Towards Correctness of Program Transformations Through Unification and Critical Pair Computation

Conrad Rau and Manfred Schmidt-Schauß

Institut für Informatik
Johann Wolfgang 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

June 12, 2010

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 is the combination of a context lemma with the computation of overlaps between program transformations and the reduction rules, which results in 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 of an application we describe a finitary and decidable unification algorithm for the combination of the equational theory of left-commutativity modelling multi-sets, context variables and many-sorted unification. Sets of equations are restricted to be almost linear, i.e. every variable and context variable occurs at most once, where we allow one exception: variables of a sort without ground terms may occur several times. Every context variable must have an argument-sort in the free part of the signature. We also extend the unification algorithm by the treatment of binding-chains in let- and letrec-environments and by context-classes. This results in a unification algorithm that can be applied to all overlaps of normal-order reductions and transformations in an extended lambda calculus with letrec that we use as a case study.

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 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 $\rightarrow$ that represents a single execution step on programs (perhaps with environment), 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 $P_1 \sim P_2 \iff \big($ for all $C : C[P_1]\Downarrow \iff C[P_2]\Downarrow\big)$, where $C[\cdot]$ 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 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 [Mil77], ciu-lemmas [FH92] and bisimulation, see e.g. [How89]. A context lemma usually shows that $R[P_1]\Downarrow \iff R[P_2]\Downarrow$ for a restricted set of reduction contexts R is sufficient to show equivalence of P_1 and P_2.
The reduction relation $\xrightarrow{*}$ is often given as a set of rules $l_i \to r_i$, where l_i, r_i are like rewriting rules, but extended with pattern 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 $\sigma(s_1 \to s_2)$, i.e. $C[\sigma(s_1)]\Downarrow \iff C[\sigma(s_2)]\Downarrow$ for all contexts $C[\cdot]$. 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 Knuth-Bendix method [KB70], but has to be adapted to an asymmetric situation, to extended instantiations and to higher-order terms. How to apply this method (with ad-hoc arguments) to an extended lambda-calculus is worked out e.g. in [SSSS08]

2 Application to a Small Extended Lambda-Calculus

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 [SS07]. Calculi that are related are in [SSSM10a,SSSM10b]. Also the paper [AF97] contains a related letrec-calculus.
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 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 $1 \le i \le n$. 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, 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 subexpression) in C. A formal definition is:

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

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

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.

(lbeta)	$((\lambda x.s)\ r) \rightarrow (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[x]) \rightarrow (\texttt{letrec}\ x = s, Env\ \ \texttt{in}\ C[s])$ where s is an abstraction or a variable
(cp-e)	$(\texttt{letrec}\ x = s, Env, y = C[x]\ \texttt{in}\ r) \rightarrow (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r)$ where s is an abstraction or a variable
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)) \rightarrow (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)\ \texttt{in}\ r)$ $\rightarrow (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$

Fig. 1. Reduction rules of L_{need}

Definition 2.2. *The* reduction rules *for the calculus L_{need} are defined in Fig. 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 (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 (meta-) bound variables. All meta-variables can be instantiated by a L_{need}-expression of the appropriate sort. A reduction rule $\rho = l \rightarrow r$ is applicable to an expression e if l can be matched to e, i.e., there is a substitution σ such that $\sigma(l) = e$. The result of the reduction is $\sigma(r)$. Note that an expression may contain several sub-expressions that can be reduced according to the reduction rules of Fig. 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 $\mathcal{R}$ *and application context* $\mathcal{A}$ *are defined by the following grammar:*

$$\begin{aligned} R \in \mathcal{R} &:= A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \\ &\mid \texttt{letrec}\ x_1 = A_1, x_2 = A_2[x_1], \ldots, x_n = A_n[x_{n-1}], Env\ \texttt{in}\ A[x_n] \\ A \in \mathcal{A} &:= [\cdot] \mid (A\ s) \qquad \textit{where } s \textit{ is an expression.} \end{aligned}$$

where x_i are variables, $A_2, \ldots, A_n$ are not the empty context and A, A_i are $\mathcal{A}$-contexts.

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

(lbeta)	$R[(\lambda x.s)\ r] \rightarrow R[\texttt{letrec}\ x = r\ \texttt{in}\ s]$
(cp-in)	$\texttt{letrec}\ y = s, Env\ \texttt{in}\ A[y] \rightarrow \texttt{letrec}\ y = s, Env\ \texttt{in}\ A[s]$ where s is an abstraction or a variable
(cp-e)	$\texttt{letrec}\ y_1 = s, y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], Env\ \texttt{in}\ A[y_n]$ $\rightarrow \texttt{letrec}\ y_1 = s, y_2 = A_2[s], \ldots, y_n = A_n[y_{n-1}], Env\ \texttt{in}\ A[y_n]$ where s is an abstraction or a variable, and $A_2, \ldots, A_n$ are non-empty $\mathcal{A}$-contexts
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)) \rightarrow (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_1\ \texttt{in}\ r), y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], Env_2\ \texttt{in}\ A[y_n]$ $\rightarrow \texttt{letrec}\ y_1 = r, Env_1, y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], Env_2\ \texttt{in}\ A[y_n]$ where $A_2, \ldots, A_n$ are non-empty $\mathcal{A}$-contexts
(lapp)	$R[((\texttt{letrec}\ Env\ \texttt{in}\ r)\ t)] \rightarrow R[(\texttt{letrec}\ Env\ \texttt{in}\ (r\ t))]$

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

The *transitive closure* of the reduction relation $\rightarrow$ is denoted as $\xrightarrow{+}$ and the *transitive and reflexive closure* of $\rightarrow$ is denoted as $\xrightarrow{*}$. Respectively we use $\xrightarrow{no,+}$ for the transitive closure of the normal order reduction relation and $\xrightarrow{no,*}$ for its reflexive-transitive closure.
Note that the normal order reduction is unique. A *weak head normal form in L_{need} (WHNF)* is either an abstraction $\lambda x.s$, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ \lambda x.s)$.
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:*

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

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 $T \subseteq \sim_c$.*

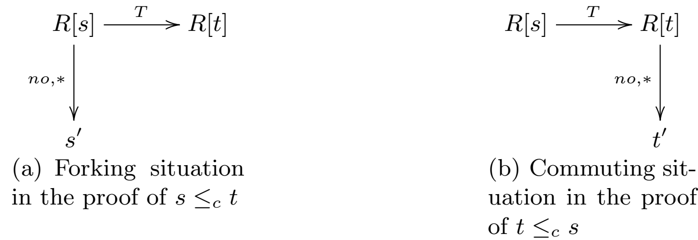
Program transformations are usually given in a format similarly to reduction rules (see Fig. 1 and Fig. 2). A program transformation T is written as $s \xrightarrow{T} t$ where s, t are "meta-expressions" i.e. expression that contain meta-variables. Note that in this setting the reduction rules of Fig. 1 are also regarded as program transformations, where the application is in any context. If necessary, the context in which a reduction rule is applied is annotated at the transformation. For example, $\xrightarrow{\mathcal{R},T}$ indicates that the transformation T is applied in a reduction context.
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 [Mil77], [SSS10],[SSSS08]), 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.

Lemma 2.8. *Let s, t be L_{need}-expressions. If for all reduction contexts R: $(R[s]{\Downarrow} \Rightarrow R[t]{\Downarrow})$, then $\forall C$: $(C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow})$; i.e. $s \leq_c t$.*

To prove the correctness of a transformation $s \xrightarrow{T} t$ one has 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 $\forall C[\cdot] : C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow}\ \wedge\ C[t]{\Downarrow} \Rightarrow C[s]{\Downarrow}$. The context lemma yields that it is sufficient to show $\forall R[\cdot] : R[s]{\Downarrow} \Rightarrow R[t]{\Downarrow}\ \wedge\ R[t]{\Downarrow} \Rightarrow R[s]{\Downarrow}$. To prove $s \sim_c t$ we assume that $s \xrightarrow{T} t$ and $R[s]{\Downarrow}$ holds, i.e. there is a WHNF s', such that $R[s] \xrightarrow{no,*} s'$ (respectively $s \xrightarrow{T} t$ and $R[t]{\Downarrow}$ holds, i.e. there is a WHNF t', such that $R[t] \xrightarrow{no,*} t'$). This situation is illustrated in Fig. 3.



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

(b) Commuting situation in the proof of $t \leq_c s$

Fig. 3. The situation in the correctness proof of $s \xrightarrow{T} t$

It remains to show that there also exists a sequence of normal order reductions from $R[t]$ (respectively from $R[s]$) to a WHNF. This can often be done by induction on the length of the given normal order reduction sequence using *sets of complete reduction diagrams.*

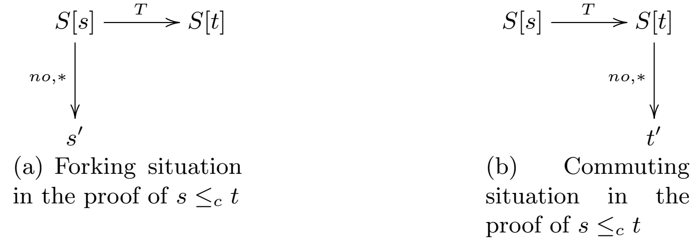
2.2 Complete Sets of Forking and Commuting Diagrams

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 $i\mathcal{S}$, where surface contexts are defined as follows:

$$\begin{array}{l} S \in \mathcal{S} ::= [\cdot] \mid (S\ s) \mid (s\ S) \mid\ (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ S) \\ \qquad\qquad \mid\ (\texttt{letrec}\ Env, x = S\ \texttt{in}\ s) \end{array}$$

In the following we use a slightly weaker context lemma for $\mathcal{S}$-contexts: to show equivalence $s \sim_c t$, it is sufficient to show $\forall S[\cdot] : S[s]\Downarrow \Rightarrow S[t]\Downarrow\ \wedge\ S[t]\Downarrow \Rightarrow S[s]\Downarrow$ for all surface contexts S. This will be helpful in closing the forking diagrams, since there are cases where the forking w.r.t. $\mathcal{R}$-contexts can only be closed with transformations in $\mathcal{S}$-contexts. The situation in the proof using surface contexts is illustrated in Fig. 4.



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

(b) Commuting situation in the proof of $s \leq_c t$

Fig. 4. part of the correctness proof for $\mathcal{S}$ of $s \xrightarrow{T} t$

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 $iX \in \{iC, i\mathcal{R}, i\mathcal{S}\}$, 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 $s \xrightarrow{iX,T} t_1 \xrightarrow{no,a_1} \ldots t_k \xrightarrow{no,a_k,} t$ of the reduction sequence RED iff the following holds:

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

The transformation consists in replacing this prefix with the result:

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

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*

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

where $k, k' \geq 0, m \geq 1, h > 1$, *such that for every reduction sequence* $t_h \xleftarrow{no} \ldots t_2 \xleftarrow{no} t_1 \xrightarrow{iX,T} t_0$, *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*

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

where $k, k' \geq 0, m \geq 1, h > 1$, *such that for every reduction sequence* $t_0 \xrightarrow{iX,T} t_1 \xrightarrow{no} \ldots \xrightarrow{no} t_h$, *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 $\xleftarrow{no,a} \cdot \xrightarrow{i\mathcal{C},llet} \quad \rightsquigarrow \quad \xrightarrow{i\mathcal{C},llet} \cdot \xleftarrow{no,a}$ is represented as

$$\begin{array}{ccc} \cdot & \xrightarrow{i\mathcal{C},llet} & \cdot \\ {\scriptstyle no,a}\downarrow & & \downarrow{\scriptstyle no,a} \\ \cdot & \dashrightarrow_{i\mathcal{C},llet} & \cdot \end{array}$$

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. An example of a single forking diagram is $\xleftarrow{no,llet-in} . \xrightarrow{i\mathcal{S},llet-e} \rightsquigarrow \xrightarrow{i\mathcal{S},llet-e} . \xleftarrow{no,llet-in}$ which we usually present as

$$\begin{array}{ccc} \cdot & \xrightarrow{i\mathcal{S},llet-e} & \cdot \\ {\scriptstyle no,llet-in}\downarrow & & \downarrow{\scriptstyle no,llet-in} \\ \cdot & \dashrightarrow_{i\mathcal{S},llet-e} & \cdot \end{array}$$

where the dashed lines indicate existentially quantified reductions.
By application of the diagram a fork between a (*no,llet-e*) an internal (*llet-in*) reduction can be closed. The forking diagram specifies two reduction sequences such that a common expression is eventually reached. The following illustrates an example application of the above diagram.

$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r))$

$\xrightarrow{no,llet-in} (\texttt{letrec}\ Env_1, Env_3, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s)\ \texttt{in}\ r)$

$\xrightarrow{i\mathcal{S} \vee no,llet-e} (\texttt{letrec}\ Env_1, Env_3, Env_2, x = s\ \texttt{in}\ r)$

the last reduction is either a no reducion if $r = A[x]$, otherwise it is a internal reduction

$\xrightarrow{i\mathcal{S},llet-e} (\texttt{letrec}\ Env_1, Env_2, x = s\ \texttt{in}\ (\texttt{letrec}\ Env_3\ \texttt{in}\ r))$

$\xrightarrow{no,llet-in} (\texttt{letrec}\ Env_1, Env_2, Env_3, x = s\ \texttt{in}\ r)$

One can view the diagram as a description of local confluence.
Notice the following: By means of the Context Lemma 2.8 it is sufficient to consider only internal reductions in $\mathcal{R}$-context. Nevertheless to close the fork in the above example it is in some cases necessary to apply a (*llet-e*) reduction in an $\mathcal{S}$-context. Therefore we often treat the slightly more general situation where a internal reduction occurs in a surface context to be able to close all diagrams.

The generation 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. We restrict our attention here to complete sets of forking diagrams because complete sets of commuting diagrams can usually be derived from them. Another reason for this restriction is that the fork of the internal reduction (or program transformation) with a no-reduction can be described by an overlap and all overlaps can be computed by (a variant of) critical pair computation via unification.

By Definition 2.9 a complete set of forking diagrams for an internal reduction $\xrightarrow{iX,T}$ comprises a set of transformation rules on reduction sequences, 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. Therefore to compute a complete set of forking diagrams for a fixed internal reduction (transformation) T we have to determine all forks of the form $\xleftarrow{no,red} \cdot \xrightarrow{T}$ where a is an no-reduction. The forks of this form are given by *overlaps* between no-reductions and the internal reduction. Informally we say that two reductions (transformations) *red* and T overlap in an expression s if s contains a *red* and a T redex. To find an overlap between a no-reduction *red* and an internal reduction T we have to determine all positions in *red* where a T-redex can occur (this follows from the uniqueness of the normal order reduction). For the computation of forks it is sufficient to consider only *critical overlaps* where an overlap does not occur at a variable position because forks described by such non-critical overlaps can always be closed by standard diagrams (Example 2.10 illustrates such a critical overlap). All critical overlaps between no-reductions *red* and a given internal reduction T can be computed by unification. The employed unification procedure will be explained in the next section.

3 A Unification Algorithm for Terms in a Combination of Sorted Equational Theories and Context

In this section we develop a unification method to compute the proper overlaps between left hand sides of normal-order reduction rules from Fig. 2 and left hand sides of transformation rules from Fig. 1. According to the context lemma for surface contexts we restrict the overlaps to the transformations in surface contexts. A complete description of the overlap is the unification equation $S[l_{T,i}] \doteq l_{no,j}$, where $l_{T,i}$ is a left hand side in Fig. 1, and $l_{no,j}$ a left hand side in Fig. 2, and S means a surface context.
We develop a unification algorithm that is applicable to terms in a combination of (restricted) equational theories with free function symbols in a many-sorted signature. The terms to be unified may contain context variables. In addition there will be context-classes which are partially ordered. As a condition for the soundness and completeness several service algorithms for context-classes are required. We will develop this in a general way in order to be able to apply this method also to other extended lambda-calculi.
The combination algorithms in [SS89,BS92] could be used. However, we only consider the theory LC that models multi-sets of bindings in letrec environments of our calculus. This theory is very special and thus there is more information on unifiers, which allows to design specialized unification rules.

3.1 Signatures, Terms and Equational Theories

Let $\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2$ be a set of theory-sorts $\mathcal{S}_1$ and free sorts $\mathcal{S}_2$. 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. 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 for all sorts S_1, S_2, there is an infinite number of context variables of sort $S_1 \to S_2$. 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 $\mathcal{T}(\mathcal{S}, \Sigma \cup \mathcal{H}, \mathcal{V})$ where $\mathcal{H} := \{[\cdot]^S \mid S \in \mathcal{S}\}$, such that there is exactly one occurrence of a hole in the context. If C is of sort S_2 and has a hole of sort S_1, then $C :: S_1 \to S_2$.
A substitution σ is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V})$, such that $\sigma(x^S)$ is a term of sort S and $\sigma(X)$ is a context of the same sort as X. Usually, we also write σ also for the mapping on terms, where every variable x in a term is replaced by $\sigma(x)$.
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.
An equational theory E is a congruence relation on $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V})$, which is denoted as $=_E$. It could also be restricted to parts of the signature. The equational theory is given by a (finite) set of axioms, which are pairs of terms of the same sort. We assume that the axioms do not contain context variables. In the usual way, the relation $=_E$ is defined as the equivalence-closure of the rewrite relation defined by the axioms. We also do not distinguish notationally between the axioms and the equational theory.

Definition 3.1. *The* pure equational theory *is defined as restricted to the local signature, i.e. to the terms $\mathcal{T}(\mathcal{S}', \Sigma_1, \mathcal{V}'\}$, where $\mathcal{S}'$ is the set of sorts that occurs as sort of subterms in the arities of theory-symbols and where $\mathcal{V}'$ is the set of variables of sorts in $\mathcal{S}'$.*
The combined equational theory *is defined on the set of terms $\mathcal{T}(\mathcal{S}, \Sigma, \mathcal{V})$, using the same set of axioms. Note that the combination is disjoint on the function symbols, but not on the sort-structure. The encoding of expression in Subsection 4.1 for example has a non-disjoint sort-structure.*

Since the unification method only works with the special theory of left-commutativity for representing multi-sets, we restrict equational theories now to to left-commutativity.

Assumption 3.2 *The equational theory E that we will use in the algorithm below is* left-commutativity *(LC) with the following sorts, symbols and axioms:*

- *The sorts are Env, and Bind, for environment and binding, where $\mathcal{S}_1 = \{Env\}$.*
- *There are the symbols*

$$\begin{array}{ll} emptyEnv :: & Env \\ env :: & Bind \times Env \to Env \end{array}$$

- *There is only one axiom: the* left-commutativity axiom

$$env(x, env(y, z)) = env(y, env(x, z))$$

  which is of theory sort Env.
- *The signature of free symbols may be extended, but there is one restriction: free function symbol with resulting sort Env are forbidden.*
  Note that we allow free function symbols that map from Env to other sorts.

It is convenient to have a notation for nested *env*-expressions: $env^*(t_1, \ldots, t_m, r)$ 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*, only the constant *emptyEnv*, variables and terms of the form $X(s)$ are possible.
The following facts can easily be verified for the theory LC:

Lemma 3.3. *Let $E = LC$. Then*

- *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 $s =_E t$, the equality $Var(s) = Var(t)$ holds.*
- *There is no equation $s =_E t$, where s is a proper subterm of t.*
- *The equation $x =_E y$ for variables x, y implies that x and y are the same variable.*

- *The equational theory E is non-collapsing, i.e, there is no equation of the form $x =_E t$, where t is not the variable x.*
- *The occurs-check is applicable in unification problems, i.e. $x \doteq t$ for $t \neq x$ is not solvable, i.e. there is no substitution σ with $\sigma(x) =_E \sigma(t)$.*
- *The equational theory has a finitary and decidable unification problem (see [DPR06,DPR98,DV99]).*

A *unification problem* is a set of equations $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, such that s_i and t_i are of the same sort for every i. A unifier σ of $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a substitution σ, such that $\sigma(s_i) =_E \sigma(t_i)$ for all i. It is called a *solution*, when $\sigma(s_i), \sigma(t_i)$ are almost ground, i.e.: only variables of sorts that have no ground terms are permitted.

Definition 3.4. *A unification problem Γ is called* almost linear*, if the following holds*

1. *every context variable occurs at most once,*
2. *every variable of a non-empty sort occurs at most once,*

Complete sets of unifiers and sets of most general unifiers are defined as usual, but only covering the unifiers such that $\sigma(s_i), \sigma(t_i)$ are almost ground (see handbook articles like [BS01]).

3.2 Properties of the Equational Theory

The analysis of the pure and combined equational theory $E = LC$ can be done by elementary methods like induction on the number of equational rewriting steps. It is also possible to adapt the construction of the amalgamated product of two equational theories (see [SS89,BS92]).
First we analyze the combined equational theory. Terms with top symbol from Σ_i are called i-terms for $i = 1, 2$. Variables are neither 1-terms nor 2-terms. A maximal (w.r.t. the subterm-ordering) i-subterm in a j-term t where $i \neq j$ is called an *alien* subterm. The equational theory according to the conditions above has the following properties, which are provable by induction on the length of equational deductions.

1. If $s =_E t$, then s, t are either both the same variable, or s, t are i-terms for some $i \in \{1, 2\}$.
2. If $s =_E t$, and s, t are not variables, then the sets of alien subterms are equal modulo $=_E$: I.e. for every alien subterm r of s, there is an alien subterm r' of t with $r =_E r'$ and vice versa.
3. The combined theory $=_{LC}$ permits an occurs-check, since the pure equational theory permits an occurs-check.
4. If $C[s] =_E t$ and s is a 2-term, i.e. with a free function symbol as top symbol, then there is a context C' and a term s' such that $C[s] =_E C'[s'], C' =_E C$, $s =_E s'$ and $C'[s'] = t$. This follows from the properties of LC, in particular the linearity of axioms of LC is required. Note that contexts remain contexts under equational reasoning w.r.t. LC.

3.3 Context-Subclasses and Unification

A context-class and subclass mechanism is required for a wide-range application of our combined unification algorithm.
There is a set of context class-symbols, say $\mathcal{C}_1, \ldots, \mathcal{C}_n$, where the semantics is a set of contexts, usually defined syntactically, such that for every (almost ground) context C it can be decided whether C belongs to $\mathcal{C}_i$. We also assume that there is a partial order on the context classes. Of course, the order on the symbols must be consistent with their set-semantics. For the unification problems we assume that context variables are labelled with a unique context class. If X is labeled with a context class $\mathcal{C}_i$, then $\sigma(X)$ must be a context that belongs to $\mathcal{C}_i$. The following conditions must hold in combination with the equational theory:

Assumption 3.5

1. *Equational deduction using the theory LC must not change the context class of almost ground contexts.*
2. *Prefix contexts C' of almost ground contexts C have the same context class as C or a smaller one.*

The following service algorithms are required with the mentioned conditions:

Assumption 3.6

1. *Matching a context and a term: Given a context variable X of context class $\mathcal{D}$, a term t and a position p in t, then there must be an algorithm that decides the question whether p is a permitted position for the context class $\mathcal{D}$. If this is the position of a variable z in t, and if $\sigma(X)$ is a prefix of the cotext $\sigma(t[[\cdot|/p)$, then the following replacement of must be complete: $X \mapsto t[X'/p]$ and $z \doteq X'[z']$, where the context class of X' is to be determined by the service algorithm. The sort of z and X' must also be determined by the sub-algorithm.*
2. *Common prefix-computation of several context variables: The following algorithm must be correct and complete: Given context variables $X_1, \ldots, X_m$ of context class $\mathcal{D}_1, \ldots, \mathcal{D}_m$, respectively. If there is no greatest lower bound of the $\mathcal{D}_i$, then fail.*
   Otherwise, let $\mathcal{D}$ be the greatest lower bound. Then the following selections can be made and may lead to a forking of the unification algorithm:
   (a) *Guess that one context variable is a prefix of the others: I.e. guess that X_j is a prefix of all others. The replacement is $X_j \mapsto X'_j$ and $X_i \mapsto X'_j X'_i$ where X'_i is of context class $\mathcal{D}_i$ for all $i \neq j$ and X'_j has context class $\mathcal{D}$. The context variables X'_i are fresh ones.*
   (b) *Guess a common prefix: Let X be a fresh context variable of context class $\mathcal{D}$. Let f be a function symbol in the signature and select an index j_i for every i. The replacement is then $X_i \mapsto X(f(z_1, \ldots z_{j_i-1}, X'_i, z_{j_i+1}, \ldots, z_k)$ where X'_i is of context class $\mathcal{D}_i$, where z_i are fresh first order variables and X, X'_i are fresh context variables. The sorts of z_i and X'_i must also be determined by the service algorithm. There may be selections of f and j_i that cannot be taken, depending on the context classes.*

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

We show that there is a complete unification algorithm for unification problems Γ where context classes are permitted with the properties as given above, and where Γ is restricted to be almost linear.
It is also assumed that Γ does not contain any context variable with an argument-sort in the equational sorts S_1.
Given a unification problem $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, the following (non-deterministic) unification algorithm will compute unifiers of Γ, where we implicitly use symmetry of $\doteq$.

Failure rules We will use the following failure rules whenever possible, where both failure rules can only be applied, if (**Guess-Empty-Context**) has been applied to all context variables in Γ.

1. (occurs-check failure) If there is a chain of equations $x_1 \doteq t_1, \ldots, x_n \doteq t_n$ in Γ, such that x_i occurs in t_{i-1} for $i = 2, \ldots, n$, x_1 occurs in t_n and at least one of the terms t_i is not a variable, then fail.
2. (sort-consistency check) If there is a context variable $X :: S_1 \to S_2$ in Γ, and there is no almost ground context of sort $S_1 \to S_2$, then fail.

Standard unification rules The following rules may operate on the unification problem:

1. (Decomposition) $\dfrac{\{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\} \cup \Gamma}{\{s_1 \doteq t_1, \ldots, s_n \doteq t_n)\} \cup \Gamma}$ if f is a free function symbol.
2. (Solve-Variable) $\dfrac{\{x \doteq t\} \cup \Gamma}{\Gamma}$ if x is of a non-empty sort. The instantiation $\{x \doteq t\}$ will be part of the computed unifier.
3. (Empty-Variable) $\dfrac{\{x \doteq y\} \cup \Gamma}{[x \mapsto y]\Gamma}$ if x is of an empty sort. The instantiation $\{x \doteq y\}$ will be part of the computed unifier.

Instantiation of Context Variables The following rules operate on the context variables at any position:

1. (**Guess-Empty-Context**) If X is a context variable of sort $S \to S$, then it is possible to guess that X is the empty context and to remove the context variable. In this case the instantiation is $X \mapsto [\cdot]$.

2. (**Theory-Contexts**) If X is a context variable with Env as resulting sort, then instantiate X as follows: $X \mapsto env(X', z)$ where X', z are fresh.
After application of these rules we can assume that there are no context variables with resulting sort Env.

Solving equations with context variables The rules for terms with contexts as top symbol are as follows:

1. Given an equation $X(s) \doteq t$, where the top symbol of t is not a context variable, guess a non-trivial position p in t that is not in an argument of a context variable, and such that the context class of X is compatible with the position p. Then
$$\frac{\{X(s) \doteq t\} \cup \Gamma}{\{X'(s) \doteq t_{|p}\} \cup \Gamma}$$
and the instantiation is $X \mapsto t[X'/p]$ where X' is a fresh context variable. The context class of X' is determined by the service algorithm.
2. Given an equation $X(s) \doteq Y(t)$, select one of the following two possibilities:
(a)
$$\frac{\{X(s) \doteq Y(t)\} \cup \Gamma}{\{s \doteq Y'(t)\} \cup \Gamma}$$
if there is a greatest lower bound D_3 of the context classes D_1, D_2 of X and Y, and if the service algorithm for context classes tells us that this is possible. The instantiation is $Y \mapsto ZY'$, $X \mapsto Z$, where Y' is a fresh context variable of the same context classes as Y, and Z has context class D_3.
(b)
$$\frac{\{X(s) \doteq Y(t)\} \cup \Gamma}{\Gamma}$$
if there is a greatest lower bound of the context classes of X and Y, and if the service algorithm for context classes tells us that the choice of the instantiation is possible. The instantiation is $\{X \mapsto Zf(z_1, \ldots, X', \ldots, z_n), Y \mapsto Zf(z_1, \ldots, Y', \ldots, z_n)$ where X', Y' are fresh context variables of the same context class as X, Y, respectively, where Z is a fresh context variable of a context class determined by the service algorithm, and where f is a function symbol in the permitted signature of arity at least 2, and the positions of X' and Y' are different. The instantiation must be provided by the service algorithm for the context classes.
If there is no such possibility, then fail.

Solving multi-set equations The following additional (non-deterministic) unification rules are sufficient to solve equations of type Env, i.e. proper multi-set-equations [DPR06,DPR98,DV99]. In the terms $env^*(s_1, \ldots, s_n, t)$, we can assume that t is not of the form $env(\ldots)$. It is also not of the form $X(\ldots)$, since these context variables can be instantiated using the rule **Theory-Contexts**. Other free function symbols are disallowed, hence t can only be a variable or the constant *emptyEnv*.

- If there is an equation $env^*(s_1, \ldots, s_n, z_1) \doteq env^*(t_1, \ldots, t_m, z_2)$, where $m, n \geq 1$, then select one of the following possibilities:
  - Select i, j and then apply the rule
$$\frac{\{env^*(s_1, \ldots, s_n, z_1) \doteq env^*(t_1, \ldots, t_m, z_2)\} \cup \Gamma}{\{s_i \doteq t_j, env^*(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, z_1) \doteq env^*(t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_m, z_2)\} \cup \Gamma}.$$
  - $$\frac{\{env^*(s_1, \ldots, s_n, z_1) \doteq env^*(t_1, \ldots, t_m, z_2)\} \cup \Gamma}{\Gamma}.$$
  where the instantiation is $\{z_1 \mapsto env^*(t_1, \ldots, t_m, z_3), z_2 \mapsto env^*(s_1, \ldots, s_n, z_3)\}$
- If there is an equation $env^*(s_1, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_m, z)$, where $m, n \geq 1$, then select i, j and apply the rule
$$\frac{\{env^*(s_1, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_m, z)\} \cup \Gamma}{\{s_i \doteq t_j, env^*(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_m, z)\} \cup \Gamma}.$$

– If there is an equation $env^*(s_1, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_m, emptyEnv)$, where $m, n \geq 1$, then select i, j and apply the rule
$$\frac{\{env^*(s_1, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_m, emptyEnv)\} \cup \Gamma}{\{s_i \doteq t_j, env^*(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n, emptyEnv) \doteq env^*(t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_m, emptyEnv)\} \cup \Gamma}.$$
– $\frac{\{env^*(s_1, \ldots, s_n, t) \doteq emptyEnv \cup \Gamma}{Fail}$ if $n \geq 1$.

We use the following convention in the conclusions of the rules state above: If in an environment $env^*(s_1, \ldots, s_n, z)$ (where z is either a variable or *emptyEnv*) the sequence $s_1, \ldots, s_n$ is empty, then we identify $env^*(s_1, \ldots, s_n, z)$ with z. E.g. $\frac{env^*(s_1, z_1) \doteq env^*(t_1, t_2, z_2)}{s_1 \doteq t_1, z_1 \doteq env^*(t_2, z_2)}$.

3.5 Properties of the LCSX-Unification Algorithm

Lemma 3.7. *The non-deterministic rule-based unification algorithm LCSX if applied to unification problems that are almost linear and where context variables have only free sorts as arguments will terminate. Moreover, there are only finitely many forking possibilities.*

Proof. The main measure μ is a tuple of several measures, where the tuples are compared lexicographically. μ_1 is the number of context variables, μ_2 is the number of context variables of sort *Env*, μ_3 is the number of variables, μ_4 is the size of the problem,i.e. the sum of the sizes of the terms, μ_4 is the sum of the sizes of env^*-terms. It is an easy task to verify that the measure is strictly decreased in every step. Also, there are only finitely many possibilities in every step.

Lemma 3.8. *The non-deterministic rule-based unification algorithm LCSX is sound and complete, i.e. every computed substitution is a unifier and every solution of the initial equation is represented by one final unifier.*

Proof. Soundness can be proved by standard methods, since rules are either instantiations or instantiations using the theory LC.
The assumptions on the service algorithms for context classes are required for the completeness.

Theorem 3.9. *The rule-based algorithm LCSX terminates if applied to unification problems that are almost linear and where context variables have only free sorts as arguments. Thus it decides unifiability of these sets of equations. Since it is sound and complete, the algorithm also computes a finite set of unifiers by gathering all possible results.*

Using this combination algorithm and the encoding below, for example all the overlaps with the normal-order rules (cp-in) and (llet-in) can be computed, but nit is not possible to compute all the overlaps. In order to compute all overlaps between all rules and transformations, a treatment of binding-chains (of the form $y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}]$ which are introduced by reduction context) is required. This is done in the next section.

4 Applying the Combined Unification Algorithm to the Example-Calculus

In this section we adapt the combination algorithm in the last section to develop a method to compute the necessary overlaps for our calculus in the case study. We compute the proper overlaps between left hand sides of normal-order reduction rules (see Fig. 2) and left hand sides of transformation rules from Fig. 1. The overlap is always the full left hand side of a transformation with a sub-expression in a surface context of a normal-order reduction rule. Overlaps where only a single meta-variable of the normal-order reduction rule is concerned can be treated without using unification. In the general case of proper overlaps the left hand sides are translated into equational problems between first-order terms: $S[l_{T,i}] \doteq l_{no,j}$, where $l_{T,i}$ is a left hand side of transformation rule and $l_{no,j}$ is a left hand side of a normal-order reduction rule. The following adaptations are required: Sorts, the equational theory of left-commutativity for the letrec-environments, context variables, context classes (of context variables), and binding-chains in letrec-environments. The latter occurs in reduction contexts and for example in the rule (cp-e). The unification algorithm LCSX has to be extended to further free function symbols and to the binding-chains. It will turn out that only the binding-chains will add new rules to the algorithm.

4.1 Encoding of Expressions as Terms

The sort and term structure according to the expression structure of the lambda calculus as given in Section 2 is as follows. There are the following sorts: $Bind, Env, Exp, BV$, for binding, environment, expression and bound variable, respectively; where $\mathcal{S}_1 = \{Env\}$ and $\mathcal{S}_2 = \{Bind, Exp, BV\}$. There are the following function symbols:

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

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.
As an example the expression ($\texttt{letrec}\ x = \lambda y.y, z = x\ x\ \texttt{in}\ z$) is encoded as

$$\begin{array}{l} let(env(bind(x, lam(y, var(y))), \\ \qquad env(bind(z, app(var(x), var(z))), \\ \qquad\quad emptyEnv)), \\ \quad var(z)) \end{array}$$

where x, y, z are variables of sort BV.
The first-order form of environment-expressions that are of the form $\{x_1 = t_1, \dots, x_k = t_k, Env\}$ is denoted by $env^*(b_1, \dots, b_m, z)$ in the following, where b_i are bindings and z is an environment variable.

4.2 Context-Subclasses and Unification

Context-subclasses are required to correctly model the normal-order reduction of our calculus. In Fig. 1 of the transformations there are only C-contexts, whereas in Fig. 2 there are also $\mathcal{A}$- and $\mathcal{R}$-contexts and also chains of bindings with parameter n. Also surface contexts $\mathcal{S}$ are required.
Grammars for $\mathcal{A}$ and $\mathcal{R}$ are $A ::= [\cdot] \mid (A\ s)$ and $R ::= A \mid (\texttt{letrec}\ Env\ \texttt{in}\ A) \mid (\texttt{letrec}\ y_1 = A_1, y_2 = A_2[y_1], \dots, y_n = A_n[y_{n-1}], Env\ \texttt{in}\ A[y_n])$, where A, A_i are $\mathcal{A}$-contexts. Thus we can replace $\mathcal{R}$-contexts by expressions containing only $\mathcal{A}$-contexts, where perhaps the rules of Fig. 2 may be splitted into several rules. Hence it is sufficient to invent procedures to deal with context variables of classes $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$ and with the binding-chains as introduced by this replacement. The partial order is $\mathcal{A} < \mathcal{S} < \mathcal{C}$. It would also be possible to have a context class $\mathcal{R}$, but for our calculus it can be omitted.
There are only the following necessary operations (here for $\mathcal{A}$ and $\mathcal{C}$-class, but it is similar for the $\mathcal{S}$-class). (i) common prefix-computation; (ii) common prefix-computation plus the first forking term. (iii) Overlapping of a context of class $\mathcal{A}$ with a term. The following observation is sufficient to design the required sub-algorithms:
$C[s] = A[t]$ for non-empty contexts C, A holds iff one of the following cases holds:

1. C is a prefix of A,$A = CA'$, C, A' are $\mathcal{A}$-contexts, $s = A'[t]$.
2. A is a prefix of C, $C = AC'$, $t = C'[s]$.
3. $C = A'[(r_1\ C')]$, $A = A'[(A''\ r_2)]$, $r_1 = A''[t]$, $r_2 = C'[s]$. Note that the function symbols and the possible positions of A'' are restricted.

If $A_1[s] = A_2[t]$ then only the prefix-cases are possible for A_1, A_2.
If $C_1[s] = C_2[t]$ then all cases are possible.
It is obvious how to translate this into a unification procedure that covers all cases.
The context classes $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{C}$ are stable under equational rewriting using the left-commutativity axiom. This is trivial for $\mathcal{C}$. For the class $\mathcal{A}$ this follows, since the LC-axiom does not change application nodes, and for $\mathcal{S}$ this holds, since lam-nodes are neither removed nor added.

4.3 Treatment of Binding-Chains

In this subsection we argue that a small set of extra rules is sufficient to remove binding-chains from unification problems $S[l_{T,i}] \doteq l_{no,j}$, where $l_{T,i}$ is a left hand side in Fig. 1, and $l_{no,j}$ a left hand side in Fig. 2.
We use $\texttt{BChain}(y_i = A_i[y_{i-1}], k_1, k_2, \mathit{Rel}])$ as a notation for a partial environment consisting of all the bindings $y_i = A_i[y_{i-1}]$, where $i = k_1 + 1, \ldots, k_2$ for positive integers k_1, k_2. If $k_1 \geq k_2$, then the environment is empty. In unification problems we also use the notation with variables and denote this as $\texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + k, N_2, \mathit{Rel}])$, where N_i are variables standing for positive integers, k is a non-negative integer constant, and $\mathit{Rel} \in \{<, \leq\}$. Here we always assume that the instantiations σ satisfy $\sigma(N_1{+}k)\ \mathit{Rel}\ \sigma(N_2)$. We will use the notation $\mathit{env}^*(\texttt{BChain}(\ldots) \mathbin{++} L, z)$ where L is an letrec-environment that does not contain a variable of sort Env and $\mathbin{++}$ denotes the union of two environments. We will use that L allows multi-set operations like reorderings.
The following unification step will be used for the unification of an environment expression with a binding-chain. An invariant is that the variables N_i may appear at most twice: at most once on the first position and at most once on the right position in a BChain-construct.
UnifBindingChain: Given an equation where one side is of the form $\mathit{env}^*(t_1, \ldots, t_m, z_1) \doteq \mathit{env}^*(L, z_2)$, where L contains a BChain-expression, then select one of the following rules for application.

1. Select an expression $\texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + k, N_2, \leq)$ from L and select one of the following two possibilities:
   (a) remove the selected BChain-expression from L. The instantiation is $N_2 \mapsto N_1 + k$, which also has to be applied to L.
   (b) replace the relation symbol $\leq$ by $<$ in the BChain-expression $\texttt{BChain}(y_i = A_i[y_{i-1}], N_1{+}k, N_2, \leq)$.
2. Select an expression t from L that is not a BChain-expression. Let L' be $L \setminus \{t\}$.
$$\frac{\{\mathit{env}^*(t_1, \ldots, t_m, z_1) \doteq \mathit{env}^*(L, z_2)\} \cup \Gamma}{\{t \doteq t_1, \mathit{env}^*(t_2, \ldots, t_m, z_1) \doteq \mathit{env}^*(L', z_2)\} \cup \Gamma}$$
3. Select an expression $\texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + k, N_2, <)$ from L:
$$\frac{\{\mathit{env}^*(t_1, \ldots, t_m, z_1) \doteq \mathit{env}^*(\texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + k, N_2) \mathbin{++} L', z_2)\} \cup \Gamma}{\begin{array}{l}\{t_1 \doteq (y_{N_3+1} = A_{N_3+1}[y_{N_3}]),\\ \mathit{env}^*(t_2, \ldots, t_m, z_1) \doteq \mathit{env}^*(\ \texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + k, N_3, \leq) \mathbin{++}\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{BChain}(y_i = A_i[y_{i-1}], N_3 + 1, N_2, \leq) \mathbin{++} L', z_2)\} \cup \Gamma\end{array}}$$
4. $$\frac{\{\mathit{env}^*(t_1, \ldots, t_m, z_1) \doteq \mathit{env}^*(L, z_2)\} \cup \Gamma}{\{\mathit{env}^*(t_2, \ldots, t_m, z_1) \doteq \mathit{env}^*(L, z_3)\} \cup \Gamma}$$
   plus the instantiation $z_2 \mapsto \mathit{env}(t_1, z_3)$.

Finally the equation with the BChain-expressions will be eliminated and an equation $z = t$ will be generated which can be removed, since it is then part of the partial solution.

Definition 4.1 (LCSXBV: Unification Algorithm with BChain). *The unification algorithm LCSX together with the additional rule* **UnifBindingChain** *is called LCSXBV.*

Note that the unification algorithms LCSX and LCSXBV in our example calculus may produce unifiers leading to illegal terms (see the example below), thus those unifiers have to be discarded.
The addition of the **UnifBindingChain**-rule to the LCSX unification algorithm is sufficient for the computation of all overlaps of the example calculus in section 2.8 because

- the unification problems that have to be solved by overlapping left hand sides of the rules in Fig. 1 with left hand sides of normal-order reductions in Fig. 2 are linear if we ignore the variables of sort BV and
- the BChain-construct occurs only on one side of an equation (in normal-order reductions) in the restricted form $\mathit{env}^*(\texttt{BChain}(y_i = A_i[s_i], N_1, N_2, \leq), t_1, \ldots, t_n, z)$.

Theorem 4.2. *The computation of all overlaps between the rules in Fig. 1 and left hand sides of normal-order reductions in Fig. 2 can be done using the algorithm LCSXBV. The unification algorithm terminates in all of these cases and computes a finite set of unifiers.*

4.4 Applying the Combined Unification Algorithm

As an example for the unification algorithm without binding-chains consider the overlap of (lapp) and (cp-in) as a part of the proof that (cp-in) is a correct program transformation in any context. This means to find overlaps of ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[x]$) within the expressions (($\texttt{letrec}\ Env'\ \texttt{in}\ s$) t). First we use an informal view: Seen as first-order terms, x, s, Env, C, t have to be treated as meta-variables, with the following restrictions: x can only be instantiated by a language-variable, s, t by expressions, Env by a sequence of bindings, C by a context. Similar as for Knuth-Bendix overlaps, the overlaps below a variable can be treated in a general way. One of the unification problems is that ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[x]$) should be made equal to ($\texttt{letrec}\ Env'\ \texttt{in}\ s$), which gives the following instantiation: $\{Env' \mapsto \{x = v, Env\}, s \mapsto C[x]\}$. The reductions are: (($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[x]$) t) $\xrightarrow{(no,lapp)}$ ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ (C[x]\ t)$), and (($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[x]$) t) $\xrightarrow{(cp-in)}$ (($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[v]$) t). It is easy to see that (($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[v]$) t) $\xrightarrow{(no,lapp)}$ ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ (C[v]\ t)$) and that ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ (C[x]\ t)$) $\xrightarrow{(cp-in)}$ ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ (C[v]\ t)$), which shows the complete reasoning for the forking diagram in this particular case. Note that a complete check for this constellation includes also applications of the rules in contexts.
Now we encode the expressions as terms: In order to overlap ($\texttt{letrec}\ x = v, Env\ \texttt{in}\ C[x]$) with ($\texttt{letrec}\ Env'\ \texttt{in}\ s$), the following encoded unification problem is obtained: $let(env(bind(x, v), Env), C(var(x))) \doteq let(Env', s)$, where the first-order variables are x, v, Env, Env', s, and C is a context variable. This gives two equations by decomposition: $env(bind(x, v), Env)) \doteq Env'$ and $C(var(x)) \doteq s$, and we only have to check whether the sort structure can instantiate C.
Another example is the overlap of a normal-order (cp-in)-left hand side with a program transformation (cp-in): ($\texttt{letrec}\ y = s, Env\ \texttt{in}\ A[y]$) and ($\texttt{letrec}\ y = s, Env\ \texttt{in}\ C[y]$) which results in the unification problem: $let(env(bind(y_1, s_1), Env_1), A(var(y_1))) \doteq let(env(bind(y_2, s_2), Env_2), C(var(y_2)))$. This gives two equations: $env(bind(y_1, s_1), Env_1) \doteq env(bind(y_2, s_2), Env_2)$ and $A(var(y_1))) \doteq C(var(y_2)))$.
There are several possible computation paths:

- One is that the terms match syntactically: I.e. $y_1 = y_2, s_1 = s_2, Env_1 = Env_2, C = A_1$.
- Another possibility is that the environment matches syntactically, but A and C are differently instantiated. That A is instantiated as a prefix of C or vice versa, which is not possible due to the sort structure, since there are no proper terms of sort BV. The other possibility is that there is a fork. The $\mathcal{A}$-context variables are restricted to applications. Hence only the function symbol *app* is possible for the fork.
  1. One possibility is $A \mapsto A'(app([\cdot], z_1))$ and that $C \mapsto A'(app(z_2, C'))$. This implies the instantiations $z_1 \mapsto C'(var(y_2))$ and $z_2 \mapsto var(y_1)$.
     The instantiated term is ($\texttt{letrec}\ y = s, Env\ \texttt{in}\ A[(y\ C'[y])]$), which represents the possibility of different targets of the copy-rules in the in-expression.
  2. The possibility that A is instantiated differently is ruled out.
- The environments are unified w.r.t. left-commutativity of *env*. Here it is sufficient to assume the instantiation $Env_1 \mapsto env(bind(y_2, s_2), Env')$ and $Env_2 \mapsto env(bind(y_1, s_1), Env')$. These possibilities can be combined with the possibilities of the unification of the equation $A(var(y_1))) \doteq C(var(y_2)))$.
  A possible instantiated expression is ($\texttt{letrec}\ y_1 = s_1, y_2 = s_2, Env\ \texttt{in}\ A[y_1\ C[y_2]]$).
  As an example of a unifier that may lead to syntactically wrong expressions, we may obtain $y_1 \mapsto y_2$, which leads to the ill-formed expression: ($\texttt{letrec}\ y_1 = s_1, y_1 = s_2, Env\ \texttt{in}\ A[y_1]$).

A further example how to deal with binding-chains is:

Example 4.3. The most complex example is the overlap of a (cp-e)-rule with a normal-order variant of the (cp-e)-rule. The unification problem is to unify the environment $env^*(x = s, y = C[x], Env)$ with $env^*(y_1 = s', \texttt{BChain}(y_i = A_i[y_{i-1}], 1, N, <), Env_2)$. One possibility for the first step is: $x = s \doteq y_{N_1+1} = A_{N_1+1}[y_{N_1}], env^*(y = C[x], Env) \doteq env^*(y_1 = s', (\texttt{BChain}(y_i = A_i[y_{i-1}], 1, N_1, \leq) +\!\!+ \texttt{BChain}(y_i = A_i[y_{i-1}], N_1 + 1, N, \leq), Env_2)$. The next step would remove the binding $y = C[x]$. The number of possibilities in eliminating the eqaution is $2 + 4 + 3 = 9$. It remains to compute the closing reduction of the forking diagram, which is out of the scope of this paper

5 Conclusion

We have provided an effective method using first-order unification with equational theories, sorts and context variables and context classes 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 extend this method to further lambda-calculi, in particular to lambda-calculi like in [SSSS08], where variable-variable bindings are transparent in the normal-order rules, and to calculi with data structures and case-expressions.

References

[AF97] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

[BS92] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1992.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In J. A. Robinson and A.Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[DPR98] Agostino Dovier, Alberto Policriti, and Gianfranco Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundam. Inform.*, 36(2-3):201–234, 1998.

[DPR06] Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Set unification. *TPLP*, 6(6):645–701, 2006.

[DV99] Evgeny Dantsin and Andrei Voronkov. A nondeterministic polynomial-time unification algorithm for bags, sets and trees. In *FoSSaCS*, pages 180–196, 1999.

[FH92] Matthias Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 103:235–271, 1992.

[How89] D. Howe. Equality in lazy computation systems. In *4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

[KB70] D. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoret. Comput. Sci.*, 4:1–22, 1977.

[SS89] Manfred Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8(1,2):51–99, 1989.

[SS07] Manfred Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

[SSS10] Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.

[SSSM10a] Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec. Frank report 40, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, April 2010.

[SSSM10b] Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *Proc. of RTA 2010*, 2010. to appear.

[SSSS08] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.