Computing Overlappings by Unification in the Deterministic Lambda Calculus LR with letrec, case, constructors, seq and variable chains

Conrad Rau and Manfred Schmidt-Schauß*
{rau,schauss}@ki.informatik.uni-frankfurt.de

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

Technical Report Frank-46

September 1, 2011

Abstract. Correctness of program transformations in extended lambda calculi with a contextual semantics is usually based on reasoning about the operational semantics which is a rewrite semantics. A successful approach to proving correctness is the combination of a context lemma with the computation of overlaps between program transformations and the reduction rules.The method is similar to the computation of critical pairs for the completion of term rewriting systems. We describe an effective unification algorithm to determine all overlaps of transformations with reduction rules for the lambda calculus LR which comprises a recursive let-expressions, constructor applications, case expressions and a seq construct for strict evaluation. The unification algorithm employs many-sorted terms, the equational theory of left-commutativity modeling multi-sets, context variables of different kinds and a mechanism for compactly representing binding chains in recursive let-expressions. As a result the algorithm computes a finite set of overlappings for the reduction rules of the calculus LR that serve as a starting point to the automatization of the analysis of program transformations.

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

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.

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

We assume given the syntax of programs $\mathcal{P}$, a deterministic reduction relation $\to \subseteq \mathcal{P} \times \mathcal{P}$ that represents a single execution step on programs and values that represent the successful end of program execution. The reduction of a program may be non-terminating due to language constructs that allow iteration or recursive definitions. For a program $P \in \mathcal{P}$ we write $P{\Downarrow}$ if there is a sequence of reductions to a value, and say P *converges* (or terminates successfully) in this case. Then equivalence of programs can be defined by $P_1 \sim P_2 \iff \big(\text{for all } C : C[P_1]{\Downarrow} \iff C[P_2]{\Downarrow}\big)$, where C is a context, i.e. a program with a hole $[\cdot]$ at a single position. Justifying the correctness of a program transformation $P \rightsquigarrow P'$ means to provide a proof that $P \sim P'$. Unfortunately, the quantification is over an infinite set: the set of all contexts, and the criterion is termination, which is undecidable in general. Well-known tools to ease the proofs are context lemmas [Mil77], ciu-lemmas [FH92] and bisimulation, see e.g. [How89].

The reduction relation $\to$ is often given as a set of rules $l_i \to r_i$ similarly to rewriting rules, but extended with different kinds of meta-variables and some other constructs, together with a strategy determining when to use which rule and at which position. In order to prove correctness of a program transformation that is also given in a rule form $s_1 \to s_2$, we have to show that $\sigma(s_1) \sim \sigma(s_2)$ for all possible rule instantiations σ i.e. $C[\sigma(s_1)]{\Downarrow} \iff C[\sigma(s_2)]{\Downarrow}$ for all contexts C. Using the details of the reduction steps and induction on the length of reductions, the hard part is to look for conflicts between instantiations of s_1 and some l_i, i.e. to compute all the overlaps of l_i and s_1, and the possible completions under reduction and transformation. This method is reminiscent of the critical pair criterion of Knuth-Bendix method [KB70] 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 *LR* which is a call-by-need lambda calculus with a letrec-construct, constructors, case-expressions and a seq-construct for strict evaluation (see [SSSS08]). 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.

2 An Extended Lambda Calculus with letrec

We will throughout use the call-by-need calculus LR from [SSSS08]. In this section we introduce its syntax and semantics.

2.1 The Call-by-Need Calculus LR

Syntax and Reduction Rules The expressions of the call-by-need lambda calculus LR from [SSSS08] consist of variables, applications, abstractions, constructor-expressions, case-expressions and recursive let-expressions.

There are finitely many constants, called constructors. The set of constructors is partitioned into (non-empty) types, i.e. we assume that a type T is the set of its constructors. For every type, we let $T = \{c_{T,i}, i = 1, \ldots, |T|\}$. Every constructor has an arity $ar(c_{T,i}) \geq 0$.
The syntax for expressions E, case alternatives Alt and patterns Pat is as follows:

$$\begin{aligned} s, s_1, \ldots, s_n \in E ::= {} & x \mid (c\ s_1 \ldots s_{ar(c)}) \mid (\texttt{seq}\ s_1\ s_2) \mid (\texttt{case}_T\ s\ Alt_1 \ldots Alt_{|T|}) \\ & \mid (s_1\ s_2) \mid (\lambda x.s) \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s) \\ Alt ::= {} & (Pat\ \rightarrow\ s) \\ Pat ::= {} & (c\ x_1 \ldots x_{ar(c)}) \end{aligned}$$

where x, x_i are variables, and where c denotes a constructor. Within each individual pattern in a case, variables are not repeated. In a `case`-expression of the form $(\texttt{case}_T \ldots)$, for every constructor $c_{T,i}, i = 1, \ldots, |T|$ of type T, there is exactly one alternative with a pattern of the form $(c_{T,i}\ y_1 \ldots y_n)$ where $n = ar(c_{T,i})$.

We assign the names *application*, *abstraction*, *constructor-application*, `seq`-*expression*, *case-expressions* or *letrec-expression* to the expressions $(s\ t)$, $(\lambda x.s)$, $(c\ s_1 \ldots s_n)$, $(\texttt{seq}\ s_1\ s_2)$, $(\texttt{case}_T\ E\ Alt_1 \ldots Alt_{|T|})$, $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$, respectively.

The pair $x = s$ of a variable x and an expression s is called a *letrec-binding* or just binding. A group of letrec-bindings, also called *environment*, is abbreviated as Env. A *value* v is defined as an abstraction or a constructor application.

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, i.e. we assume that there are no empty letrec-environments.

Letrec-bindings are recursive, i.e., the scope of x_j in ($\texttt{letrec}\ x_1 = s_1, \ldots, x_{n-1} = s_{n-1}\ \texttt{in}\ s_n$) are all expressions s_i with $1 \leq i \leq n$. Variable binding primitives are λ, letrec, and patterns in alternatives of case-expressions. Free and bound variables in expressions and α-renamings are defined as usual.

The set of free variables in t is denoted as $FV(t)$. We use the distinct variable convention (DVC), i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly α-rename bound variables in the result if necessary.

We use the notation $\{x_{g(i)} = s_{h(i)}\}_{i=m}^{n}$ for the chain $x_{g(m)} = s_{h(m)}, x_{g(m+1)} = s_{h(m+1)}, \ldots, x_{g(n-1)} = s_{h(n-1)}$ of bindings, e.g. $\{x_{i+1} = s_i\}_{i=m}^{n}$ means the bindings $x_{m+1} = s_m, x_{m+2} = s_{m+1} \ldots, x_n = s_{n-1}$, where all the x_i are distinct variables. Notice, that chains run from m to $n-1$ in contrast to [SSSS08], where they run from m to n. The reason for this lies in the unification algorithm, where we need to split chains, which is more conveniently done on our modified chains.

A *context* C is an expression according to the syntax of LR where the symbol $[\cdot]$, the *hole*, is also allowed as expression, such that $[\cdot]$ occurs exactly once (as sub-expression) in C. We distinguish the following different context classes:

Definition 2.1. Application contexts $\mathcal{A}$, general contexts $\mathcal{C}$, reduction contexts $\mathcal{R}$ *and* surface contexts $\mathcal{S}$ *are defined by the following grammars:*

$$
\begin{array}{lll}
A \in \mathcal{A} & ::= & [\cdot] \mid (A\ s) \mid (\texttt{case}_T\ A\ \text{alts}) \mid (\texttt{seq}\ A\ s) \\
R \in \mathcal{R} & ::= & A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \mid \texttt{letrec}\ y_1 = A_1, Env\ \texttt{in}\ A[y_1] \\
 & & \mid \texttt{letrec}\ y_1 = A_1, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env\ \texttt{in}\ A[y_n] \\
S \in \mathcal{S} & ::= & [\cdot] \mid (S\ s) \mid (s\ S) \mid (c\ s_1\ \ldots s_{i-1}\ S\ s_{i+1} \ldots s_{ar(c)}) \\
 & & \mid\ (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ S) \mid\ (\texttt{letrec}\ Env, y = S\ \texttt{in}\ s) \\
 & & \mid\ \texttt{seq}\ S\ s \mid\ \texttt{seq}\ s\ S \mid\ \texttt{case}_T\ S\ alts \mid (\texttt{case}_T\ s\ alts(Pat \to S)\ alts) \\
C \in \mathcal{C} & ::= & [\cdot] \mid (C\ s) \mid (s\ C) \mid (c\ s_1\ \ldots s_{i-1}\ C\ s_{i+1} \ldots s_{ar(c)}) \mid (\lambda x.C) \\
 & & \mid\ (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ C) \mid\ (\texttt{letrec}\ Env, x = C\ \texttt{in}\ s) \\
 & & \mid (\texttt{seq}\ C\ s) \mid (\texttt{seq}\ s\ C) \mid \texttt{case}_T\ C\ alts \mid (\texttt{case}_T\ s\ alts(Pat \to C)\ alts)
\end{array}
$$

where s, s_i *denote expressions.*

Given a term t and a context C, we write $C[t]$ for the LR-expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted. Note that α-renaming of contexts is restricted.

Remark 2.2. A reduction context R may contain a chain of the form $\{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}$, where the A_i could be the empty context. This differs from the calculus in [RSS11] where these application-contexts are required to be non-empty.

Definition 2.3. *The unrestricted reduction rules for the calculus* LR *are defined in Figures 1 and 2. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet), and we speak also of the rules (cp), (lll), (seq), and (case).*

(lbeta) $((\lambda x.s)\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in) $(\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \ \texttt{in}\ C[x_m])$
$\to (\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \ \texttt{in}\ C[v])$
where v is an abstraction
(cp-e) $(\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env, y = C[x_m]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env, y = C[v]\ \texttt{in}\ r)$
where v is an abstraction
(llet-in) $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r))$
$\to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e) $(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp) $((\texttt{letrec}\ Env\ \texttt{in}\ t)\ s) \to (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$
(lcase) $(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ t)\ alts) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ t\ alts))$
(seq-c) $(\texttt{seq}\ v\ t) \to t$ if v is a value
(seq-in) $(\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \texttt{in}\ C[(\texttt{seq}\ x_m\ t)])$
$\to (\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \texttt{in}\ C[t])$
if v is a value
(seq-e) $(\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env, y = C[(\texttt{seq}\ x_m\ t)]\ \texttt{in}\ r)$
$\to (\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env, y = C[t]\ \texttt{in}\ r)$
if v is a value
(lseq) $(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$

Fig. 1: Unrestricted reduction rules, part a

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

The normal order reduction of LR in [SSSS08] is defined via a search for a normal-order redex and placing labels in the expression. We will give an equivalent, explicit definition using contexts, since this explicit definition will be the basis for the computation of overlaps of reductions and transformation rules.

Definition 2.4. *Normal order reduction* $\xrightarrow{no}$ *(called no-reduction for short) is defined by the reduction rules in Figure 3 and 4. There are special cases for constructors of arity* $= 0$*, or when parts of the environment are not available or omitted, which can easily be derived from these rules by instantiation. After instantiation of rules, empty environments are not permitted.*

Note that the normal order reduction is unique. A *weak head normal form in* LR *(WHNF)* is defined as either an abstraction $\lambda x.s$, or a constructor application $(c\ s_1 \ldots s_n)$ or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is an abstraction or a constructor application.

(case-c) $(\texttt{case}_T\ (c_i\ \overrightarrow{t})\ \ldots((c_i\ \overrightarrow{y}) \to t)\ldots) \to (\texttt{letrec}\ y_1 = t_1, \ldots, y_n = t_n\ \texttt{in}\ t)$
where $n = ar(c_i) \geq 1$

(case-c) $(\texttt{case}_T\ c_i\ \ldots\ (c_i \to t)\ldots) \to t \quad \text{if } ar(c_i) = 0$

(case-in) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t}), \{x_{i+1} = x_i\}_{i=1}^m, Env$
$\texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots((c_i\ \overrightarrow{z})\ldots \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), y_1 = t_1, \ldots, y_n = t_n, \{x_{i+1} = x_i\}_{i=1}^m, Env$
$\texttt{in}\ C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ t)]$
where $n = ar(c_i) \geq 1$ and y_i are fresh variables

(case-in) $\texttt{letrec}\ x_1 = c_i, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \texttt{in}\ C[\texttt{case}_T\ x_m\ \ldots\ (c_i \to t)\ldots]$
$\to \texttt{letrec}\ x_1 = c_i, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \texttt{in}\ \ C[t]$
if $ar(c_i) = 0$

(case-e) $\texttt{letrec}\ x_1 = (c_i\ \overrightarrow{t}), \{x_{i+1} = x_i\}_{i=1}^m,$
$u = C[\texttt{case}_T\ x_m\ \ldots((c_i\ \overrightarrow{z}) \to r_1)\ldots], Env$
$\texttt{in}\ r_2$
$\to \texttt{letrec}\ x_1 = (c_i\ \overrightarrow{y}), y_1 = t_1, \ldots, y_n = t_n, \{x_{i+1} = x_i\}_{i=1}^m,$
$u = C[(\texttt{letrec}\ z_1 = y_1, \ldots, z_n = y_n\ \texttt{in}\ r_1)], Env$
$\texttt{in}\ r_2$
where $n = ar(c_i) \geq 1$ and y_i are fresh variables

(case-e) $\texttt{letrec}\ x_1 = c_i, \{x_{i+1} = x_i\}_{i=1}^m, u = C[\texttt{case}_T\ x_m\ \ldots\ (c_i \to r_1)\ldots], Env$
$\texttt{in}\ r_2$
$\to \texttt{letrec}\ x_1 = c_i, \{x_{i+1} = x_i\}_{i=1}^m \ldots, u = C[r_1], Env\ \texttt{in}\ r_2$
if $ar(c_i) = 0$

Fig. 2: Unrestricted reduction rules, part b

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

2.2 Chains of Bindings in letrec Environments

Chains of the form $\{x_{i+1} = A_{i+1}[x_i]\}_{i=m}^n$ play a particular role in reduction rules. The binding chain starts with the binding $x_{m+1} = A_{m+1}[x_m]$, i.e. x_m denotes a variable that does not occur at a binder position inside the chain (it is free in the chain but may occur at a binder position outside the chain, as it usually does in the reduction rules). The last letrec-binding in the chain is $x_n = A_n[x_{n-1}]$, which means that x_n denotes a binder.

In the reduction rules from figure 1, 2, 3 and 4 there are two different types of binding chains:

(no, lbeta) $R[(\lambda x.s)\ r] \to R[\texttt{letrec}\ x = r\ \texttt{in}\ s]$
(no, cp-in) $\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \texttt{in}\ A[x_m]$
$\to \texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, Env\ \texttt{in}\ A[v]$
where v is an abstraction.
(no, cp-e) $\texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, y_1 = A_1[x_m],$
$\{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env\ \texttt{in}\ A[y_n]$
$\to \texttt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m}, y_1 = A[v],$
$\{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env\ \texttt{in}\ A[y_n]$
where v is an abstraction and A_1 is a non-empty context.
(no, llet-in) $(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r))$
$\to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(no, llet-e) $\texttt{letrec}\ y_1 = (\texttt{letrec}\ Env_1\ \texttt{in}\ r), \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env_2\ \texttt{in}\ A[y_n]$
$\to \texttt{letrec}\ y_1 = r, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}, Env_1, Env_2\ \texttt{in}\ A[y_n]$
(no, lapp) $R[((\texttt{letrec}\ Env\ \texttt{in}\ r)\ t)] \to R[(\texttt{letrec}\ Env\ \texttt{in}\ (r\ t))]$
(no, lcase) $R[\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ r)\ alts)]$
$\to R[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ r\ alts))]$
(no, lseq) $R[\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ r)\ s)] \to R[(\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ r\ s))]$

Fig. 3: Normal order reduction rules of LR, part 1

1. *N-chains* of the form $\{y_{i+1} = A_{i+1}[y_i]\}_{i=m}^{n}$ where A_{i+1} is a (possible empty $\mathcal{A}$ context). They occur only in normal order reduction rules and are used to specify the exact position of the normal order redex. In the reduction rules such chains are accompanied by a leading binding $y_m = s$, where the form of s varies from rule to rule. We call this binding the *origin* of the chain.
2. *Var-chains* of the form $\{x_{i+1} = x_i\}_{i=m}^{n}$ occur in the cp reduction rules, seq-rules and case-rules of the calculus LR. When a var-chain occurs in a reduction rule, it is always accompanied by a leading binding $x_m = v$, which we call *origin* of the var-chain.

Var-chains are special instances of A-chains where all application contexts are empty and the leading binding is of the special form $x = v$ where v is a value. Both types of chains can be characterized by a relation on their bindings.

2.3 Contextual Equivalence

The semantic foundation of our calculus LR is the equality of expressions defined by contextual equivalence.

Definition 2.5 (Contextual Preorder and Equivalence). *Let s, t be* LR*-expressions. Then:*

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

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

$$
\begin{array}{ll}
\text{(no, case)} & R[(\mathtt{case}_T\ (c_i\ \overrightarrow{t})\ \ldots\ (c_i\ \overrightarrow{x} \to t)\ldots)] \\
& \quad \to R[\mathtt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \mathtt{in}\ t] \quad \text{if } ar(c_i) \geq 1 \\
\text{(no, case)} & \mathtt{letrec}\ x_1 = c\ \overrightarrow{t}, \{x_{i+1} = x_i\}_{i=1}^m, Env \\
& \quad \mathtt{in}\ A[\mathtt{case}_T\ x_m\ c\ \overrightarrow{y} \to t, alts] \\
& \quad \to \mathtt{letrec}\ z_1 = t_1, \ldots, z_n = t_n, x_1 = c\ \overrightarrow{z}, \{x_{i+1} = x_i\}_{i=1}^m, Env \\
& \quad \mathtt{in}\ A[\mathtt{letrec}\ y_1 = z_1, \ldots, y_n = z_n\ \mathtt{in}\ t] \\
\text{(no, case)} & \mathtt{letrec}\ x_1 = c\ \overrightarrow{t}, \{x_{i+1} = x_i\}_{i=1}^m, y_1 = A[\mathtt{case}_T\ x_m\ c\ \overrightarrow{u} \to t, alts], \\
& \quad \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^n, Env\ \mathtt{in}\ A[y_n] \\
& \quad \to \mathtt{letrec}\ z_1 = t_1, \ldots, z_n = t_n, x_1 = c\ \overrightarrow{z}, \{x_{i+1} = x_i\}_{i=1}^m, \\
& \quad\quad y_1 = A[\mathtt{letrec}\ u_1 = z_1, \ldots, u_n = z_n\ \mathtt{in}\ t], \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^n, \\
& \quad\quad Env\ \mathtt{in}\ A[y_n] \\
\text{(no, seq)} & R[(\mathtt{seq}\ v\ s] \to R[t] \quad \text{if } v \text{ is a value} \\
\text{(no, seq)} & \mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \mathtt{in}\ A[\mathtt{seq}\ x_m\ t] \\
& \quad \to \mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \mathtt{in}\ A[t] \quad \text{if } v \text{ is a value} \\
\text{(no, seq)} & \mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, y_1 = A[\mathtt{seq}\ x_m\ t], \\
& \quad \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^n, Env\ \mathtt{in}\ A[y_n] \\
& \quad \to \mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, \\
& \quad\quad y_1 = A[t], \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^n, Env\ \mathtt{in}\ A[y_n] \quad \text{if } v \text{ is a value}
\end{array}
$$

Fig. 4: Normal order reduction rules of LR, part 2

Program transformations are usually given in a format similarly to reduction rules, as in Figure 1, 2 and Figure 3 and 4. A program transformation T is written as $s \xrightarrow{T} t$. Here we restrict our attention for the sake of simplicity to the program transformations that are given by the reduction rules in Figure 1, 2.

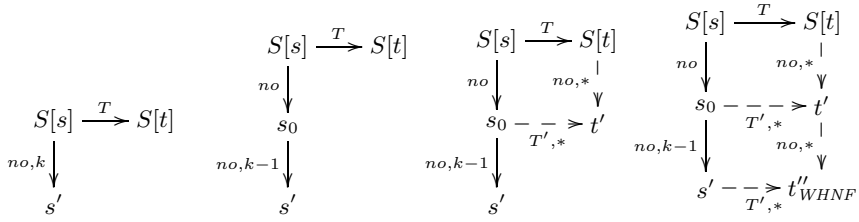
An important tool to prove contextual equivalence is a *context lemma* (see for example [Mil77], [SSS10],[SSSS08]), which allows to restrict the class of contexts that have to be considered in the definition of the contextual equivalence from general $\mathcal{C}$ to $\mathcal{R}$ contexts.

However, often $\mathcal{S}$-contexts are more appropriate for computing overlaps and closing the diagrams, for example there are cases, where the forking diagrams cannot be closed using reductions in $\mathcal{R}$-contexts. To use transformations in all possible contexts will lead to diagrams which in several cases prevent induction proofs on the lengths of reductions, since duplicated reductions may be required. The $\mathcal{S}$-contexts do not permit holes in abstractions, so the major source of duplicating reductions is omitted. The extension of the reasoning to all contexts is done using the context lemma for surface contexts. So we will use $\mathcal{S}$-contexts in the following for transformations instead of $\mathcal{R}$-contexts.

Lemma 2.7. *Let s, t be* LR*-expressions and S a context of class $\mathcal{S}$. $(S[s]{\Downarrow} \Rightarrow S[t]{\Downarrow})$ iff $\forall C : (C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow})$; i.e. $s \leq_c t$.*

Proof. A proof of this lemma when the contexts are in class $\mathcal{R}$ is in [SSS10] (and also in [SSSS08]). Since every $\mathcal{R}$-context is also an $\mathcal{S}$-context, the lemma holds.

To prove the correctness of a transformation $s \xrightarrow{T} t$ we have to prove that $s \sim_c t$, which is equivalent to $s \leq_c t \ \wedge \ t \leq_c s$. By Definition 2.5 this amounts to showing $\forall C : C[s]{\Downarrow} \Rightarrow C[t]{\Downarrow} \ \wedge \ C[t]{\Downarrow} \Rightarrow C[s]{\Downarrow}$. The context lemma yields that it is sufficient to show $\forall S : S[s]{\Downarrow} \Rightarrow S[t]{\Downarrow} \ \wedge \ S[t]{\Downarrow} \Rightarrow S[s]{\Downarrow}$. We restrict our attention here to $S[s]{\Downarrow} \Rightarrow S[t]{\Downarrow}$ because $S[t]{\Downarrow} \Rightarrow S[s]{\Downarrow}$ could be treated in a similar way. To prove $s \sim_c t$ we assume that $s \xrightarrow{T} t$ and $S[s]{\Downarrow}$ holds, i.e. there is a WHNF s', such that $S[s] \xrightarrow{no,k} s'$ (see Figure 5(a)). It remains to show that there also exists a sequence of normal order reductions from $S[t]$ to a WHNF. This can often be done by induction on the length k of the given normal order reduction $S[s] \xrightarrow{no,k} s'$ using *complete sets of reduction diagrams*. Therefore we split $S[s] \xrightarrow{no,k} s'$ into $S[s] \xrightarrow{no} s_o \xrightarrow{no,k-1} s'$ (see Figure 5(b)). Then an applicable *forking diagram* defines how the fork $s_0 \xleftarrow{no} S[s] \xrightarrow{T} S[t]$ can be closed specifying two sequences of transformations such that a common expression t' is eventually reached: one starting from $S[t]$ consisting only of no-reductions and one starting from s_0 consisting of some other reductions (that are not normal order) denoted by T' in Figure 5(c).



(a) Forking in the proof of $s \leq_c t$ (b) Splitting the no-sequence (c) Application of a forking diagram (d) Inductive proof of $s \leq_c t$

Fig. 5: Sketch of the correctness proof for $s \xrightarrow{T} t$

A set of forking diagrams for a transformation T is *complete* if the set comprises an applicable diagram for every forking situation. If we have a complete set of forking diagrams we often can inductively construct a terminating reduction sequence for $S[t]$ if $S[s]{\Downarrow}$ (as indicated in Figure 5(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 [SSSS08]). We restrict our attention to complete sets of forking diagrams.

2.4 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 LR are called *internal* and denoted by a label i. An internal reduction in a reduction context is marked by $i\mathcal{R}$, and an internal reduction in a surface context by $i\mathcal{S}$.

A *reduction sequence* is of the form $t_1 \to \ldots \to t_n$, where t_i are LR-expressions and $t_i \to t_{i+1}$ is a reduction as defined in Definition 2.3. In the following definition we describe transformations on reduction sequences. Therefore we use the notation

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

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

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 2.8.

• *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*

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

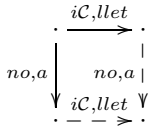
where $k, k' \geq 0, m \geq 1, h > 1$, *such that for every reduction sequence* $t_0 \xrightarrow{iX,T} t_1 \xrightarrow{no} \ldots \xrightarrow{no} t_h$, *where* t_h *is a WHNF, at least one of the transformation rules is applicable to a prefix of the sequence.*

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.

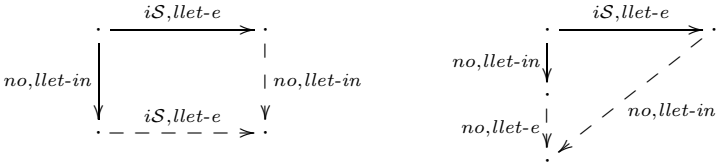
For example, the forking diagram $\xleftarrow{no,a} \cdot \xrightarrow{i\mathcal{C},llet} \quad \rightsquigarrow \quad \xrightarrow{i\mathcal{C},llet} \cdot \xleftarrow{no,a}$ is represented as



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 2.9. Example forking diagrams are



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

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

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

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

The first step in the computation of a complete set of forking diagrams for a transformation T is the determination of all forks of the form $\xleftarrow{no,red} \cdot \xrightarrow{i\mathcal{S},T}$ where *red* is an no-reduction and T is not a normal order reduction (but a transformation in an $\mathcal{S}$-context). Such forks are given by *overlaps* between no-reductions and the transformation. Informally we say that *red* and T overlap in an expression s if s contains a normal order redex *red* and a T redex in a surface context. To find an overlap between an no-reduction *red* and a transformation T it is sufficient, by definition of the normal order reduction, to determine all surface-positions in *red* where a T-redex can occur. This covers all overlaps (the critical and the non critical). Note that complications are the multi-set property of the letrec environments, and the instantiations of the context classes.

We devise an algorithm that computes complete sets of forks for the presented calculus. The main goal of the algorithm is to compute all overlaps between left hand sides of transformations rules and left hand normal order reduction rules. The algorithm has different phases:

1. Translate/encode left hand sides of reduction rules into a first-order term representation and use it to generate unification problems that describe all overlaps.
2. Solve the unification problems (in a almost first order way);
3. then check if no expressions from different α-equivalence classes were equated.
4. Instantiate the unification problems that describe the forks with the computed solutions and translate them back to yield all forks in the LR calculus.

2.5 The Transformations

In this section we give the transformations that we want to show correct using unification. These are the rules in Figure 6 which are variants of the unrestricted rules in Figures 1, and the (unchanged) case-rules in Figure 2.

Therefore we also need the deep general $\mathcal{DC}$-contexts, which are the contexts in $\mathcal{C}$, but not in $\mathcal{S}$. The technical reason is that the (cp)-rules for general contexts appears too expressive, and cannot be shown to be correct using the technique of using a context lemma, induction on the length of reductions and overlap diagrams.

These deep contexts of class $\mathcal{DC}$ can defined as all contexts of the form $C[D_1]$, where D_1 is a context according to the following grammar:

$$(\lambda x.C) \mid (\mathtt{case}_T\ s\ alts(Pat \to C)\ alts)$$

where s, s_i denote expressions, and C general contexts.

(lbeta) $((\lambda x.s)\ r) \to (\mathtt{letrec}\ x = r\ \mathtt{in}\ s)$

(cp-in-S) $(\mathtt{letrec}\ x = v, Env\ \ \mathtt{in}\ S[x]) \to (\mathtt{letrec}\ x_1 = v, Env\ \ \mathtt{in}\ S[v])$
where v is an abstraction

(cp-in-D) $(\mathtt{letrec}\ x = v, Env\ \ \mathtt{in}\ D[x]) \to (\mathtt{letrec}\ x_1 = v, Env\ \ \mathtt{in}\ D[v])$
where v is an abstraction

(cp-e-S) $(\mathtt{letrec}\ x = v, Env, y = S[x]\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ x = v, Env, y = S[v]\ \mathtt{in}\ r)$
where v is an abstraction

(cp-e-D) $(\mathtt{letrec}\ x = v, Env, y = D[x]\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ x = v, Env, y = D[v]\ \mathtt{in}\ r)$
where v is an abstraction

(llet-in) $(\mathtt{letrec}\ Env_1\ \mathtt{in}\ (\mathtt{letrec}\ Env_2\ \mathtt{in}\ r))$
$\to (\mathtt{letrec}\ Env_1, Env_2\ \mathtt{in}\ r)$

(llet-e) $(\mathtt{letrec}\ Env_1, x = (\mathtt{letrec}\ Env_2\ \mathtt{in}\ s_x)\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ Env_1, Env_2, x = s_x\ \mathtt{in}\ r)$

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

(lcase) $(\mathtt{case}_T\ (\mathtt{letrec}\ Env\ \mathtt{in}\ t)\ alts) \to (\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{case}_T\ t\ alts))$

(seq-c) $(\mathtt{seq}\ v\ t) \to t$ if v is a value

(seq-in) $(\mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \mathtt{in}\ C[(\mathtt{seq}\ x_m\ t)])$
$\to (\mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env\ \mathtt{in}\ C[t])$
if v is a constructor application

(seq-e) $(\mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env, y = C[(\mathtt{seq}\ x_m\ t)]\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ x_1 = v, \{x_{i+1} = x_i\}_{i=1}^m, Env, y = C[t]\ \mathtt{in}\ r)$
if v is a constructor application

(lseq) $(\mathtt{seq}\ (\mathtt{letrec}\ Env\ \mathtt{in}\ s)\ t) \to (\mathtt{letrec}\ Env\ \mathtt{in}\ (\mathtt{seq}\ s\ t))$

where $C \in \mathcal{C}, \mathcal{S} \in \mathcal{S}, \mathcal{D} \in \mathcal{DC}$

Fig. 6: Transformation rules

It is no restriction to prove correctness only for the transformations in Fig. 6 instead of the transformations in Fig. 1:

Proposition 2.10. *Correctness of the transformations in Figure 6 implies the correctness of the transformations in Figure 1.*

Proof. The correctness of (cp-in-S) and (cp-in-D) implies the correctness of (cp-in): All contexts are covered, hence the transformation

$$(\texttt{letrec}\ x = v, Env\ \ \texttt{in}\ C[x]) \to (\texttt{letrec}\ x_1 = v, Env\ \ \texttt{in}\ C[v])$$

is correct. The correctness of copying over the chain can also be derived using induction on the length of the variable-chain, since the transformations can also be applied in the backward direction (i.e. cp-in-S and cp-in-D applied forwards and backwards can simulate cp-in reductions).

Similar arguments apply to (seq-e) and (seq-in): The correctness of (seq-e) and (seq-in) for abstraction follows from the correctness of (seq-c) and the the correctness of the cp-transformations. □

Note that the variable chains in (seq-e) and (seq-in) cannot be omitted in this way: there is no rule that permits to copy constructor applications. Using further transformation rules as in [SSSS08] may be an alternative, but is a deviation which will not be explored here.

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

In this next sections 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

$$S[l_T] \doteq l_{no},$$

where l_T is a left hand side in Figure 6, 2, and l_{no} a left hand side in Figure 3, 4 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 special constructs to mirror the syntax of the reduction rules in the lambda calculus, and to represent the rule-schemas as a finite set of extended first-order rules. 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, and iii) a special construct $\mathsf{Ch}(\ldots)$ to represent binding chains of variable length as they occur in reduction rules.

The presented unification algorithm is applicable to terms with the mentioned extra constructs.

3.1 Many Sorted Terms and Contexts

Let $\mathcal{S} = \{S_1, S_2, \ldots\}$ be a set of *sort symbols* and Σ be a *many sorted* signature of function symbols, where for each function symbol an arity sort $sa : \Sigma \to \mathcal{S}^*$

and a result sort $sr : \Sigma \to \mathcal{S}$ is given. If $sa(f) = (S_1, \ldots, S_n), sr(f) = S$ we usually write $f :: S_1 \times \ldots \times S_n \to S$ and call it *the sort of* f.

***FO*-terms:** Let $\mathcal{V}_{FO}$ be a $\mathcal{S}$-sorted set of *first order variables*, where $\mathcal{V}_{FO}^S$ are pairwise disjoint sets of variables of sort S. The set of *$\mathcal{S}$-sorted first order terms* $\mathcal{T}_{FO}(\Sigma)$ is inductively defined by: $\mathcal{V}_{FO}^S \in \mathcal{T}_{FO}^S(\Sigma)$ if $S \in \mathcal{S}$ and $f(s_1, \ldots, s_n) \in \mathcal{T}_{FO}^S(\Sigma)$ if $f \in \Sigma$, $f :: S_1 \times \ldots \times S_n \to S$ and $s_i \in \mathcal{T}_{FO}^{S_i}(\Sigma)$ for $i = 1, \ldots, n$. We also use $sr(x)$ to denote the sort of the *FO*-variable x.

***CV*-terms:** For a set of sorts symbol $\mathcal{S}$ and each pair of sorts $S = (S_1, S_2) \in \mathcal{S}$ we define a set of *context variables* $\mathcal{V}_{CV}^S = \{X, Y, Z, \ldots\}$ of sort $S_1 \to S_2$, and we set $\mathcal{V}_{CV} = \bigcup_{S \in \mathcal{S} \times \mathcal{S}} \mathcal{V}_{CV}^S$. Context variables can be regarded as unary function symbol, so we abuse the notation *sa* and *sr* to denote their arity sort and result sort. With $\mathcal{T}_{CV}(\Sigma)$ we denote the following inductively defined set of $\mathcal{S}$-sorted terms over $\mathcal{V}_{FO} \cup \mathcal{V}_{CV}$: $\mathcal{T}_{FO}^S \subseteq \mathcal{T}_{CV}^S$ for $S \in \mathcal{S}$ and $X(s) \in \mathcal{T}_{CV}^S(\Sigma)$ if $X \in \mathcal{V}_{CV}$ $X :: S_1 \to S$ and $s \in \mathcal{T}_{CV}^{S_1}(\Sigma)$. *Positions* are strings of integers defining paths in terms. $s|_p$ is the subterm of s at position p and $s[t]_p$ is the term where $s|_p$ is replaced by t at position p. If s is a term, then $Var_O(s)$ is the set of O-variables occurring in s, with $O \in \{FO, CV\}$ and we set $Var(t) = Var_{FO}(t) \cup Var_{CV}(t)$.

Contexts are terms with one hole. As notation we use $t[\cdot]_p$, where p is the position of the hole. The sort of $t[\cdot]_p$ is $S_1 \to S_2$ if $t \in \mathcal{T}_{CV}^{S_2}$ and $sr(t|_p) = S_1$.

A term s without occurrences of variables is called *ground*. We also allow sorts without any ground term, also called *empty sorts*. These are sorts, such that there is no function symbol f in Σ with $sr(f) = S$. The term s is called *almost ground*, if for every variable $x \in \mathcal{V}_{FO}^S$ in s, the sort of x is an empty sort.

An *FO-substitution* is a mapping $\sigma_{FO} : \mathcal{V}_{FO} \to \mathcal{T}_{FO}$ such that $\sigma(x) = x$ except for a finite set and $\forall x \in \mathcal{V}_{FO} : sr(x) = sr(\sigma_{FO}(x))$. Analogously a *CV-substitution* maps context variables to context of the correct sort. We confuse σ with the tuple $(\sigma_{FO}, \sigma_{CV})$ and extend it to arbitrary terms in the usual way.

3.2 Encoding of LR-Expressions as Terms

The sort and term structure according to the expression structure of the lambda calculus LR (from Section 2.1) is as follows. Let $\mathcal{S}_1 = \{Exp, BV, Bind\}$ be a set of *free sorts* for expressions, bound variables, bindings (of the form $x = s$) and $\mathcal{S}_2 = \{Env\}$ be a set of *theory sorts* for environments (i.e. sets of bindings); we set $\mathcal{S}_{LR} = \mathcal{S}_1 \cup \mathcal{S}_2$. The following *free function* symbols are used to encode the corresponding LR-expressions in Σ_{LR}:

$$\Sigma_1 = \left\{ \begin{array}{l} app :: Exp \times Exp \to Exp,\ var :: BV \to Exp, \\ lam :: BV \times Exp \to Exp,\ seq :: Exp \times Exp \to Exp, \\ let :: Env \times Exp \to Exp,\ \ bind :: BV \times Exp \to Bind \end{array} \right\}$$

The *LR*-calculus further contains *constructor-* and *case expressions*, which we encode by the following function symbols in Σ_1:

1. For every constructor c from LR, there is a free function symbol c of arity $ar(c)$ of sort $\underbrace{Exp \times \ldots \times Exp}_{ar(c)} \to Exp$.

2. For every type T, there is a function symbol $case_T$ of arity $1 + \sum_{i=1,\ldots,|T|}(ar(c_{T,i})+1)$. The sort of $case_T$ is $Exp \times BV \times \ldots \times BV \times Exp \ldots \to Exp$. The first argument is the to-be-cased expression, then there are $|T|$ groups of arguments, one group for every constructor. The group for the constructor c is represented by the sorts $\underbrace{BV \times \ldots \times BV}_{ar(c)} \times Exp$, where we assume, that the sequence of the constructors is fixed for every type.

For example, for type `List`, there is a function symbol $case_{\texttt{List}}$ of sort $Exp \times BV \times BV \times Exp \times Exp \to Exp$. Argument 1 is for the expressions to be cased, arguments 2, 3 are the pattern variables of the list-constructor, argument 4 is the result expression for the list-constructor, and argument 5 is the result expression for the empty-list constructor. The function symbol $case_{Bool}$ is of sort $Exp \times Exp \times Exp \to Exp$, which can be interpreted like an *if-then-else* with three arguments.

In addition there are two theory function symbols:

$$\Sigma_2 = \left\{ emptyEnv :: Env,\ env :: Bind \times Env \to Env \right\}$$

We set $\Sigma_{LR} = \Sigma_1 \cup \Sigma_2$. 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.

Definition 3.1. *We use the name* FO-LR *to denote the $\mathcal{S}_{LR}$-sorted set of terms $\mathcal{T}_{FO}(\Sigma_{LR})$ over the $\mathcal{S}_{LR}$-sorted set of first order variables $\mathcal{V}_{FO}$. These are the terms without context variables, and without variable chains (introduced below).*

The language CV-LR *is the language* FO-LR *extended by context variables of type Exp → Exp. I.e. other context variables are not permitted.*

Definition 3.2. *Given an* LR*-expression t (without any meta-constructs like context-symbols, or variable chains), the translation into a* FO-LR*-term is denoted as $[\![t]\!]$.*

Given an FO-LR*-term s, the backtranslation into* LR*, which is unique, is denoted as $[\![s]\!]^-$. In case this results in a (syntactically) illegal* LR*-expression, we simply say that the backtranslation is not defined (on this expression). Otherwise, we say that s is an* LR-syntactically correct *term. If the backtranslated expression $[\![t]\!]$ satisfies the DVC in* LR*, then we also say that t satisfies the DVC.*

We use the notion syntactically correct instead of LR-syntactically correct.

3.3 Context Classes

Context classes are required to correctly model the overlappings in LR. The transformations in Figure 6, 2 contain only C-contexts, whereas in Figure 3, 4 there are also $\mathcal{A}$- and $\mathcal{R}$-contexts, and the overlapping also requires surface contexts $\mathcal{S}$.

FO-LR contexts are encodings of the respective contexts of $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{S}$-context where the hole is of sort *Exp*. We also say that they belong to context class $\mathcal{C}$, $\mathcal{A}$, or $\mathcal{S}$, respectively. If an almost ground *FO-LR* context C is of context class $\mathcal{C}$, $\mathcal{A}$, or $\mathcal{S}$, then this is denoted as $C \in \mathcal{C}$, $C \in \mathcal{A}$, or $C \in \mathcal{S}$, respectively. There is a natural partial order on context classes: $\mathcal{A} < \mathcal{S} < \mathcal{C}$, reflecting the subset ordering. The minimal context class of a context C is denoted as $cc(C)$. Also arbitrary non-almost ground contexts in *CV-LR* are associated to context classes. If all context variables occurring in a context $t[\cdot]_p$ have a context class, then the context class of $t[\cdot]_p$ can easily be determined (via backtranslation to *LR*, where context variables are translated to arbitrary context of the appropriate class).

For a term context $t[\cdot]_p$ that has no counterpart in *LR*, *cc* is undefined, e.g. $cc(lam([\cdot], s)) = \texttt{CUD}$. We define *CUD* as the greatest element in the partial order of the context classes, i.e. $\mathcal{A} < \mathcal{S} < \mathcal{C} < CUD$ (this is used in the side conditions of the **DC** unification rule from **??**).

3.4 Context Variables

Similarly to *FO-LR*-contexts, context variables come equipped with a context class, which is either $\mathcal{A}, \mathcal{S}$ or $\mathcal{C}$. With $cc(X)$ we denote the minimal class of $X \in \mathcal{V}_{CV}$. Substitutions have to respect the context class of context variables: if X is a context variable, then $\sigma(X)$ must be a context with $cc(\sigma(X)) \leq cc(X)$.

3.5 Encoding of letrec-environments

To model the multi-set property of letrec-environments, i.e., that bindings can be reordered, we use the equational theory of *left-commutativity* (abbreviated by *LC*).

Definition 3.3. *The equational theory* LC *of a left-commutative function symbol env is defined by the following axiom:*

$$LC_{env} := \{env(x, env(y, z)) = env(y, env(x, z))\}.$$

It is denoted by $=_{LC}$. *We also define* $=_{LC}$ *on* FO-LR*-contexts, in the natural way, which is without problems, since the application of* LC*-axioms keeps the number of holes.*

For the *LC*-theory and unification modulo *LC* see [DPR06,DPR98,DV99]).

Lemma 3.4 (Properties of *LC*). *Let* s, t *be* FO-LR *terms.*

1. $s =_{LC} t$ *implies that* $|s| = |t|$, $Var(s) = Var(t)$ *and the root symbols of* s, t *agree.*
2. *For all* $n \geq 0$ *and* $f \in \Sigma_2^n$: $f(s_1, \ldots, s_n) =_{LC} f(t_1, \ldots, t_n)$ iff $s_1 =_{LC} t_1, \ldots, s_n =_{LC} t_n$.

3. *If* $s =_{LC} t$, *and the root symbol of* s *is* env, *then* s, t *are of the form* $s = env(s_1, env(s_2 \dots env(s_n, s_{n+1}) \dots))$, *and* $t = env(t_1, env(t_2 \dots env(t_n, t_{n+1}) \dots))$, *where all* s_i, t_i, $i = 1, \dots, n+1$ *do not have* env *as root symbol, and the following holds: (i)* $s_{n+1} =_{LC} t_{n+1}$, *where either* $s_{n+1} = t_{n+1} = y$ *for some variable* y *or* $s_{n+1} = t_{n+1} = emptyEnv$; *(ii) there is a bijection* $\phi : \{1, \dots, n\} \to \{1, \dots, n\}$ *such that* $s_i =_{LC} t_{\phi(i)}$ *for all* $i = 1, \dots, n$.
4. $env(s_1, s_2) =_{LC} env(t_1, t_2)$ iff $s_1 =_{LC} t_1 \wedge s_2 =_{LC} t_2$ *or* $\exists z : s_2 =_{LC} env(t_1, z) \wedge t_2 =_{LC} env(s_1, z)$.
5. *If* $t[\cdot]_p =_{LC} s[\cdot]_q$, *for* FO-LR-*application contexts* $t[\cdot]_p, s[\cdot]_q$, *then* $p = q$. *If* $t[x]_p =_{LC} s[y]_q$ *for variables* x, y, *then also* $x =_{LC} y$ *and hence* $x = y$.
6. *If* $bind(y, t_1[var(x_1)]_p) =_{LC} bind(z, t_2[var(x_2)]_q)$ *and* $cc(t_1[\cdot]_p) = cc(t_2[\cdot]_q) = \mathcal{A}$, *then* $y = z$, $x_1 = x_2$, $t_1[\cdot]_p =_{LC} t_2[\cdot]_q$, *and* $p = q$.

Proof. The $\Leftarrow$-directions of claims 2 and 4 are trivial. For item 1 and the $\Rightarrow$-directions of items 2 and 4, one can use the fact, that $s =_{LC} t$ implies that there exists an $n \geq 0$ such that $s \overset{n}{\leftrightarrow}_{LC} t$.

The claim of item 3 can easily be proved by induction on the length of an LC-equality deduction $env(s_1, s_2) \overset{n}{\leftrightarrow}_{LC} env(t_1, t_2)$. The claim of item 4 is a consequence of item 3. The claim of item 5 follows from previous items, by induction on the structure of the contexts, and since application contexts do not have a function symbol env as a root symbol of a subcontext on the hole path. □

It is convenient to have a notation for nested env-expressions: $env^*(\{t_1, \dots, t_m\} \cup r)$ denotes the term $env(t_1, env(t_2, \dots, env(t_m, r) \dots))$, where we assume that the root symbol of r is different from env. Due to our assumptions on terms of sort *Env* and the notation, only the constant *emptyEnv*, and a variable of sort *Env* are possible for r. We also use the notation $env^*(M_1 \cup \dots \cup M_k \cup r)$, where we always assume that the rightmost expression in the union is of type *Env*. The convenience of the notation can be seen in the following situations: If r is a variable, then instantiating it with $env^*(M' \cup r')$ results in $env^*(M_1 \cup \dots \cup M_k \cup M' \cup r')$. The components in the multi-set may only be expressions of type *Bind*, i.e., variables of type *Bind* or expressions with top symbol *bind*. (We will later allow additional constructs (chains) in the multi-sets).
Lemma 3.4 immediately implies:

Lemma 3.5. *For* FO-LR-*terms the following holds:*
$env^*(\{s_1, \dots, s_m\} \cup r_1) =_{LC} env^*(\{t_1, \dots, t_n\} \cup r_2)$ iff $m = n$, $r_1 = r_2 = y$ *for some variable* y *or* $r_1 = r_2 = emptyEnv$; *and there is a bijection* $\phi : \{1, \dots, n\} \to \{1, \dots, n\}$, *such that* $s_i =_{LC} t_{\phi(i)}$.

We write $env(M \cup r)$ to denote the environment term where $M = \{s_1, \dots, s_n\}$ is a set of (encodings of) bindings.

3.6 The Predecessor Relation

Definition 3.6 (Predecessor relation on bindings). *We consider again the language of* FO-LR *terms. Let* $env^*(M \cup r)$ *be an env-term with two bindings* s_1, s_2 *as elements of* M.

We define the predecessor relation $\prec$ *for bindings (in the same environment):* $s_1 \prec s_2$ iff s_1 *is of the form* $bind(x, s)$ *and* s_2 *is of the form* $bind(y, t[var(x)]_p)$ *and* $x \neq y$, *where* x, y *are* BV*-sorted variables,* s *is some term of sort Exp and* $t[\cdot]_p$ *is an application context (i.e.* $cc(t[\cdot]_p) = \mathcal{A}$*).*

The predecessor relation describes the criterion by which bindings are chained in environment terms: Two binding terms (terms with root symbol *bind*) are *chained* if they both have different binders and one binder occurs in the bound expression s of the other binding at a position p such that $s[\cdot]_p$ is an application context. A sequence of bindings $s_1 \prec s_2 \prec \ldots \prec s_n$ is called a (first order) *binding chain.*

Example 3.7. We have $bind(x, app(var(y), r)) \prec bind(z, app(var(x), r'))$. It is also possible that the variables y, z are equal. In this case we have $bind(x, app(var(y), r)) \prec bind(y, app(var(x), r'))$, as well as $bind(y, app(var(x), r')) \prec bind(x, app(var(y), r))$, which means that there may be cycles.
Note that the relation $bind(y, t[var(x)]_p) \prec bind(z, t'[var(y)]_q)$ implies that the position q is unique, since we only consider application contexts.

Lemma 3.8. *For a* LR*-syntactically correct* FO-LR*-term the following properties hold:*

1. *In a term* $env^*(\{s_1, \ldots, s_m\} \cup r)$, *the terms* s_i *are bindings of the form* $bind(x_i, s_i')$, *where* x_i *is a* BV*-sorted variable, and the variables* x_i *are pairwise different.*
2. *The term* r *is either a variable of sort Env or the constant emptyEnv.*

Lemma 3.9. *For a* LR*-syntactically correct* FO-LR*-term with a subterm* $env^*(M \cup r)$, *the following holds:*

1. *For every binding* $s \in M$*: if there is a further binding* $s' \in M$ *with* $s' \prec s$, *then* s' *is unique.*
2. *For every binding* $s \in M$*: There is a unique descending chain* $s \succ s_1 \succ s_2 \ldots$ *with* $s_i \in M$. *Either the chain does not terminate, or there is a unique minimal binding* $s' \in M$ *reachable from* s *via* $\succ$*-steps.*

Proof. The claims follow from syntactical correctness and the definition of $\prec$. □

Note that the reverse of Lemma 3.9(1) does not hold, sind for example for $s_1 := bind(x, r_1), s_2 := bind(y, app(x, r_2))$, $s_3 := bind(z, app(x, r_3))$, we have $s_1 \prec s_2$ and $s_1 \prec s_3$, and s_1, s_2, s_3 are permitted to occur in the same environment term in an LR-syntactically correct term, since x, y, z are different variables.

The syntactical correctness of terms impose a restriction on the possibilities that bindings chained by the predecessor relation $\prec$ can be equated in environment terms by the LC congruence. We will elaborate on this in the following example and lemmas.

Example 3.10. Suppose the following equations between syntactically correct environment terms with chained bindings are given:

$$env(bind(x_1, s), env(bind(x_2, t[var(x_1)]_p), r)) \\ =_{LC} env(bind(y_1, s'), env(bind(y_2, t'[var(y_1)]_q), u))$$
$$bind(x_2, t[var(x_1)]_p) =_{LC} bind(y_2, t'[var(y_1)]_q)$$

Where $cc(t[\cdot]_p) = cc(t'[\cdot]_q) = \mathcal{A}$ and $x_1 \neq x_2$. By Lemma 3.4 (6) we have $x_1 = y_1, x_2 = y_2, t[\cdot]_p = t'[\cdot]_q$ and $p = q$. Now suppose, that $bind(x_1, s) \neq_{LC} bind(y_1, s')$. Then by Lemma 3.4 there is a z, such that $r =_{LC} env(bind(y_1, s'), z)$ and $u =_{LC} env(bind(x_1, s), z)$. But this violates the assumption of syntactical correctness, since both environment terms would contain the variable y_1 twice at a binder position, hence for the above example $bind(x_2, t[x_1]_p) =_{LC} bind(y_2, t'[y_1]_q)$ implies $bind(x_1, s) =_{LC} bind(y_1, s')$.

Lemma 3.11. *Let $s := env^*(M_1 \cup r_1)$ and $t := env^*(M_2 \cup r_2)$ be* LR*-syntactically correct environment terms with $s_1, s_2 \in M_1$, $t_2 \in M_2$, $s_1 \prec s_2$, $s =_{LC} t$ and $s_2 =_{LC} t_2$. Then there is some $t_1 \in M_2$ with $t_1 \prec t_2$ and $s_1 =_{LC} t_1$.*

Proof. Lemma 3.5 implies that there is some $t_1 \in M_2$ with $s_1 =_{LC} t_1$. Syntactic correctness, the preconditions of this lemma, and Lemma 3.4 imply that $s_2 = bind(x, s_2')$, and $t_2 = bind(x, t_2')$ for some variable x, and $s_2' =_{LC} t_2'$. Moreover, $s_2' = s_2''[y_1]_p$, and $t_2' = t_2''[y_1]_p$ and $s_2''[\cdot]_p, t_2''[\cdot]_p$ are application contexts, because s_2, t_2 have predecessors in M_1, M_2. Now $s_1 \prec s_2$ and $t_1 \prec t_2$ imply that $s_1 = bind(y_1, s_1')$ and $t_1 = bind(y_1, t_1')$. Syntactical correctness implies that M_1 contains at most one term with binding variable y_1, and the same for M_2. From $s =_{LC} t$ and Lemma 3.4 it follows that $s_1 =_{LC} t_1$ must hold. □

Definition 3.12. *In a syntactically correct environment term $env^*(\{s_1, \ldots, s_n\} \cup M \cup r)$ with $s_1 \prec \ldots \prec s_n$ we call $s' \in \{s_1, \ldots, s_n\} \cup M$ the* origin *of the chain if s' is the minimal binding in $\{s_1, \ldots, s_n\} \cup M$ reachable from s_i through $\succ$-steps.*

In the language FO-LR*, a binding $bind(x, t)$ is called a* value binding*, if t is a value, i.e. if t is the encoding of an abstraction or of a constructor application.*

Note that $s' \prec s$ is not possible for a value binding s.

Lemma 3.13. *Let $env^*(\{s_1, \ldots, s_m\} \cup M_1 \cup r_1)$ and $env^*(\{t_1, \ldots, t_n\} \cup M_2 \cup r_2)$ be* LR*-syntactically correct environment terms with $s_1 \prec s_2 \prec \ldots \prec s_m$ and $t_1 \prec t_2 \prec \ldots \prec t_n$.*

If $env^*(\{s_1, \ldots, s_m\} \cup M_1 \cup r_1) =_{LC} env^*(\{t_1, \ldots t_n\} \cup M_2 \cup r_2)$ and $s_i =_{LC} t_j$ for some $1 \leq i \leq m, 1 \leq j \leq n$ then

1. *If* $i > j$ *then* $s_{i-j+1} =_{LC} t_1, \ldots, s_{i-1} =_{LC} t_{j-1}$.
2. *If* $i < j$ *then* $s_1 =_{LC} t_{j-i+1}, \ldots, s_{i-1} =_{LC} t_{j-1}$.
3. *If* $i = j$ *then* $s_1 =_{LC} t_1, \ldots, s_{i-1} =_{LC} t_{j-1}$.
4. *If* s_1 *and* t_1 *are origins of the chains, respectively, and* s_1 *or* t_1 *is a value binding, then* $i = j$, $s_1 =_{LC} t_1, \ldots, s_{i-1} =_{LC} t_{i-1}$ *and* s_1, t_1 *are both value bindings.*

Proof. This follows by iterated application of Lemma 3.11. Item 4 follows from Lemma 3.11 and since in addition value bindings cannot be equal to other non-value bindings in the chains. □

3.7 Encoding of Binding Chains with Variable Length

We extend the set of *CV-LR*-terms by a special construct $\mathsf{VCh}, \mathsf{NCh}$ (see below) to encode *chains of bindings* of variable length (as they occur in *LR*). We denote this set of terms with the special chain constructs by $\mathcal{T}_{CH}$ (or by *CH-LR*).

Remark 3.14 (Encoding through the use of schematization). The constructs for binding chains describe (possibly infinite) sets of terms. They bear some similarities to term schematizations used in [Sal92,Her92,HG97]. The main difference is that our schematization describes only a very specific set of terms whereas other schematizations (like *R*-terms or primal grammars) can be used to describe arbitrary terms with a recurrent structure. Also our schematization allows the introduction of new variables, a feature that is not supported by *R*-terms. Primal grammars with marked variables allow the generation of new variables, but for them unification is undecidable [Sal93], whereas our schematization has a decidable unification problem (under some conditions).

Syntax: Let $\mathcal{N} = \{1, 2, \ldots\}$ be the set of natural numbers and $\mathcal{V}_{\mathcal{N}} = \{l_1, l_2, \ldots\}$ the set of $\mathcal{N}$-variables. The set of $\mathcal{N}$-terms $\mathcal{T}_{\mathcal{N}}$ is defined as the smallest set such that $\mathcal{N} \cup \mathcal{V}_{\mathcal{N}} \subseteq \mathcal{T}_{\mathcal{N}}$ and $e_1 + e_2 \in \mathcal{T}_{\mathcal{N}}$ when $e_1, e_2 \in \mathcal{T}_{\mathcal{N}}$. We use the two special symbols for *chains*: VCh and NCh which can be regarded as a function symbols of arity 3 that take as arguments two BV-sorted variables and a $\mathcal{T}_{\mathcal{N}}$-term. The symbol Ch is used to denote either chain construct. The occurrence of these constructs is restricted to environment terms, i.e. in a term $env^*(M \cup r)$ a chain can occur in M; informally the Ch-construct is a context of sort $Env \to Env$, and hence can be seen as $BV \times BV \times Int \to (Env \to Env)$. Our union-notation for env^* also permits the view that the resulting sort is Set of *Binds*.

Remark on Occurrence and Use of Chains: Note that the occurrences of the constructs $\mathsf{VCh}, \mathsf{NCh}$ is rather limited in equations: There may be at most one NCh-construct and at most two VCh-constructs, and both are in the top letrec of equations. These number of occurrences is not increased in the data structure. The following definition of semantics exploits that there is at most one occurrence of an NCh-construct.

For several occurrences of NCh, the definitions would have to be generalized.

Semantics of *CH-LR*-terms The chain constructs are used to represent special sets of *CV-LR*-terms: Sets of binding terms with variable size where the bindings are connected. In a chain expression $\mathsf{NCh}(x, y, l)$ the variables x and y denote variables that can occur somewhere else (for example in a superterm), and represent the end- and start-point of the chain. The $\mathcal{N}$-term l controls the size of the set of bindings. The process of *unfolding* a chain into a sequence of bindings can be formalized in the following way:

$$\begin{array}{ll} \mathit{unfold}(\mathsf{NCh}(x,y,i)) & = \mathsf{NCh}(x,y,i) \\ \mathit{unfold}(\mathsf{NCh}(x,y,1)) & = \{\mathit{bind}(y, A(\mathit{var}(x)))\} \\ \mathit{unfold}(\mathsf{NCh}(x,y,n)) & = \{\mathit{bind}(z, A(\mathit{var}(x)))\} \cup \mathit{unfold}(\mathsf{NCh}(z,y,n-1)) \\ \mathit{unfold}(\mathsf{NCh}(x,y,l_1+l_2)) & = \mathit{unfold}(\mathsf{NCh}(x,z,l_1)) \cup \mathit{unfold}(\mathsf{NCh}(z,y,l_2)) \end{array}$$

where $i \in \mathcal{V}_{\mathcal{N}}, n \in \mathcal{N}, l_1, l_2 \in \mathcal{T}_{\mathcal{N}}$, z is a fresh BV-sorted variable and A is a fresh context variable of class $\mathcal{A}$.

Unfolding VCh-constructs is defined analogously, with the difference that binding terms are unfolded omitting application context variables, i.e. $\mathit{unfold}(\mathsf{VCh}(x, y, 1)) = \{\mathit{bind}(y, \mathit{var}(x))\}$ and so on. The operation of unfolding arbitrary Ch-terms is denoted by unfolT. We use the abbreviation $\mathsf{NCh}(x, y, \overline{l})$ instead of $\mathit{unfold}(\mathsf{NCh}(x, y, l))$ and $\overline{s}$ instead of $\mathit{unfolT}(s)$. Thus unfolT is defined by $\overline{v} = v$, if v is a (first order or context) variable; $\overline{\mathsf{Ch}(x, y, l)} = \mathsf{Ch}(x, y, \overline{l})$ and $\overline{f(s_1, \ldots, s_n)} = f(\overline{s_1}, \ldots, \overline{s_n})$. In contrast to [Sal92] the unfolding of a chain may introduce new (BV sorted) variables and context variables. With $\mathit{IVar}(\mathsf{Ch}(x, y, l)) := \mathit{Var}(\mathsf{Ch}(x, y, \overline{l})) \setminus \{x, y\}$ we denote the set of variables introduced through the unfolding of the chain construct. The variables introduced through the unfolding of chains in a Ch-term are chosen as distinct to all others variables in the context where the unfolding takes place, i.e. $\mathit{Var}(t) \cap \mathit{IVar}(\overline{t}) = \emptyset$.

An $\mathcal{N}$-substitution is a mapping $\sigma_{\mathcal{N}} : \mathcal{V}_{\mathcal{N}} \to \mathcal{T}_{\mathcal{N}}$. We use σ to denote the triple $(\sigma_{FO}, \sigma_{CV}, \sigma_{\mathcal{N}})$, which is a slight extension of substitutions.

The application of substitutions to arbitrary Ch-terms is defined as follows (with $O \in \{FO, CV, \mathcal{N}\}$): $\sigma(x) = \sigma_O(x)$, where $x \in \mathcal{V}_O$, $\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n))$, where f is an n-ary function symbol and $s_i \in \mathcal{T}_{FO}$, $\sigma(X(s)) = \sigma_{CV}(X)(\sigma(t))$ and $\sigma(\mathsf{Ch}(x, y, l)) = \mathsf{Ch}(\sigma_{FO}(x), \sigma_{FO}(y), \sigma_{\mathcal{N}}(l))$. We say a substitution σ is O-ground if there occur no O-variables in the image of σ. Unfoldings are also extended to FO and substitutions, i.e. $\overline{\sigma} = \{x_1 \mapsto \overline{s_1}, \ldots, x_n \mapsto \overline{s_n}\}$.

The set of *CV-LR*-terms represented by the *CH-LR*-term r is defined as

$$\mathcal{T}_{CV}(r) = \{\overline{\sigma_{\mathcal{N}}(r)} \mid \sigma_{\mathcal{N}} \text{ is } \mathcal{N}\text{-ground and } \mathit{dom}(\sigma_{\mathcal{N}}) = \mathit{Var}_{\mathcal{N}}(r)\}$$

and the set of *FO-LR*-terms represented by r is

$$\mathcal{T}_{FO}(r) = \{\sigma_{CV}(s) \mid s \in \mathcal{T}_{CV}(r), \sigma_{\mathcal{V}_{CV}} \text{ is } CV\text{-ground and } \mathit{dom}(\sigma_{CV}) = \mathit{Var}_{CV}(s)\}.$$

Example 3.15. The chain $\mathsf{NCh}(x, y, l)$ stands for the following sets of *CH-LR*-terms: $\{\{\mathit{bind}(y, A_1(\mathit{var}(x)))\}$ when choosing $\{l \mapsto 1\}$, and $\{\mathit{bind}(x_1, A_1(\mathit{var}(x))), \mathit{bind}(y, A_2(\mathit{var}(x_1)))\} \ldots\}$ when choosing $\{l \mapsto 2\}$.

Remark 3.16. For chains the variable names introduced through an unfolding are somehow irrelevant, i.e. we allow renaming of such introduced variables.

However, this representation semantics would not work for the NCh-constructs during unification, if context variable names are connected to another NCh-constructs, for example if a NCh-construct is syntactically duplicated and the the two copies should be the same. The reason is that the unfolding introduces fresh names of free context variables.

Luckily, in all the considered unification problems between left hand sides (see Definition **??**) there will be at most one occurrence of an NCh-construct, which is never connected to another such NCh-construct.

Lemma 3.17. *Let* $\mathsf{Ch}(x,y,l), \mathsf{Ch}(x,y,l')$ *be two chains (of the same type, i.e. both* VCh *or* NCh*) and* σ *be a substitution, such that* $\sigma(l) = l'$*. Then there exists a variable renaming* $\rho : IVar(\mathsf{Ch}(x,y,\sigma(l))) \to IVar(\mathsf{Ch}(x,y,l'))$ *(i.e. a bijection between the variables introduced through the unfolding of both chains) such that* $\rho(\mathsf{Ch}(x,y,\overline{\sigma l})) = \mathsf{Ch}(x,y,\overline{l'})$*.*

Proof. Through induction over l'. We treat only the case where $\mathsf{Ch} = \mathsf{NCh}$. For $l' = i \in \mathcal{V}_{\mathcal{N}}$ we set $\rho = \emptyset$. For $l' = 1$ let the unfoldings be $\mathsf{NCh}(x,y,\overline{\sigma l}) = \{bind(y, A(var(x)))\}$ and $\mathsf{NCh}(x,y,\overline{l'}) = \{bind(y, A'(var(x)))\}$. Then we set $\rho = \{A \mapsto A'\}$. In the case $l' = 1 + n$ let the unfoldings be $\mathsf{NCh}(x,y,\overline{\sigma l}) = \mathsf{NCh}(x,z,\overline{1}) \cup \mathsf{NCh}(z,y,\overline{n}) = \{bind(y, A(var(z)))\} \cup \mathsf{NCh}(z,y,\overline{n})$ and $\mathsf{NCh}(x,y,\overline{l'}) = \{bind(y, A'(var(z')))\} \cup \mathsf{NCh}(z',y,\overline{n})$ respectively. We set $\rho = \{z \mapsto z', A \mapsto A'\}$ and compose this renaming with the renaming yielding from the induction hypothesis applied to the renaming of two chains of length n. □

In general there does not exist a bijective renaming between two unfolded chains of different types. However, after instantiating all context variables in an NCh, the chains may become renamings of each other. Due to the construction of the unification algorithm, this may happen only after a complete expansion of the chain making the context variables explicit.

We regard two chains (of the same type) as equivalent when they unfold to the same set of bindings, modulo renaming of the variables that are introduced through the unfolding, i.e. when they have the same starting and ending (BV sorted) variables and the same length. This is semantically justified.

Definition 3.18. *Two* CH-LR*-terms* s_1 *and* s_2 *are* equivalent modulo LC *(*$s_1 =_{LC} s_2$*) iff for all* $\mathcal{N}$*-ground substitutions* $\sigma_{\mathcal{N}}$ *with* $dom(\sigma_{\mathcal{N}}) = Var_{\mathcal{N}}(s_1) \cup Var_{\mathcal{N}}(s_2)$ *there exists a renaming* ρ *from the variables introduced through the unfolding of* s_1 *to those in* s_2 *such that* $\rho(\overline{\sigma_{\mathcal{N}}(s_1)}) =_{LC} \overline{\sigma_{\mathcal{N}}(r_2)}$*.*

Example 3.19. Let $s = env^*(\mathsf{VCh}(x,z_1,l_1) \cup \{bind(z_2, var(z_1))\} \cup \mathsf{VCh}(z_2,y,l_2) \cup r)$ and $t = env^*(\mathsf{VCh}(x,y,l_1 + 1 + l_2) \cup r)$ and suppose $IVar(s) = IVar(\mathsf{VCh}(x,z_1,l_1)) \cup \mathsf{VCh}(z_2,y,l_2) \cup \{z_1,z_2\}$ then s and t are equivalent under the renaming $\rho : IVar(s) \to IVar(t)$ (constructed as in the proof of lemma 3.17).

The bindings introduced through the unfolding of chains are ordered by the predecessor relation $\prec$.

Lemma 3.20. *If* $\mathsf{Ch}(x, y, \overline{l}) = \{s_1, \ldots, s_n\}$ *then* $s_1 \prec \ldots \prec s_n$*, and these are the only* $\prec$*-relations if* $x \neq y$*.*

Proof. Follows from the definition of chain unfolding.

We extend Definition 3.2 of the translation $[\![\cdot]\!] : LR \to FO\text{-}LR$ to translate LR-meta-expressions (used in the definition of the reduction rules) into *CH-LR* terms (see also Fig. 12).

Definition 3.21. *We define the two sets* lhs_T, lhs_{no} *of encoded left hand sides (*lhs*) of unrestricted and normal order reduction rules, respectively, of the calculus LR. In order to keep the rules finite, we restrict the types to the type* `Bool` *with constructors* `True`, `False` *and* `List` *with constructors* `Nil` *and* `Cons`*.*

1. lhs_T
   is the following set of encodings of left hand sides of an unrestricted LR *reduction rule (see figures 6, 2), where first the rules are instantiated:*
   (a) *The phrase "v is a value" will lead to instantiations into an abstraction* $\lambda x.t$ *and constructor terms, one possibility for every constructor.*
   (b) *Every mention of constructor c is instantiated to every possible constructor. This is also done accordingly with cases and types.*
   (c) *Rules with a* $\{x_{i+1} = x_i\}_{i=1}^m$*-expression are further instantiated resulting in two rules: one for* $m = 1$ *without a chain, and one for the case* $m > 1$*. Then the usual translation* $[\![\cdot]\!]$ *applies, but the chains* $\{x_{i+1} = x_i\}_{i=1}^m$ *(for the case* $m > 1$*) are translated into* $\mathsf{VCh}(x_1, x_m, N)$*, where* N *is a new integer variable. Note that the number* m *does not play any role here.*
2. lhs_{no}
   is the set of first-order encodings of left hand sides of normal order LR *reduction rule (see figures 3, 4) with the following procedure: For every left hand side, the following instantiations will generate variants of the rules:*
   (a) *The phrase "v is a value" will lead to instantiations into an abstraction* $\lambda x.t$ *and constructor terms, one possibility for every constructor.*
   (b) *Every mention of constructor c is instantiated to every possible constructor. This is also done accordingly with cases and types.*
   (c) *For rules which contains the the symbol R (reduction context), there will be four instances where R is replaced by one of the following possibilities:*
      i. A*,*
      ii. (`letrec` Env `in` A)*,*
      iii. (`letrec` $y_1 = A, Env$ `in` $A_2[y_1]$)*,*
      iv. (`letrec` $y_1 = A, \{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^m, Env$ `in` $A'[y_m]$)*.*
   (d) *Rules with an occurrence of* $\{x_{i+1} = x_i\}_{i=1}^m$*-expression or* $\{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^n$*-expressions are further instantiated to distinguish the cases* $m = 1$*, where the chain is omitted, and the case* $m > 1$*, as well as*

$n = 1$ and the case $n > 1$. Then $[\![\cdot]\!]$ is used for the first-order encoding, where the chain $\{x_{i+1} = x_i\}_{i=1}^{m}$ (for the case $m > 1$) is translated into $\mathsf{VCh}(x_1, x_m, N)$, where N is a new variable, and the chain $\{y_{i+1} = A_{i+1}[y_i]\}_{i=1}^{n}$ is translated into $\mathsf{NCh}(y_1, y_n, N')$ where N' is a fresh first-order variable. Note that neither n nor m play a role in the encoding.
For the left hand side of the rule cp-e a constraint will be added, saying that the context variable A_1 is not empty.

That the restriction to type `Bool` and `List` is sufficient has to be argued on a meta-level.

In an *env*-term $t = env^*(\mathsf{Ch}(x, y, l) \cup M \cup r)$ the chain $\mathsf{Ch}(x, y, l)$ has an origin in t if $\mathsf{Ch}(x, y, \overline{1})$ has an origin in t.

Definition 3.22. *A term $t \in \mathcal{T}_{CH}$ satisfies the* T-chain-restrictions *if*

1. *In t there occurs at most one* VCh*-construct and no* NCh*-constructs.*
2. *If a* VCh *occurrs in t then it has an origin, which is a value binding.*

A term $t \in \mathcal{T}_{CH}$ satisfies the N-chain-restrictions *if*

1. *In t there occurs at most one* VCh*-construct and at most one* NCh*-construct.*
2. *Every* Ch*-consruct in t has an origin.*
3. *If a* VCh *occurrs in t its origin is a value binding.*
4. *If the origin of a* NCh *in t is a value binding then the env-term, in which the chain occurs is of the form*

$$(env^*(\{bind(x, A(var(z)))\} \cup \mathsf{NCh}(x, y, k) \cup Env)$$

where A context variable of class $\mathcal{A}$, that is constraint to as non empty.

Lemma 3.23. *All $l_T \in lhs_T$ satisfy the T-chain-restrictionsand all $l_{no} \in lhs_{no}$ satisfy the N-chain-restrictions.*

Proof. Through inspection of the reduction rules in figures 6, 2, 3 and 4. The cases for item **??** are the normal order reduction rules (no, cp-e), (no, case) and (no, seq), wich are of the form

$$let(env^*(\{bind(x', v)\} \cup \mathsf{VCh}(x', z, k') \cup \{bind(x, A(var(z)))\} \cup \mathsf{NCh}(x, y, k) \cup Env), r)$$

□

The intuition behind item **??** of the above lemma is the following: For the encoded reduction rules, if the origin of a NCh-construct is a value binding then this NCh-chain is connected (trough $\prec$) over a binding ($bind(x, t[y']_p)$, where $t[\cdot]_p \neq [\cdot]$) to a VCh which is terminated by a value binding.

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

4.1 Unification of CH-LR-Terms

If two *CH-LR* terms s_1 and s_2 are to be unified, each term of the unfoldings in $\mathcal{T}_{CV}(s_1)$ and $\mathcal{T}_{CV}(s_2)$ have to be checked against each other, thus typically leading to a infinite set of unifiers. The goal of *CH-LR* unification is to compute (a finite complete set of) *unifiers* (that are substitutions that solve unification problems between *CH-LR*-terms modulo LC) thus yielding finite descriptions of sets of unifiers.

A *unification problem* is a pair (Γ, Δ), where $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a set of equations between *CH-LR* terms such that the terms s_i and t_i are of the same sort for every i, and every context variable is labelled with a context class symbol ($\mathcal{A}, \mathcal{S}$ or $\mathcal{C}$). Δ is a set of constraints: it consists of context variables that must not be instantiated by the empty context.

A *solution* σ of (Γ, Δ), with $\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a substitution σ according to the following conditions:

i) It instantiates variables by terms and context variables by contexts of the correct context class that are nontrivial if contained in Δ.
ii) It replaces chain constructs $\mathsf{Ch}(x, y, N)$ by a set of bindings according to the unfolding definition, and in case of $\mathsf{NCh}(x, y, N)$ the context variables are also replaced by ground contexts. In proofs, we add the exact information on the form $\mathsf{NCh}(x, y, N) \mapsto \mathit{bindingSet}$.
iii) $\sigma(s_i), \sigma(t_i)$ are almost ground for all i.
iv) $\sigma(s_i) =_{LC} \sigma(t_i)$ for all i.

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

Definition 4.1. *We consider the set of unification problems*

$$IP := \{\{S(s_1) \doteq s_2\} \mid s_1 \in lhs_T, s_2 \in lhs_{no}\}$$

where S is a context variable of context-class S. The terms s_1, s_2 are assumed to be variable disjoint, which can be achieved by renaming. The initial set Δ of context variables only contains the A_1-context in case s_2 comes from a (cp-e)-reductions. The pair (Γ, Δ) with $\Gamma \in IP$ is called an initial *LR*-forking-problem.

Proposition 4.2. *The following holds for each $P \in \{\{S(s_1) \doteq s_2\} \mid s_1 \in lhs_T, s_2 \in lhs_{no}\}$:*

1. *They are almost linear*
2. *There is at most one occurrence of a* NCh*-construct*
3. *There are at most two occurrences of* VCh*-constructs.*
4. *There are no variables of type Bind.*

Definition 4.3. *A* final unification problem S *derived from an initial LR-forking-problem* (Γ, Δ) *is either Fail or a set of equations* $s_1 \doteq t_1, \ldots, s_n \doteq t_n$, *such that* $S = S_{BV} \cup S_{\neg BV}$, *and every equation in* S_{BV} *is of the form* $x \doteq y$ *where* x, y *are of sort* BV *and every equation in* $S_{\neg BV}$ *is of the form* $x \doteq t$, *where* x *is not of sort* BV.

Proposition 4.4. *Given an initial LR-forking-problem* (Γ, Δ). *Then the equations in* $S_{\neg BV}$ *in the final unificaiotn problem are in DAG-solved form.*

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

Definition 4.5. *A final unification problem* S *derived from* Γ satisfies the distinct variable convention *(DVC), if for every derived solution* σ, *all terms in* $\sigma(\Gamma)$ *satisfy the DVC.*

Proposition 4.6. *The DVC-property of a final unification problem is decidable.*

Proof. If $t_1 \doteq t_2$ is the initial problem, then apply the substitution σ derived from S to t_1. The DVC is violated iff the following condition holds: Let M_{BV} be the set of BV-variables occurring in $\sigma(t_1)$. If $\sigma(t_1)$ makes two variables in M_{BV} equal, then the DVC is violated.

Example 4.7. We give an example that is not an initial one, but can also be treated: 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. Using the DVC-check, our unification can efficiently check alpha-equivalence of pure lambda-expressions that satisfy the DVC.

4.2 The Unification Rules

We proceed by describing a unification algorithm starting with initial *LR*-unification problems (Γ, Δ). It is intended to be complete for all common instances that represent *LR*-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.

Note that the initial equation is almost linear, hence the rules can be presented in a simplified way. For example substitution application can be avoided during unification.

We implicitly use symmetry of $\doteq$ if not stated otherwise. We divide Γ into a solved part S, (a final unification problem), and a still to be solved part P. We permit also context-equations in the solved part S. We usually omit Δ in the notation if it is not changed by the rule.

The following non-deterministic unification rules have to form:

$$\textbf{Name}\ \frac{Sys \qquad C}{\begin{array}{c}1)\ Sys_1\\ 2)\ Sys_2\\ \dots\\ n)\ Sys_n\end{array}}$$

where $Sys = S; P; \Delta$ is a system that consists of a set of solved equations S a set of still to be solved equations P and a set of constraints Δ on context variables and variable chains. There may also be additional conditions C that must be satisfied for the rule **Name** to be applied. If a given system *sys* matches the form of *Sys* and fulfils the conditions C then *sys* can be transformed into a system Sys_i where $1 \leq i \leq n$. They systems $S_1 \dots S_n$ represent the non-deterministical choises of transformations (they may be given implictly, e.g. "select an i with $1 \leq i \leq n$"). If a rule introduces new variables into a system, then these are chosen as fresh, i.e. distinct from all variables already present in the system.

Standard unification rules The standard unification rules can be seen in Figure 7.

The rules **Solve** pushes an equation in solved form ($x \doteq s$) into the set of solved equations S without an occurs-check or emlimination of x from the rest problem (this is due to the almost linearity of the initial LR-forking problems).

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

Dec $\dfrac{S;\ \{f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)\} \uplus P}{S;\ \{s_1 \doteq t_1, \dots, s_n \doteq t_n\} \cup P}$ If $f \neq env$

Fail $\dfrac{S;\ \{f(\dots) \doteq g(\dots)\} \uplus P}{Fail}$ If $f \neq g$.

Stuck-Fail $\dfrac{S;\ P \qquad P \neq \emptyset}{Fail}$ and no unification rule is applicable to P.

DVC-Fail $\dfrac{S;\ \emptyset}{Fail}$ If S is final and the DVC is violated w.r.t. the initial problem.

Fig. 7: Standard unification rules

The **DVC-Fail** rule discards unifiers that equate terms of different alpha-equivalence classes or capture free bound variables w.r.t. the initial *LR*-forking problem (see section 6.1).

Solving equations with context variables: Figure 8 shows unification rules to deal with equations that contain context variables. There is the rule **Empty-C** that guesses an arbitrary context variable in the problem P as empty or not empty (in which case the variable is marked by insertion into the constraint set Δ). The aim of the other rules is context decomposition of equations $X(s) \doteq f(t_1, \ldots, t_n)$, where we instantiate the prefix of X to f and guess where the hole can appear in this prefix, thereby taking into account the context class of X. The rules of Figure 8 enumerate all possible choices for f and hole positions $i, \ldots, n$ in dependence of the context class of X.

In the rules we use the notation $f(s_1, \ldots, s_n)[C]_i$ to denote the context $f(s_1, \ldots, s_{i-1}, C, s_{i+1}, \ldots, s_n)$.

Alternate Rules to solve $X(s) = f(t_1, \ldots, t_n)$: The rule **DC** from figure 9 subsumes the rules **Dec-C-App-Seq**, **Dec-C**, **Dec-C-Lam**, **Fail-C-Lam**, **Fail-C-Var**, **Dec-C-Case-1** and **Dec-C-Case-2**.

Explanation of the rules: To solve an equation of the form $X(s) \doteq f(t_1, \ldots, t_n)$ we use **DC** to guess an position p where s can appear in $f(t_1, \ldots, t_n)$. Let $p = i \cdot q$ with $i = 1, \ldots, n$, then s occurs in t_i at position q, which is recursively determined through solving $X'(s) \doteq t_i$. The position where the hole occurs in $f(t_1, \ldots, t_n)$ has to comply with the context class of X, which is stated in the side condition. E.g. to $X(s) \doteq lam(x, t)$ where $cc(X) = \mathcal{S}$ the rule **DC** is not applicable, because 1. $lam(X', t)$ is not an legal context (the hole can not occur at an position with a subterm of sort BV see 3.3, i.e. $cc(lam(X', t)) = CUD$) and 2. $cc(X) = \mathcal{S} < cc(lam(x, X')) = \mathcal{C}$ (i.e. in a $\mathcal{S}$ context the hole can not appear in the body of an abstraction).

We need the additional rule **DCL** to guess the position of the hole (which accommodates s) deep into the right hand side of an letrec-binding, which can not be achieved by **DC** because there are no context variables of sort $Exp \to Env$ (all context variables and context must be of sort $Exp \to Exp$ see 3.3)

The failure rules are mimicked through the fact, that the side condition of rule **DC** prevent certain kinds of applications, and therefore unification gets stuck, e.g. $X(s) \doteq var(x)$ can not be transformed by **DC**, because $cc(var(X')) = CUD > cc(X)$.

To solve equations with context variables of the form $X(s) \doteq Y(t)$, with $X, Y \in \Delta$, we use the rules from Figure 10. The idea is that either there is common prefix of the two contexts (in which case we guess this position via **Merge-Prefix**) or there is none (in which case we guess a common part of the two contexts and then a function symbol where the hole positions fork via **Merge-Fork**).

Empty-C $$\frac{S;\ P;\ X \notin \Delta \qquad X \text{ occurs in } P}{1)\ S;\ P;\ \{X\} \cup \Delta \qquad 2)\ \{X \doteq [\cdot]\} \cup S;\ \{X \mapsto [\cdot]\}P;\ \Delta}$$

Dec-C-App-Seq $$\frac{S;\ \{X(s) \doteq f(t_1, t_2)\} \uplus P;\ X \in \Delta \quad f \in \{app, seq\}}{\{X \doteq f(X', t_2)\} \cup S;\ \{X'(s) \doteq t_1\} \cup P;\ \Delta}$$
X' is a fresh context variable of the same context class as X.

Dec-C $$\frac{S;\ \{X(s) \doteq f(t_1, t_2)\} \uplus P;\ X \in \Delta \quad cc(X) \neq \mathcal{A},\ f \in \{let, app, seq\}}{\{X \doteq f(t_1, X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P;\ \Delta}$$
X' is a fresh context variable of the same context class as X.

Dec-C-Let $$\frac{S;\ \{X(s) \doteq let(t_1, t_2)\} \uplus P;\ X \in \Delta \quad cc(X) \neq \mathcal{A}}{\begin{array}{l}\{X \doteq let(env^*(\{bind(x, X')\} \cup z), t_2)\} \cup S;\\ \{env^*(\{bind(x, X'(s))\} \cup z) \doteq t_1\} \cup P;\ \Delta\end{array}}$$
X' is fresh of the same class as X and x, z are fresh variables of appropriate sort.

Dec-C-Lam $$\frac{S;\ \{X(s) \doteq lam(t_1, t_2)\} \uplus P;\ X \in \Delta \quad cc(X) = \mathcal{C}}{\{X \doteq lam(t_1, X')\} \cup S;\ \{X'(s) \doteq t_2\} \cup P;\ \Delta}$$
X' is a fresh context variable of the same class as X.

Fail-C-Lam $$\frac{S;\ \{X(s) \doteq lam(t_1, t_2)\} \uplus P;\ X \in \Delta \quad cc(X) \neq \mathcal{C}}{Fail}$$

Fail-C-Var $$\frac{S;\ \{X(s) \doteq var(x)\} \uplus P;\ X \in \Delta}{Fail}$$

Dec-C-Cons $$\frac{S;\ \{X(s) \doteq c(s_1, \ldots, s_n)\} \uplus P;\ X \in \Delta \qquad cc(X) \neq \mathcal{A}}{\begin{array}{l}\text{select an } i,\ 1 \leq i \leq n\text{:}\\ \{X \doteq c(s_1, \ldots, s_n)[X']_i\} \cup S;\ \{X'(s) \doteq s_i\} \uplus P;\ \Delta\end{array}}$$
where $n = ar(c)$ and X' is fresh of the same class as X.

Fail-C-Cons $$\frac{S;\ \{X(s) \doteq c(s_1, \ldots, s_{ar(c)})\} \uplus P;\ X \in \Delta \qquad cc(X) = \mathcal{A}}{Fail}$$

Dec-C-Case-1 $$\frac{S;\ \{X(s) \doteq case_T(s_1, \ldots, s_n)\} \uplus P;\ X \in \Delta}{\{X \doteq case_T(X', s_1, \ldots, s_n)\} \cup S;\ \{X'(s) \doteq s_1\} \uplus P;\ \Delta}$$
where $n = ar(case_T)$ and X' is fresh of the same class as X.

Dec-C-Case-2 $$\frac{S;\ \{X(s) \doteq case_T(s_1, \ldots, s_n)\} \uplus P;\ X \in \Delta \quad cc(X) \neq \mathcal{A}}{\begin{array}{l}\text{select an } i,\ 2 \leq i \leq n \text{ such that } so(s_i) = Exp\text{:}\\ \{X \doteq case_T(s_1, \ldots, s_n)[X']_i\} \cup S;\ \{X'(s) \doteq s_i\} \uplus P;\ \Delta\end{array}}$$
where $n = ar(case_T)$ and X' is fresh of the same class as X.

Fig. 8: Unification rules to solve equations with context variables

$$\textbf{DC}\ \frac{\{X(s) \doteq f(t_1,\ldots,t_n)\} \uplus P; X \in \Delta}{\begin{array}{l}\text{select an } i,\ 1 \le i \le n \\ \{X \doteq f(t_1,\ldots,t_n)[X']_i,\ X'(s) \doteq t_i\} \cup P\end{array}} \quad \begin{array}{c} cc(X) = cc(X') \ge cc(f(t_1,\ldots,t_n)[X']_i) \\ 1 \le i \le n\end{array}$$

$$\textbf{DCL}\ \frac{S;\ \{X(s) \doteq let(t_1,t_2)\} \uplus P;\ X \in \Delta}{\begin{array}{l}\{X \doteq let(env^*(\{bind(x,X')\} \cup z), t_2)\} \cup S; \\ \{env^*(\{bind(x,X'(s))\} \cup z) \doteq t_1\} \cup P;\ \Delta\end{array}} \quad \begin{array}{c} cc(X) = cc(X') \in \{\mathcal{S},\mathcal{C}\} \\ so(z) = Env,\ \text{fresh}\end{array}$$

Fig. 9: Decomposition rules to solve of the form $X(s) \doteq f(t_1,\ldots,t_n)$

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

Y' is a fresh variable of the same class as Y, and Z has context class $\min(cc(X), cc(Y))$.

Merge-Fork-A

$$\frac{S;\ \{X(s) \doteq Y(t)\} \uplus P;\ X,Y \in \Delta \qquad cc(Y) \ne \mathcal{A}}{\begin{array}{l}1)\ S;\ \{X \doteq Z(app(X',Y'(t))), Y \doteq Z(app(X'(s),Y'))\} \cup P;\ \Delta \\ 2)\ S;\ \{X \doteq Z(seq(X',Y'(t))), Y \doteq Z(seq(X'(s),Y'))\} \cup P;\ \Delta \\ 3)\ S;\ \{X \doteq Z(case_T(s_1,\ldots,s_n)[X']_1, [Y'(t)]_i), \\ \qquad Y \doteq Z(case_T(s_1,\ldots,s_n)[X'(s)]_1, [Y']_i)\} \cup P;\ \Delta \\ \quad so(case_T) = S_1 \times \ldots \times S_n \to S, S_1 = S_i = Exp, 2 \le i \le n\end{array}}$$

where X', Y' are fresh variables of the same class as X, Y, respectively, and Z is a fresh context variable of context class $\min(cc(X), cc(Y))$.

Merge-Fork-C

$$\frac{S;\ \{X(s) \doteq Y(t)\} \uplus P;\ X,Y \in \Delta \qquad cc(X) \ne \mathcal{A} \ne cc(Y)}{\begin{array}{l}1)\ S;\ \{X \doteq Z(app(X',Y'(t))), Y \doteq Z(app(X'(s),Y'))\} \cup P;\ \Delta \\ 2)\ S;\ \{X \doteq Z(seq(X',Y'(t))), Y \doteq Z(seq(X'(s),Y'))\} \cup P;\ \Delta \\ 3)\ S;\ \{X \doteq Z(let(env^*(\{bind(x,X')\} \cup z), Y'(t))), \\ \qquad Y \doteq Z(let(env^*(\{bind(x,X'(s))\} \cup z), Y'))\} \cup P;\ \Delta \\ 4)\ S;\ \{X \doteq Z(let(env^*(\{bind(x,X'), bind(y,Y'(t))\})\} \cup z), w), \\ \qquad Y \doteq Z(let(env^*(\{bind(x,X'(s)), bind(y,Y')\} \cup z), w))\} \cup P;\ \Delta \\ 5)\ S;\ \{X \doteq Z(case_T(s_1,\ldots,s_n)[X']_i[Y'(t)]_j), \\ \qquad Y \doteq Z(case_T(s_1,\ldots,s_n)[X'(s)]_i, [Y']_j)\} \cup P;\ \Delta \\ \quad so(case_T) = S_1 \times \ldots \times S_n \to S, S_i = S_j = Exp, 2 \le i \le n, 1 \le j \le n, i \ne j \\ 6)\ S;\ \{X \doteq Z(c(s_1,\ldots,s_{ar(c)})[X']_i[Y'(t)]_j), \\ \qquad Y \doteq Z(c(s_1,\ldots,s_{ar(c)})[X'(s)]_i[Y']_j)\} \cup P;\ \Delta \\ \quad i,j \in \{1,\ldots,ar(c)\}, i \ne j\end{array}}$$

where 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 $\min(cc(X), cc(Y))$. The variables w, x, y, z are also fresh and of the appropriate sort.

Fig. 10: Unification rules for context merging

Rules for Multi-Set Equations. The additional (non-deterministic) unification rules in Figure 11 are sufficient to solve nontrivial equations of type *Env*, i.e. proper multi-set-equations, which must be of the form $env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)$, where r_1, r_2 are variables or the constant *emptyEnv*.

Solve-Env $\dfrac{S;\ \{env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \uplus P}{\{r_1 \doteq env^*(M_2 \cup z_3), r_2 \doteq env^*(M_1 \cup z_3)\} \cup S;\ P}$
if r_1, r_2 are variables and z_3 is a fresh variable.

Dec-Env $\dfrac{S;\ \{env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \uplus P \qquad t_1 \in M_1, t_2 \in M_2}{S;\ \{t_1 \doteq t_2, env^*((M_1 \setminus \{t_1\}) \cup r_1) \doteq env^*((M_2 \setminus \{t_2\}) \cup r_2)\} \uplus P}$

Fail-Env $\dfrac{S;\ \{env^*(L \cup r) \doteq emptyEnv\} \uplus P}{Fail}$
if L is nonempty, i.e contains at least one binding or at least one Ch-expression.

Dec-Chain
$\dfrac{\{env^*(M_1 \cup r_1) \doteq env^*(\mathsf{Ch}(x,y,l) \cup M_2 \cup r_2)\} \uplus P \qquad s \in M_1 \text{ is a binding term}}{\begin{array}{l} 1)\ \{l \doteq 1, s \doteq bind(y, A(var(x))), env^*(M_1 \setminus \{s\} \cup r_1) \doteq env^*(M_2 \cup r_2)\} \cup P \\ 2)\ \{l \doteq 1{+}l_1, s \doteq bind(z, A(var(x))), \\ \quad env^*(M_1 \setminus \{s\} \cup r_1) \doteq env^*(\mathsf{Ch}(z,y,l_1) \cup M_2 \cup r_2)\} \cup P \\ 3)\ \{l \doteq l_1{+}1, s \doteq bind(y, A(var(z))), \\ \quad env^*(M_1 \setminus \{s\} \cup r_1) \doteq env^*(\mathsf{Ch}(x,z,l_1) \cup M_2 \cup r_2)\} \cup P; \\ 4)\ \{l \doteq l_1{+}1{+}l_2, s \doteq bind(z_2, A(var(z_1))), \\ \quad env^*(M_1 \setminus \{s\} \cup r_1) \doteq env^*(\mathsf{Ch}(x,z_1,l_1) \cup \mathsf{Ch}(z_2,y,l_2) \cup M_2 \cup r_2)\} \cup P \end{array}$
where z, z_1, z_2 are fresh variables of sort BV and A is either a fresh context variable of class $\mathcal{A}$ if Ch=NCh or $[\cdot]$ if Ch=VCh and l_1, l_2 are fresh $\mathcal{N}$ variables.

U-Chain
$\dfrac{\begin{array}{l} \{env^*(\{bind(x_1,s_1)\} \cup \mathsf{VCh}(x_1,y_1,l_1) \cup M_1 \cup r_1) \doteq \\ \quad env^*(\{bind(x_2,s_2)\} \cup \mathsf{VCh}(x_2,y_2,l_2) \cup M_2 \cup r_2)\} \uplus P; \\ dis(\mathsf{VCh}(x_1,y_1,l_1), \mathsf{VCh}(x_2,y_2,l_2)) \notin \Delta \end{array}}{\begin{array}{l} 1)\ \{l_1 \doteq l_2, bind(x_1,s_1) \doteq bind(x_2,s_2), y_1 \doteq y_2, \\ \quad env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \cup P; \Delta \\ 2)\ \{l_1 \doteq l{+}l_1', l_2 \doteq l{+}l_2', bind(x_1,s_1) \doteq bind(x_2,s_2), \\ \quad env^*(\mathsf{VCh}(z,y_1,l_1') \cup M_1 \cup r_1) \doteq env^*(\mathsf{VCh}(z,y_2,l_2') \cup M_2 \cup r_2)\} \cup P; \\ \quad dis(\mathsf{VCh}(z,y_1,l_1'), \mathsf{VCh}(z,y_2,l_2')) \cup \Delta \end{array}}$
where z is a fresh variable of sort BV, l, l_1', l_2' are fresh $\mathcal{N}$-variables and $\mathsf{VCh}(z,y_1,l_1'), \mathsf{VCh}(z,y_2,l_2')$ are disjunct.

Fig. 11: Unification rules to solve multi-set equations

The rule **Dec-Chain** covers the cases where a non-chain binding s is equated with a chain binding. The possibilities are: 1) The chain consists only of one binding which is equated with s, or 2) the first binding of the chain is equated with binding s, or 3) the last chain binding is equated with s, or 4) a binding from the middle of the chain is equated with s and the original chain is split around this

externalized binding. All of these cases require that some of the internal chain variables (context and BV-sorted) are made explicit. These variables are always chosen as fresh (i.e. not occurring anywhere else in the unification problem).

For the unification rule **U-Chain** we extend our sets of constraints Δ to also contain constraints on chains of the form $dis(\mathsf{VCh}(x_1, y_1, l_1), \mathsf{VCh}(x_2, y_2, l_2))$ denoting that these two chains are disjunct, i.e. if $env^*(\mathsf{VCh}(x_1, y_1, l_1) \cup r_1) =_{LC} env^*(\mathsf{VCh}(x_2, y_2, l_2) \cup r_2)$ then r_1 is of the form $(\mathsf{VCh}(x_2, y_2, l_2) \cup r_3)$ and r_2 is of the form $(\mathsf{VCh}(x_1, y_1, l_1) \cup r_3)$.

In case ii) of rule **U-Chain** the chains are identical. Case i) of the **U-Chain** rule describes that the origins and some initial part of two var-chains are equal, i.e. two chains are equated beginning from their starting point up to some point from where they are disjoint. The possibilities of unifying chains among each other seem rather restricted, after all should it not be possible to equate two arbitrary bindings from two different chains? The rule seems incomplete not to take into account such possibilities. Nevertheless from lemma 3.13, definition 3.21 and lemma 3.23 it follows that the rule covers all possibilities of unifying chains among each other: All other possibilities of unifying bindings of chains lead to solutions representing (LR-syntactically incorrect terms).

5 Termination

On initial LR-forking problems the unification algorithm terminates. This is manly due to the almost linearity of those unification problems and the special restrictions on the occurrence of chains (lemma 3.23).

Theorem 5.1. *For initial LR-forking problems problems, the unification algorithm terminates.*

Proof. Let $\Gamma = (P, \Delta)$ be an initial LR-forking problem and $\mu = (\mu_1, \mu_2, \mu_3)$ be an associated complexity measure where

- μ_1 is the number of occurrences of the function symbol *let* in P.
- μ_2 is the size of P, i.e. $\sum_{(s \doteq t) \in P}(|t| + |s|)$ where $|\cdot|$ is the usual definition of the size of a term, with the modification

$$|env^*(M \cup r)| = 7m + m' + |r| + \sum_{t_i \in M} |t_i|$$

  where m is the number of *bind*-expressions in ML and m' is the number of Ch-expressions in M.
  We set $\mu_2(Fail) = 0$
- μ_3 is the number of context variables in P that are not constraint as not empty ($Var_{CV}(P) \setminus \Delta$)

Each application of an unification rule (except for the **Merge-Fork** rules) decreases μ regarding the lexicographic order.

The multi-equation rules in rule **Dec-Chain** have to be analyzed. The new constructed bind-term has size 5, so the sub-cases 1) – 3) strictly reduce the size. The sub-case 4) adds 6 to the size due to new sub-terms, and removes 7 since t_1 is a non-Ch-expression and removed from the multi-set.

Notice that by lemmas 3.23 and 6.15 the rule **U-Chain** is applicable only once to initial *LR*-forking problems: Because after the application, the value binding origins of the chains are removed from the env^*-terms.

The rules **Merge-Fork-A** and **Merge-Fork-C** both increase μ_2 and μ_3 and they may as well increase μ_1 (depending on the non-deterministic choice). Nevertheless all equations that result from the application of those rules are of the form $X \doteq Z(f(s_1, \ldots, s_n)[X']_i, [Y']_j)$ and on such equations only the rule **Empty-C** can be applied (**Empty-C** at most 3 times) or **Solve** can be applied which moves those equations to the solved part, thereby eventually decreasing μ. □

6 Soundness and Completeness

6.1 Correct Handling of Bound Variables

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

Example 6.1. Unifying (the first-order encodings of) $\lambda x.\lambda y.x$ and $\lambda u.\lambda v.v$ the unification succeeds:

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

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

Example 6.2. Substitutions in $\mathcal{T}_{CH}$ are not capture avoiding. E.g. unifying the encodings of $\lambda x.y$ and $\lambda z.z$ (where we assume $x \neq y$) yields $\{x \doteq z, y \doteq z\}$ as a final problem. The corresponding substitution again equates two terms that are not α-equivalent in the original higher order calculus.

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

With $[\![\cdot]\!]$: Meta-*LR* $\to \mathcal{T}_{CH}$ we denote the translation from Meta-*LR* expressions into their encoding; $[\![\cdot]\!]^-$: $\mathcal{T}_{CH} \to$ Meta-*LR* is the inverse translation.

Meta-variables in expression are translated into variables of the appropriate sort. The context classes are chosen according to their intention in the rules. A definition of $[\![\cdot]\!]$ can be seen in Fig. 12.

$$
\begin{aligned}
[\![x]\!] &= \mathit{var}(x)\\
[\![(\lambda x.s)]\!] &= \mathit{lam}(x, [\![s]\!])\\
[\![(s_1\ s_2)]\!] &= \mathit{app}([\![s_1]\!], [\![s_2]\!])\\
[\![\mathtt{seq}\ s_1\ s_2]\!] &= \mathit{seq}([\![s_1]\!], [\![s_2]\!])\\
[\![(c\ s_1 \ldots s_n)]\!] &= c([\![s_1]\!], \ldots, [\![s_n]\!])\\
[\![(\mathtt{case}_T\ s\ At_1 \ldots At_{|T|})]\!] &= \mathit{case}_T([\![s]\!], [\![At_1]\!], \ldots, [\![At_{|T|}]\!])\\
[\![(\mathtt{letrec}\ \mathit{Env}\ \mathtt{in}\ s)]\!] &= \mathit{let}([\![\mathit{Env}]\!], [\![s]\!])\\
[\![x = s]\!] &= \mathit{bind}(x, [\![s]\!])\\
[\![\{\ \}]\!] &= \mathit{emptyEnv}\\
[\![\{x_1 = s_1, \ldots, x_n = s_n\}]\!] &= \mathit{env}([\![x_1 = s_1]\!], [\![\{\ldots, x_n = s_n\}]\!])
\end{aligned}
$$

(a) Translation of *LR*-expressions into many sorted terms (*FO-LR*)

$$
\begin{aligned}
[\![s]\!] &= s :: S\\
&\text{translate meta variable } s \text{ to term variable of an appropriate sort } S\\
[\![A[s]]\!] &= X([\![s]\!]) \quad X \text{ is a context variables of class } \mathcal{A}\\
[\![S[s]]\!] &= X([\![s]\!]) \quad X \text{ is a context variables of class } \mathcal{S}\\
[\![C[s]]\!] &= X([\![s]\!]) \quad X \text{ is a context variables of class } \mathcal{C}\\
[\![\{x_{i+1} = x_i\}_{i=k}^{m}]\!] &= \mathsf{VCh}(x_k, x_m, l)\\
[\![\{y_{i+1} = A_{i+1}[y_i]\}_{i=k}^{m}]\!] &= \mathsf{NCh}(y_k, y_m, l)\\
&\text{the } x\text{'s and } y\text{'s are } BV \text{ sorted and } l \text{ is an integer variable}
\end{aligned}
$$

(b) Translation of additional syntactic constructs (into *CH-LR*)

Fig. 12: Encoding of *LR*-reductions

Lemma 6.3. *If $s \in \mathcal{T}_{CH}$ is almost ground and no* Ch*-constructs occur in s then $[\![s]\!]^- \in LR$.*

Definition 6.4. *An LR expression s* satisfies the DVC *iff all free variables in s are distinct from bound variables and all bound variables in s are distinct.*

For a term $s \in \mathcal{T}_{CH}$ let τ be a substitution such that

$$
\begin{aligned}
\tau(t) &= a && \text{for all variables } t \in \mathit{Var}_{\mathit{Exp}}(s) \text{ and } a \text{ is a constant}\\
\tau(\mathit{Env}) &= \mathit{emptyEnv} && \text{for all environment variables } \mathit{Env} \in \mathit{Var}_{\mathit{Env}}(s)\\
\tau(X) &= [\cdot] && \text{for all context variables } X \text{ in } s\\
\tau(\mathsf{Ch}(N, M)) &= y_m = y_n && \text{for all } \mathsf{Ch}\text{-constructs in } s
\end{aligned}
$$

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

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

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

Definition 6.5 (DVC-check). *Assume $s,t \in \mathcal{T}_{CH}$ and $\Gamma = \{s \doteq t\}$ is an initial unification problem and $S \neq Fail$ is a final system derived from Γ.*

Then the following two rules check if the substitution σ_S derived from S satisfies the DVC w.r.t. the initial problem $\Gamma = \{s \doteq t\}$.

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

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

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

The DVC-check is decidable: If $s \doteq t$ is the initial problem, then apply the substitution σ_S derived from a final problem S to s. Then check if $\tau\sigma_S(s)$ satisfies the DVC where τ is the substitution from definition 6.4 that ensures that the resulting term is ground. (The DVC-check can be done on the representation of the solutions, not all ground instances have to be checked).

The DVC-check can not detect the capture of free variables as in example 6.2. **Convention:** To avoid capture of free variables by a substitution, terms in an initial unification problem must adhere to the closedness condition (Definition 6.4).

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

Definition 6.6 (DVC-solution). *Let $\Gamma = \{s_1 \doteq t_1, \dots s_n \doteq t_n\}$ and σ be a solution of Γ. Then σ is a* DVC-solution *iff $\sigma(s_i)$ satisfies the DVC for all i.*

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

6.2 Soundness and Completeness

Definition 6.7. *Let $\Gamma = (P, \Delta)$, $P = \{s_1 \doteq t_1, \dots s_n \doteq t_n\}$ be a unification problem. Then we define the set of solutions modulo* LC *of Γ as*

$$U_{LC}(\Gamma) := \{\sigma \mid \sigma \textit{ is a solution of } s_i \doteq t_i \textit{ for } i = 1, \dots, n\}.$$

The set of DVC-solutions *is defined by*

$$U_{LC}^{DVC}(\Gamma) := \{\sigma \mid \sigma \in U_{LC}(\Gamma) \textit{ and } \sigma(s_i) \textit{ satisfies the DVC for } i = 1, \dots, n\}.$$

We set $U_{LC}^{DVC}(\mathit{Fail}) = \emptyset$.

Remark 6.8. In the following proofs we can safely ignore the sets S, P of an unification system because equations in these two sets, that are not explicitly mentioned, are not changed by unification rules.

Lemma 6.9. *The standard unification rules* ***Solve***, ***Dec***, ***Trivial*** *and the failure rules* ***Fail***, ***Fail-C-Lam***, ***Fail-C-Var***, ***Fail-C-Cons*** *and* ***Fail-Env****, preserve the set of DVC-solutions. I.e. If Γ' is derived from Γ with one of the above rules, then $U_{LC}^{DVC}(\Gamma) = U_{LC}^{DVC}(\Gamma')$.*

Proof. For the standard rules this is straightforward.

(**Fail-C-Lam**) If **Fail-C-Lam** is applicable to Γ then there is an equation $X(s) \doteq \mathit{lam}(t_1, t_2)$ in Γ where X belongs to the context class $\mathcal{A}$ or $\mathcal{S}$. Such an equation has no solution, because by Definition 2.1 neither of the context classes $\mathcal{A}, \mathcal{S}$ permit the occurrence of the hole in the body of an abstraction.

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

Lemma 6.10. *The rules of the unification algorithm are correct, i.e. if Γ' is derived from Γ using one of the rules of the unification algorithm, then $U_{LC}^{DVC}(\Gamma) \supseteq U_{LC}^{DVC}(\Gamma')$.*

Proof. For the rules mentioned in lemma 6.9 this is obvious.

If the set of DVC-solutions of Γ' is empty (e.g. for $\Gamma' = \mathit{Fail}$) then $U_{LC}^{DVC}(\Gamma) \supseteq \emptyset = U_{LC}^{DVC}(\Gamma')$ holds. Therefore the **Fail**-rules are correct and we assume $U_{LC}^{DVC}(\Gamma') \neq \emptyset$.

If a rule introduces variables of sort BV (as is the case for **Dec-C-Let**, **Merge-Fork-A**, **Merge-Fork-C**, **Dec-Chain**, **U-Chain**) then the variables introduced are chosen in such a way, that the DVC is satisfied by the resulting terms (i.e. a unification step never introduces an DVC-error into a problem).

We show some unification rules as correct. The other rules can be proved correct similarly straightforward. If a rule involves non-deterministic choice, then for each derivable system correctness has to be shown.

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

Correctness of (**Dec-Chain**): We look at case 2) of **Dec-Chain** and we assume that the chain construct is NCh, i.e. for a solution σ we have:

$$\begin{aligned} \sigma(l) &= \sigma(1{+}l_1) \\ \sigma(s) &=_{LC} \sigma(\mathit{bind}(z, A(\mathit{var}(x)))) \\ \sigma(\mathit{env}^*(M_1 \setminus \{s\} \cup r_1)) &=_{LC} \sigma(\mathit{env}^*(\mathsf{NCh}(z, y, l_1) \cup M_2 \cup r_2)) \end{aligned}$$

The equation $\sigma(l) = \sigma(1{+}l_1)$ indicates, that the chain $\mathsf{NCh}(x, y, l)$ (from the premise of the rule) was split into a leading binding and a remaining chain of

length l_1. If we explicitly store the chain-split in the equation system of the conclusion, like this:

$$\{\mathsf{NCh}(x,y,l) \doteq \{bind(z,A(var(x)))\} \cup \mathsf{NCh}(z,y,l_1)\} \cup \\ \{\mathsf{NCh}(x,y,l) \mapsto \{bind(z,A(var(x)))\} \cup \mathsf{NCh}(z,y,l_1)\}(S)$$

then by assumption we know that σ is a solution of this equation, i.e.

$$\sigma(\mathsf{NCh}(x,y,l)) =_{LC} \sigma(\{bind(z,A(var(x)))\} \cup \mathsf{NCh}(z,y,l_1))$$

holds and this σ also solves the equation in the premise of **Dec-Chain**, i.e.:

$$\begin{aligned} & \sigma(env^*(M_1 \cup r_1)) \\ = \; & env^*(\{\sigma(s)\} \cup \sigma(M_1 \setminus \{s\}) \cup \sigma(r_1)) \\ =_{LC} \; & env^*(\{\sigma(bind(z,A(var(x))))\} \cup \sigma(\mathsf{NCh}(z,y,l_1)) \cup \sigma(M_2) \cup \sigma(r_2)) \\ = \; & env^*(\sigma(\mathsf{NCh}(x,y,l)) \cup \sigma(M_2) \cup \sigma(r_2)) \\ = \; & \sigma(env^*(\mathsf{NCh}(x,y,l) \cup M_2 \cup r_2)) \end{aligned}$$

For the other three cases of the **Dec-Chain** rule (and if the chain is VCh) the proof is analogous.

Correctness of **U-Chain**: Case 1) of **U-Chain** states that the two chains are equal. The correctness is obvious.

A solution σ of case 2) satisfies:

$$\begin{aligned} \sigma(l_1) &= \sigma(l{+}l_1') \\ \sigma(l_2) &= \sigma(l{+}l_2') \\ \sigma(bind(x_1,s_1)) &=_{LC} \sigma(bind(x_2,s_2)) \Rightarrow \sigma(x_1) = \sigma(x_2) \\ \sigma(env^*(\mathsf{VCh}(z,y_1,l_1') \cup M_1 \cup r_1)) &=_{LC} \sigma(env^*(\mathsf{VCh}(z,y_2,l_2') \cup M_2 \cup r_2)) \end{aligned}$$

which means, that the two chains in the premise of the rule are cut in half and the initial parts are equated (starting from their origins), and the two tails are disjunct. If we again explicitly store the split of the chains in the solved part of the conclusion:

$$\mathsf{VCh}(x_1,y_1,l_1) \doteq \mathsf{VCh}(x_1,z,l) \cup \mathsf{VCh}(z,y_1,l_1') \\ \mathsf{VCh}(x_2,y_2,l_2) \doteq \mathsf{VCh}(x_2,z,l) \cup \mathsf{VCh}(z,y_2,l_2')$$

then it is easy to see that σ is also a solution of the equations in the premise of the rule **U-Chain**. □

We use the notation $\Gamma \Longrightarrow \Gamma'$ to denote that the unification problem Γ is transformed into the problem Γ' by means of a unification rule. $\Gamma \Longrightarrow^* \Gamma'$ denotes a finite sequence of transformations and $\Gamma \Longrightarrow^T \Gamma'$ to denote a transformation with rule T.

Theorem 6.11 (Soundness). *If $\Gamma \Longrightarrow^* \Gamma'$ and Γ' is a final unification problem then $U_{LC}^{DVC}(\Gamma) \supseteq U_{LC}^{DVC}(\Gamma')$.*

Proof. Either Γ' is *Fail*, in which case the claim follows directly, or $\Gamma' \neq$ *Fail* in which case we perform induction on the length of the transformation to solved form using lemma 6.10.

Now that we established soundness of the unification algorithm, we show its completeness.

Lemma 6.12. *Let* $\Gamma = (\{X(s) \doteq f(t_1, \ldots, t_n)\}, \Delta = \emptyset)$ *be an almost linear unification problem with* $X(s), f(t_1, \ldots, t_n) \in \mathcal{T}_{CH}$ *and let* σ *be a solution of* Γ *(i.e.* $\sigma \in U^{DVC}_{LC}(\Gamma)$*).*

Then there exists a unification rule (or a sequence of rules) such that $\Gamma \Longrightarrow \Gamma'$ *(or* $\Gamma \Longrightarrow^* \Gamma'$*) and there exists a substitution* τ *with* $dom(\tau) = Var(\Gamma) \setminus Var(\Gamma')$ *such that and* $\tau\sigma \in U^{DVC}_{LC}(\Gamma')$*.*

Proof. The context variable X can be the empty context, in this case $\sigma(s) =_{LC} f(\sigma(t_1), \sigma(t_2))$ holds and σ is also a solution of the unification problem $\{X \doteq [\cdot]\} \cup \{X \mapsto [\cdot]\}(\{X(s) \doteq f(t_1, t_2)\})$, which results from $X(s) \doteq f(t_1, t_2)$ by application of the case 2) of the rule **Empty-C**. If X is not the empty context we can first transform Γ with case 1) of rule **Empty-C** and insert X into the constraint set Δ thereby constraining the context variable as not empty. Now we are in the case, that there is a sequence of transformations to Γ' and we go trough the cases for the function symbol f and the the context class of X to show that for each possible solution σ of $\{X(s) \doteq f(t_1, \ldots, t_n)\}, \{X\} \uplus \Delta$ there exists a transformation that keeps the solution.

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

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

Case $f = let$ then the class of X can be either a $\mathcal{S}$ or a $\mathcal{C}$ (if X is of class $\mathcal{A}$, then $X(s) \doteq let(t_1, t_2)$ has no solution). We have two cases: 1. The hole may occur in the second argument of *let*. In this case a solution is

also a solution of the equation transformed by the rule **Dec-C**. 2. Additionally to the above case the hole can appear in the first argument of *let* i.e. $\sigma(X) =_{LC} let(\sigma(t_1[\cdot]_p), \sigma(t_2))$ and $\sigma(s) =_{LC} \sigma(t_1|_p)$ for some position $p \in Pos(t_1)$ such that $t_1[\cdot]_p$ is an admissible $\mathcal{S}$ contexts ($\mathcal{C}$ context respectively). Since *let* is of sort $Env \to Exp \to Env$ the head symbol of the context $t_1[\cdot]_p$ must be of sort *Env* and in t_1 the hole can occur only on the right hand side of variable-expression binding, hence we can conclude $\sigma(t_1[\cdot]_p) =_{LC} \sigma(env^*(\{bind(x, (t_1|_q)[\cdot]_r)\} \cup z))$ for some fresh variables x, z of appropriate sort and and some positions q, r such that $p = qr$ (i.e. $q = 1.2$). When we set $\tau = \{X' \mapsto (t_1|_q)[\cdot]_r\}$ then $\sigma\tau$ is a solution for the conclusion $\{X \doteq let(env^*(\{bind(x, X')\} \cup z), t_2), env^*(\{bind(x, X'(s))\} \cup z) \doteq t_1\}$ of the **Dec-C-Let** rule.

Case $f = c(s_1, \ldots, s_n)$. If the context class of X is $\mathcal{A}$ then this equation has no solution, in which case the rule **Fail-C-Cons** applies. Else the hole of the solution of X may appear under any argument position of the constructor c, which is handled by the rule **Dec-C-Cons**.

Case $f = case_T$ here **Dec-C-Case-1** covers the possible solutions when X is of class $\mathcal{A}$, then the hole of $\sigma(X)$ can appear only under the first argument of the $case_T$ term. **Dec-C-Case-2** covers the possible solutions if X is of class $\mathcal{S}$ or $\mathcal{C}$, then the hole of $\sigma(X)$ can appear under any argument position of the $case_T$ term where the sort is not BV.

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

Lemma 6.13. *Let $\Gamma = (\{X(s) \doteq Y(t)\}, \Delta = \emptyset)$ be an almost linear unification problem with $X(s), Y(t) \in \mathcal{T}_{CH}$ and let σ be a solution of Γ (i.e $\sigma \in U_{LC}^{DVC}(\Gamma)$).*

Then there exists a unification rule (or a sequence of rules) such that $\Gamma \Longrightarrow \Gamma'$ (or $\Gamma \Longrightarrow^* \Gamma'$) and there exists a substitution τ with $dom(\tau) = Var(\Gamma') \setminus Var(\Gamma)$ such that and $\tau\sigma \in U_{LC}^{DVC}(\Gamma')$.

Proof. Assume $\sigma \in U_{LC}^{DVC}(\{X(s) \doteq Y(t)\})$.

Either one (or both) of the context variables may be empty the empty context, in which case we use **Empty-C** to guess the respective context variable (or both) as empty. (E.g. one case is $X = [\cdot]$, i.e. $\sigma(s) =_{LC} \sigma(Y(t))$ and σ is also a solution if we transform Γ by **Empty-C** 2) where we choose X as empty.)

If both context variables are not the empty context we know, that $\sigma(X) =_{LC} s_p[\cdot]_p, \sigma(Y) =_{LC} t_q[\cdot]_q$ and $s_p[\sigma(s)]_p =_{LC} t_q[\sigma(t)]_q$ holds. Let p_0 be the greatest common prefix of p, q. We have to distinguish two cases:

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

Let $\tau = \{Y' \mapsto u[\cdot]_{p'}, X' \mapsto s_p[\cdot]_p\}$. Then $\tau\sigma$ is a solution for the conclusion of **Merge-Prefix** (where we first applied **Empty-C** twice to constrain X and Y as non empty).

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

To show completeness of the unification rules that handle environment terms with chains, it is helpful to reformulate the properties from lemma 3.23: that the occurrence of chains in initial *LR*-forking problems are restricted.

Definition 6.14. *An equation $s \doteq t$ between terms $s, t \in \mathcal{T}_{CH}$ satisfies the IP-chain-restictions if*

- *in each terms s and t occurs at most one* VCh*-construct that has an origin that is a value binding and*
- *in only one of the terms s or t there occurs an* NCh*-construct that has an origin that is an value binding. In this case the term is of the form as in definition 3.22 (item 4).*

The next lemma shows, that the number of chains in an initial *LR*-forking-problem can only increase during unification by splitting chains (via **Dec-Chain** and **U-Chain**).Therefore for *LR*-forking problems the number of chains that have an value binding as origin can only decrease (from one to zero).

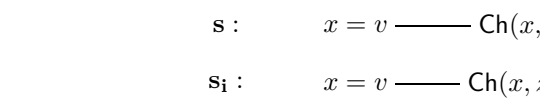
Lemma 6.15. *Let $s, t \in \mathcal{T}_{CH}$ be terms such that one (say s) satisfies the T-chain-restrictions and the other (say t) satisfies the N-chain-restrictions (from definition 3.22) and let $\{s \doteq t\}$ be an almost linear unification problem.*
Then for all possible sequences of unification transformations

$$\{s \doteq t\} \Longrightarrow \Gamma_1 \Longrightarrow \ldots \Longrightarrow \Gamma_n$$

to a final system Γ_n all equations $s_i \doteq t_i$ in Γ_j (for $j = 1, \ldots, n$) satisfy the IP-chain-restictions and each VCh*-construct in s_i, t_i that does not have an value binding as an origin is originated through the split of some initial chains in s and t.*

Proof. Through induction over the length of the transformation to a final system, using the almost linearity (i.e. there is no copying of chains), T-chain-restrictions and N-chain-restrictions and the form of the unification rules, where the critical rules are those that modify (i.e. split) chains: **Dec-Chain** and **U-Chain**. □

The following diagram shows an example, where in a term s an initial chain $\mathsf{Ch}(x, y, l)$ with a value binding origin is cut during unification (e.g. by **Dec-Chain** 4) into a leading chain $\mathsf{Ch}(x, z_1, l_1)$ and a trailing chain $\mathsf{Ch}(z_2, y, l_2)$ (which has no value binding origin in s_i).



Lemma 6.16. *Let $\Gamma = (P = \{env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)\} \uplus P', \Delta)$ be an almost linear unification problem where r_1, r_2 are variables of sort Env and M_1, M_2 are sets of bindings and chain constructs, such that Γ either satisfies the T-chain-restrictions and the N-chain-restrictionsor it was derived form a unification problem that satisfied both.*

Let σ be a solution of Γ ($\sigma \in U^{DVC}_{LC}(\Gamma)$). Then either

1. *Γ is already in solved form or*
2. *there exists a unification rule (or a sequence of rules) such that $\Gamma \Longrightarrow \Gamma'$ (or $\Gamma \Longrightarrow^* \Gamma'$) and there exists a substitution τ with $dom(\tau) = Var(\Gamma') \setminus Var(\Gamma)$ such that $\tau\sigma \in U^{DVC}_{LC}(\Gamma')$.*

Proof. By structural induction on M_1 and M_2.
Case $M_1 = \emptyset$ and $M_2 = \emptyset$, i.e. both sequences are empty, then by definition of the env^* function symbol $env^*(r_1) = r_1$ and $env^*(r_2) = r_2$ and the equation $r_1 \doteq r_2$ is in solved form.
Case $M_1 = \emptyset$ and $M_2 = \{t\} \cup M_2'$: The equation $env^*(r_1) = r_1 \doteq env^*(\{t\} \cup M_2' \cup r_2)$ is in solved form.
Case $M_1 = \{x\} \cup M_1'$ and $M_2 = \emptyset$: The equation $env^*(r_1) = r_1 \doteq env^*(\{x\} \cup M_2' \cup r_2)$ is in solved form.
Case M_1 and M_2 are both not the empty, i.e. there are x, y such that $x \in M_1$ and $y \in M_2$. We analyze the cases for x and y (where we use $M_1' = M_1 \setminus \{x\}$ and $M_2' = M_2 \setminus \{y\}$):

1. x is a binding term, i.e. $x = bind(z, s)$ and
   y **is a binding term:** $y = bind(z', t)$, then $\sigma(env^*(\{bind(z, s)\} \cup M_1' \cup r_1)) =_{LC} \sigma(env^*(\{bind(z', t)\} \cup M_2' \cup r_2))$ holds and by lemma 3.5 we can conclude that either
   (a) $\sigma(bind(z, s)) =_{LC} \sigma(bind(z', t))$ and $\sigma(env^*(M_1' \cup r_1)) =_{LC} \sigma(env^*(M_2' \cup r_2))$. We chose **Dec-Env** as the unification rule and to transform Γ into $\Gamma' = \{bind(z, s) \doteq bind(z', t), env^*(M_1' \cup r_1) \doteq env^*(M_2' \cup r_2)\}$ of which σ is also a solution.
   (b) Or there exist a z such that $\sigma(r_2) =_{LC} \sigma(env^*(M_1 \cup z))$ and $\sigma(r_1) =_{LC} \sigma(env^*(M_2 \cup z))$. If we apply **Solve-Env** to Γ then σ solves the derived Γ'.
   y **is a chain construct** i.e. $y = \mathsf{Ch}(x_1, y_1, l_1)$, then $\sigma(env^*(\{bind(z, s)\} \cup M_1' \cup r_1)) =_{LC} \sigma(env^*(\mathsf{Ch}(x_1, y_1, l_1) \cup M_2' \cup r_2))$ holds where the chain of

length l_1 is unfolded to all ground instances of binding chains (before application of σ):

$$\begin{array}{ll}\mathsf{NCh}(x_1, y_1, \overline{l_1}) = & \\ bind(y_1, A(var(x_1)))) & \text{if } l_1 = 1 \text{ or} \\ bind(z_1, A_1(var(x_1))), s_1, \ldots, s_{l_1-2}, bind(y_1, A_{l_1}(var(z_{l_1-1}))) & \text{if } l_1 \geq 2\end{array}$$

where the intermediate bindings introduced through the unfolding are connected via the predecessor relation $\prec$, i.e.

$$\mathsf{uch} := bind(z_1, A_1(var(x_1))) \prec s_1 \prec \ldots \prec s_{l_1-2} \prec bind(y_1, A_{l_1}(var(z_{l_1-1}))).$$

We call this first order binding chain uch (unfold chain).
By lemma 3.5 we conclude now that either there exists a z such that $\sigma(r_1) =_{LC} \sigma(env^*(M_2 \cup z))$ and $\sigma(r_2) =_{LC} \sigma(env^*(M_1 \cup z))$ in which case we can apply **Solve-Env** to derive Γ' for that σ is a solution.
Or the binding term $bind(z, s)$ is lc-congruent (under σ) to some binding from the chain $\mathsf{Ch}(x_1, y_1, l_1)$, i.e.

$$\begin{array}{r}\sigma(bind(z, s)) =_{LC} \sigma(u) \quad \text{for } u \in \mathsf{uch} \text{ and} \\ \sigma(env^*(M_1' \cup r_1)) =_{LC} \sigma(env^*(\mathsf{uch} \setminus \{u\} \cup M_2' \cup r_2))\end{array}$$

There are now four different cases for u:

1) If $l_1 = 1$ (i.e. the chain is of length one) then $\mathsf{uch} = u = bind(y_1, A(var(x_1)))$ and if we apply case 1) of **Dec-Chain** to Γ then σ is a solution for the resulting unification problem Γ'.
- If $l_1 \geq 2$ then u may either be 2) the start-binding $bind(z_1, A_1(var(x_1)))$ of uch or 3) some intermediate binding s_i for $i = 1, \ldots, l-2$ or 4) the end-binding $bind(y_1, A_{l_1}(var(z_{l_1-1})))$ of uch. For each of the cases there is non-deterministic choice in the **Dec-Chain** rule, such that the solution is preserved. The position where u is taken from the (unfold) chain of bindings uch is represented by equations over the length l_1 of the (not unfold) chain $\mathsf{Ch}(x_1, y_1, l_1)$ in the conclusions of the **Dec-Chain** rule.

2. x is a chain construct $x = \mathsf{Ch}(x_1, y_1, l_1)$.
If y is a binding term then this case is symmetrical to the one above.
y is a chain construct: $y = \mathsf{Ch}(x_2, y_2, l_2)$. By lemma 6.15 and the assumptions only one of the chains can be a NCh, w.l.o.g. we assume $x = \mathsf{NCh}$, then y is a VCh and

$$\sigma(env^*(\mathsf{NCh}(x_1, y_1, l_1) \cup M_1' \cup r_1)) =_{LC} \sigma(env^*(\mathsf{VCh}(x_2, y_2, l_2) \cup M_2' \cup r_2))$$

holds. Then the chains are disjunct in the solution, i.e.

$$\begin{array}{r}\sigma(r_1) =_{LC} \sigma(env^*(\mathsf{VCh}(x_2, y_2, l_2) \cup z)) \\ \sigma(r_2) =_{LC} \sigma(env^*(\mathsf{NCh}(x_1, y_1, l_1) \cup z)) \\ \sigma(env^*(M_1' \cup z)) =_{LC} \sigma(env^*(M_2' \cup z)).\end{array}$$

Which means, that no chain-bindings can be equated between a NCh and a VCh. We show this by contradiction using lemma 3.13, which states, that if any two bindings from the (unfolded) chains are equal, then all their predecessors are equal until the origin of one chain is reached[1]. We distinguish the cases for the origin:

(a) The origin is a value binding, then by lemma 3.13 the origins of both chains are equal (modulo lc under σ) and both origins have to be value bindings. Now we conclude by lemma 6.15 that there is a binding $b := bind(z, A(var(x_1))) \in M_1'$ (where A must not be the empty context) that is a predecessor of the leading binding of the unfold chain $\mathsf{NCh}(x_1, y_1, l_1)$ which is equal under σ to some binding b' which is unfold from $\mathsf{VCh}(x_2, y_2, l_2)$. But for any binding $b' := bind(z', (var(z'')))$ unfold form the chain $\mathsf{VCh}(x_2, y_2, l_2)$ the equation $b =_{LC} b'$ can never hold.

(b) The origin is a non-value binding. By lemma 6.15 we can conclude that this chain $\mathsf{Ch}(z, \ldots)$ results from the split of some initial chain. Hence the leading binding of $\mathsf{Ch}(z, \ldots)$ has a predecessor $bind(z, t)$ in the original problem (but not in the term (local) $env^*(\mathsf{VCh}(x_2, y_2, l_2) \cup M_2' \cup r_2)$). Now lemma 3.13 provides a contradiction for the original problem (i.e. the original problem now contains LR-syntactically incorrect terms, thereby raising a DVC-failure).

As both chains must be disjunct under σ we can apply **Solve-Env** to derive a unification problem that preserves the σ.

Now we assume, that x and y **are both** VCh**-constructs** and

$$\sigma(env^*(\mathsf{VCh}(x_1, y_1, l_1) \cup M_1' \cup r_1)) =_{LC} \sigma(env^*(\mathsf{VCh}(x_2, y_2, l_2) \cup M_2' \cup r_2))$$

holds. By assumptions we know, that either

(a) M_1' contains the origin of the chain $\mathsf{VCh}(x_1, y_1, l_1)$ and M_2' contains the origin of $\mathsf{VCh}(x_2, y_2, l_2)$ respectively (which are both value bindings). In this case we conclude by lemma 3.13 that starting form their origins some initial parts of both chains are equal until some point, from which they are disjunct This case is covered by the case 2. of rule **U-Chain**. Or both chains are completely identical, which is covered by case 1. of **U-Chain**.

(b) Or just one or neither of the chains has an origin that is an value binding in M_1' (M_2' respectively), but then by lemma 6.15 those chains are originated by splits, i.e. in the initial problem they have predecessors, which are not equated with the respective predecessors from the other chain. By lemma 3.13 such an solution contradicts the assumption that the solution of the initial problem is LR-syntactically correct. I.e. the chains originated by splits can only by disjunct from each other (and from initial parts of non split chains) in the solution. This case is again covered by **Solve-Env**. □

[1] We can apply lemma 3.13 here because the chain constructs are unfolded to first-order binding chains.

To an initial problem the rule **U-Chain** can be applied only once, because there are at most two chains that have value bindings as origins in an initial problem and after an application of **U-Chain** there are no more such chains with value origins in the problem.

Theorem 6.17 (Completeness). *Let $\Gamma = (P, \Delta)$ be an LR-initial-forking problem. For each $\theta \in U_{LC}^{DVC}(\Gamma)$ there exists a (finite) sequence of transformations $\Gamma \Longrightarrow \Gamma_1 \Longrightarrow \ldots \Longrightarrow \Gamma_n$ and a substitution τ with $dom(\tau) = Var(\Gamma_n) \setminus Var(\Gamma)$ such that $\tau(\Gamma_n)$ is a final system that represents θ.*

Proof. By structural induction on P. For almost linear problems the unification algorithm terminates (by theorem 5.1) with a unification problem that is either final or *Fail*.

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

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

If Γ_i is not a final problem, then it contains some equations that are not solved and can still be transformed by unification rules. Or the problem is stuck in which case the **Stuck-Fail** rule applies and detects this case. We go trough the cases for these equations.

Case $f(s_1, \ldots s_m) \doteq g(t_1, \ldots, t_n)$ where f, g are free function symbols. Either $f = g$ and $m = n$, and then the rule **Dec** can be applied, which by Lemma 6.9 does not modify the set of solutions. Or $f \neq g$ then the **Fail** rule applies which also does not change the set of solutions.

This holds for all unsolved equations to which unification rules can be applied that do not modify the set of solutions (i.e. the rules covered in Lemma 6.9).

Case $X(s) \doteq f(t_1, \ldots, t_n)$. Then by lemma 6.12 we know there exist a unification rule that transforms the equation (-set) into another set, while keeping the solution.

Case $X(s) \doteq Y(t)$ is treated in lemma 6.13.

Case $env^*(M_1 \cup r_1) \doteq env^*(M_2 \cup r_2)$ where M_1, M_2 may contain binding terms and chain constructs and the equation satisfies the IP-chain-restictions (by assumption that we sarted with an initial *LR*-forking problem and by lemma 6.15). Then the rules **Solve-Env**, **Dec-Env** **Dec-Chain** and **U-Chain** can be applied wich by lemma 6.16 ensure completeness of the algorithm. □

Theorem 6.18. *The rule-based algorithm terminates if applied to initial LR-forking-problems. Thus it decides unifiability of these sets of equations. Since it is sound and complete, and the non-deterministic forking possibilities of the algorithm are finite, the algorithm also computes a finite and complete set of final unification problems by gathering all possible results.*

Theorem 6.19. *The computation of all overlaps between the rules in Figures 6 and 2 and left hand sides of normal order reductions in Figures 3 and 4 explained in definition 3.21 can be done using the unification algorithm.*

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.

7 Further Transformation Rules

The method of overlap computation we employed here, is in general also applicable to other transformations, if they satisfie the following conditions: Their left-hand-sides can be represented by the unification term language together with possible side conditions (i.e. $[\![lhs]\!]$ exists and the side conditions must also be encode-able in unification problems, like A is not empty), the translated lhs of the transformations have to be *almost linear* (see section 4.1) and they further have to satisfy the T-chain-restrictions (from definition 3.22).

Figure 13 shows some additional transformations which are considered in [SSSS08]. Note that these are all proved as correct in [SSSS08], but nevertheless are a challenging testbed for the unification algorithm and the subsequent automatic complete induction prover for diagrams.

We comment on the applicability of the overlap computation to the extra transformations:

gc1,gc2 These rules require an extra condition on the occurrences of variables, which currently can not be treated in the unification algorithm.

ucp1,ucp2,ucp3 Also, these rules have restrictions on the number of occurrences of the replaced variable x, which currently cannot be treated in the unification algorithm.

Other rules in Fig. 13 can all be processed by the unification algorithm, since all the conditions are satisfied.

The addition of these restrictions to the algorithm would also make changes necessary to the unification rules and consequently also to completeness and termination proof. We leave this as further work.

(gc1) $(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}, Env\ \texttt{in}\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ t)$
if for all $i : x_i$ does not occur in Env nor in t

(gc2) $(\texttt{letrec}\ \{x_i = s_i\}_{i=1}^{n}\ \texttt{in}\ t) \to t$
if for all $i : x_i$ does not occur in t

(cpx-in) $(\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x])$
$\to (\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y])$ where y is a variable and $x \neq y$

(cpx-e) $(\texttt{letrec}\ x = y, z = C[x], Env\ \texttt{in}\ t)$
$\to (\texttt{letrec}\ x = y, z = C[y], Env\ \texttt{in}\ t)$ where y is a variable and $x \neq y$

(cpax) $(\texttt{letrec}\ x = y, Env\ \texttt{in}\ s)$
$\to (\texttt{letrec}\ x = y, Env[y/x]\ \texttt{in}\ s[y/x])$
where y is a variable, $x \neq y$ and $y \in FV(s, Env)$

(cpcx-in) $(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \texttt{in}\ C[x])$
$\to (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, Env\ \texttt{in}\ C[c\ \overrightarrow{y}])$

(cpcx-e) $(\texttt{letrec}\ x = c\ \overrightarrow{t}, z = C[x], Env\ \texttt{in}\ t)$
$\to (\texttt{letrec}\ x = c\ \overrightarrow{y}, \{y_i = t_i\}_{i=1}^{ar(c)}, z = C[c\ \overrightarrow{y}], Env\ \texttt{in}\ t)$

(abs) $(\texttt{letrec}\ x = c\ \overrightarrow{t}, Env\ \ \texttt{in}\ s) \to (\texttt{letrec}\ x = c\ \overrightarrow{x}, \{x_i = t_i\}_{i=1}^{ar(c)}, Env\ \ \texttt{in}\ s)$
where $ar(c) \geq 1$

(abse) $(c\ \overrightarrow{t}) \to (\texttt{letrec}\ \{x_i = t_i\}_{i=1}^{ar(c)}\ \texttt{in}\ c\ \overrightarrow{x})$ where $ar(c) \geq 1$

(xch) $(\texttt{letrec}\ x = t, y = x, Env\ \texttt{in}\ r) \ \to\ (\texttt{letrec}\ y = t, x = y, Env\ \texttt{in}\ r)$

(ucp1) $(\texttt{letrec}\ Env, x = t\ \texttt{in}\ S[x]) \to (\texttt{letrec}\ Env\ \texttt{in}\ S[t])$

(ucp2) $(\texttt{letrec}\ Env, x = t, y = S[x]\ \texttt{in}\ r) \to (\texttt{letrec}\ Env, y = S[t]\ \texttt{in}\ r)$

(ucp3) $(\texttt{letrec}\ x = t\ \texttt{in}\ S[x]) \to S[t]$
where in the (ucp)-rules, x has at most one occurrence in $S[x]$ and no occurrence in Env, t, r; and S is a surface context

(lwas) $W^{-}_{(1)}[(\texttt{letrec}\ Env\ \texttt{in}\ s)] \to (\texttt{letrec}\ Env\ \texttt{in}\ W^{-}_{(1)}[s])$
where $W^{-}_{(1)}$ is of main depth 1 and the hole is not contained in an abstraction nor in a `letrec`-expression

(cpcxnoa) $(\texttt{letrec}\ x = c\ x_1 \ldots x_m, Env\ \texttt{in}\ C[x])$
$\to (\texttt{letrec}\ x = c\ x_1 \ldots x_m, Env\ \texttt{in}\ C[c\ x_1 \ldots x_m])$

(case-cx) $(\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env\ \texttt{in}\ C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ alts])$
$\to \texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env$
$\texttt{in}\ C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]$

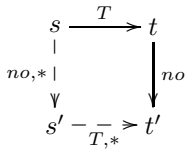
(case-cx) $\texttt{letrec}\ x = (c_{T,j}\ x_1 \ldots x_n), Env,$
$y = C[\texttt{case}_T\ x\ ((c_{T,j}\ y_1 \ldots y_n) \to s)\ alts]\ \texttt{in}\ r$
$\to\ \texttt{letrec}\ x = (c\ x_1 \ldots x_n), Env,$
$y = C[(\texttt{letrec}\ y_1 = x_1, \ldots, y_n = x_n\ \texttt{in}\ s)]\ \texttt{in}\ r$

(case-cx) like (case) in all other cases

Fig. 13: Extra Transformation Rules

8 Extending the Unification Algorithm to Commuting Diagrams

Commuting diagrams (as defined in section 2.4) are of the form



They describe the situation that a reduction sequence that consists of a transformation T and a normal order reduction is turned into a sequence of no-reductions followed by a sequence of transformations (i.e. the transformation T is swapped behind the normal order reduction).

Commuting diagrams can be derived as a forking diagrams using the following observation: It is sufficient to determine all overlaps of T^{-1} with normal order reductions , i.e. determine all forks of the form $s' \xleftarrow{no} s \xrightarrow{T^{-1}} t$, which correspond to the commuting sequence $t \xrightarrow{T} s \xrightarrow{no} s'$. So we can use the same unification algorithm provided the conditions are met. However, right hand sides of our considered transformations are different from the left hand sides.

Our encoding of expressions also applies to right hand sides of transformations see Definition 3.21. However, several right hand sides violate conditions for initial unification problems. Problematic cases are the cp-transformations, that may contain variables standing for values twice, e.g. in the right hand side of (cp-e-S) ($\texttt{letrec}\ x = v, Env, y = S[v]\ \texttt{in}\ r$) the meta-variable v occurs twice. A naive encoding into $\mathcal{T}_{CH}$ would violate the almost linearity condition, which is crucial for termination and completeness of the unification algorithm. Another violation is that the (llet)-rules have two variables in the environment: Env_1, Env_2, which cannot be treated by the current set of rules.

Since proper non-linearity prohibits the use of our unification algorithm, we choose to encode a slight variant of right-hand sides of the (cp)-rules, where one v is translated into $lam(x, s)$ and the other into $lam(x', s')$, and then the unifier is checked whether it instantiates s, s' or not, and for the two environment variables, we extend the rule **Solve-Env**.

Definition 8.1. *We define the set rhs_T of encoded right hand sides (*rhs*) of unrestricted reduction rules of the calculus LR.*

rhs_T is the following set of encodings of right hand sides of an unrestricted LR *reduction rule (see figures 6, 2), where first the rules are instantiated:*

1. *The phrase "v is a value" will lead to instantiations into an abstraction $\lambda x.t$ and constructor terms, one possibility for every constructor. If a right hand side contains two occurrences of the same variable v and the side condition "v is a value", we instantiate one occurrence with $\lambda x.t$ and the other one with $\lambda x'.t'$ (where all variables are fresh).*
2. *The other instantiations are done as in definition 3.21.*

Remark 8.2. After this change, all terms in rhs_T satisfy the conditions concerning chains from 3.22 and they are almost linear. One difference is that they may contain environment terms with more than one variable of sort *Env*, e.g. (`letrec` Env_1, Env_2 `in` r) the rhs of (llet-in). This and the item 1 from above must be kept in mind during unification (i.e. this requires a slight modification of the unification algorithm).

The initial *LR*-commuting-problems describe all commuting sequences between transformations and reductions of *LR*.

Definition 8.3. *We consider the set of unification problems*

$$IPC := \{\{S(r) \doteq l\} \mid r \in rhs_T, l \in lhs_{no}\}$$

where S is a context variable of context-class S. The terms r, l are assumed to be variable disjoint, which can be achieved by renaming. The initial set Δ of context variables only contains the A_1-context in case l comes from a (cp-e)-reductions. The pair (Γ, Δ) with $\Gamma \in IPC$ is called an initial *LR*-commuting-problem.

To solve initial *LR*-commuting-problems we have to slightly modify the unification algorithm to address the two problems from remark 8.2:

- Two occurrences of the same variable v in one term: This is addressed during encoding where the two occurrences of v are encoded as $lam(x, s)$ and $lam(x', s')$ respectively. For all computed unifiers σ of initial commuting problems that contain two such terms, we have to check if s, s' are instantiated by different terms. If so, the unifier σ must be discarded, and unification fails: In this case our unification algorithm cannot compute the complete set of unifiers. We expect that this does not happen in the calculus LR with the transformations in Figure 6. Experimental results for the calculus L_{need} ([RSS11]) support this conjecture.
- Two environment variables may be in the same environment in a right hand side of a rule, as in

  (llet-in) (`letrec` Env_1 `in` (`letrec` Env_2 `in` r)) $\to$ (`letrec` Env_1, Env_2 `in` r)
  (llet-e) (`letrec` $Env_1, x =$ (`letrec` Env_2 `in` s_x) `in` r)
  $\to$ (`letrec` $Env_1, Env_2, x = s_x$ `in` r)

  Overlapping the left-hand side of cp-rules, e.g. (cp-in): (`letrec` $x_1 = v, \{x_{i+1} = x_i\}_{i=1}^{m} Env$ `in` $C[x_m]$) with the above right hand sides is not possible with the current unification rules. Extending the rule **Solve-Env** without precaution must take care of all partitions of the variable chain into the two variables Env_1, Env_2, which would result in an infinite number of solutions, and hence nontermination. The observation that the scoping of Env_1, Env_2 is restricted (variables bound in Env_2 have their scope in Env_2, but not in Env_1), permits to represent the possible unifiers and to extend the unification rules to a terminating and complete unification algorithm. A

further restriction is that the two environment variable case only appears in the right hand side of transformations which do not contain chains.
If two variables r_1, r_2 of sort *Env* occur in one environment-term we modify **Solve-Env** to deal with the problem state

$$S;\ \{env^*(M_1 \cup \{r_1, r_2\}) \doteq env^*(M_2 \cup r_3\} \uplus P$$

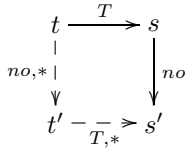
where we assume that the bindings in M_1, M_2 are not further unified.
The **Solve-Env**-variant can first be applied to remove M_1:

$$\frac{S;\ \{env^*(M_1 \cup \{r_1, r_2\}) \doteq env^*(M_2 \cup r_3)\} \uplus P}{\{r_3 \doteq env^*(M_1 \cup z_3)\} \cup S;\ \{env^*(\{r_1, r_2\}) \doteq env^*(M_2 \cup z_3)\} \uplus P}$$

Then the following possibilities may be chosen:

1. The remaining single bindings in M_2 have to be in r_1, or r_2.
2. There may be a chain or even two chains (for example (no,cp-e)) in M_2. The scoping considerations allow us to make only a single split: Select one split in the chain (the two chains seen as one), and assing one part to r_1 and the other part to r_2.

Unfortunately, this is not sufficient, since the computed overlap may be invalid, because applying the transformation backwards may lead to expressions violating the DVC.
Thus a final check is necessary, whether all the computed expressions s, s', t satisfy the DVC after instantiation.



Hence complete (and finite) sets of commuting diagrams for *LR* transformations can be computed using a slight variant of the encoding to deal with duplicate v-variables and a modified **Solve-Env**-rule together with an extended DVC-check.

9 Conclusion

We investigated the extended call-by-need λ-calculus *LR* from [SSSS08] which is a core language of pure Haskell. The calculus is equipped with a contextual semantics for program equivalence that naturally leads to the notion of correctness of program transformations. One crucial step in the proof of the correctness of a program transformation is the determination of complete sets of forking and commuting diagrams. In [SSSS08] the those diagram sets where generated by hand. Our presented method is able to automatically compute the overlaps between the core transformations of *LR* and the normal order reductions, thereby determining a complete set of forks for forking diagrams.

For the computation of the overlaps we translate the transformations and normal order reduction of LR into the term language $\mathcal{T}_{CH}$ that captures the special syntactic constructs of the LR reductions, as they are: 1. sorts, 2. context variables of different context classes, 3. commutativity of bindings in letrec-environments, 4. bound variables and 5. chains of bindings.

All overlaps in LR are then described by the special set of unification problems, the initial LR forking-problems. We presented a unification algorithm to solve those problems and proofed its termination, soundness and completeness. Thereby we showed, that the computation of of all overlaps of the core transformations of the LR calculus is possible and yields a finite and complete sets of forks. The presented method can also be used to compute overlaps of additional transformations (from section 7), if their left-hand-sides can be encoded into $\mathcal{T}_{CH}$ and satisfy some additional restrictions on variable occurrences (i.e. almost linearity) and binding chains (i.e. they T-chain-restrictions from definition 3.22).

If we slightly modify the rules of our unification algorithm we can also use it to compute all commuting sequences for the core transformations in LR . Hence the unification algorithm is a crucial part in the automatization of correctness proofs for program transformations because can be used to determine the all forks and commuting sequences which have to be closed to generate complete diagram sets.

Outlook: The next steps in the automatization of correctness proofs are closing of diagrams and automatic induction. The closing of forking diagrams for the simpler calculus L_{need} is described in [RSS11]. There matching is used to reduce terms and a search procedure is employed to find a common reduct that joins the overlaps. This method is able to automatically close all the determined overlaps in LR. We conjecture that a similar procedure can close the diagrams in LR.

To automatize induction, our future research investigates the following ideas: A diagram can be interpreted as an rewrite rule (on strings), that rewrites a sequence of reductions into another sequence. And a complete set of diagrams can be interpreted as a TRS D. To automatically verify induction (e.g. on the length of the normal order reduction) it has to be checked, if D is a terminating TRS. This can be done using a tool that automatically proofs termination of term rewrite systems (TRS).

References

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.

Her92. Miki Hermann. On the relation between primitive recursion, schematization and divergence. In *ALP*, pages 115–127, 1992.

HG97. Miki Hermann and Roman Galbavý. Unification of infinite sets of terms schematized by primal grammars. *Theor. Comput. Sci.*, 176(1–2):111–158, 1997.

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.

RSS11. Conrad Rau and Manfred Schmidt-Schauß. Towards correctness of program transformations through unification and critical pair computation. Frank report 41, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main, January 2011.

Sal92. Gernot Salzer. The unification of infinite sets of terms and its applications. In *LPAR 1992*, volume 624 of *LNCS*, pages 409–420, 1992.

Sal93. Gernot Salzer. On the relationship between cycle unification and the unification of infinite sets of terms. In Wayne Snyder, editor, *UNIF 1993*, 1993.

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.

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