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

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.

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 $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ 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 $P_1 \sim P_2 \iff (\text{for all } C : C[P_1]\downarrow \iff C[P_2]\downarrow)$, where $C$ is a context, i.e. a program with a hole $[\_]$ 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 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.

---

\*This author is supported by the DFG under grant SCHM 986/9-1.
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 [9], ciu-lemmas [6] and bisimulation, see e.g. [7].

The reduction relation $\rightarrow$ is often given as a set of rules $l_i \rightarrow r_i$ 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 $s_1 \rightarrow s_2$, we have to show that $\sigma(s_1) \sim \sigma(s_2)$ for all possible rule instantiations $\sigma$ 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 [8] 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 $L_{\text{need}}$ 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 5 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 $L_{\text{need}}$

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

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

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

We assign the names application, abstraction, or letrec-expression to the expressions $(s t)$, $(\lambda x.s)$, $(\text{letrec } x_1 = s_1, \ldots, x_n = s_n \text{ in } t)$, 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 $(\text{letrec } x_1 = s_1, \ldots, x_{n-1} = s_{n-1} \text{ in } s_n)$ are all expressions $s_i$ with $1 \leq i \leq n$. Free and bound variables in expressions and $\alpha$-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 $\alpha$-rename bound variables in the result if necessary.

A context $C$ is an expression from $L_{\text{need}}$ 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 $C$ are defined by the following grammar:

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

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

**Definition 2.2** The unrestricted reduction rules for the calculus $L_{\text{need}}$ 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).

![Figure 1](image-url)

The reduction rules of $L_{\text{need}}$ contain different kinds of meta-variables. The meta-variables $r, s, s_i, t$ denote arbitrary $L_{\text{need}}$-expressions. $Env, Env_1, Env_2$ represent letrec-environments and $x, y$ denote bound
variables. All meta-variables can be instantiated by an $L_{\text{need}}$-expression of the appropriate syntactical form. A reduction rule $\rho = l \rightarrow r$ 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.

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

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

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.

![Figure 2: Normal order reduction rules of $L_{\text{need}}$](image)

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

The transitive closure of the reduction relation $\rightarrow$ is denoted as $\Rightarrow$ and the transitive and reflexive closure of $\rightarrow$ is denoted as $\Rightarrow^*$. Respectively we use $\xRightarrow{no,\rightarrow}$ for the transitive closure of the normal order reduction relation, $\xRightarrow{no,s}$ 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 $t \xRightarrow{no,s} t'$ 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 $L_{\text{need}}$ is the equality of expressions defined by contextual equivalence.
Definition 2.5 (Contextual Preorder and Equivalence) Let \( s, t \) be \( L_{\text{need}} \)-expressions. Then:

\[
\begin{align*}
    s \leq c t & \iff \forall C : C[s]\Downarrow \Rightarrow C[t]\Downarrow \\
    s \sim c t & \iff s \leq c t \land t \leq c s
\end{align*}
\]

Definition 2.6 A program transformation \( T \subseteq L_{\text{need}} \times L_{\text{need}} \) is a binary relation on \( L_{\text{need}} \)-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.

An important tool to prove contextual equivalence is a context lemma (see for example [9], [13], [15]), which allows to restrict the class of contexts that have to be considered in the definition of the contextual equivalence from general \( C \) to \( \mathbb{R} \)-contexts.

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

Lemma 2.7 Let \( s, t \) be \( L_{\text{need}} \)-expressions and \( S \) a context of class \( \mathbb{S} \). \((S[s]\Downarrow \Rightarrow S[t]\Downarrow) \iff \forall C : (C[s]\Downarrow \Rightarrow C[t]\Downarrow) \); i.e. \( s \leq c t \).

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

To prove the correctness of a transformation \( s \xrightarrow{T} t \) we have to prove that \( s \sim_c t \iff s \leq c t \land t \leq c s \) which by Definition 2.5 amounts to showing \( \forall C : C[s]\Downarrow \Rightarrow C[t]\Downarrow \land C[t]\Downarrow \Rightarrow C[s]\Downarrow \). The context lemma yields that it is sufficient to show \( \forall S : S[s]\Downarrow \Rightarrow S[t]\Downarrow \land S[t]\Downarrow \Rightarrow S[s]\Downarrow \). 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 \( S[s] \xrightarrow{\text{no},k} s' \) (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 \( S[s] \xrightarrow{\text{no},k} s' \) using complete sets of reduction diagrams. Therefore we split \( S[s] \xrightarrow{\text{no},k} s' \) into \( S[s] \xrightarrow{\text{no}} S[0] \xrightarrow{\text{no},k-1} s' \) (see Figure 3(b)). Then an applicable forking diagram defines how the fork \( S[0] \xrightarrow{\text{no}} S[s] \xrightarrow{T} S[t] \) can be closed specifying two sequences of transformations such that a common expression \( i' \) is eventually reached: one starting from \( S[0] \) 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).

![Figure 3: Sketch of the correctness proof for \( s \xrightarrow{T} t \)]](image-url)
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.

**Example 2.8** Example forking diagrams are

\[
\begin{align*}
\text{letrec } \text{Env}_1, x &= (\text{letrec } \text{Env}_2 \text{ in } s) \text{ in } (\text{letrec } \text{Env}_3 \text{ in } r) \\
&\xrightarrow{\text{no,let-in}} (\text{letrec } \text{Env}_1, \text{Env}_3, x = (\text{letrec } \text{Env}_2 \text{ in } s) \text{ in } r) \\
&\xrightarrow{iS,\text{let-e}} (\text{letrec } \text{Env}_1, \text{Env}_3, \text{Env}_2, x = s \text{ in } r) \\
\text{the last reduction is either an no-reduction if } r = A[x], \text{ otherwise it is an internal reduction} \\
&\xrightarrow{iS,\text{let-e}} (\text{letrec } \text{Env}_1, \text{Env}_2, x = s \text{ in } (\text{letrec } \text{Env}_3 \text{ in } r)) \\
&\xrightarrow{\text{no,let-in}} (\text{letrec } \text{Env}_1, \text{Env}_2, \text{Env}_3, x = s \text{ in } r)
\end{align*}
\]

The square diagram covers the case, where (no,let-in) is followed by an internal reduction. The triangle diagram covers the other case, where the reduction following (no,let-in) is (no,let-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 $\xrightarrow{\text{no,red}} \frac{iS,T}{\text{red}}$ where red is an no-reduction and $T$ is not a normal order reduction (but a transformation in an $S$-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 2.8 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 2.7) we restrict the overlaps to the transformations applied in surface contexts. A complete description of a single overlap is the unification equation \( S[t_i] = l_{no, j} \), where \( t_i \) 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 \( S = S_1 \uplus S_2 \) be the disjoint union of a set of theory-sorts \( S_1 \) and a set of free sorts \( S_2 \). We assume that \( Exp \) is a sort in \( S_2 \). 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 \( f : S_1 \times \ldots \times S_n \rightarrow S_{n+1} \), 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 \( S_i \). Note, however, that there may be function symbols \( f \in \Sigma_i \) that have argument-sorts from \( S_j \), 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 \( X, Y, Z, \ldots \), 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 \( Exp \rightarrow Exp \). Let \( \mathcal{V} = \mathcal{V}^0 \uplus \mathcal{V}^1 \). The set of terms \( \mathcal{T}(S, \Sigma, \mathcal{V}) \) is the set of terms built according to the grammar \( T(x, \ldots, t_n) \rightarrow X(t) \), 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 \( \mathcal{T}(Exp, \Sigma \cup \{ \}, \mathcal{V}^1) \) such that there is exactly one occurrence of 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 \( \Sigma \) where the resulting sort is the sort of \( x \), and hence no ground term of this sort.

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

3.2 Encoding of \( L_{\text{need}} \)-Expressions as Terms

The sort and term structure according to the expression structure of the lambda calculus \( L_{\text{need}} \) (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 \( S_1 = \{ Env \} \) and \( S_2 = \{ Bind, Exp, BV \} \). There are the following function symbols:

<table>
<thead>
<tr>
<th>theory function symbols (( \Sigma_1 ))</th>
<th>free function symbols (( \Sigma_2 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>emptyEnv :: Env</td>
<td>let :: Env \times Exp \rightarrow Exp</td>
</tr>
<tr>
<td>env :: Bind \times Env \rightarrow Env</td>
<td>app :: Exp \times Exp \rightarrow Exp</td>
</tr>
<tr>
<td></td>
<td>lam :: BV \times Exp \rightarrow Exp</td>
</tr>
<tr>
<td></td>
<td>bind :: BV \times Exp \rightarrow Bind</td>
</tr>
<tr>
<td></td>
<td>var :: BV \rightarrow Exp</td>
</tr>
</tbody>
</table>

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 \( (\text{letrec } x = \lambda y.z \cdot x \times z) \) is encoded as \( \text{let}(env^* (\{ \text{bind}(x, \text{lam}(y, \text{var}(y))), \text{bind}(z, \text{app}(\text{var}(x), \text{var}(z))) \}) \cup \text{emptyEnv}, \text{var}(z)) \), 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 \([5, 4]\)). The equational theory LC is a congruence relation on the terms, which is denoted as \( =_{\text{LC}} \). The pure equational theory is defined as restricted to the axiom-signature, i.e. to the terms \( T(\{ Env, Bind \}, \Sigma_1, \mathcal{V}_{\text{Env}} \cup \mathcal{V}_{\text{Bind}}) \), where \( \mathcal{V}_S \) is the set of variables of sort \( S \). The combined equational theory is defined on the set of terms \( T(S, \Sigma, \mathcal{V}_S) \). 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 \( T(S, \Sigma, \mathcal{V}_S) \):

- The terms in the LC-axioms are built only from \( \Sigma_1 \)-symbols and variables, and the axioms relate two terms of equal sort which must be in \( S_1 \).
- For every equation \( s =_{\text{LC}} t \), the equality \( \text{Var}(s) = \text{Var}(t) \) holds.
- The equational theory LC is non-collapsing, i.e., there is no equation of the form \( x =_{\text{LC}} t \), where \( t \) is not the variable \( x \).
- If \( C[s] =_{\text{LC}} t \) and \( s \) has a free function symbol as top symbol, then there is a context \( C' \) and a term \( s' \) such that \( C[s] =_{\text{LC}} C'[s'] \) 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 \([5, 4]\)).

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 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 BCh\( n_1, n_2 \) for \( n_1, n_2 \geq 1 \) represents the following binding chain:

\[
\text{bind}(y_{n_1+1}, A_{n_1+1}(\text{var}(y_{n_1+1}))), \text{bind}(y_{n_1+2}, A_{n_1+2}(\text{var}(y_{n_1+1}))), \ldots, \text{bind}(y_{n_2}, A_{n_2}(\text{var}(y_{n_2-1})))
\]

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 \( \text{env}^*(\ldots \cup \text{BCh}(N_1, N_2) \cup r) \).

Context-classes are required to correctly model the overlappings in \( L_{\text{need}} \). The transformations in Figure 1 contain only C-contexts, whereas in Figure 2 there are also A- and R-contexts, and the overlapping also requires surface contexts \( S \). The grammar definition of A-, R- and S-contexts (definition 2.3) justifies the replacement of R-contexts by expressions containing only A-contexts and BCh-expressions.
C. Rau and M. Schmidt-Schauß

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

Let \( C \) be decided whether \( S \) and \( A \) context variables. The unification algorithm must know how to deal with context variables of classes \( S \) and \( C \). The partial order on context classes is \( \mathcal{A} < S < C \). For every almost ground context \( C \) it can be decided whether \( C \) belongs to \( \mathcal{A} \) (or \( S \), respectively). We will use the facts that equational deduction w.r.t. \( \mathcal{L} \)C 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 \( \mathcal{A} \), \( S \) and \( C \)).

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 \( (\Gamma, \Delta) \), 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 \( \text{BCh} \)-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 \( \Delta_1 \) and a set \( \Delta_2 \) 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 \( \Delta_1 \) consists of context variables that must not be instantiated by the empty context, and that the constraints \( \Delta_2 \) hold for \( \sigma(N_i) \) after instantiating with \( \sigma \).

A solution \( \sigma \) of \( (\Gamma, \Delta) \), with \( \Gamma = \{s_1 = t_1, \ldots, s_n = t_n\} \) is a substitution \( \sigma \) 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 \( \Delta_1 \), and the integer variables \( N_i \) by positive integers according to the constraint \( \Delta_2 \). ii) \( \sigma(s_i), \sigma(t_i) \) are almost ground for all \( i \). It is assumed that the \( \text{BCh} \)-constructs \( \text{BCh}(n_1, n_2) \) are expanded into a binding chain as explained above, iii) \( \sigma(s_i) =_{\mathcal{L}C} \sigma(t_i) \) for all \( i \).

A unification problem \( \Gamma \) 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 \( \Pi_T \) be the set of left hand sides of reduction rules from Figure 4 and \( \Pi_{no} \) 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_i = A, Env in A[y_i]), (letrec y_N1 = A, BCh(N1, N2), Env in A[y_N1]) 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 \( \text{var}(z) \) and an abstraction \( \lambda.x.t \) where \( t \) denotes a meta-variable for an arbitrary expression. With \( \Pi'_T, \Pi'_{no} \) we denote the sets where left hand sides of rules are encoded as terms.

We consider the set of unification problems \( \Gamma_1 = \{S(l_{T,i}) = l_{no,j} \mid l_{no,j} \in \Pi'_{no}\} \) with \( l_{T,i} \in \Pi_T \) and \( S \) is a surface context variable. The sets \( \Pi'_T \) and \( \Pi'_{no} \) are assumed to be variable disjoint, which can be achieved by renaming. The initial set \( \Delta_1 \) of context variables only contains the \( A_2 \)-context from the (cp-e)-reductions, and \( \Delta_2 \) may contain some initial constraints from the rules. The pairs \( (\Gamma_1, \Delta) \) are called the initial \( L_{\text{need}} \)-forking-problems.

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

**Definition 4.2** A final unification problem \( S \) of an initial \( \Gamma \) is a set of equations \( s_1 \doteq t_1, \ldots, s_n \doteq t_n \), such that \( S = S_{BV} \cup S_{-BV} \), and every equation in \( S_{BV} \) is of the form \( x \doteq y \) where \( x, y \) are of sort \( BV \) and every equation in \( S_{-BV} \) is of the form \( x \doteq t \), where \( x \) is not of sort \( BV \), and the equations in \( S_{-BV} \) are in DAG-solved form.
Given a final unification problem \( S \), the represented solutions \( \sigma \) could be derived by first instantiating the integer variables, expanding the BCh-constructs into binding chains, instantiating all context variables and variables that are not of sort \( BV \) and then turning the equations into substitutions.

A final unification problem \( S \) derived from \( \Gamma \) satisfies the distinct variable convention (DVC), if for every derived solution \( \sigma \), 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 \( \sigma \) 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 \( \text{BCh}(N_1, N_2) \) occurring in \( \sigma(t_1) \) we add the variable \( \gamma_{N_2} \) 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. Thus a variant of our unification can efficiently check alpha-equivalence of lambda-expressions that satisfy the DVC.

We proceed by describing a unification algorithm starting with initial \( L_{need} \)-unification problems \((\Gamma, \Delta)\). 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 \( \Gamma \) in a solved part \( S \), (a final unification problem), and a still to be solved part \( P \). We usually omit \( \Delta \) in the notation if it is not changed by the rule.

**Standard unification rules.**

\[
\begin{align*}
\text{Dec} & \quad S; \{ f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n) \} \cup P \\
& \quad S; \{ s_1 \doteq t_1, \ldots, s_n \doteq t_n \} \cup P & \text{If } f \text{ is a free function symbol (i.e. } f \neq \text{env).}
\end{align*}
\]

\[
\begin{align*}
\text{Solve} & \quad S; \{ x \doteq t \} \cup P \\
& \quad \{ x \doteq t \} \cup S; P & \text{Trivial}
\end{align*}
\]

\[
\begin{align*}
\text{Fail} & \quad S; \{ f(...) \doteq g(...) \} \cup P \\
& \quad \text{DVC-Fail}
\end{align*}
\]

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

**Solving equations with context variables.** The rules for terms with contexts as top symbol using their context classes are as follows: The following rule operates on context variables at any position:

\[
\begin{align*}
\text{Empty-C} & \quad S; P; \Delta_1 \\
& \quad \text{select one of the following possibilities} \\
& \quad S; P; \{ X \} \cup \Delta_1 \quad \text{or} \quad \{ X \mapsto [ ] \} \cup S; \{ X \mapsto [ ] \}; P; \Delta_1
\end{align*}
\]

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 \( \text{Exp} \). There are the following possibilities:

\[
\begin{align*}
\text{Dec-CA} & \quad S; \{ X(s) \doteq \text{app}(t_1, t_2) \} \cup P \\
& \quad \{ X \mapsto \text{app}(X', t_2) \} \cup S; \{ X'(s) \doteq t_1 \} \cup P \\
& \quad X' \text{ is a fresh context variable of the same context class as } X.
\end{align*}
\]
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:

**Dec-CC**

$$S; \{X(s) \doteq f(t_1, t_2)\} \Downarrow P$$

$$X \mapsto f(t_1, X') \cup S; \{X'(s) \doteq t_2\} \cup P$$

if $t_2$ is of sort $\text{Exp}$.

$X'$ 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 $f \in \{\text{let}, \text{app}\}$.

**Dec-CL**

$$S; \{X(s) \doteq \text{let}(t_1, t_2)\} \Downarrow P$$

$$X \mapsto \text{env}^*(\{\text{bind}(x, X')\} \cup z, t_2) \cup S; \{\text{env}^*(\{\text{bind}(x, X'(s))\} \cup z) \doteq t_1\} \cup P$$

If $X$ is of context class $S$ or $C$, $X'$ is a fresh context variable of the same context class as $X$.

**Dec-Lam**

$$S; \{X(s) \doteq \lambda(t_1, t_2)\} \Downarrow P$$

$$X \mapsto \text{lam}(t_1, X') \cup S; \{X'(s) \doteq t_2\} \cup P$$

If $X$ is of class $C$, $X'$ is a fresh context variable of the class $C$.

**Fail-Lam**

$$S; \{X(s) \doteq \lambda(t_1, t_2)\} \Downarrow P$$

**Fail-Var**

$$S; \{X(s) \doteq \text{var}(x)\} \Downarrow P$$

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-P**

$$S; \{X(s) \doteq Y(t)\} \Downarrow P$$

$$Y \mapsto Z Y', X \mapsto Z \cup S; \{s \doteq Y'(t)\} \cup P$$

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

**Merge-FA**

$$S; \{X(s) \doteq Y(t)\} \Downarrow P$$

$$X \mapsto Z(\text{app}(X', Y'(t))), Y \mapsto Z(\text{app}(X'(s), Y')) \cup S; P$$

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-FC**

$$S; \{X(s) \doteq Y(t)\} \Downarrow P$$

choose either of the following possibilities

$$\{X \mapsto Z(\text{app}(X', Y'(t))), Y \mapsto Z(\text{app}(X'(s), Y'))\} \cup S; P$$

$$\{X \mapsto Z(\text{let}(\text{env}^*(\{\text{bind}(x, X')\} \cup z, Y'(t))), Y \mapsto Z(\text{let}(\text{env}^*(\{\text{bind}(x, X'(s))\} \cup z, Y'))\} \cup S; P$$

$$\{X \mapsto Z(\text{let}(\text{env}^*(\{\text{bind}(x, X'), \text{bind}(y, Y'(t))\} \cup z, w)), Y \mapsto Z(\text{let}(\text{env}^*(\{\text{bind}(x, X'(s)), \text{bind}(y, Y')\} \cup z, w))\} \cup S; P$$

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 $\text{Env}$, i.e., proper multi-set-equations, which must be of the form $\text{env}^*(L_1 \cup r_1) \doteq \text{env}^*(L_2 \cup r_2)$, where $r_1, r_2$ are variables or the constant $\text{emptyEnv}$. We will use the notation $L$ for sub-lists in $\text{env}^*$-expressions and the notation $L_1 \cup L_2$ for union. In the terms $\text{env}^*(L \cup t)$, we assume that $t$ is not of the form $\text{env}(\ldots)$. 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 $\text{emptyEnv}$. The components in the multi-set may be expressions of type $\text{Bind}$, i.e., variables or expressions with top symbol $\text{bind}$, or a $\text{BCh}(...)$-component that represents several terms of type $\text{Bind}$. We also use the
convention that in the conclusions of the rules an empty environment $env^r(\{ \} \cup r)$ 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 $BCh(N_1,N_2)$-construct occurs in a term in $P$, it occurs in an $env^r$-list, hence there is also a binding $y_{N_i} = s$ in the $env^r$-list, and the list is terminated with a variable derived from the environment-variable $Env$. In equations, the $BCh(\ldots)$-components initially appear only on one side, which cannot be changed by the unification. Also the $env^r$-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 $BCh(\ldots)$-components.

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

\begin{align*}
\text{Solve-E} & \quad S; \{ env^r(L_1 \cup r_1) \equiv env^r(L_2 \cup r_2) \} \cup P \quad \text{if } r_1, r_2 \text{ are variables; } z_3 \text{ is a fresh variable.} \\
\text{Dec-E} & \quad S; \{ env^r(L_1 \cup r_1) \equiv env^r(L_2 \cup r_2) \} \cup P \\
& \quad S; \{ t_1 \equiv t_2, env^r(L_1 \setminus \{ t_1 \} \cup r_1) \equiv env^r(L_2 \setminus \{ t_2 \} \cup r_2) \} \cup P \\
& \quad \text{if } L_1 \text{ and } L_2 \text{ contain binding expressions } t_1 \text{ and } t_2.
\end{align*}

Dec-Ch

\[ S; \{ env^r(L_1 \cup r_1) \equiv env^r(BCh(N_1,N_2) \cup L_2 \cup r_2) \} \cup P; (\Delta_1, \{ N_1 < N_2 \} \cup \Delta_2) \]

select one of the following possibilities

(i) $S; \{ t_1 \equiv bind(y_{N_2},A_{N_2}(\text{var}(y_{N_1}))), env^r(L_1 \setminus \{ t_1 \} \cup r_1) \equiv env^r(L_2 \cup r_2) \} \cup P; \{ A_{N_2} \} \cup \Delta_1, \{ N_1 + 1 = N_2 \} \cup \Delta_2$

(ii) $S; \{ t_1 \equiv bind(y_{N_2},A_{N_2}(\text{var}(y_{N_1}))), env^r(L_1 \setminus \{ t_1 \} \cup r_1) \equiv env^r(BCh(N_3,N_2) \cup L_2 \cup r_2) \} \cup P; \{ A_{N_2} \} \cup \Delta_1, \{ N_1 + 1 = N_3, N_3 < N_2 \} \cup \Delta_2$

(iii) $S; \{ t_1 \equiv bind(y_{N_2},A_{N_2}(\text{var}(y_{N_1}))), env^r(L_1 \setminus \{ t_1 \} \cup r_1) \equiv env^r(BCh(N_1,N_3) \cup BCh(N_4,N_2) \cup L_2 \cup r_2) \} \cup P; \{ A_{N_2} \} \cup \Delta_1, \{ N_1 < N_3, N_3 + 1 = N_2 \} \cup \Delta_2$

(iv) $S; \{ t_1 \equiv bind(y_{N_2},A_{N_2}(\text{var}(y_{N_1}))), env^r(L_1 \setminus \{ t_1 \} \cup r_1) \equiv env^r(BCh(N_1,N_3) \cup BCh(N_4,N_2) \cup L_2 \cup r_2) \} \cup P; \{ A_{N_2} \} \cup \Delta_1, \{ N_1 < N_3, N_3 + 1 = N_4, N_4 < N_2 \} \cup \Delta_2$

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.

\begin{align*}
\text{Fail-E} & \quad S; \{ env^r(L \cup t) \equiv \text{emptyEnv} \} \cup P \quad \text{if } L \text{ is nonempty, i.e contains at least one binding or at least one BCh-expression.}
\end{align*}

An invariant of the rules that deal with $BCh$ is that the variables $N_i$ may appear at most twice in $\Gamma$; at most twice explicit in $\Delta_2$ and at most once in $BCh$-expressions.

\subsection{Properties of the LCSX-Unification Algorithm}

\textbf{Lemma 4.4} For initial problems, the algorithm LCSX terminates.

\textit{Proof.} For this we can ignore the rules that change $\Delta$.

The following measure is used, which is a lexicographical combination of several component measures: $\mu_1$ is the number of occurrences of \textit{let} in $P$; the second component $\mu_2$ is the following size-measure, where $env^r(L \cup r)$ has measure $7m + m' + \sum \mu_2(t_i) + \mu_2(r)$ where $m$ is the number of \textit{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-Ch 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.

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

\[ \text{Proof.} \quad \text{Soundness can be proved by standard methods, since rules are either instantiations or instantiations using the theory LC.} \]

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 \( x \equiv t \)) are just marked as such, i.e. moved to a set of solved equations. Solving equations \( X(s) \equiv t \) 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 \( s \equiv s \), then it will be removed, and if it is of the form \( f(...) \equiv f(...) \) 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 [5]), we see that the rules are complete. \]

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

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

\section{Running the Unification Algorithm LCSX}

\[ \text{Example 5.1} \quad \text{The goal is to compute a complete set of forks for the transformation (cp-e)} \]

\[ \text{(letrec } x = s, \text{Env}, \text{z = } C[x] \text{ in r)} \rightarrow \text{(letrec } x = s, \text{Env}, \text{z = } C[s] \text{ in r)} \]

\[ \text{from Figure 4] We instantiate the meta-variable } s \text{ by the expression } \lambda w.t \text{ and translate the left hand side of the rule into the term language, resulting in the following initial forking problem to be solved} \]

\[ \{S(\text{let(env*)} \{\{\text{bind}(x, \text{lam}(w,t)), \text{bind}(z, C(\text{var}(x)))\} \cup \text{Env}, r)) \equiv l_{no,j}\} \]

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

\[ S(\text{let(env*)} \{\{\text{bind}(x, \text{lam}(w,t)), \text{bind}(z, C(\text{var}(x)))\} \cup \text{Env}_1, r)) \]

\[ \equiv \text{let(env*)} \{\{\text{bind}(x', \text{lam}(w',t')), \text{bind}(y_{N_1}, A_{N_1}(\text{var}(x')))\} \cup \text{BCh}(N_1, N_2) \cup \text{Env}_2, A(y_{N_2})\} \]

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 = [\] via the Empty-C-rule and then using decomposition for let which leads to \( r = A(y_{N_2}) \) and the equation
Dec-Ch

More information can be found in [10].

searches for closing reduction sequences. Via this method we were able to close (almost transformations and no-reductions. The program computes 1214 overlaps for the it is sufficient to compute final systems. The (possibly infinite) set of ground solutions is not required.

Dec-E

Merge-FA

Dec steps, two for bind and one for lam) we arrive at the system

C(var(x)) = A_N \{var(Y_N)\}, z = y_N, x = x, w = w, t = t', Env_2 = env^\ast({\text{bind}(x, \text{lam}(w, t))}) \cup Env_3, Env_1 = env^\ast({\text{bind}(x, \text{lam}(w', t'))}) \cup BCh(N_1, N_2) \cup BCh(N_4, N_2) \cup Env_3).

Next we apply Merge-FA to the first equation, yielding

C = Z(app(A'_N, var(Y_N)), C'), A_N \doteq Z(app(A'_N, C'(var(x))))

where Z, A'_N are of context class A and C' is of context class C. The final representation is:

S_{\text{BV}} = \{ S \doteq [], r \doteq A(y_N), C \doteq Z(\ldots), t \doteq t', A_N \doteq Z(\ldots), Env_2 \doteq env^\ast(\ldots), Env_1 \doteq env^\ast(\ldots) \}

\text{S_{\text{BV}} = \{ z \doteq y_N, x \doteq x', w \doteq w' \}}

The resulting expression is:

letrec x' = (\lambda w'. t'), Y_N = A_N \{x'\}, \{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_1},

Y_N = Z(app(A'_N, var(Y_N)), C'[x'])), \{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_2}, Env_2 \in A[Y_N]

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

\text{letrec x' = (\lambda w'. t'), Y_N = A_N \{x'\},

\{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_1}, Y_N = Z(app(A'_N, var(Y_N)), C'[x'])), \{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_2}, Env_2 \in A[Y_N]}

\text{letrec x' = (\lambda w'. t'), Y_N = A_N \{x'\}, \{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_1}, Y_N = Z(app(A'_N, var(Y_N)), C'[x'])), \{ y_{i+1} = A_{i+1}[y_i] \}_{i=1}^{N_2}, Env_2 \in A[Y_N]}

This fork can be closed by the sequence \text{No.cp-e-c} \rightarrow \text{Is.cp-e-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.

We implemented the presented unification algorithm LCSX in Haskell to compute all forks between transformations and no-reductions. The program computes 1214 overlaps for the L_{\text{need}} calculus, and also searches for closing reduction sequences. Via this method we were able to close (almost) all forks. The complete sets of forking diagrams for the transformations llet and cp is in Figure 4. The implementation is available at: http://www.ki.informatik.uni-frankfurt.de/research/dfg-diagram/en. More information can be found in [10].

\text{1Some simple commuting diagrams for cp reductions are not automatically closed, due to renaming of bound variables.}
6 Conclusion and Further Work

We have provided a 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 rules, and to calculi with data structures and case-expressions.

References