no,llet-e)* and the transformation *(llet-in)* can be closed. The forking diagrams specify two reduction sequences such that a common expression is eventually reached. The following reduction sequence illustrates an application of the above diagram:

$$
\begin{array}{l}
(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r)) \\
\hline
\quad \xrightarrow{no,llet\text{-}in} (\texttt{letrec}\ Env_1, Env_3, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ r) \\
\xrightarrow{i\mathcal{S} \vee no,llet\text{-}e} (\texttt{letrec}\ Env_1, Env_3, Env_2, x = s\ \texttt{in}\ r) \\
\quad \text{the last reduction is either an no-reduction if } r = A[x], \text{ otherwise it is an internal reduction} \\
\hline
\quad \xrightarrow{i\mathcal{S},llet\text{-}e} (\texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r)) \\
\quad \xrightarrow{no,llet\text{-}in} (\texttt{letrec}\ Env_1, Env_2, Env_3, x = s\ \texttt{in}\ r)
\end{array}
$$

The square diagram covers the case, where *(no,llet-in)* is followed by an internal reduction. The triangle diagram covers the other case, where the reduction following *(no,llet-in)* is *(no,llet-e)*. One can view the forking diagram as a description of local confluence.

The computation of a complete set of diagrams by hand is cumbersome and error-prone. Nevertheless the diagram sets are essential for proving correctness of a large set of program transformations in this setting. For this reason we are interested in automatic computation of complete diagram sets.

The first step in the computation of a complete set of forking diagrams for a transformation T is the determination of all forks of the form $\xleftarrow{no,red} \cdot \xrightarrow{i\mathcal{S},T}$ where *red* is an no-reduction and T is not a normal order reduction (but a transformation in an $\mathcal{S}$-context). Such forks are given by *overlaps* between no-reductions and the transformation. Informally we say that *red* and T overlap in an expression s if s contains a normal order redex *red* and a T redex (in a surface context). To find an overlap between an no-reduction *red* and a transformation T it is sufficient, by definition of the normal order reduction, to determine all surface-positions in *red* where a T-redex can occur. For the computation of all forks we have to consider only *critical overlaps* where an overlap does not occur at a variable position (Example 2.10 illustrates such a critical overlap). Forks stemming from non-critical overlaps at variable positions can always be closed by a predefined set of standard diagrams. All (critical) overlaps between no-reductions and a given transformation T can be computed by a variant of critical pair computation based on unification. The employed unification procedure will be explained in the next section.

3 Encoding Expressions as Terms in a Combination of Sorted Equational Theories and Context

In this section we develop a unification method to compute proper overlaps for forking diagrams. According to the context lemma for surface contexts (Lemma 2.8) we restrict the overlaps to the transformations applied in surface contexts. A complete description of a single overlap is the unification equation $S[l_{T,i}] \doteq l_{no,j}$, where $l_{T,i}$ is a left hand side in Figure 1, and $l_{no,j}$ a left hand side in Figure 2, and S means a surface context. To solve these unification problems we translate the meta-expressions from transformations and no-reduction rules into many sorted terms with some special constructs to mirror the syntax of the reduction rules in the lambda calculus. The constructs are i) context variables of different context classes $\mathcal{A}, \mathcal{S}$ and $\mathcal{C}$, ii) a left-commutative function symbol *env* to model that bindings in letrec-environments can be rearranged iii) a special construct `BCh`$(\ldots)$ to represent binding chains of variable length as they occur in no-reduction rules.

The presented unification algorithm is applicable to terms with the mentioned extra constructs. We do not use the general unification combination algorithms in [11,2], since we only have a special theory LC that models multi-sets of bindings in letrec-environments of our calculus, and moreover, it is not clear how to adapt the general combination method to context classes and binding chains.

3.1 Many Sorted Signatures, Terms and Contexts

Let $\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2$ be the disjoint union of a set of theory-sorts $\mathcal{S}_1$ and a set of free sorts $\mathcal{S}_2$. We assume that *Exp* is a sort in $\mathcal{S}_2$. Let $\Sigma = \Sigma_1 \uplus \Sigma_2$ be a many-sorted signature of (theory- and free) function symbols, where every function symbol comes with a fixed arity and with a single sort-arity of the form $f : S_1 \times \ldots \times S_n \to S_{n+1}$, where S_i for $i = 1, \ldots, n$ are the argument-sorts and S_{n+1} is called resulting sort. For every $f \in \Sigma_i$ for $i = 1, 2$ the resulting sort must be in $\mathcal{S}_i$. Note, however, that there may be function symbols $f \in \Sigma_i$ that have argument-sorts from $\mathcal{S}_j$, for $i \neq j$. There is a set $\mathcal{V}^0$ of first-order variables that are 0-ary and have a fixed sort and are ranged over by $x, y, z, \ldots$, perhaps with indices. We write x^S if the variable x has the sort S. There is also a set $\mathcal{V}^1$ of context-variables which are unary and are ranged over by X, Y, Z, perhaps with indices. We assume that for every sort S, there is an infinite number of variables of this sort, and that there is an infinite number of context variables of sort $Exp \to Exp$. Let $\mathcal{V} = \mathcal{V}^0 \cup \mathcal{V}^1$. The set of terms $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V})$ is the set of terms built according to the grammar $x \mid f(t_1, \ldots, t_n) \mid X(t)$, where sort conditions are obeyed. Let $Var(t)$ be the set of first-order variables that occur in t and let $Var^1(t)$ be the set of context variables that occur in t. A context C is a term in $\mathcal{T}(Exp, \Sigma \cup [\cdot], \mathcal{V})$ such that there is exactly one occurrence of a the special hole constant $[\cdot]$ in the context and the sort at the position of the hole is *Exp*.

A term s without occurrences of variables is called *ground*. We also allow sorts without any ground term, also called *empty sorts*, since this is required in our encoding of bound variables. The term s is called *almost ground*, if for every variable x in s, there is no function symbol in Σ where the resulting sort is the sort of x, and hence no ground term of this sort.

A substitution σ is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$, such that $\sigma(x^S)$ is a term of sort S and $\sigma(X)$ is a context. As usual we extend σ to terms, where every variable x in a term is replaced by $\sigma(x)$.

3.2 Encoding of L_{need}-Expressions as Terms

The sort and term structure according to the expression structure of the lambda calculus L_{need} (from section 2.1) is as follows. There are the following sorts: $Bind, Env, Exp, BV$, for bindings, environments,

expressions and bound variables, respectively; where $\mathcal{S}_1 = \{Env\}$ and $\mathcal{S}_2 = \{Bind, Exp, BV\}$. There are the following function symbols:

theory function symbols (Σ_1)		free function symbols (Σ_2)	
$emptyEnv ::$	Env	$let :: Env \times Exp \to Exp$	$bind :: BV \times Exp \to Bind$
$env ::$	$Bind \times Env \to Env$	$app :: Exp \times Exp \to Exp$	$var :: BV \to Exp$
		$lam :: BV \times Exp \to Exp$	

Note that there are free function symbols that map from Env to Exp, but there is no free function symbol that maps to Env. Note also that there is no function symbol with resulting sort BV, hence this is an empty sort, and every term of sort BV is a variable.

It is convenient to have a notation for nested env-expressions: $env^*(\{t_1, \ldots, t_m\} \cup r)$ denotes the term $env(t_1, env(t_2, \ldots, env(t_m, r)\ldots))$, where r is not of the form $env(s, t)$. Due to our assumptions on terms of sort Env and the sort of context variables, only the constant $emptyEnv$ and variables are possible for r.

As an example the expression (`letrec` $x = \lambda y.y, z = x\ x$ `in` z) is encoded as $let(env^*(\{bind(x, lam(y, var(y))), bind(z, app(var(x), var(z)))\} \cup emptyEnv), var(z))$, where x, y, z are variables of sort BV.

To model the multi-set property of letrec-environments, i.e., that bindings can be reordered, we use the equational theory *left-commutativity* (LC) with the following axiom: $env(x, env(y, z)) = env(y, env(x, z))$ (for the LC-theory and unification modulo LC see [6,5,4]). The equational theory LC is a congruence relation on the terms, which is denoted as $=_{LC}$. The *pure equational theory* is defined as restricted to the axiom-signature, i.e. to the terms $\mathcal{T}(\{Env, Bind\}, \Sigma_1, \mathcal{V}_{Env} \cup \mathcal{V}_{Bind})$, where $\mathcal{V}_S$ is the set of variables of sort S. The *combined equational theory* is defined on the set of terms $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$. Note that it is a disjoint combination w.r.t. the function symbols, but not w.r.t. the sorts.

The following facts about the theory LC can easily be verified:

Lemma 3.1. *For the equation theory LC, the following holds in $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$:*

- *The terms in the LC-axioms are built only from Σ_1-symbols and variables, and the axioms relate two terms of equal sort which must be in $\mathcal{S}_1$.*
- *For every equation $s =_{LC} t$, the equality $Var(s) = Var(t)$ holds.*
- *The equational theory LC is non-collapsing, i.e, there is no equation of the form $x =_{LC} t$, where t is not the variable x.*
- *If $C[s] =_{LC} t$ and s has a free function symbol as top symbol, then there is a context C' and a term s' such that $C[s] =_{LC} C'[s'], C' =_{LC} C$, $s =_{LC} s'$ and $C'[s'] = t$. This follows from general properties of combination of equational theories and properties of the theory LC.*
- *The equational theory LC has a finitary and decidable unification problem (see[6,5,4]).*

In order to capture binding chains of variable length as they occur in the definition of the no-reduction rules (Figure 2) the syntax construct $\texttt{BCh}(N_1, N_2)$ is introduced, where N_i are integer variables that can be instantiated with $N_1 \mapsto n_1$, $N_2 \mapsto n_2$, where $0 < n_1 < n_2$. An instance $\texttt{BCh}(n_1, n_2)$ for $n_1, n_2 \geq 1$ represents the following binding chain: $bind(y_{n_1+1}, A_{n_1+1}(var(y_{n_1})))$, $bind(y_{n_1+2}, A_{n_1+2}(var(y_{n_1+1})))$, $\ldots$, $bind(y_{n_2}, A_{n_2}(var(y_{n_2-1})))$, where the names y_i, A_i are reserved for these purposes and are all distinct. The `BCh`-expressions are permitted only in the env^*-notation, like a sub-multi-set, and we denote this for example as $env^*(\ldots \cup \texttt{BCh}(N_1, N_2) \cup r)$.

Context-classes are required to correctly model the overlappings in L_{need}. The transformations in Figure 1 contain only C-contexts, whereas in Figure 2 there are also $\mathcal{A}$- and $\mathcal{R}$-contexts, and the overlapping also requires surface contexts $\mathcal{S}$. The grammar definition of $\mathcal{A}$-, $\mathcal{R}$- and $\mathcal{S}$-contexts (definition 2.3)

justifies the replacement of $\mathcal{R}$-contexts by expressions containing only $\mathcal{A}$-contexts and BCh-expressions. Thereby some rules of Figure 2 may be split into several rules. The context class $\mathcal{C}$ means all contexts and $\mathcal{S}$ means all contexts where the hole is not in an abstraction. In the term encoding, these translate to context variables. The unification algorithm must know how to deal with context variables of classes $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$. The partial order on context classes is $\mathcal{A} < \mathcal{S} < \mathcal{C}$. For every almost ground context C it can be decided whether C belongs to $\mathcal{A}$ (or $\mathcal{S}$, respectively). We will use the facts that equational deduction w.r.t. LC does not change the context class of almost ground contexts, and that prefix and suffix contexts of almost ground contexts C have the same context class as C (among $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$).

4 A Unification Algorithm LCSX for Left-Commutativity, Sorts and Context-Variables

We define unification problems and solutions as extension of equational unification (see [3]).

A *unification problem* is a pair (Γ, Δ), where $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, the terms s_i and t_i are of the same sort for every i and may also contain BCh-expressions, every context variable is labelled with a context class symbol, and $\Delta = (\Delta_1, \Delta_2)$ is a constraint consisting of a set of context variables Δ_1 and a set Δ_2 of equations and inequations of the form $N_i + 1 = N_j$ and $N_i < N_j$ for the integer variables N_i. The intention is that Δ_1 consists of context variables that must not be instantiated by the empty context, and that the constraints Δ_2 hold for $\sigma(N_i)$ after instantiating with σ.

A *solution* σ of (Γ, Δ), with $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a substitution σ according to the following conditions: i) it instantiates variables by terms, context variables by contexts of the correct context class that are nontrivial if contained in Δ_1, and the integer variables N_i by positive integers according to the constraint Δ_2. ii) $\sigma(s_i), \sigma(t_i)$ are almost ground for all i. It is assumed that the BCh-constructs $\texttt{BCh}(n_1, n_2)$ are expanded into a binding chain as explained above, iii) $\sigma(s_i) =_{LC} \sigma(t_i)$ for all i.

A unification problem Γ is called *almost linear*, if every context variable occurs at most once and every variable of a non-empty sort occurs at most once in the equations.

Definition 4.1. *Let Π_T be the set of left hand sides of reduction rules from Figure 1 and Π_{no} the set of left hand sides of no-reduction rules from Figure 2 where the reduction contexts R in (lbeta) and (lapp) are instantiated by the four possibilities for R: A, (*letrec *Env in A), (*letrec *$y_1 = A, Env$ in $A_2[y_1]$), (*letrec *$y_{N_1} = A, \texttt{BCh}(N_1, N_2), Env$ in $A[y_{N_2}]$) with constraint $N_1 < N_2$. The meta-variable s in the cp rules (that can be either a variable or an abstraction) is instantiated by $var(z)$ and an abstraction $\lambda x.t$ where t denotes a meta-variable for an arbitrary expression. With Π'_T, Π'_{no} we denote the sets where left hand sides of rules are encoded as terms.*

We consider the set of unification problems $\Gamma_i = \{S(l_{T,i}) \doteq l_{no,j} \mid l_{no,j} \in \Pi'_{no}\}$ with $l_{T,i} \in \Pi'_T$ and S is a surface context variable. The sets Π'_T and Π'_{no} are assumed to be variable disjoint, which can be achieved by renaming. The initial set Δ_1 of context variables only contains the A_2-context from the (cp-e)-reductions, and Δ_2 may contain some initial constraints from the rules. The pairs (Γ_i, Δ) are called the initial L_{need}-forking-problems*.*

Note that initial L_{need}-forking-problems are almost linear, there is at most one BCh-construct, which is in the environment of the topmost let-expression, and there are no variables of type *Bind*.

Definition 4.2. *A* final unification problem *S of an initial Γ is a set of equations $s_1 \doteq t_1, \ldots, s_n \doteq t_n$, such that $S = S_{BV} \cup S_{\neg BV}$, and every equation in S_{BV} is of the form $x \doteq y$ where x, y are of sort BV and every equation in $S_{\neg BV}$ is of the form $x \doteq t$, where x is not of sort BV, and the equations in $S_{\neg BV}$ are in DAG-solved form.*

Given a final unification problem S, the represented solutions σ could be derived by first expanding the `BCh`-constructs into binding chains accorind to the constraints in Δ_2, then turning the equations into substitutions, instantiating the integer variables and then instantiating all context variables and variables that are not of sort BV. Note that there may be infinitely many represented solutions for a single final unification problem.

A final unification problem S derived from Γ *satisfies the distinct variable convention* (DVC), if for every derived solution σ, all terms in $\sigma(\Gamma)$ satisfy the DVC. This property is decidable: If $t_1 \doteq t_2$ is the initial problem, then apply the substitution σ derived from S to t_1. The DVC is violated if the following condition holds: Let M_{BV} be the set of BV-variables occurring in $\sigma(t_1)$. For every `BCh`-construct $\texttt{BCh}(N_1, N_2)$ occurring in $\sigma(t_1)$ we add the variable y_{N_2} to M_{BV}. If $\sigma(t_1)$ makes two variables in M_{BV} equal, then the DVC is violated, and the corresponding final problem is discarded.

Example 4.3. Unifying (the first-order encodings of) $\lambda x.\lambda y.x$ and $\lambda u.\lambda v.v$, the unification succeeds and generates an instance that represents $\lambda x.\lambda x.x$, which does not satisfy the DVC. After the addition of a DVC-check, our unification can efficiently check alpha-equivalence of pure lambda-expressions that satisfy the DVC.

We proceed by describing a unification algorithm starting with initial L_{need}-unification problems (Γ, Δ). It is intended to be complete for all common instances that represent L_{need}-expressions that satisfy the DVC, i.e. where all bound variables are distinct and the bound variables are distinct from free variables. Final unification problems that lead to expressions that do not satisfy the DVC are discarded.

Given an initial unification problem $\Gamma = \{s_1 \doteq t_1\}; \Delta$, the (non-deterministic) unification algorithm described below will non-deterministically compute a final unification problem S or fail. A finite complete set of final unification problems can be attained by gathering all final unification problems in the whole tree of all non-deterministic choices. We implicitly use symmetry of $\doteq$ if not stated otherwise. We divide Γ in a solved part S, (a final unification problem), and a still to be solved part P. We usually omit Δ in the notation if it is not changed by the rule.

Standard unification rules.

Dec $\dfrac{S;\ \{f(s_1,\ldots,s_n) \doteq f(t_1,\ldots,t_n)\} \uplus P}{S;\ \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \cup P}$ If f is a free function symbol (i.e. $f \neq env$).

Solve $\dfrac{S;\ \{x \doteq t\} \uplus P}{\{x \doteq t\} \cup S;\ P}$ **Trivial** $\dfrac{S;\ \{s \doteq s\} \uplus P}{S;\ P}$

Fail $\dfrac{S;\ \{f(\ldots) \doteq g(\ldots)\} \uplus P \text{ and } f \neq g}{\mathit{Fail}}$ **DVC-Fail** $\dfrac{S;\ \emptyset}{\mathit{Fail}}$ If S is final and the DVC is violated w.r.t. the initial problem.

Note that the occurs-check is not necessary, since P is almost linear and an equation $x \doteq t$ for variables x of sort BV implies that t is a variable.

Solving equations with context variables. The following rule operates on context variables at any position:

Empty-C $\dfrac{S;\ P;\ \Delta_1}{\begin{array}{l}\text{select one of the following possibilities}\\ S;\ P;\ \{X\} \cup \Delta_1 \quad \text{or} \quad \{X \mapsto [\cdot]\} \cup S;\ \{X \mapsto [\cdot]\}P;\ \Delta_1\end{array}}$ If X occurs in P and $X \notin \Delta_1$.

Assume there is an equation $X(s) \doteq t$, where the top symbol of t is not a context variable and $X \in \Delta_1$. Note that the sort of $X(s)$ is *Exp*. There are the following possibilities:

Dec-C-App $\dfrac{S;\ \{X(s) \doteq app(t_1, t_2)\} \uplus P}{\{X \mapsto app(X', t_2)\} \cup S;\ \{X'(s) \doteq t_1\} \cup P}$

X' is a fresh context variable of the same context class as X.

Dec-C $\dfrac{S;\ \{X(s) \doteq f(t_1, t_2)\} \uplus P}{\{X \mapsto f(t_1, X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P}$ if t_2 is of sort *Exp*.

X' is a fresh context variable of the same context class as X (it may only be $\mathcal{C}$ or $\mathcal{S}$) and f is a function symbol such that $f \in \{let, app\}$.

Dec-C-Let $\dfrac{S;\ \{X(s) \doteq let(t_1, t_2)\} \uplus P}{\{X \mapsto let(env^*(\{bind(x, X')\} \cup z), t_2)\} \cup S;\ \{env^*(\{bind(x, X'(s))\} \cup z) \doteq t_1\} \cup P}$

If X is of context class $\mathcal{S}$ or $\mathcal{C}$. X' is a fresh context variable of the same context class as X.

Dec-C-Lam $\dfrac{S;\ \{X(s) \doteq lam(t_1, t_2)\} \uplus P}{\{X \mapsto lam(t_1, X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P}$

If X is of class $\mathcal{C}$. X' is a fresh context variable of the class $\mathcal{C}$.

Fail-C-Lam $\dfrac{S;\ \{X(s) \doteq lam(t_1, t_2)\} \uplus P}{Fail}$ **Fail-C-Var** $\dfrac{S;\ \{X(s) \doteq var(x)\} \uplus P}{Fail}$

If X is of class $\mathcal{A}$ or $\mathcal{S}$.

Given an equation $X(s) \doteq Y(t)$, with $X, Y \in \Delta_1$, let $\mathcal{D}$ be the smaller one of the context classes of X, Y. Then select one of the following possibilities:

Merge-Prefix $\dfrac{S;\ \{X(s) \doteq Y(t)\} \uplus P}{\{Y \mapsto ZY', X \mapsto Z\} \cup S;\ \{s \doteq Y'(t)\} \cup P}$

Y' is a fresh context variable of the same context class as Y, and Z has context class $\mathcal{D}$.

Merge-Fork-A $\dfrac{S;\ \{X(s) \doteq Y(t)\} \uplus P}{\{X \mapsto Z(app(X', Y'(t)), Y \mapsto Z(app(X'(s), Y'))\} \cup S;\ P}$

If exactly one of the context classes of X, Y is $\mathcal{A}$. W.l.o.g. let X be of context class $\mathcal{A}$. X', Y' are fresh context variables of the same context class as X, Y, respectively, and Z is a fresh context variable of context class $\mathcal{A}$.

Merge-Fork-C

$$\frac{S;\ \{X(s) \doteq Y(t)\} \uplus P}{\begin{array}{l}\text{choose either of the following possibilities}\\ \{X \mapsto Z(app(X', Y'(t))), Y \mapsto Z(app(X'(s), Y'))\} \cup S;\ P\\ \{X \mapsto Z(let(env^*(\{bind(x, X')\} \cup, z), Y'(t))), Y \mapsto Z(let(env^*(\{bind(x, X'(s))\} \cup z), Y'))\} \cup S;\ P\\ \{X \mapsto Z(let(env^*(\{bind(x, X'), bind(y, Y'(t))\} \cup z), w)),\\ \ Y \mapsto Z(let(env^*(\{bind(x, X'(s)), bind(y, Y')\} \cup z), w))\} \cup S;\ P\end{array}}$$

If the context classes of X, Y are different from $\mathcal{A}$. X', Y' are fresh context variables of the same context class as X, Y, respectively and Z is a fresh context variable of context class $\mathcal{D}$. The variables w, x, y, z are also fresh and of the appropriate sort.

Rules for Multi-Set Equations. The following additional (non-deterministic) unification rules are sufficient to solve nontrivial equations of type *Env*, i.e. proper multi-set-equations, which must be of the form $env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)$, where r_1, r_2 are variables or the constant *emptyEnv*. We will use the notation L for sub-lists in env^*-expressions and the notation $L_1 \cup L_2$ for union. In the terms $env^*(L \cup t)$,

we assume that t is not of the form $env(\ldots)$. It is also not of the form $X(\ldots)$ due to the sort assumptions. Other free function symbols are disallowed, hence t can only be a variable or the constant *emptyEnv*. The components in the multi-set may be expressions of type *Bind*, i.e., variables or expressions with top symbol *bind*, or a $\texttt{BCh}(\ldots)$-component that represents several terms of type *Bind*. We also use the convention that in the conclusions of the rules an empty environment $env^*(\{\,\} \cup r)$ without any bindings and just a variable r is identified with r. Note that the lists allow multi-set operations like reorderings.

Due to the initial encoding of reduction rules, if a $\texttt{BCh}(N_1, N_2)$-construct occurs in a term in P, it occurs in an env^*-list, hence there is also a binding $y_{N_1} = s$ in the env^*-list, and the list is terminated with a variable derived from the environment-variable *Env*. In equations, the $\texttt{BCh}(\ldots)$-components initially appear only on one side, which cannot be changed by the unification. Also the env^*-list is an immediate sub-term of a top let-expression, which may change after applying unification rules. Due to these conditions, we assume that the left term in the equation does not contain $\texttt{BCh}(\ldots)$-components.

If there is an equation $env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)$, then select one of the following possibilities:

Solve-Env $\dfrac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)\} \uplus P}{\{r_1 \mapsto env^*(L_2 \cup z_3), r_2 \mapsto env^*(L_1 \cup z_3)\} \cup S;\ P}$ If r_1, r_2 are variables z_3 is a fresh variable.

Dec-Env If L_1 and L_2 contain binding expressions, then select a binding expression t_1 from L_1 and a binding expression t_2 from L_2.

$$\frac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(L_2 \cup r_2)\} \uplus P}{S;\ \{t_1 \doteq t_2, env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(L_2 \setminus \{t_2\} \cup r_2)\} \uplus P}$$

Dec-Chain

$$\frac{S;\ \{env^*(L_1 \cup r_1) \doteq env^*(\texttt{BCh}(N_1, N_2) \cup L_2 \cup r_2)\} \uplus P;\ (\Delta_1, \{N_1 < N_2\} \cup \Delta_2)}{\text{select one of the following possibilities}}$$

(i) $S;\ \{t_1 \doteq bind(y_{N_2}, A_{N_2}(var(y_{N_1}))),$
$env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(L_2 \cup r_2)\} \cup P;\ \{A_{N_2}\} \cup \Delta_1, \{N_1 + 1 = N_2\} \cup \Delta_2$
(ii) $S;\ \{t_1 \doteq bind(y_{N_3}, A_{N_3}(var(y_{N_1}))),$
$env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\texttt{BCh}(N_3, N_2) \cup L_2 \cup r_2)\} \cup P;\ \{A_{N_3}\} \cup \Delta_1, \{N_1 + 1 = N_3, N_3 < N_2\} \cup \Delta_2$
(iii) $S;\ \{t_1 \doteq bind(y_{N_2}, A_{N_2}(var(y_{N_3}))),$
$env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\texttt{BCh}(N_1, N_3) \cup L_2 \cup r_2)\} \cup P;\ \{A_{N_2}\} \cup \Delta_1, \{N_1 < N_3, N_3 + 1 = N_2\} \cup \Delta_2$
(iv) $S;\ \{t_1 \doteq bind(y_{N_4}, A_{N_4}(var(y_{N_3}))),$
$env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(\texttt{BCh}(N_1, N_3) \cup \texttt{BCh}(N_4, N_2) \cup L_2 \cup r_2)\} \cup P;$
$\{A_{N_4}\} \cup \Delta_1, \{N_1 < N_3, N_3 + 1 = N_4, N_4 < N_2\} \cup \Delta_2$

Where $y_{N_2}, y_{N_3}, y_{N_4}, A_{N_2}, A_{N_3}, A_{N_4}, N_3, N_4$ are fresh variables of appropriate sort.

Fail-Env $\dfrac{S;\ \{env^*(L \cup t) \doteq emptyEnv\} \uplus P}{Fail}$.

If L is nonempty, i.e contains at least one binding or at least one BCh-expression.

An invariant of the rules that deal with BCh is that the variables N_i may appear at most twice in Γ; at most twice explicit in Δ_2 and at most once in BCh-expressions.

4.1 Properties of the LCSX-Unification Algorithm

Lemma 4.4. *For initial problems, the algorithm LCSX terminates.*

Proof. For this we can ignore the rules that change Δ.

The following measure is used, which is a lexicographical combination of several component measures: μ_1 is the number of occurrences of *let* in P; the second component μ_2 is the following size-measure, where $env^*(L \cup r)$ has measure $7m + m' + \sum \mu_2(t_i) + \mu_2(r)$ where m is the number of *bind*-expressions in L and and m' is the number of `BCh`-expressions in L.

The critical applications are the guessing rules for equations with top-context variables, and the rules for multi-equations. The context variable-guessing either decreases the size or the number of occurrences of let. The multi-equation rules in rule **Dec-Chain** have to be analyzed. The new constructed bind-term has size 5, so the subcases (i) – (iii) strictly reduce the size. The subcase (iv) adds 6 to the size due to new sub-terms, and removes 7 since t_1 is a non-`BCh`-expression and removed from the multi-set.

Now we state the properties of the algorithm and give short sketchy proofs. More detailed analyses and more explanations are in Section 5.

Lemma 4.5. *The non-deterministic rule-based unification algorithm LCSX is sound and complete in the following sense: every computed final unification problem that leads to an expression satisfying the DVC represents a set of solutions and every solution of the initial unification problem that represents an expression satisfying the DVC is represented by one final system of equations.*

Proof. Soundness can be proved by standard methods, since rules are either instantiations or instantiations using the theory LC.

Completeness can be proved, if every rule is shown to be complete, and if there are no stuck unification problems that have solutions. The **Solve** rules are complete since solved variables (in equations of the form $x \doteq t$) are just marked as such, i.e. moved to a set of solved equations. Solving equations $X(s) \doteq t$ is complete: if t is a variable, then it can be replaced; if t is a proper term of type *Exp*, then all cases are covered by the rules. In the case that the equation is $X(s) = Y(t)$, the rules are also complete, and also respect the context classes of X, Y. If the equation is $s \doteq s$, then it will be removed, and if it is of the form $f(\ldots) \doteq f(\ldots)$ then decomposition applies. In the case that the top symbol is *env*, the rules for multi-equations apply, i.e., the rules for env^*. Using the properties of the equational theory LC and the considerations in [6,5]), we see that the rules are complete.

Theorem 4.6. *The rule-based algorithm LCSX terminates if applied to initial L_{need}-forking-problems. Thus it decides unifiability of these sets of equations. Since it is sound and complete, and the forking possibilities are finite, the algorithm also computes a finite and complete set of final unification problems by gathering all possible results.*

Theorem 4.7. *The computation of all overlaps between the rules in Figure 1 and left hand sides of normal order reductions in Figure 2 can be done using the algorithm LCSX. The unification algorithm terminates in all of these cases and computes a finite set of final unification problems and hence all the critical pairs w.r.t. our normal order reduction.*

5 Soundness and Completeness

5.1 Correct Handling of Bound Variables

L_{need} is a higher order calculus with bound variables and the usual notion of α-equivalence. When we encode reduction rules of L_{need} into $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ for unification we loose information about bound names. Furthermore the LCSX unification algorithm has no notion of bound variables. An undesirable consequence is that we may equate terms in $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ that are not (α-)equivalent in L_{need}. We give two example for this:

Example 5.1. Unifying (the first-order encodings of) $\lambda x.\lambda y.x$ and $\lambda u.\lambda v.v$ the unification succeeds:

$$\begin{aligned} & \{lam(x, lam(y, var(x))) \doteq lam(u, lam(v, var(v)))\} \\ \Longrightarrow^{3\times\mathbf{Dec}} & \{x \doteq u, y \doteq v, x \doteq v\} =: S. \end{aligned}$$

By coalescing the variables in S we get the unifier $\sigma_S = \{x \mapsto v, y \mapsto v, u \mapsto v\}$ that equates the two terms by an instance that represents $\lambda v.\lambda v.v$. The original expressions are not α-equivalent but the computed solution equates them.

Due to their first order nature, substitutions in $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ are not capture avoiding. E.g. unifying the encodings of $\lambda x.y$ and $\lambda z.z$ (where we assume $x \neq y$) yields $\{x \doteq z, y \doteq z\}$ as a final problem. The corresponding substitution again equates two terms that are not α-equivalent in the original higher order calculus.

To fix this mistreatment of bound names we introduced a method, called *DVC-check*, to discard unifiers that equate expressions from different α-equivalence classes or that provoke capture of variables in the wrong scope. We proceed by explaining the notions that are required to formulate this method.

Recall, that we deal with the following calculi (which are all different representations of the L_{need} calculus defined in section 2.1):

- L_{need} is the call-by-need lambda calculus with recursive letrec expressions.
- Meta-L_{need} denotes the syntax of the L_{need} calculus enriched with meta-variables, that is used to define the reduction rules of L_{need}. We have different sorts of meta-variables for: 1. Variables 2. Expressions 3. Letrec-environments and 4. Contexts.
- $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ is the term language used to encode Meta-L_{need} expressions as terms defined in section 3. This encoding is used to unify left hand sides of transformations with left hand sides of normal order reduction rules.

With $[\![\cdot]\!] : \text{Meta-}L_{need} \to \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ we denote the translation from Meta-L_{need} expressions into their encoding; $[\![\cdot]\!]^- : \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0) \to \text{Meta-}L_{need}$ is the inverse translation. Meta-variables in expression are translated into variables of the appropriate sort. The context classes are chosen according to their intention in the rules. A definition of $[\![\cdot]\!]$ is as follows:

$$\begin{aligned}
[\![x]\!] &= var(x) \\
[\![(s\ t)]\!] &= app([\![s]\!], [\![t]\!]) \\
[\![(\lambda x.s)]\!] &= lam(x, [\![s]\!]) \\
[\![(\texttt{letrec}\ Env\ \texttt{in}\ s)]\!] &= let([\![Env]\!], [\![s]\!]) \\
[\![\{Env, x_1 = s_1, \ldots, x_n = s_n\}]\!] &= [\![\{x_1 = s_1, \ldots, x_n = s_n, Env\}]\!] \\
[\![\{x_1 = s_1, \ldots, Env, \ldots, x_n = s_n\}]\!] &= [\![\{x_1 = s_1, \ldots, x_n = s_n, Env\}]\!] \\
[\![\{x_1 = s_1, x_2 = s_2, \ldots, x_n = s_n\}]\!] &= env([\![x_1 = s_1]\!], [\![\{x_2 = s_2, \ldots, x_n = s_n\}]\!]) \\
[\![\{x = s, Env\}]\!] &= env([\![x = s]\!], [\![Env]\!]) \\
[\![\{x = s\}]\!] &= env([\![x = s]\!], emptyEnv) \\
[\![x = s]\!] &= bind(x, [\![s]\!]) \\
[\![\{y_k = A_k[x], \{y_{i+1} = A_{i+1}[y_i]\}_{i=k}^{M}\}]\!] &= [\![\{y_N = A_N[x], \{y_{i+1} = A_{i+1}[y_i]\}_{i=N}^{M}\}]\!] \\
&\text{where } k \text{ is an integer constant and } N, M \text{ are integer variables.} \\
[\![\{y_{i+1} = A_{i+1}[y_i]\}_{i=N}^{M}]\!] &= \texttt{BCh}(N, M)
\end{aligned}$$

Lemma 5.2. *If $s \in \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ is almost ground and no `BCh`-constructs occur in s then $[\![s]\!]^- \in L_{need}$.*

Definition 5.3. *An L_{need} expression s* satisfies the DVC *iff all free variables in s are distinct from bound variables and all bound variables in s are distinct.*

For a term $s \in \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ let τ be a substitution such that

$$\begin{array}{rl} \tau(t) = a & \text{for all variables } t \in Var_{Exp}(s) \text{ and } a \text{ is a constant} \\ \tau(Env) = emptyEnv & \text{for all environment variables } Env \in Var_{Env}(s) \\ \tau(X) = [\cdot] & \text{for all context variables } X \text{ in } s \\ \tau(\texttt{BCh}(N, M)) = y_m = y_n & \text{for all } \texttt{BCh}\text{-constructs in } s \end{array}$$

A term $s \in \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ satisfies the DVC *iff $[\![\tau(s)]\!]^-$ satisfies the DVC.*
A term $s \in \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ satisfies the closedness condition *iff $[\![\tau(s)]\!]^-$ is closed.*

As we work with α-equivalence classes of terms in L_{need}, we can assume by convention that in an L_{need}-expression all free variables are different from bound variables. We also choose to work with representatives in which all bound variables are distinct. Therefore we can assume that in an initial unification problem all terms satisfy the DVC.

We will also assume that the terms that are obtained after instantiating with a solution satisfy the DVC.

Definition 5.4 (DVC-check). *Assume $s, t \in \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V}^0)$ and $\Gamma = \{s \doteq t\}$ is an initial unification problem and $S \neq Fail$ is a final system derived from Γ.*

Then the following two rules check if the substitution σ_S derived from S satisfies the DVC w.r.t. the initial problem Γ.

$$\textbf{DVC-Success}\ \frac{S}{S}\ \textit{if } \sigma_S(s) \textit{ satisfies the DVC.}$$

$$\textbf{DVC-Fail}\ \frac{S}{Fail}\ \textit{if } \sigma_S(s) \textit{ does not satisfy the DVC.}$$

*If the rule **DVC-Fail** is applicable to a final system we speak of a DVC-check failure w.r.t. the initial unification problem $\{s \doteq t\}$.*

The DVC-check is decidable: If $s \doteq t$ is the initial problem, then apply the substitution σ_S derived from a final problem S to s. Then check if $\tau\sigma_S(s)$ satisfies the DVC where τ is the substitution from definition 5.3 that ensures that the resulting term is ground. (The DVC-check can be done on the representation of the solutions, not all ground instances have to be checked).

The DVC-check can not detect the capture of free variables as in example 5.1.

Convention 5.5 *To avoid capture of free variables by a substitution, terms in an initial unification problem must adhere to the closedness condition (Definition 5.3).*

Solutions of unification problems that violate the DVC may not respect alpha-equivalence in the original L_{need} calculus in a correct way. Hence we have to adapt the notion of solutions.

Definition 5.6 (DVC-solution). *Let $\Gamma = \{s_1 \doteq t_1, \ldots s_n \doteq t_n\}$ and σ be a solution of Γ. Then σ is a* DVC-solution *iff $\sigma(s_i)$ satisfies the DVC for all i.*

Convention 5.7 *As we are only interested in solutions that do not collapse expressions from different α-equivalence classes, we henceforth obey to the convention, that when we speak of σ as a solution of a unification problem Γ we mean that σ is a DVC-solution of Γ.*

5.2 Soundness and Completeness

Definition 5.8. *Let (Γ, Δ), $\Gamma = \{s_1 \doteq t_1, \dots s_n \doteq t_n\}$ be a unification problem. Then we define the set of solutions modulo* LC *of Γ as $U_{LC}(\Gamma) := \{\sigma | \sigma(s_i) =_{LC} \sigma(t_i), 1 \leq i \leq n\}$, where the $\sigma(s_i)$ are almost ground.*

The set of DVC-solutions *is defined by $U_{LC}^{DVC} := \{\sigma | \sigma(s_i) =_{LC} \sigma(t_i) \wedge \sigma(s_i)$ satisfies the DVC$, 1 \leq i \leq n\}$.*

Remark 5.9. In the following proofs we can safely ignore the sets S, P of an unification system because equations in these two sets, that are not explicitly mentioned, are not changed by unification rules.

Lemma 5.10. *The standard unification rules **Dec**, **Trivial** and the failure rules **Fail**, **Fail-C-Lam**, **Fail-C-Var**, **Fail-Env**, **DVC-Fail** preserve the set of DVC-solutions. I.e. If Γ' is derived from Γ with one of the above rules, then $U_{LC}^{DVC}(\Gamma) = U_{LC}^{DVC}(\Gamma')$.*

Proof. For the standard rules this is straightforward.

(**Fail-C-Lam**) If **Fail-C-Lam** is applicable to Γ then there is an equation $X(s) \doteq lam(t_1, t_2)$ in Γ where X belongs to the context class $\mathcal{A}$ or $\mathcal{S}$. Such an equation has no solution, because by definition 2.3 neither of the context classes $\mathcal{A}, \mathcal{S}$ permit the occurrence of the hole in the body of an abstraction.

(**Fail-C-Var**) The equation $X(s) \doteq var(x)$ has no solution, because the symbol *var* is of sort $BV \to Exp$ and in a context the hole can only appear at an term position of sort *Exp* not an position of sort BV.

Lemma 5.11. *The rules of the unification algorithm are correct, i.e. if $\Gamma \Longrightarrow \Gamma'$ using one of the rules of the unification algorithm, then $U_{LC}^{DVC}(\Gamma) \supseteq U_{LC}^{DVC}(\Gamma')$.*

Proof. For the rules mentioned in lemma 5.10 this is obvious.

If the set of DVC-solutions of Γ' is empty then $U_{LC}^{DVC}(\Gamma) \supseteq \emptyset = U_{LC}^{DVC}(\Gamma')$ holds. Therefore we assume $U_{LC}^{DVC}(\Gamma') \neq \emptyset$.

If a rule introduces variables of sort BV (as is the case for **Dec-C-Let**,**Merge-Fork-C**,**Dec-Chain**) then the variables introduced can be chosen in such a way, that the DVC is satisfied by the resulting terms.

We show two unification rules as correct. The other rules can be proved correct similarly straightforward.

To prove that **Dec-C-App** is a correct unification rule, we assume that σ is a solution of $\{X \doteq app(X', t_2), X'(s) \doteq t_1\}$, i.e. $\sigma(X) =_{LC} app(\sigma(X'), \sigma(t_2))$ and $\sigma(X'(s)) =_{LC} \sigma(t_1)$. Such a solution σ also solves the equation $X(s) \doteq app(t_1, t_2)$ which can be seen by simple instantiation $\sigma(X(s)) = app(\sigma(X'(s)), \sigma(t_2)) =_{LC} app(\sigma(t_1), \sigma(t_2)) = \sigma(app(t_1, t_2))$.

In the correctness proof of the rule **Dec-Chain** we only show correctness for alternative i) (the other alternatives are proved similarly). Therefore assume that σ is a solution of

$$\{t_1 \doteq bind(y_{N_2}, A_{N_2}(var(y_{N_1}))), env^*(L_1 \setminus \{t_1\} \cup r_1) \doteq env^*(L_2 \cup r_2)\}$$

that satisfies the constraint $\{A_{N_2}\} \cup \Delta_1, \{N_1 + 1 = N_2\} \cup \Delta_2$. I.e. the following holds: $\sigma(t_1) =_{LC} \sigma(bind(y_{N_2}, A_{N_2}(var(y_{N_1}))))$ and $\sigma(env^*(L_1 \setminus \{t_1\} \cup r_1)) =_{LC} \sigma(env^*(L_2 \cup r_2))$. Applying σ to the original equation $env^*(L_1 \cup r_1) \doteq env^*(\texttt{BCh}(N_1, N_2) \cup L_2 \cup r_2)$ yields $\sigma(env^*(L_1 \cup r_1)) = env^*(\sigma(L_1 \cup r_1)) = env^*(\sigma(t_1) \cup \sigma(L_1 \setminus \{t\} \cup r_1)) =_{LC} env^*(\sigma(bind(y_{N_2}, A_{N_2}(var(y_{N_1})))) \cup \sigma(L_2 \cup r_2)) =$ [1] $env^*(\sigma(\texttt{BCh}(N_1, N_2)) \cup \sigma(L_2 \cup r_2)) = \sigma(env^*(\texttt{BCh}(N_1, N_2) \cup L_2 \cup r_2))$.

[1] This follows by instantiation of the `BChain` according to the constraint $N_1 + 1 = N_2$.

Theorem 5.12 (Soundness).
If $\Gamma \Longrightarrow^* \Gamma'$ and Γ' is a final unification problem then $U^{DVC}_{LC}(\Gamma) \supseteq U^{DVC}_{LC}(\Gamma')$.

Proof. By induction on the length of the transformation to solved form using lemma 5.11.

Theorem 5.13 (Completeness). *Let Γ be an almost linear unification problem. For each $\theta \in U^{DVC}_{LC}(\Gamma)$ there exists a sequence of LCSX transformations $\Gamma \Longrightarrow \Gamma_1 \Longrightarrow \ldots \Longrightarrow \Gamma_n$ such that Γ_n is a final system that represents θ.*

Proof. For almost linear problems the LCSX algorithm terminates (by Lemma 4.4) with a unification problem that is either final or *Fail*.

If Γ is in solved form (a final system) then it is of the form $\Gamma = \{x_1 \doteq y_1, \ldots, x_m \doteq y_m, z_1 \doteq t_1, \ldots, z_n \doteq t_n\}$. If the DVC-check fails on this set of equations, then Γ has no DVC-solution. Otherwise all DVC-solutions of this systems are represented by σ_Γ (the substitution that can be derived from Γ).

It remains to show that for each Γ_i, which is not a final unification problem, and every solution $\sigma \in U^{DVC}_{LC}(\Gamma_i)$ of Γ_i there exists a LCSX transformation $\Longrightarrow$ such that $\Gamma_i \Longrightarrow \Gamma_{i+1}$ and $\sigma \in U^{DVC}_{LC}(\Gamma_i)$.

If Γ_i is not a final problem, then it contains some equations that are not solved and can still be transformed by LCSX unification rules. We go trough the cases for these equations.

Case $f(s_1, \ldots s_m) \doteq g(t_1, \ldots, t_n)$ where f, g are free function symbols. Either $f = g$ and $m = n$, and then the rule **Dec** can be applied, which by Lemma 5.10 does not modify the set of solutions. Or $f \neq g$ then the **Fail** rule applies which also does not change the set of solutions.

This holds for all unsolved equations to which unification rules can be applied that do not modify the set of solutions (i.e. the rules covered in Lemma 5.10).

Case $X(s) \doteq f(t_1, t_2)$. We assume that $\sigma \in U^{DVC}_{LC}(\{X(s) \doteq f(t_1, t_2)\})$. The context variable X can be the empty context, in this case $\sigma(s) =_{LC} f(\sigma(t_1), \sigma(t_2))$ holds and σ is also a solution of the unification problem $\{X \doteq [\cdot]\} \cup \{X \mapsto [\cdot]\}(\{X(s) \doteq f(t_1, t_2)\})$, which results from $X(s) \doteq f(t_1, t_2)$ by application of the *ii*) case of the rule **Empty-C**. If X is not the empty context (i.e. $X \in \Delta_1$) we have to go trough the cases for the function symbol f and the the context class $\mathcal{D}$ of X.

Case $f = app$ i.e. $\sigma(X(s)) =_{LC} app(\sigma(t_1), \sigma(t_2))$ holds. 1. Assume $\mathcal{D} = \mathcal{A}$. Since X is not the empty context and in a $\mathcal{A}$ context the hole can appear only in the first argument of *app*, we can conclude $\sigma(X) =_{LC} app(\sigma(t_1[\cdot]_p), \sigma(t_2))$ and $\sigma(s) =_{LC} \sigma(t_1|_p)$ for some position p in $Pos(t_1)$ such that $t_1[\cdot]_p$ is a context of class $\mathcal{A}$. With $\tau = \{X' \mapsto t_1[\cdot]_p\}$ we have a solution $\tau\sigma$ for the left hand side of rule **Dec-C-App** i.e. $\tau\sigma$ solves $\{X \doteq app(X', t_2), X'(s) \doteq t_1\}$. 2. Now we assume $\mathcal{D} = \mathcal{S}$ or $\mathcal{D} = \mathcal{C}$. Since every $\mathcal{A}$ is also an $\mathcal{S}$ context ($\mathcal{C}$ context respectively) the above case applies as well. In addition, according to the definition of the two context classes, the hole can also occur in the second argument of *app*. Therefore we can conclude $\sigma(X) =_{LC} app(\sigma(t_1), \sigma(t_2[\cdot]_p))$ for some position p in $Pos(t_2)$ such that $t_2[\cdot]_p$ is a context of class $\mathcal{S}$ ($\mathcal{C}$ respectively). If we set $\tau = \{X' \mapsto t_2[\cdot]_p\}$ we have a solution $\tau\sigma$ that also solves the equation in the conclusion of rule **Dec-C** i.e. $\{X \doteq app(t_1, X'), X'(s) \doteq t_2\}$.

Case $f = lam$ According to the definition of the context classes $\mathcal{A}, \mathcal{S}$ and $\mathcal{C}$ the hole is only admissible in a body of an abstraction for contexts of class $\mathcal{C}$. Hence $X(s) \doteq lam(t_1, t_2)$ has only a solution if X is of class $\mathcal{C}$ (the two rules that cover all possible solutions of the equation are **Dec-C-Lam** and **Fail-C-Lam**).

Case $f = let$ then $\mathcal{D}$ can be either a $\mathcal{S}$ or a $\mathcal{C}$ context (if $\mathcal{D}$ is a $\mathcal{A}$ context, then $X(s) \doteq let(t_1, t_2)$ has no solution). We have two cases: 1. The hole may occur in the second argument of *let*. In this case a solution is also a solution of the equation transformed by the rule **Dec-C**. 2. Additionally to the above case the hole can appear in the first argument of *let* i.e. $\sigma(X) =_{LC} let(\sigma(t_1[\cdot]_p), \sigma(t_2))$ and $\sigma(s) =_{LC} \sigma(t_1|_p)$ for some position $p \in Pos(t_1)$ such that $t_1[\cdot]_p$ is an admissible $\mathcal{S}$ contexts

($\mathcal{C}$ context respectively). Since *let* is of sort $Env \to Exp \to Env$ the head symbol of the context $t_1[\cdot]_p$ must be of sort *Env* and in t_1 the hole can occur only on the right hand side of variable-expression binding, hence we can conclude $\sigma(t_1[\cdot]_p) =_{LC} \sigma(env^*(\{bind(x,(t_1|_q)[\cdot]_r)\} \cup z))$ for some fresh variables x, z of appropriate sort and and some positions q, r such that $p = qr$ (i.e. $q = 1.2$). When we set $\tau = \{X' \mapsto (t_1|_q)[\cdot]_r\}$ then $\sigma\tau$ is a solution for the conclusion $\{X \doteq let(env^*(\{bind(x,X')\} \cup z), t_2), env^*(\{bind(x, X'(s))\} \cup z) \doteq t_1\}$ of the **Dec-C-Let** rule.

Case $f = env$ **or** $f = bind$**.** This case can not occur since *env* is of sort $Bind \to Env \to Env$ (*bind* is of sort $BV \to Exp \to Bind$ respectively) and X is of sort *Exp*. Therefore an equation $X(s) \doteq env(t_1, t_2)$ ($X(s) \doteq bind(t_1, t_2)$ respectively) is not well sorted and has no well sorted solution.

Case $X(s) \doteq Y(t)$. Assume $\sigma \in U^{DVC}_{LC}(\{X(s) \doteq Y(t)\})$. Hence $\sigma(X) =_{LC} s_p[\cdot]_p, \sigma(Y) =_{LC} t_q[\cdot]_q$ and $s_p[\sigma(s)]_p =_{LC} t_q[\sigma(t)]_q$ holds. Let p_0 be the greatest common prefix of p, q. We have to distinguish two cases:

First case: $p_0 = p$ **or** $p_0 = q$ W.l.o.g. we assume $p_0 = p$ (in this case X is a prefix of Y, the other case is symmetrical). Then $t_q[\cdot]_q$ can be written as $t_q[\cdot]_q =_{LC} s_p[u[\cdot]_{p'}]_p$ where $u[\sigma(s)]_{p'} =_{LC} \sigma(t)$.
Let $\tau = \{Y' \mapsto u[\cdot]_{p'}, X' \mapsto s_p[\cdot]_p\}$. Then $\tau\sigma$ is a solution for the conclusion of **Merge-Prefix**.

Second case: p_0 **is distinct from** p **and** q Sketch: In this case we have incomparable positions of the holes in the solution. Guess where the least common ancestor of the two positions p, q is: p_0 (i.e. $p = p_0p'$ and $q = p_0q'$). The context above this position is Z. The funciton symbol at this position is f: a binary function symbol that can accommodate two holes at different positions directly under the root position and has resulting sort *Exp*, i.e. $f \in app, let$. Below f there are two contexts X', Y' before we find the terms s, t.
The context classes of X, Y also have to be taken into account: if exactly one context is a $\mathcal{A}$ context then the case is covered by **Merge-Fork-A** (f can only be *app* and the hole of the $\mathcal{A}$ context is in the first argument of *app*). If both contexts have a context class greater than $\mathcal{A}$ then the additional cases are covered by the possibilities of the rule **Merge-Fork-C**.

Case $env^*(s_1, \dots s_m) \doteq env^*(t_1, \dots, t_n)$. Then the rules **Solve-Env**, **Dec-Env** and **Dec-Chain** can be applied. A proof of their completeness is in [6].

6 Running the Unification Algorithm LCSX

Example 6.1. The goal is to compute a complete set of forks for the transformation (cp-e)

$$(\texttt{letrec}\ x = s, Env, z = C[x]\ \texttt{in}\ r) \ \to (\texttt{letrec}\ x = s, Env, z = C[s]\ \texttt{in}\ r)$$

from Figure 1. We instantiate the meta-variable s by the expression $\lambda w.t$ and translate the left hand side of the rule into the term language, resulting in the following initial forking problem to be solved

$$\{S(let(env^*(\{bind(x, lam(w,t)), bind(z, C(var(x)))\} \cup Env), r)) \doteq l_{no,j}\}.$$

where $l_{no,j}$ is an encoded left hand side of an no-reduction rule. We pick a single equation from this set:

$$\begin{aligned} &S(let(env^*(\{bind(x, lam(w,t)), bind(z, C(var(x)))\} \cup Env_1), r)) \\ \doteq\ &let(env^*(\{bind(x', lam(w',t')), bind(y_{N_1}, A_{N_1}(var(x')))\} \cup \texttt{BCh}(N_1, N_2) \cup Env_2), A(y_{N_2})) \end{aligned}$$

which describes the overlaps between the (cp-e) transformation and the normal order (cp-e-c) reduction. No we compute one possible final problem via the presented unification algorithm. A nontrivial possibility is to choose $S = [\cdot]$ via the **Empty-C**-rule and then using decomposition for *let* which leads to $r = A(y_{N_2}))$ and the equation

$$\begin{aligned} &env^*(\{bind(x, lam(w,t)), bind(z, C(var(x)))\} \cup Env_1) \\ \doteq\ &env^*(\{bind(x', lam(w',t')), bind(y_{N_1}, A_{N_1}(var(x')))\} \cup \mathtt{BCh}(N_1, N_2) \cup Env_2). \end{aligned}$$

One choice for the next step (via the rule **Dec-Chain**) results in the equations:

$$\begin{aligned} &bind(z, C(var(x)) \doteq bind(y_{N_4}, A_{N_4}(var(y_{N_3}))), \quad env^*(\{bind(x, lam(w,t))\} \cup Env_1) \\ \doteq\ &env^*(\{bind(x', lam(w',t')), bind(y_{N_1}, A_{N_1}(var(x')))\} \cup (\mathtt{BCh}(N_1, N_3) \cup \mathtt{BCh}(N_4, N_2) \cup Env_2) \end{aligned}$$

where one binding is taken from the $\mathtt{BCh}(N_1, N_2)$-construct and the chain is split around this binding into two remaining chains. The two bindings $bind(x, lam(w,t))$ and $bind(x', lam(w',t'))$ are unified (via **Dec-Env**) and then we solve the equation between the environments (**Solve-Env**) and (after three additional **Dec**-steps, two for *bind* and one for *lam*) we arrive at the system

$$\begin{aligned} &C(var(x)) \doteq A_{N_4}(var(y_{N_3})), z \doteq y_{N_4}, x \doteq x', w \doteq w', t \doteq t', Env_2 \doteq Env_3, \\ &Env_1 \doteq env^*(\{bind(y_{N_1}, A_{N_1}(var(x')))\} \cup \mathtt{BCh}(N_1, N_3) \cup \mathtt{BCh}(N_4, N_2) \cup Env_3). \end{aligned}$$

Next we apply **Merge-Fork-A** to the first equation, yielding

$$C \doteq Z(app(A'_{N_4}(var(y_{N_3})), C')), A_{N_4} \doteq Z(app(A'_{N_4}, C'(var(x))))$$

where Z, A'_{N_4} are of context class $\mathcal{A}$ and C' is of context class $\mathcal{C}$. The final representation is:

$$\begin{aligned} &S_{\neg BV} = \{t \doteq t', S \doteq [\cdot], r \doteq A(y_{N_2}), C \doteq Z(\ldots), A_{N_4} \doteq Z(\ldots), Env_2 \doteq Env_3, Env_1 \doteq env^*(\ldots)\} \\ &S_{BV} = \{z \doteq y_{N_4}, x \doteq x', w \doteq w'\} \end{aligned}$$

The resulting expression is:

$$\begin{aligned} &\mathtt{letrec}\ x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, \\ &\qquad y_{N_4} = Z[app(A'_{N_4}(var(y_{N_3})), C'[x'])], \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_3\ \mathtt{in}\ A[y_{N_2}] \end{aligned}$$

The corresponding fork is given by reducing the expression with (no,cp-e-c) and (cp-e) respectively

$$\begin{array}{c} \mathtt{letrec}\ x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, y_{N_4} = Z[app(A'_{N_4}[var(y_{N_3})], C'[x'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_3\ \mathtt{in}\ A[y_{N_2}] \end{array}$$

$$\swarrow^{no,cp\text{-}e\text{-}c} \qquad \searrow^{i\mathcal{S},cp\text{-}e}$$

$$\begin{array}{c} \mathtt{letrec}\ x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[\lambda \mathbf{w}'.\mathbf{t}'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, y_{N_4} = Z[app(A'_{N_4}[var(y_{N_3})], C'[x'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_3\ \mathtt{in}\ A[y_{N_2}] \end{array} \qquad \begin{array}{c} \mathtt{letrec}\ x' = (\lambda w'.t'), y_{N_1} = A_{N_1}[x'], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_1}^{N_3}, \\ y_{N_4} = Z[app(A'_{N_4}[var(y_{N_3})], C'[\lambda \mathbf{w}'.\mathbf{t}'])], \\ \{y_{i+1} = A_{i+1}[y_i]\}_{i=N_4}^{N_2}, Env_3\ \mathtt{in}\ A[y_{N_2}] \end{array}$$

This fork can be closed by the sequence $\xrightarrow{i\mathcal{S},cp\text{-}e} \cdot \xleftarrow{no,cp\text{-}e\text{-}c}$. Notice that for the determination of all forks it is sufficient to compute final systems. The (possibly infinite) set of ground solutions is not required.

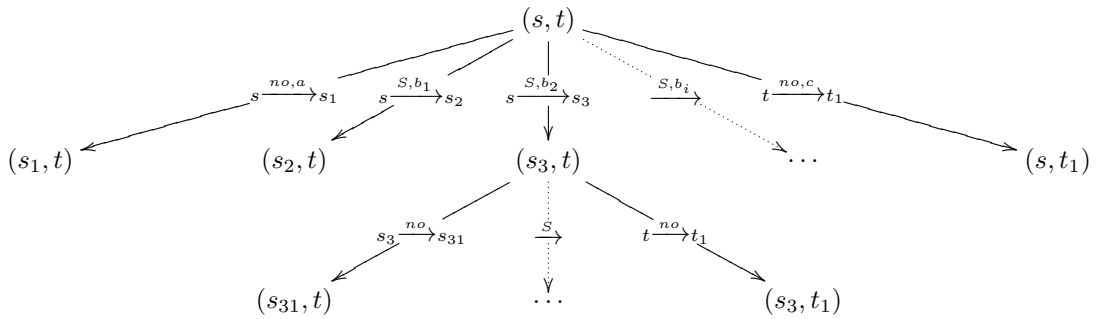
7 Computed Forks and Forking Diagrams

7.1 Automatic Closing of Forks

We implemented the presented unification algorithm in Haskell to compute all forks between transformations and no-reductions. The implementation uses searching as a method to close the forks. To close a fork $s \xleftarrow{no} \cdot \xrightarrow{iS} t$ the meta term s can be reduced with internal- or no-reductions whereas the term t can only be no-reduced. We build a search tree with root node (s, t) and for every node (s_i, t_j) in the tree we add the following children:

- (s_l, t_j), if $s_i \xrightarrow{no,a} s_l$ for a no-reduction a. And
- (s_l, t_j), for all possible (not necessarily internal) transformations in surface context $s_i \xrightarrow{S,b} s_l$. And
- (s_i, t_k), if $t_j \xrightarrow{no,c} t_k$ for a no-reduction c.

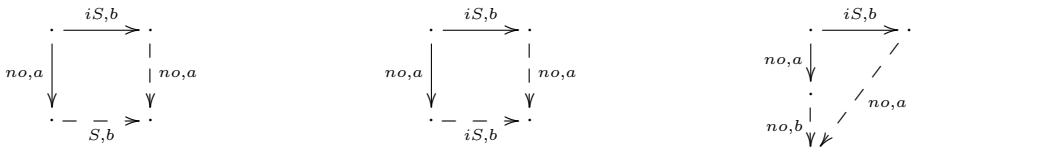
The structure of such a search tree can be depicted as follows



We search such a tree via iterative deepening depth-first search (with a successively increasing depth-bound) for a node where $s_i =_{lc} t_j$. The path to the node describes a reduction sequence that closes the fork between s and t. (This method is generally not sufficient (complete) to close all diagrams because the meta terms are not guaranteed to terminate, but for the present calculus it is adequate to close all computed forks).

To reduce meta terms we use matching. A meta term s *S-reduces* to a term s', written as $s \xrightarrow{S} s'$, if there is a transformation rule $l_T \to r_T$ (from Figure 1) such that the matching problem $s \leq^? S(l_T)$, where S is a context variable of sort $\mathcal{S}$, has a solution σ. The result of the reduction is $s' = \sigma(S(r_T))$. A meta term s *no-reduces* to s', denoted by $s \xrightarrow{no} s'$, if there is a normal order reduction rule $l_{no} \to r_{no}$ (from Figure 2) such that the matching problem $s \leq^? l_{no}$ has a solution σ. The result of the reduction is $s' = \sigma(r_{no})$.

The $\mathcal{S}$ context that encloses the left hand side of the transformation rule can not ensure that a closing reduction is internal. Therefore the automatic closing procedure computes diagrams, where the closing reduction sequence at the bottom of the diagram (after the normal order reduction) is in a $\mathcal{S}$ context. Such a diagram, as can be seen in Figure 4(a), subsumes diagrams as depicted in 4(b) and 4(c).



(a) S-reduction closes (b) internal S-reduction closes (c) no-reduction closes

Fig. 4.

To solve a matching problem $s \leq^? t$ the standard unification rules and the rules **DVC-Fail**, **Empty-C** as described in section 4 are used. The following rule is added

Match-Var-Fail $S;\ \{x \leq^? s\} \uplus P \Longrightarrow \mathit{Fail}$

If s is not a variable (its head symbol is a function symbol or a context variable)

Given an equation $t \leq^? X(s)$ if $t \neq Y(t')$ (i.e. the head symbol of s is a function symbol) then the **Dec-** and **Fail-**rules from section can be used to solve the equation. For an equation $Y(t) \leq^? X(s)$ we replace the **Merge-**rules with the following two rules:

Match-C $S;\ \{Y(t) \leq^? X(s)\} \uplus P \Longrightarrow \{X \mapsto Y\} \cup S;\ \{t \leq^? s\} \cup P$

If the context class of Y is less or equal to the context class of X.

Match-Fail $S;\ \{Y(t) \leq^? X(s)\} \uplus P \Longrightarrow \mathit{Fail}$

If the context class of Y is greater or equal to the context class of X.

7.2 Computed Forking Diagrams

We give a overview of the computed complete diagram sets for the transformations of the L_{need} calculus as defined in Figure 1. Rather than enumerate all computed diagrams we give a summarized compilation with some typical examples.

There are no critical overlaps for internal *lbeta* and *lapp* transformations. All non critical overlaps for the two transformations are covered by diagrams as pictured in Figure 4. A typical case for a diagram of type 4(a) is the closing sequence for the overlapping of (internal) lbeta transformation with a normal order lletin reduction:

$$\begin{array}{l} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (\texttt{letrec}\ E_{11}\ \texttt{in}\ ((\lambda x_1.s_1)\ r_1)))] \\ \hline \xrightarrow{no,lletin} Z_{12}^A[(\texttt{letrec}\ \{E_{10}, E_{11}\}\ \texttt{in}\ ((\lambda x_1.s_1)\ r_1))] \\ \xrightarrow{S,lbeta} Z_{12}^A[(\texttt{letrec}\ \{E_{10}, E_{11}\}\ \texttt{in}\ (\texttt{letrec}\ \{x_1 = r_1, \}\ \texttt{in}\ s_1))] \\ \hline \xrightarrow{iS,lbeta} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (\texttt{letrec}\ E_{11}\ \texttt{in}\ (\texttt{letrec}\ \{x_1 = r_1, \}\ \texttt{in}\ s_1)))] \\ \xrightarrow{no,lletin} Z_{12}^A[(\texttt{letrec}\ \{E_{10}, E_{11}\}\ \texttt{in}\ (\texttt{letrec}\ \{x_1 = r_1, \}\ \texttt{in}\ s_1))] \end{array}$$

The closing S-reduction is either normal order, if Z_{12} is the empty context, or internal, if Z_{12} is not empty.

We give some examples of the computed forks and the corresponding computed closing sequences.

An example reduction sequence for diagram 5(b) is:

$$\begin{array}{l} (\texttt{letrec}\ \{x_{13} = (\texttt{letrec}\ E_{10}\ \texttt{in}\ (\texttt{letrec}\ E_6\ \texttt{in}\ s_5)), E_{14}\}\ \texttt{in}\ A_8^A[x_{13}]) \\ \xrightarrow{no,llete} (\texttt{letrec}\ \{x_{13} = (\texttt{letrec}\ E_6\ \texttt{in}\ s_5), E_{10}, E_{14}\}\ \texttt{in}\ A_8^A[x_{13}]) \\ \xrightarrow{no,llete} (\texttt{letrec}\ \{x_{13} = s_5, E_6, E_{10}, E_{14}\}\ \texttt{in}\ A_8^A[x_{13}]) \\ \hline \xrightarrow{i,S} (\texttt{letrec}\ \{x_{13} = (\texttt{letrec}\ \{E_6, E_{10}\}\ \texttt{in}\ s_5), E_{14}\}\ \texttt{in}\ A_8^A[x_{13}]) \\ \xrightarrow{no,llete} (\texttt{letrec}\ \{x_{13} = s_5, E_6, E_{10}, E_{14}\}\ \texttt{in}\ A_8^A[x_{13}]) \end{array}$$

An example for diagram 5(c) is:

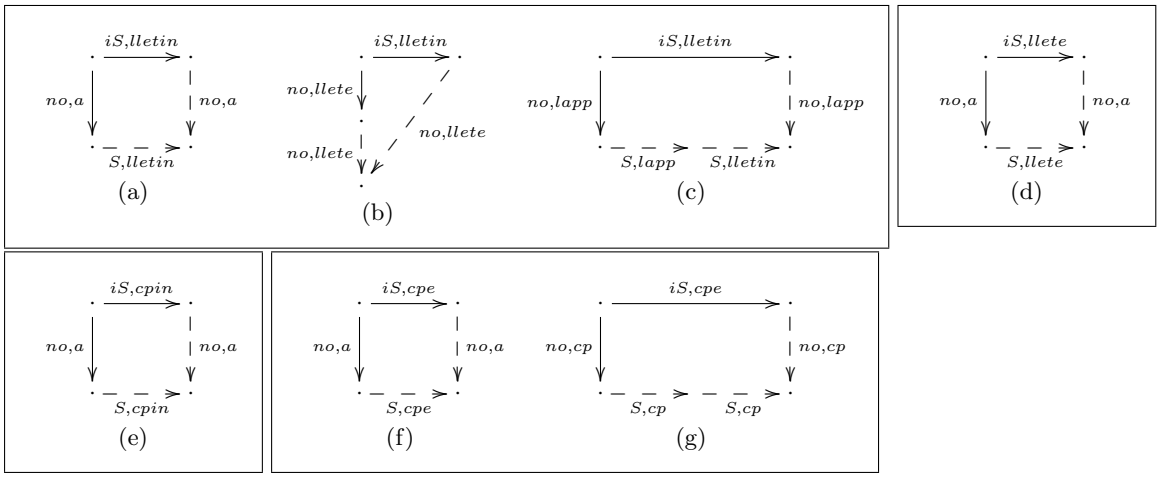


Fig. 5. Complete sets of forking diagrams for internal transformations llet and cp.

$$\begin{array}{l} Z_{12}^A[((\texttt{letrec}\ E_{10}\ \texttt{in}\ (\texttt{letrec}\ E_6\ \texttt{in}\ s_5))\ s_{10})] \\ \hline \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ ((\texttt{letrec}\ E_6\ \texttt{in}\ s_5)\ s_{10}))] \\ \xrightarrow{S,lapp} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (\texttt{letrec}\ E_6\ \texttt{in}\ (s_5\ s_{10})))] \\ \xrightarrow{S,lletin} Z_{12}^A[(\texttt{letrec}\ \{E_6, E_{10}\}\ \texttt{in}\ (s_5\ s_{10}))] \\ \hline \xrightarrow{iS,lletin} Z_{12}^A[((\texttt{letrec}\ \{E_6, E_{10}\}\ \texttt{in}\ s_5)\ s_{10})] \\ \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ \{E_6, E_{10}\}\ \texttt{in}\ (s_5\ s_{10}))] \end{array}$$

The closing S-reductions are either normal order, if the $\mathcal{A}$-context Z_{12} is the empty context, or internal otherwise.

A typical case for 5(d) is:

$$\begin{array}{l} Z_{12}^A[((\texttt{letrec}\ E_{10}\ \texttt{in}\ t_{10})\ (\texttt{letrec}\ \{x_7 = (\texttt{letrec}\ E_7\ \texttt{in}\ s_7), E_8\}\ \texttt{in}\ r_7))] \\ \hline \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (t_{10}\ (\texttt{letrec}\ \{x_7 = (\texttt{letrec}\ E_7\ \texttt{in}\ s_7), E_8\}\ \texttt{in}\ r_7)))] \\ \xrightarrow{S,llete} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (t_{10}\ (\texttt{letrec}\ \{x_7 = s_7, E_7, E_8\}\ \texttt{in}\ r_7)))] \\ \hline \xrightarrow{iS,llete} Z_{12}^A[((\texttt{letrec}\ E_{10}\ \texttt{in}\ t_{10})\ (\texttt{letrec}\ \{x_7 = s_7, E_7, E_8\}\ \texttt{in}\ r_7))] \\ \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ E_{10}\ \texttt{in}\ (t_{10}\ (\texttt{letrec}\ \{x_7 = s_7, E_7, E_8\}\ \texttt{in}\ r_7)))] \end{array}$$

The closing S-reduction is either normal order, if Z_{12}^A is the empty context, otherwise it is internal.

An example for a computed closing sequence for 5(e) is:

$$\begin{array}{l} Z_{12}^A[((\texttt{letrec}\ \{x_3 = s_3, E_3\}\ \texttt{in}\ x_3)\ s_{13})] \\ \hline \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ \{x_3 = s_3, E_3\}\ \texttt{in}\ (x_3\ s_{13}))] \\ \xrightarrow{S,cpin} Z_{12}^A[(\texttt{letrec}\ \{x_3 = s_3, E_3\}\ \texttt{in}\ (s_3\ s_{13}))] \\ \hline \xrightarrow{iS,cpin} Z_{12}^A[((\texttt{letrec}\ \{x_3 = s_3, E_3\}\ \texttt{in}\ s_3)\ s_{13})] \\ \xrightarrow{no,lapp} Z_{12}^A[(\texttt{letrec}\ \{x_3 = s_3, E_3\}\ \texttt{in}\ (s_3\ s_{13}))] \end{array}$$

A typical case for diagram 5(g) is:

$$
\begin{array}{l}
(\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[x_4]), y_{11} = A^A_{10}[y_{10}], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}]) \\
\hline
\xrightarrow{no,cpe} (\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[x_4]), y_{11} = A^A_{10}[(\lambda z'_{10}.C^C_{16}[x_4])], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}]) \\
\xrightarrow{S,cpe} (\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[s_4]), y_{11} = A^A_{10}[(\lambda z'_{10}.C^C_{16}[x_4])], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}]) \\
\xrightarrow{S,cpe} (\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[s_4]), y_{11} = A^A_{10}[(\lambda z'_{10}.C^C_{16}[s_4])], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}]) \\
\hline
\xrightarrow{iS,cpe} (\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[s_4]), y_{11} = A^A_{10}[y_{10}], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}]) \\
\xrightarrow{no,cpe} (\texttt{letrec}\ \{x_4 = s_4, y_{10} = (\lambda z_{10}.C^C_{16}[s_4]), y_{11} = A^A_{10}[(\lambda z'_{10}.C^C_{16}[s_4])], E_{15}\}\ \texttt{in}\ A^A_8[y_{11}])
\end{array}
$$

The two closing S-reductions are always internal although the matching method used for reduction does not ensure this.

Not all forks could be closed automatically by the described procedure. Those forks are always of the same type: Overlaps with cp-rules, for example

$$
\begin{array}{l}
(\texttt{letrec}\ \{y_{14} = (\lambda z_{14}.t_{14}), y_{15} = Z^A_{17}[(A^A_{15}[y_{14}]\ C^C_{16}[y_{14}])], E_{14}\}\ \texttt{in}\ A^A_{12}[y_{15}]) \\
\hline
\xrightarrow{no,cpe} (\texttt{letrec}\ \{y_{14} = (\lambda z_{14}.t_{14}), y_{15} = Z^A_{17}[(A^A_{15}[(\lambda z'_{14}.t_{14})]\ C^C_{16}[y_{14}])], E_{14}\}\ \texttt{in}\ A^A_{12}[y_{15}]) \\
\hline
\xrightarrow{iS,cpe} (\texttt{letrec}\ \{y_{14} = (\lambda z_{14}.t_{14}), y_{15} = Z^A_{17}[(A^A_{15}[y_{14}]\ C^C_{16}[(\lambda z'_4.t_{14})])], E_{14}\}\ \texttt{in}\ A^A_{12}[y_{15}])
\end{array}
$$

The reason the automatic closing fails here, is the renaming of bound variables during application of cp-rules.

7.3 Statistics

We present some statistics about the number of computed forks. Two aspects about these numbers have to be kept in mind:

- Not all computed overlaps are critical: Trivial overlaps at root positions and overlaps at variable positions are computed as well.
- There are some redundant unifiers (mainly because of the rules **Dec-Chain** and **Empty-C**), therefore the sets of computed forks (respectively the sets of diagrams) are not minimal, but complete due to the completeness of the unification algorithm.

Transformation	**Computed Forks**	**Transformation**	**Computed Forks**
lbeta	113	llete	117
lapp	97	cpin	350
lletin	101	cpe	436

8 Conclusion and Further Work

We have provided a effective method using first-order unification with equational theories, sorts, context variables and context classes and binding chains of variable length to compute all critical overlaps between a set of transformation rules and a set of normal order rules in a call-by-need lambda calculus with letrec-environments. Further work is to apply this method to further transformations and also to extend the method in order to make it applicable to other program calculi as in [15], where variable-variable bindings are present in the normal order rules, and to calculi with data structures and case-expressions.

References

1. Z. M. Ariola & M. Felleisen (1997): *The call-by-need lambda calculus.* *J. Funct. Programming* 7(3), pp. 265–301.
2. Franz Baader & Klaus U. Schulz (1992): *Unification in the union of disjoint equational theories: Combining decision procedures.* In: *11th CADE 92*, *LNCS* 607, Springer, pp. 50–65.
3. Franz Baader & Wayne Snyder (2001): *Unification Theory.* In: J. A. Robinson & A.Voronkov, editors: *Handbook of Automated Reasoning*, Elsevier and MIT Press, pp. 445–532.
4. Evgeny Dantsin & Andrei Voronkov (1999): *A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees.* In: *FoSSaCS*, pp. 180–196.
5. Agostino Dovier, Alberto Policriti & Gianfranco Rossi (1998): *A Uniform Axiomatic View of Lists, Multisets, and Sets, and the Relevant Unification Algorithms.* Fundam. Inform. 36(2-3), pp. 201–234.
6. Agostino Dovier, Enrico Pontelli & Gianfranco Rossi (2006): *Set unification.* *TPLP* 6(6), pp. 645–701. Available at `http://dx.doi.org/10.1017/S1471068406002730`.
7. Matthias Felleisen & R. Hieb (1992): *The revised report on the syntactic theories of sequential control and state.* *Theoret. Comput. Sci.* 103, pp. 235–271.
8. D. Howe (1989): *Equality in lazy computation systems.* In: *4th IEEE Symp. on Logic in Computer Science*, pp. 198–203.
9. D. Knuth & P. B. Bendix (1970): *Simple word problems in universal algebra.* In: J. Leech, editor: *Computational problems in abstract algebra*, Pergamon Press, pp. 263–297.
10. R. Milner (1977): *Fully abstract models of typed λ-calculi.* *Theoret. Comput. Sci.* 4, pp. 1–22.
11. Manfred Schmidt-Schauß (1989): *Unification in a Combination of Arbitrary Disjoint Equational Theories.* *J. Symbolic Computation* 8(1,2), pp. 51–99.
12. Manfred Schmidt-Schauß (2007): *Correctness of Copy in Calculi with Letrec.* In: *Term Rewriting and Applications (RTA-18)*, *LNCS* 4533, Springer, pp. 329–343.
13. Manfred Schmidt-Schauß & David Sabel (2010): *On generic context lemmas for higher-order calculi with sharing.* *Theoret. Comput. Sci.* 411(11-13), pp. 1521 – 1541.
14. Manfred Schmidt-Schauß, David Sabel & Elena Machkasova (2010): *Simulation in the Call-by-Need Lambda-Calculus with letrec.* In: Christopher Lynch, editor: *Proc. of 21st RTA 2010*, *LIPIcs* 6, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 295–310. Available at `http://dx.doi.org/10.4230/LIPIcs.RTA.2010.295`.
15. Manfred Schmidt-Schauß, Marko Schütz & David Sabel (2008): *Safety of Nöcker's Strictness Analysis.* *J. Funct. Programming* 18(04), pp. 503–551.