Unification of Program Expressions with Recursive Bindings

Post-Conference Version from November 22, 2016

Manfred Schmidt-Schauß
Goethe-University, Frankfurt am Main
schauss@ki.cs.uni.frankfurt.de

David Sabel*
Goethe-University, Frankfurt am Main
sabel@ki.cs.uni.frankfurt.de

ABSTRACT

This paper presents an algorithm for unification of meta-expressions of higher-order lambda calculi with recursive bindings. The meta-language uses higher-order abstract syntax. Besides usual unification variables for expressions and term variables, there are context-variables for generalized shapes of contexts, environment variables for sets of bindings, and (flexible) chain-variables as they, for instance, occur in formal descriptions of the operational semantics of lazy functional programming languages with shared environments. To exclude solutions with unintend scoping, the algorithm takes advantage of so-called non-capture constraints. The expressiveness of the meta-language comprises reduction contexts to support reasoning on program evaluation under reduction strategies. The non-deterministic unification algorithm runs in polynomial time provided certain restrictions on the number of occurrences of unification variables hold. The deterministic version of the algorithm will output a finite and concise set of representatives of all solutions. Results on an implementation of the algorithm are presented. The experiments focus on computing critical pairs of equations from program calculi modelling lazy functional languages which support the resoning on the correctness of program transformations.

CCS Concepts

•**Theory of computation** → **Program verification;** *Operational semantics;* •**Software and its engineering** → **Functional languages;**

Keywords

unification, lambda calculus, letrec, operational semantics, critical pairs, call-by-need

*The second author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA 2908/3-1.

This paper originally appeared as

Manfred Schmidt-Schauß and David Sabel. Unification of program expressions with recursive bindings. In Germanàn Vidal, editor, PPDP '16: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, pages 160–173, New York, NY, USA, 2016. ACM.

Compared to the conference version, this version of the paper contains some modifications and clarifications. These changes are marked by blue font.
Last change: November 22, 2016

1. INTRODUCTION

Motivation. The development of deductive techniques to reason about programs and their semantics is an important topic, since it supports the correct construction of programs in programming languages and software systems. For instance, those techniques may be applied in the compilation of programs to built smart and correct compilers, but also in verification of programs where correct reasoning on programs is an indispensable requirement.

We are in particular interested in lazy functional programming languages like Haskell, since – due to the declarative programming paradigm – the result of a program is specified, but not the exact sequence of computations. This opens options for optimizations, since these languages have a large set of correct program transformations. The semantics of lazy functional programming languages can be modeled by extended lambda calculi with a call-by-need operational semantics (see e.g. [2, 1, 21, 20]). Such a calculus consists of the syntax of the language and a formal description of the operational semantics which e.g. may be defined as a small-step, or a big-step reduction relation, or by an abstract machine.

The goal of this paper is to support syntactic reasoning in those languages by providing a unification algorithm for unifying meta-expressions as they occur in program calculi descriptions.

A common construct found in the core languages of modern functional programming languages is a cyclic let (called letrec), which allows to express mutually recursive bindings as well as sharing of expressions. For instance, the following Haskell-expression (written desugared) applies the identity function to a list of Boolean values, using a cyclic let to define the recursive `map`-function, the identity `id`, the input `list` and the output `result`:

```
let map    = \f xs -> case xs of
                        []     -> []
                        (y:ys) -> (f y):(map f ys)
    id     = \x -> x
    list   = True:False:[]
    result = map id list
in result
```

The order of the bindings in such a let-expression is irrelevant, i.e. the bindings are seen as a *set* of bindings. Clearly, this commutativity of bindings has to be respected by a unification algorithm. Call-by-need evaluation of the expression is quite tricky, for instance, for a first computation step, the map-function has to be inlined in the right hand side of binding result. This search for the redex is usually modeled by a rather complex notion of reduction contexts (e.g. [1, 20]) or by using unwinding in abstract machines and storing the continuations on a stack (e.g. [21]). In the formal description of calculus rules, meta-variables for sets of bindings, meta-notation for the redex search, binders with scoping, and contexts occur (we give an example in Sect. 2). All of them have to be respected by the unification algorithm and the corresponding meta-language in use.

For our meta-language, we will use higher-order abstract syntax [9] to present binders and augment it by meta-symbols for variables, expressions, contexts, and most importantly by variables for letrec-environments and special environment chains, as they occur in semantics descriptions for program calculi with cyclic let. Even though the inspiration and application comes from call-by-need lambda-calculi, the techniques are adaptable also to other higher-order calculi. In the unification algorithm, the meta-symbols become unification variables. The unification algorithm is given by non-deterministic rules, which treat all these syntactic constructs, where we introduce several restrictions on the problems to ensure termination of the unification algorithm.

Results. The main achievement of this paper is the construction of a practically useful unification algorithm for a parametrized meta-notation that allows to unify meta-expressions as they occur in the description of the operational semantics of program calculi with recursive bindings and shared bindings, and outputs a finite set of constraints as solutions. The expressivity of the language comprises meta-variables for variables, expressions, environments, contexts, constructs to express the positions of call-by-need reduction, non-emptiness constraints for context variables and environments, and non-capture constraints for expressions in contexts to exclude name captures in possible solutions. The meta-language is parametrized by context classes and can be used for various calculi.

We prove that our unification algorithm is sound and complete and that it runs in non-deterministic polynomial time (Theorem 5.12) and the determinized version produces a finite set of constraints. In fact the problem itself is NP-complete (see Theorem 5.14).

We also have implemented the meta-language and the algorithm, and show some experimental results, where our examples stem from applying unification to automatize the proof of the correctness of program transformations: here, critical pairs between the standard calculus reduction steps and the transformation steps can be computed by unification (see Sect. 2 for more details).

Related Work. Unification of first-order terms and variants thereof is well-known (see e.g. [3]). Higher-order unification [5] is unification modulo β- or $\beta\eta$-equivalence and thus different from our purely syntactic approach. Higher-order abstract syntax was introduced in [9] and used for implementing higher-order unification and matching, where also recursive bindings are modeled, using higher-order abstract syntax, but where the bindings are ordered and not treated as a set as in our approach. Compared to these previous works, our approach addresses semantic descriptions of program calculi and thus combines several unification techniques and introduces novel kinds of unification variables, representing parts of letrec environments and chains.

The extension of first-order unification by context unification without restriction on occurrences is shown decidable and in PSPACE [6] (see [14] for an earlier overview). Our use of context variables is far more restricted, since they occur at most once.

A further technique for unification and (syntactic) reasoning about higher-order expressions and binders is nominal unification [23], which can solve equations between lambda-expressions with variables w.r.t. α-equivalence in polynomial time. However, our focused functional languages have cyclic letrec in their syntax (see e.g. [4] for a discussion on reasoning with more general name binders, and [22] for a formalization of general binders in Isabelle). A nominal unification algorithm for higher-order expressions including a letrec construct is described in [15], which, however, cannot deal with environment, context and chain variables. Thus it cannot be used for our aims, since our meta-language must cover meta-expressions that can express reduction strategies to determine the required positions. Hence, a more expressive language is required and hence also an adapted unification algorithm, while unification w.r.t. α-equivalence is not necessary for our purposes.

A previous attempt to unify expressions with letrec appeared in [11], however, with several restrictions like linearity of expressions, an insufficient representation of letrec-environments and an incomplete modeling of the bindings. Besides a more generic meta-language which is parametrized by contexts, we improve upon [11] in several respects: (i) the letrec environments are treated as multisets in a natural way and can contain more than one environment-variable; (ii) the problems may be non-linear: unification-variables for bound variables may occur several times, and unification-variables for expressions may occur at most twice, which permits to cover more requests for computing overlaps.

Outline. In Sect. 2 we motivate our design decisions for the meta-language, which is then introduced in Sect. 3. We define the letrec unification problem and introduce and illustrate our unification algorithm UnifLRS in Sect. 4. In Sect. 5 we show correctness of the algorithm and prove NP-completeness of the letrec unification problem. In Sect. 6 we report on an implementation of UnifLRS. We conclude in Sect. 7.

2. MOTIVATING DESIGN DECISIONS

In this section, we briefly discuss the diagram-based proof method to show correctness of program transformations [7, 24, 20] as an application where our unification algorithm UnifLRS can be used as a subprocedure. We also illustrate our design decisions using a minimalistic call-by-need lambda calculus.

The so-called diagram method can roughly be summarized as follows: first all critical pairs between calculus reductions and transformation steps have to be computed (where often only strategy-specific positions instead of all positions are overlapped), then the critical pairs have to be joined (in a specific form) which leads to a complete set of diagrams. Finally, the diagrams can be used in an inductive proof to show correctness of the transformation (this step can be automatized as described in [10]). In call-by-need lambda calculi with letrec, the diagram method was e.g. used to prove correctness of program transformations in [20, 12], to show for

Expressions e where v, v_i, w, w_i are variables

$e ::= w \mid \lambda w.e \mid (e_1\ e_2) \mid \texttt{letrec}\ w_1{=}e_1, \ldots, w_n{=}e_n\ \texttt{in}\ e$ $\qquad$ $\mathit{Env} ::= \emptyset \mid w_1{=}e_1, \ldots, w_n{=}e_n$

Application contexts A and reduction contexts R

$A ::= [\cdot] \mid (A\ e)$ $\qquad$ $R ::= A \mid \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ A \mid \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m, \mathit{Env}\ \texttt{in}\ A_0[w_1]$

Normal-order reduction $\xrightarrow{no}$

(no,lbeta) $R[((\lambda w.e_1)\ e_2)] \rightarrow R[\texttt{letrec}\ w{=}e_2\ \texttt{in}\ e_1]$

(no,cp-in) $\texttt{letrec}\ \{w_i{=}w_{i+1}\}_{i=1}^{m-1}, w_m{=}\lambda w.e, \mathit{Env}\ \texttt{in}\ A_0[w_1] \rightarrow \texttt{letrec}\ \{w_i{=}w_{i+1}\}_{i=1}^{m-1}, w_m{=}\lambda w.e, \mathit{Env}\ \texttt{in}\ A_0[\lambda w.e]$

(no,cp-e) $\texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m[v_1], \{v_j{=}v_{j+1}\}_{j=1}^{n-1}, v_n{=}\lambda w.e, \mathit{Env}\ \texttt{in}\ A[w_1]$
$\rightarrow \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m[\lambda w.e], \{v_j{=}v_{j+1}\}_{j=1}^{n-1}, v_n = \lambda w.e, \mathit{Env}\ \texttt{in}\ A[w_1]$
where $A_m \neq [\cdot]$ and $m \geq 1, n \geq 1$

(no,llet-in) $\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ \texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ e \rightarrow \texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ e$

(no,llet-e) $\texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}(\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ e), \mathit{Env}_2\ \texttt{in}\ A_0[w_1]$
$\rightarrow \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}e, \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ A_0[w_1]$

(no,lapp) $R[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e_1)\ e_2] \rightarrow R[\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (e_1\ e_2)]$

Some program transformations

(lbeta) $((\lambda w.e_1)\ e_2) \rightarrow \texttt{letrec}\ w{=}e_2\ \texttt{in}\ e_1$

(cp-in) $\texttt{letrec}\ w{=}\lambda v.e, \mathit{Env}\ \texttt{in}\ C[w] \rightarrow \texttt{letrec}\ w{=}\lambda v.e, \mathit{Env}\ \texttt{in}\ C[\lambda v.e]$

(cp-e) $\texttt{letrec}\ w_1{=}\lambda v.e, w_2{=}C[w_1], \mathit{Env}\ \texttt{in}\ e' \rightarrow \texttt{letrec}\ w_1{=}\lambda v.e, w_2{=}C[\lambda v.e], \mathit{Env}\ \texttt{in}\ e'$

(llet-in) $\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ (\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ e) \rightarrow \texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ e$

(llet-e) $\texttt{letrec}\ \mathit{Env}_1, x{=}(\texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ e_1)\ \texttt{in}\ e_2 \rightarrow \texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2, x{=}e_1\ \texttt{in}\ e_2$

(lapp) $((\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e_1)\ e_2) \rightarrow \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (e_1\ e_2)$

(gc1) $\texttt{letrec}\ w_1{=}e_1, \ldots, w_n{=}e_n, \mathit{Env}\ \texttt{in}\ e \rightarrow \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e$ if for all $i : w_i$ does not occur in Env, e

(gc2) $\texttt{letrec}\ w_1{=}e_1, \ldots, w_n{=}e_n\ \texttt{in}\ e \rightarrow e,$ if for all $i : w_i$ does not occur in e

(cpx-in) $\texttt{letrec}\ w{=}v, \mathit{Env}\ \texttt{in}\ C[w] \rightarrow \texttt{letrec}\ w{=}v, \mathit{Env}\ \texttt{in}\ C[v],$ if $v \neq w$

(cpx-e) $\texttt{letrec}\ w{=}v, w'{=}C[w], \mathit{Env}\ \texttt{in}\ e \rightarrow \texttt{letrec}\ w{=}v, w'{=}C[v], \mathit{Env}\ \texttt{in}\ e,$ if $v \neq w$

(xch) $\texttt{letrec}\ w{=}e, v{=}w, \mathit{Env}\ \texttt{in}\ e' \rightarrow \texttt{letrec}\ v{=}e, w{=}v, \mathit{Env}\ \texttt{in}\ e'$

Figure 1: The calculus L_{need}

correct program transformations that these improve the run-time behavior of programs in [17], and to show equivalence of reduction strategies e.g. in [19]. Similar diagram methods were e.g. used in [7, 24] to prove meaning preservation of program transformations Computing the critical pairs is usually done manually by a case-analysis of all possible overlaps. One application of our new unification algorithm is to contribute to the automation of this step.

To get an impression of a description of a call-by-need calculus, in Fig. 1 the call-by-need lambda calculus with `letrec` L_{need} [18] is shown, i.e. its syntax, small-step operational semantics, called normal-order reduction, and a selection of program transformations are shown. This should be understood as a running example, not as the overall target of our algorithm. Indeed, we are interested in more complex calculi, which e.g. model the full core language of Haskell, like the LR-calculus in [20, 17] (which extends L_{need} by constructors, `case`-expressions, and Haskell's `seq`-operator) and our meta-language and algorithm are designed to treat those more sophisticated calculi. However, for justifying our design decisions, we discuss examples of overlaps in the calculus L_{need}. Consider the rule (llet-in) which transforms expressions $(\texttt{letrec}\, \mathit{Env}_1\, \texttt{in}\, (\texttt{letrec}\, \mathit{Env}_2\, \texttt{in}\, e))$ into $(\texttt{letrec}\, \mathit{Env}_1, \mathit{Env}_2\, \texttt{in}\, e)$. Here, $\mathit{Env}_1, \mathit{Env}_2$ are meta-variables representing any (non-empty) `letrec`-environment and e is a meta-variable any expression. As a normal-order reduction, the rule (llet-in) can be applied only at the top of expressions, and as a program transformation, it can be applied anywhere in the expression. Critical pairs can be found by examining the overlaps between the left-hand sides (lhs) with the calculus rules, respecting the strategy. A nontrivial overlap with itself is that $(\texttt{letrec}\, \mathit{Env}_2\, \texttt{in}\, e)$ equals $(\texttt{letrec}\, \mathit{Env}'_1\, \texttt{in}\, (\texttt{letrec}\, \mathit{Env}'_2\, \texttt{in}\, e'))$, which is possible if $\mathit{Env}_2 = \mathit{Env}'_1$ and $e = (\texttt{letrec}\, \mathit{Env}'_2\, \texttt{in}\, e')$. The corresponding (also joining) diagram is:

$$\begin{tikzcd}
\begin{array}{l}\texttt{letrec}\, \mathit{Env}_1\, \texttt{in} \\ (\texttt{letrec}\, \mathit{Env}'_1\, \texttt{in} \\ (\texttt{letrec}\, \mathit{Env}'_2\, \texttt{in}\, e'))\end{array} \arrow[r, "llet"] \arrow[d, "{no,llet}"'] & \begin{array}{l}\texttt{letrec}\, \mathit{Env}_1\, \texttt{in} \\ (\texttt{letrec}\, \mathit{Env}'_1; \mathit{Env}'_2\, \texttt{in}\, e')\end{array} \arrow[d, dashed, "{no,llet}"] \\
\begin{array}{l}\texttt{letrec}\, \mathit{Env}_1; \mathit{Env}'_1 \\ \texttt{in}\, (\texttt{letrec}\, \mathit{Env}'_2\, \texttt{in}\, e')\end{array} \arrow[r, dashed, "llet"'] & \begin{array}{l}(\texttt{letrec}\, \mathit{Env}_1; \\ \mathit{Env}'_1; \mathit{Env}'_2\, \texttt{in}\, e')\end{array}
\end{tikzcd}$$

where the vertical arrows are the normal-order reductions and the horizontal ones are the program transformations. Scoping problems may be omitted in this reasoning step, however, an example for the nontrivial information which may be required by other reasoning steps (e.g. for computing joins) are guarantees that the binders in Env'_2 cannot bind free variables in Env_1. Another part of the reasoning on correctness requires to look for the reversed transformation rules, and hence also overlaps of the corresponding lhs and/or rhs are required.

The rule (cpx-in) transforms $(\texttt{letrec}\, w = v, \mathit{Env}\, \texttt{in}\, C[w])$ into $(\texttt{letrec}\ w = v, \mathit{Env}\ \texttt{in}\ C[v])$. Here C is a meta-variable for any context. Besides including this notion in our meta-language, also the information that neither w nor v is captured in the hole of C has to be included. Both cases will be dealt with by so-called non-capture constraints of the form (e, C), which forbid the capture of free and bound variables in $e\sigma$ by $C\sigma$ under a unifier σ.

In the right hand side $\texttt{letrec}\, w = \lambda v.e, \mathit{Env}\, \texttt{in}\, C[\lambda v.e]$ of rule (cp-in) the expression $\lambda v.e$ occurs twice. Hence, we will allow such occurrences syntactically, and thus some non-linearity of the input unification program is required. In our

$$
\begin{array}{rl}
x, y \in \mathbf{Var} ::= & X \mid Y \mid \mathsf{x} \mid \mathsf{y} \\
s, t \in \mathbf{Expr} ::= & S \mid D^{\mathcal{K}}[s] \mid \mathtt{letrec}\, env \,\mathtt{in}\, s \mid (f\, r_1 \ldots r_{ar(f)}) \\
& \text{where } r_i \text{ is } o_i \in \mathbf{HExpr}^{n_i} \text{ if } oar(f)(i) = n_i > 0, \\
& \quad r_i \text{ is } s_i \in \mathbf{Expr} \text{ if } oar(f)(i) = 0, \text{ and} \\
& \quad r_i \text{ is } x_i \in \mathbf{Var}, \text{ if } oar(f)(i) = \mathbf{Var}. \\
o \in \mathbf{HExpr}^1 ::= & x.s \quad \text{where } s \in \mathbf{Expr} \\
o \in \mathbf{HExpr}^n ::= & x.o_1 \quad \text{if } o_1 \in \mathbf{HExpr}^{n-1} \text{ and } n > 1 \\
b \in \mathbf{Bind} ::= & x.s \quad \text{where } s \in \mathbf{Expr} \\
env \in \mathbf{Env} ::= & \emptyset \mid E; env \mid Ch^{\mathcal{K}}[x, s]; env \mid b; env
\end{array}
$$

The constructs $X, Y, S, D^{\mathcal{K}}, E, Ch^{\mathcal{K}}$ are meta-variables.

Figure 2: Syntax of LRSX

modeling, any occurrences of meta-variables for variables and at most two occurrences of meta-variables for expressions are permitted.

Finally, the definition of normal-order reduction demands further expressivity of the meta-notation, to represent chains of `letrec`-bindings, as they occur in for instance in form of $\{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m}$ in the definition of reduction contexts R in Fig. 1. As an example, consider the L_{need}-expression $\mathtt{letrec}\, v_1{=}(v_2\, v_2), v_2{=}v_3, v_3{=}((\lambda u.u)\, w)\, \mathtt{in}\, (v_1\, u')$. The reduction position of redex $((\lambda u.u)\, w)$ in the `letrec`-environment can be represented schematically, by an environment $([\cdot]{=}(v_2\, v_2), v_2{=}v_3, v_3{=}[\cdot])$ that has two holes and represents a chain of bindings (for the example the first hole has to be filled by v_1 and the second hole by $((\lambda u.u)\, w)$).

To cover all those environment-chains, our syntax allows to write $Ch^A[v_1, (\lambda u.u)\, w]$ where Ch^A is a meta-variable for environment-chains parametrized by A-contexts.

3. THE META-LANGUAGE

We introduce the syntax of meta-expressions and explain the semantics and the intention of the notation. Our syntax for (ground) language expressions is a general one for functional programming languages that are extended lambda calculi. It comprises for example the syntax of [20, 17, 8, 2] – e.g., in [8], the argument position in applications are restricted to variables, which can be represented in our meta-syntax. Our meta-language LRSX will be parametrized over a set $\mathcal{F}$ of function symbols and – to support generality of the algorithm – over a finite set $\overline{K}$ of context classes. Contexts and context classes will be explained in more detail later in Definition 3.3, since for the definition of the syntax of LRSX they only appear as index on unification-variables.

Definition 3.1 (Syntax of the meta-language LRSX)**.** *For a set $\overline{K}$ of context classes and a set $\mathcal{F}$ of function symbols, the syntax of the language LRSX$(\overline{K}, \mathcal{F})$ is defined by the grammar in Fig. 2. It uses four syntactic categories of objects (called types):* **Var** *is a countably-infinite set of* variables*,* **HExpr** *are higher-order expressions,* **Env** *are letrec-environments, and* **Bind** *are* `letrec`*-bindings*[1]*.*

Elements o of **HExpr** *have an* order *$order(o) \in \mathbb{N}_0$, where $\mathbf{HExpr}^n$ denotes the elements of* **HExpr** *of order n, and where $\mathbf{HExpr}^0$ may be abbreviated as* **Expr***.*

Every function symbol $f \in \mathcal{F}$ has a syntactic type of the form $f : \tau_1 \to \ldots \to \tau_n \to \mathbf{Expr}$, where τ_i may be **Var***, or $\mathbf{HExpr}^{k_i}$; n is called the* arity *of f, denoted $ar(f)$; and the* order arity *$oar(f)$ is the n-tuple $\langle \delta_1, \ldots, \delta_n \rangle$, where $\delta_i = k_i \in \mathbb{N}_0$, or $\delta_i = \mathbf{Var}$, depending on the type of f. We write $oar(f)(i)$ to extract the order arity of the i^{th} argument of f (i.e. $oar(f)(i) = \delta_i$). For every function symbol f, there is also a* set of strict positions *$SPOS(f) \subseteq \{i \mid 1 \le i \le ar(f), oar(f)(i) = 0\}$. In $\mathcal{F}$ there is at least a unary operator* `var` *of type* **Var** $\to$ **Expr** *which lifts variables to expressions, where $ar(\mathtt{var}) = 1$, $oar(\mathtt{var}) = \langle \mathbf{Var} \rangle$, and $SPOS(\mathtt{var}) = \emptyset$.*

The syntax comprises concrete term variables, unification variables, and additionally we use meta-symbols to denote syntactic constructs. Thus, we carefully use different fonts and lower- or upper-case letters to distinguish them: concrete term-variables of type **Var** *are denoted by* x, y*, and x, y are used as meta-symbols to denote both – a concrete term variable or a unification variable. Similarly, lower case letters s, t denote expressions, env denotes environments, and b denotes bindings.*

Upper-case letters always denote unification variables. The different kinds of unification variables are: unification variables X, Y of type **Var***, unification variables S of type* **Expr***, E of type* **Env** *standing for (partial) environments, $D^{\mathcal{K}}$ for a context variable of context class $\mathcal{K} \in \overline{K}$, and two-hole environment-context variables $Ch^{\mathcal{K}}[\cdot, \cdot]$ which are of the type* **Var** $\to$ **Expr** $\to$ **Env***, and occur with a context-class index $\mathcal{K}$, a* **Var***-argument x, and an* **Expr***-argument s.*

An LRSX*-expression s is called* ground *(an* LRS-expression*), iff it does not contain any unification variable.*

Note that we use the same notation $x.s$ for binders in letrec-environments and for higher-order expressions, which will not lead to confusion, since the meaning is always clear from the context. We also are a bit sloppy in the notation of environments by using ";" for constructing and concatenating environments and by not notationally distinguishing between a single binding and environments consisting of a single binding.

Example 3.2. *We illustrate the meta-language by some examples. Abstractions are represented by $\lambda{\in}\mathcal{F}$ with $ar(\lambda){=}1$, $oar(\lambda){=}\langle 1 \rangle$, and $SPOS(\lambda){=}\emptyset$. Since $oar(\lambda){=}\langle 1 \rangle$, the function symbol λ must be applied to a higher-order expression of order 1. For instance, the identity function is represented by applying λ to* x.(`var` x) *written as* λ x.(`var` x)*. Applications can be represented by a function symbol* `app` *with $ar(\mathtt{app}){=}2$, $oar(\mathtt{app}){=}\langle 0, 0 \rangle$, and $SPOS(\mathtt{app}){=}\ \{1\}$.*

If $oar(f)(i){=}\mathbf{Var}$, then the i^{th} argument of function symbol f must be a variable. For instance, with the definition `appx` $\in \mathcal{F}$*, $oar(\mathtt{appx}) = \langle 0, \mathbf{Var} \rangle$ we introduce an application which allows any expression as first argument, but restricts the second argument to variables (e.g. in [8] such applications occur). For example,* `appx` (λ(x.`var` x)) y *is a valid expression, while* `appx` (λ(x.`var` x)) (λ.(y.`var` y)) *is invalid.*

Constructors can be represented by a function symbol f where $SPOS(f){=}\emptyset$ and $oar(f)$ is a tuple of only 0-s (or of **Var***-s if constructor arguments are restricted to variables as e.g. in [8]). For example, the list constructors are* `nil` *and* `cons` *with $ar(\mathtt{nil}){=}0$, $oar(\mathtt{nil}){=}\langle\rangle$, $ar(\mathtt{cons}){=}2$, $oar(\mathtt{cons}){=}\langle 0, 0 \rangle$, and $SPOS(\mathtt{cons}){=}SPOS(\mathtt{nil}){=}\emptyset$. Also case-constructs are representable, e.g., let $case_{list} \in \mathcal{F}$ with $ar(case_{list}){=}3$, $oar(case_{list}) = \langle 0, 0, 2 \rangle$, $SPOS(case_{list}){=}\{1\}$. The first argument of $case_{list}$ is the to-be-cased expression, the second argument is the case-alternative for the empty list, and the third argument is the case-alternative for the*

[1]We could omit the type **Bind** since it is a synonym of $\mathbf{HExpr}^1$, but to increase readability, we include both types.

non-empty list, which is a higher-order expression of order 2, since it binds two variables: one for the head and one for the tail of the list. The map-function can be written as

```
letrec
  map.λf.λxs.case_list (var xs) nil
               y.ys.(cons
                      (app (var f) (var y))
                      (app (app (var map) (var f)) (var ys)))
in (var map)
```

Definition 3.3 (Contexts and context classes)**.** Context expressions *are defined like expressions, where the new constant symbol* $[\cdot] : \mathbf{Expr}$ *(the hole) is permitted, and where we assume that the hole occurs exactly once in contexts. With* D *we denote unification-variables for contexts,* $\mathbb{D}$ *always represents a concrete context (i.e. an* LRS*-expression with a hole, called an* LRS-context*), and by* d *we denote* LRSX*-contexts, i.e. contexts, that may contain unification-variables.*

The operation plugging an expression s *in the hole of* d *is written as* $d[s]$*. Contexts with* $k > 1$ *holes are denoted using several hole symbols* $[\cdot_1], \dots, [\cdot_k]$*, where every hole symbol occurs exactly once.*

A context class $\mathcal{K}$ *is a set of contexts, where we assume that a context class comes with a grammar for defining the corresponding contexts as well as with particular algorithms (see Definition 4.5 and Fig. 5). We allow the context class Triv which consists only of the empty context* $[\cdot]$*.*

Definition 3.4 (Scoping rules and sets of variables)**.** *For any syntactic object* r*, let* $UV(r)$ *be the set of unification variables occurring in* r*. In a higher-order expression* $x.r$*, the scope of* x *is* r*. The scope of* x *in* letrec $x.s; env$ in s' *or* letrec $Ch^{\mathcal{K}}[x,s]; env$ in s' *is* s*,* env*,* $Ch^{\mathcal{K}}$ *and* s' *(and thus* letrec*-bindings are recursive).*

We use $FV(r)$ *to denote the set of variables* x *that are not bound by some higher-order binder, a let-binding, or the variable* x *in* $Ch^{\mathcal{K}}[x,s]$*, and we use* $BV(r)$ *to denote the set of bound variables. We write* $Var(r)$ *for* $FV(r) \cup BV(r)$*.*

For a context d*,* $CV(d)$ *(the captured variables) denotes the set of variables* x *which become bound if plugged into the hole of* d*.* $Ch^{\mathcal{K}}$ *is only permitted for context classes* $\mathcal{K}$ *such that* $CV(\mathbb{D}^{\mathcal{K}}) = \emptyset$ *for all* $\mathbb{D}^{\mathcal{K}}$*.*

We say that a context class $\mathcal{K}$ *has a* non-binding concretization*, if there exists a context* $\mathbb{K} \neq [\cdot]$ *of class* $\mathcal{K}$ *s.t.* $CV(\mathbb{K}) = \emptyset$*.*

Besides syntactic equality there are two further notions of equivalence of expressions: one is pure α-equivalence, and the other one interprets environments like *multisets* of bindings.

Definition 3.5. *Let* $\sim_{let}$ *be the reflexive-transitive closure of permuting bindings in a* letrec*-environment, and with* $\sim_\alpha$ *(called* extended α-equivalence*) we denote the reflexive-transitive closure of combining* $\sim_{let}$ *and* α*-equivalence.*

Definition 3.6. Admissible *context-classes* $\mathcal{K}$ *must satisfy:*

1. *Class* $\mathcal{K}$ *is closed under* $\sim_\alpha$ *and* $\sim_{let}$*, i.e. if* $\mathbb{D} \in \mathcal{K}$ *and* $\mathbb{D}' \sim_\alpha \mathbb{D}$ *or* $\mathbb{D}' \sim_{let} \mathbb{D}$*, then* $\mathbb{D}' \in \mathcal{K}$*.*
2. *The context class* $\mathcal{K}$ *must contain the empty context.*
3. *There is a linear algorithm that checks for a given* LRSX*-context* d *and context class* $\mathcal{K}$*, whether* d *is a* $\mathcal{K}$*-context.*

Definition 3.7 (Semantics of unification variables)**.** *Unification variables represent ground expressions, environments, and contexts. The semantics of unification variables* X, Y *are all concrete variables of type* **Var***, expression variables* S *represent any ground expression of type* **Expr***, and the semantics of environment variables* E *are all ground environments of type* **Env***. The semantics of a context variable* $D^{\mathcal{K}}$ *are all contexts of context class* $\mathcal{K}$*. Semantically, the construct* $Ch^{\mathcal{K}}[x,s]$ *stands for* $x.\mathbb{D}^{\mathcal{K}}[s]$ *or chains*

$$x.\mathbb{D}_1^{\mathcal{K}}[(\mathtt{var}\ \mathsf{x}_1)]; \mathsf{x}_1.\mathbb{D}_2^{\mathcal{K}}[(\mathtt{var}\ \mathsf{x}_2)]; \dots; \mathsf{x}_n.\mathbb{D}_n^{\mathcal{K}}[s]$$

with fresh variables x_i*, and contexts* $\mathbb{D}_i^{\mathcal{K}}$ *from the context class* $\mathcal{K}$*. If the context class is Triv, then the notation* $\mathbb{D}^{\mathcal{K}}[s]$ *simply means* s*. A substitution* ρ *maps (a finite set of) unification variables to variables, expressions, environments, and contexts respecting their types. A substitution is* ground *iff it maps all variables of its domain to* LRS*-expressions. For substituting* $Ch^{\mathcal{K}}[y,s]$*, we write* $Ch^{\mathcal{K}}[y,s] \mapsto d[y,s]$*, and mean the substitution of* $Ch^{\mathcal{K}}$ *by* d*, where* $Ch^{\mathcal{K}}$ *and* d *are environment-contexts with two holes.*

Substitutions *for* $Ch^{\mathcal{K}}$ *must be of the form*

$$\{Ch^{\mathcal{K}} \mapsto [\cdot_1].D_1^{\mathcal{K}}[(\mathtt{var}\ x_1)]; x_1.D_2^{\mathcal{K}}[(\mathtt{var}\ x_2)]; \dots; x_n.D_n^{\mathcal{K}}[\cdot_2]\}.$$

Substitutions $\{D^{\mathcal{K}} \mapsto d\}$ *are only valid if all valid instances of* d *are contexts in* $\mathcal{K}$*.*

As a restriction on the problems and solutions, we will employ the following convention:

Definition 3.8. *An expression* s *satisfies the* weak distinct variable convention *(weak DVC) iff any let-bound variable does not occur twice as a binder in a single* letrec*-environment. An expression* s *satisfies the* (strong DVC) *iff the sets of bound and free variables are disjoint* and all *binders bind different variables.*

We use this definition for concrete variables as well as for unification-variables representing variables. Note that double occurrences of let-bound variables may occur during unification, e.g., the expressions letrec X.var Y, Y.var X in S_1 and letrec X'.var X', Y'.var Y' in S_2 both fulfill the weak DVC. Unifying the expressions leads to the substitution $\sigma = \{S_1 \mapsto S_2, X \mapsto X', Y \mapsto X'\}$. However, σ is not a solution, since it violates the weak distinct variable convention: in

$$\begin{array}{rl} & \sigma(\mathtt{letrec}\ X.\mathtt{var}\ Y, Y.\mathtt{var}\ X\ \mathtt{in}\ S_1) \\ = & \mathtt{letrec}\ X'.\mathtt{var}\ X', X'.\mathtt{var}\ X'\ \mathtt{in}\ S_2 \end{array}$$

the binder for let-variable X' occurs twice in the same environment.

4. THE UNIFICATION ALGORITHM

4.1 The Letrec Unification Problem

We motivate the proposal for our unification algorithm and start with simple examples.

Example 4.1. *Consider equating the expressions* $\lambda v_1.\lambda v_2.v_1$ *and* $\lambda w_1.\lambda w_2.w_2$ *which is written as* $\lambda X_1.\lambda X_2.(\mathtt{var}\ X_1) \doteq \lambda X_3.\lambda X_4.(\mathtt{var}\ X_4)$ *in our syntax. Unifying this naively leads to* $X_3 \doteq X_1, X_4 \doteq X_2, X_4 \doteq X_1$*, hence the instance of the input expressions is* $\lambda X_1.\lambda X_1.(\mathtt{var}\ X_1)$*. We may forbid such instances using so-called* non-capture constraints *of the form* (s,d)*, meaning that the bound and free variables of expression* s *are not captured by the hole of context* d*. Thus* *the above example together with the non-capture constraint* $(\mathtt{var}\ X_1, \lambda X_2.[\cdot])$ *has no solution, since* $X_2 = X_1$ *violates the constraint.*

As another example, consider $\lambda X_1.\mathtt{var}\ X_1 \doteq \lambda X_2.\mathtt{var}\ X_3$*. Our algorithm delivers a solution, which equates* X_1, X_2*, and* X_3*. However, by using non-capture constraints, one can ensure that the "free" occurrence of* X_3 *must be a free variable in any solution. For the input* $\lambda X_1.\mathtt{var}\ X_1 \doteq \lambda X_2.\mathtt{var}\ X_3$ *and constraint* $(\mathtt{var}\ X_3, \lambda X_2.[\cdot])$*, our unification algorithm instantiates the constraint to* $(\mathtt{var}\ X_1, \lambda X_1.[\cdot])$ *and then detects a failure.*

A third example, slightly modified and adapted (and renamed on one side) from [23], is the equation

$$\lambda X_1.\lambda X_2.(\mathtt{var}\ X_2, S_1, \mathtt{var}\ X_2) \doteq \lambda X_2.\lambda X_1.(S_1, S_2, \mathtt{var}\ X_1),$$

where no capture constraints are necessary. The solution w.r.t. our criterion of syntactic equivalence is $\mathtt{var}\ X_1 = \mathtt{var}\ X_2 = S_1 = S_2$*, whereas nominal unification [23] produces* $S_1 = \mathtt{var}\ X_1, S_2 = \mathtt{var}\ X_2$*, since nominal unification solves w.r.t.* α*-equivalence.*

We now formally define the unification problem:

Definition 4.2. *Let* Γ *be a finite set of* unification equations $s \doteq s'$ *where* s, s' *are* LRSX*-expressions that fulfill the weak DVC. Let* Δ_1 *be a finite set of context variables (called* non-empty context constraints*),* Δ_2 *be a finite set of environment variables* E *(called* non-empty environment constraints*),* Δ_3 *be a finite set of pairs* (t, d) *(called* non-capture constraints*, NCCs, for short) where* t *is an* LRSX*-expression and* d *is an* LRSX*-context. If* Γ *fulfills the following* occurrence restrictions*, then* $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$ *is called a* letrec unification problem*:*

1. *Every variable of type* S *occurs at most twice in* Γ*.*
2. *Every variable of kind* E*,* $Ch^{\mathcal{K}}$*,* $D^{\mathcal{K}}$ *occurs at most once in* Γ*.*
3. $Ch^{\mathcal{K}}$*-variables may occur only in one equation in* Γ *which must be of the form* $s \doteq d[\mathtt{letrec}\ Ch_1^{\mathcal{K}}[x_1, s_1]; \ldots; Ch_k^{\mathcal{K}}[x_k, s_k];\ env\ \mathtt{in}\ t]$ *s.t.* d, t, s, env, s_i *do not contain any* Ch*.*

A substitution ρ *(mapping a finite set of unification variables to variables, expressions, environments, and contexts, resp.) is called a* unifier *of* $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$ *iff:*

1. *For all* $(s \doteq t) \in \Gamma$*:* $s\rho \sim_{let} t\rho$ *and* $s\rho, t\rho$ *fulfill the weak DVC.*
2. *For all* $D \in \Delta_1$*:* $D\rho \neq [\cdot]$ *and for all* $E \in \Delta_2$*:* $E\rho \neq \emptyset$*.*
3. *For all NCCs* $(s, d) \in \Delta_3$*:* $(Var(s\rho) \cap CV(d\rho)) = \emptyset$*.*

A substitution ρ *is called a* solution *(or a* ground unifier*) for* P *if* ρ *is unifier and for all* $(s \doteq t) \in \Gamma$ *and all* $(s', d) \in \Delta_3$ *the expressions* $s\rho$*,* $t\rho$*, and* $d[s']\rho$ *are ground. The problem* P *is called* solvable *iff there exists a solution for* P*.*
A set M *of unifiers is called* complete *for* P*, if for every ground unifier* ρ *of* P*, there is some* $\rho_i \in M$ *and some ground substitution* σ*, s.t. for every unification variable* U *occurring in* P*:* $U\rho \sim_{let} U(\rho_i \circ \sigma)$*.* □

The following theorem shows that our letrec-unification problem is (at least) infinitary. The consequence is that computing a complete set of unifiers (as substitutions) is in general impossible.

Theorem 4.3. *There are letrec unification problems* P*, s.t. any complete set* M *of unifiers is infinite.*

Proof. The claim is that there are problems P, s.t. every complete set M of P consisting of unifiers is infinite. Let the equation P be $E_1; E_2 \doteq Ch^{\mathcal{K}}[y, (\mathtt{var}\ \mathsf{x})]$. Then for every n there is a ground solution ρ_n that expands $Ch^{\mathcal{K}}[y, (\mathtt{var}\ \mathsf{x})]$ to exactly n bindings (a chain), of the form

$$\mathsf{y}.(\mathtt{var}\ \mathsf{y}_1); \mathsf{y}_1.(\mathtt{var}\ \mathsf{y}_2); \ldots \mathsf{y}_{n-1}.(\mathtt{var}\ \mathsf{x}),$$

and E_1 is mapped to all bindings at odd position in the chain, while E_2 is mapped to all bindings at even position in the chain. The claim is that this series of solutions cannot be represented by finitely many unifiers:

Let θ be a unifier in the complete set, such that ρ_n is an instance. Since θ must be a correct solution, there are no solution components E in $E_i\theta$: Any E as a solution component (in the top environment) of $E_1\theta$ can be instantiated with a binding $y'.(\mathtt{var}\ \mathsf{x}')$ with a fresh variable x', but this cannot be in the instance, of $Ch^{\mathcal{K}}$, since the variable x' must occur in a let-binder, since instances of $Ch^{\mathcal{K}}$ are chains of bindings. The same holds for $E_2\theta$. Next assume that $Ch^{\mathcal{K}'}[y', s']$ is a solution component (in the top environment) of $E_1\theta$. Obviously s' must be of the form $(\mathtt{var}\ \mathsf{x}_2)$. In addition, ρ_n must be an instance of θ. This implies that $Ch^{\mathcal{K}'}[y', s']$ consists of only one binding of $E_i\theta$, since the bindings in $E_i\theta$ are not connected. Let n be the number of components E in $E_1\theta$ and m be the number of components $Ch^{\mathcal{K}}$ in $E_1\theta$. We have shown that θ can only have an instance ρ_i with at most $n + m$ bindings. A finite set $\theta_1, \ldots, \theta_m$ cannot cover the full series ρ_i, since this would imply an upper bound on the number of bindings in ρ_i. □

Since complete sets of unifiers may be infinite, our unification algorithm will not output such sets. Instead it will generate as output a (finite) complete set of (solvable) constraints, which consist of a substitution together with a set of equations of the simple form $E_1; \ldots; E_n \doteq Ch^{\mathcal{K}}[y, s]$. Note that in Theorem 5.5 we show that there is a (non-deterministic) polynomial decision algorithm for solvability of these constraints.

4.2 Structure of the Unification Algorithm

The unification algorithm UnifLRS is formulated as a set of non-deterministic rules on a data structure that consists of a set of equations $s_1 \doteq s_2$, and further constraints (see [3] for general notions of unification). The input of the algorithm consists of a letrec unification problem.

Definition 4.4. *The data structure for unification consists of several components:*

1. *Sol, a substitution consisting of different single mappings.*
2. Γ*, a set of (higher-order) expression-equations* $r \doteq r'$*, environment equations* $env \doteq env'$*, binding equations* $b \doteq b'$*, and variable equations* $x \doteq x'$*.*
3. $\Delta = (\Delta_1, \Delta_2, \Delta_3, \Delta_4)$*, where* Δ_1 *and* Δ_2 *are the non-emptiness constraints for context variables and environment variables, respectively, and* Δ_3 *are the non-capture constraints as described in Definition 4.2, and* Δ_4 *is a set of environment-equations* $env_1 \doteq env_2$ *that are seen as delayed equations.* □

For a letrec unification problem as input, $\Gamma, \Delta_1, \Delta_2$, and Δ_3 are instantiated accordingly, and the components Sol and Δ_4 are initialized as $\emptyset$.

The unification algorithm applies the unification rules (described in Section 4.4) non-deterministically until no more rule is applicable or a *FAIL* occurs. The output of the unification algorithm (on one non-deterministic run) is either *FAIL* or $(Sol, \emptyset, \Delta)$.

The unification rules are designed to respect the non-emptiness constraints in Δ_1 and Δ_2, and thus never instantiate a context variable in Δ_1 with the empty context, or an environment variable in Δ_2 with the empty environment. Furthermore, some rules require a context variable or an environment variable to be non-empty, and thus, if necessary, context or environment variables are sometimes non-deterministically guessed to be either empty or non-empty. The NCCs in Δ_3 are instantiated by the solution and their satisfiability is checked by the failure rules (see Definition 4.9 below). The generated delayed environment equations in Δ_4 are also instantiated by the solution and their satisfiability is checked once, by the final Δ_4-satisfiability check (Definition 4.6).

4.3 Prerequisites for Context Classes

Since we permit various context classes, we make a general formulation by requiring several non-deterministic (i.e. set-producing) subalgorithms that must come with the context classes as interfaces. They mainly specify two things: (i) how contexts d of a class $\mathcal{K}$ are split into $d = d_0[d_1]$ where d_0 is a flat context (i.e. the hole is in depth 1), and thus determine the possible positions of d_1 and the context class of d_1; and (ii) the common prefix-context d of two contexts d_1, d_2, i.e. for some d_1', d_2': $d_1 = d[d_1']$ and $d_2 = d[d_2']$. These are related to the rules (c2), (c3), and (c4) in Fig. 5.

Definition 4.5. *For the system $\overline{\mathcal{K}}$ of context classes, there are four non-deterministic subalgorithms, called $CCSv_{Df}$, $CCSv_{Dlet}$, $CCSv_{DDpref}$, and $CCSv_{DDfork}$, where we require that these are executable in constant time:*

1. *$CCSv_{Df}(\mathcal{K}_1, f)$ delivers the minimal set of possibilities $(i, \mathcal{K}_2)$ s.t. on any equation $D_1^{\mathcal{K}_1}[s] \doteq f\ r_1 \dots r_n$, non-deterministic branching with the substitutions*
$$\{D_1^{\mathcal{K}_1} \mapsto (f\ r_1 \dots r_{i-1}\ (x_1.\dots x_k.D_2^{\mathcal{K}_2})\ r_{i+1} \dots r_n)\}$$
and fresh variables x_i is an (n.d.) sound and complete operation

2. *$CCSv_{Dlet}(\mathcal{K}_1)$ delivers exactly the minimal set of possibilities of the form $(\texttt{in}, \mathcal{K}_2)$ and $(env, \mathcal{K}_3)$ such that on any equation of the form $D_1^{\mathcal{K}_1}[s] \doteq (\texttt{letrec}\ env\ \texttt{in}\ s')$ non-deterministic branching with substitutions*
$$\{D_1^{\mathcal{K}_1} \mapsto \texttt{letrec}\ env\ \texttt{in}\ D_2^{\mathcal{K}_2}\} \text{ and}$$
$$\{D_1^{\mathcal{K}_1} \mapsto \texttt{letrec}\ y.D_3^{\mathcal{K}_3}; E\ \texttt{in}\ s'\}$$
and fresh variables y and E is an (n.d.) sound and complete operation.

3. *$CCSv_{DDpref}(\mathcal{K}_1, \mathcal{K}_2)$ delivers the minimal set of possibilities $\mathcal{K}_3, \mathcal{K}_4$, s.t. for any equation $D_1^{\mathcal{K}_1}[s] \doteq D_2^{\mathcal{K}_2}[t]$ the substitution $\{D_1^{\mathcal{K}_1} \mapsto D_3^{\mathcal{K}_3}, D_2^{\mathcal{K}_2} \mapsto D_3^{\mathcal{K}_3} D_4^{\mathcal{K}_4}\}$ and replacing the equation by $s \doteq D_4^{\mathcal{K}_4}[t]$ is an (n.d.) sound and complete operation covering the solutions σ that make $D_1^{\mathcal{K}_1}\sigma$ a prefix of $D_2^{\mathcal{K}_2}\sigma$.*

4. *$CCSv_{DDfork}(\mathcal{K}_1, \mathcal{K}_2)$ delivers the minimal set of possibilities $(d, \mathcal{K}_3, \mathcal{K}_4, \mathcal{K}_5)$ such that for every equation of the form $D_1^{\mathcal{K}_1}[s] \doteq D_2^{\mathcal{K}_2}[t]$ branching with substitutions*
$$\{D_1^{\mathcal{K}_1} \mapsto D_3^{\mathcal{K}_3}[d[D_4^{\mathcal{K}_4}[\cdot], D_5^{\mathcal{K}_5}[t]]],$$
$$D_2^{\mathcal{K}_2} \mapsto D_3^{\mathcal{K}_3}[d[D_4^{\mathcal{K}_4}[s], D_5^{\mathcal{K}_5}[\cdot]]]\}$$
and removing the equation is an (n.d.) sound and complete operation covering the solutions σ such that neither $D_1^{\mathcal{K}_1}\sigma$ is a prefix of $D_2^{\mathcal{K}_2}\sigma$. nor vice versa. The two-hole context $d[\cdot_1, \cdot_2]$ for $f \in \mathcal{F}$ and $(k, l) \in \{(1,2), (2,1)\}$ is of one of the forms:
   - *$(f\ r_1 \dots r_{i-1}\ \ e_k\ r_{i+1} \dots r_{j-1}\ \ e_l\ r_{j+1} \dots r_{ar(f)})$, where r_h for $h \notin \{i,j\}$ is a higher-order expression of the form $x_{h,1}.\dots x_{h,n_h}.s_h$, and e_k, e_l are: $e_k = x_{k,1}.\dots x_{k,n_k}.[\cdot_k]$ and $e_l = x_{k,1}.\dots x_{i,n_l}.[\cdot_k]$, or*
   - *$(\texttt{letrec}\ y.[\cdot_k]; env'\ \texttt{in}\ [\cdot_l])$, or*
   - *$(\texttt{letrec}\ y_1.[\cdot_k];\ y_2.[\cdot_l]\ ; env'\ \texttt{in}\ s)$.*

4.4 The Unification Rules

We present the unification rules of UnifLRS. Rules are written as inference rules $\frac{P_0}{P_1 \mid \dots \mid P_n}$ with the meaning that for problem P_0 the algorithm branches into derived problems $P_1, \dots, P_n$. This non-determinism is disjunctive non-determinism, i.e. to find all unifiers all branches have to be inspected. We also write $\big|_{i \in I} P_i$ where $I = \{i_1, \dots, i_m\}$ is an index set, instead of $P_{i_1} \mid \dots \mid P_{i_m}$. The non-determinism between single rules in the set of inference rules is conjunctive non-determinism, i.e. the order of applying the rules is irrelevant for finding a unifier. We assume that all equations in Γ are symmetric and thus assume that rules mentioning an equation $s \doteq s'$ are also applicable to $s' \doteq s$. The instantiation of Δ by σ is defined as $\Delta\sigma = (\Delta_1, \Delta_2, \Delta_3\sigma, \Delta_4\sigma)$. Variables that appear in the problems $P_1, \dots, P_n$, but not in P_0, are always fresh ones.

The *first-order rules* are in Fig. 3. They include the usual unification rules to move solved unification variables into *Sol* and for decomposing syntactic constructs.

Rules for environment equations are in Fig. 4. They treat environment concatenation as associative-commutative, i.e. environments are treated as multisets of bindings. The rules act on an environment equation $env_1 \doteq env_2$. Equations with a binding on one side of the equation, i.e. $env_1 \doteq b; env_2$, are treated by rule (env-b) which guesses b in env_1: a binding b may be present in env_1, b may be part of an E-variable, or b may be one binding of a $Ch^{\mathcal{K}}$-chain (which leads to four subcases: $Ch^{\mathcal{K}}$ consists only of this binding, the binding is the prefix, a proper infix, or the suffix of the chain). The rules (env-E1), (env-E2), and (env-E3) treat equations where only E-variables play a role, the case $E \doteq env$ leads to a substitution provided the non-emptiness constraints are satisfied, or a new emptiness constraint has to be guessed. Environment equations without bindings and where one side contains several $Ch^{\mathcal{K}}$-variables and/or several E-variables are processed in one blow by rule (env-Ch), which splits into several equations with exactly one $Ch^{\mathcal{K}}$-variable, which are then moved to the constraint equations Δ_4, which guarantees a finite set of solutions, and which are checked by the Δ_4-satisfiability check Definition 4.6.

$$\textsf{(f1)}\ \frac{(Sol, \Gamma \dot\cup \{X \doteq x\}, \Delta)}{(Sol \circ \{X \mapsto x\}, \Gamma[x/X], \Delta[x/X])} \quad \textsf{(f3)}\ \frac{(Sol, \Gamma \dot\cup \{x.r \doteq x'.r'\}, \Delta)}{(Sol, \Gamma \dot\cup \{x \doteq x', r \doteq r'\}, \Delta)} \quad \textsf{(f5)}\ \frac{(Sol, \Gamma \dot\cup \{S \doteq s\}, \Delta)}{(Sol \circ \{S \mapsto s\}, \Gamma[s/S], \Delta[s/S])}\ \text{if } S \text{ is not a proper sub-expression of } s$$

$$\textsf{(f2)}\ \frac{(Sol, \Gamma \dot\cup \{\mathsf{x} \doteq \mathsf{x}\}, \Delta)}{(Sol, \Gamma, \Delta)} \quad \textsf{(f4)}\ \frac{(Sol, \Gamma \dot\cup \{\texttt{letrec}\ env_1\ \texttt{in}\ s_1 \doteq \texttt{letrec}\ env_2\ \texttt{in}\ s_2\}, \Delta)}{(Sol, \Gamma \dot\cup \{env_1 \doteq env_2, s_1 \doteq s_2\}, \Delta)} \quad \textsf{(f6)}\ \frac{(Sol, \Gamma \dot\cup \{f\ r_1 \ldots r_n \doteq f\ r'_1 \ldots r'_n\}, \Delta)}{(Sol, \Gamma \dot\cup \{r_1 \doteq r'_1, \ldots, r_n \doteq r'_n\}, \Delta)}$$

Figure 3: First-order rules

$$\textsf{(env-b)}\ \frac{(Sol, \Gamma \dot\cup \{b_1; env_1 \doteq env\}, \Delta)}{\begin{array}{l}
\Big|_{\forall b_2; env_2 = env} (Sol, \Gamma \dot\cup \{b_1 \doteq b_2; env_1 \doteq env_2\}, \Delta) \\
\mid \Big|_{\forall E; env_2 = env \wedge env_2 \neq \emptyset} (Sol \circ \{E \mapsto b_1; E'\}, \Gamma \dot\cup \{env_1 \doteq E'; env_2\}, \Delta[(b_1; E')/E]) \\
\mid \Big|_{\forall Ch^{\mathcal{K}}[y,s]; env_2 = env} (Sol \circ \{Ch^{\mathcal{K}}[y,s] \mapsto y.D^{\mathcal{K}}[s]\}, \Gamma \dot\cup \{b_1 \doteq y.D^{\mathcal{K}}[s], env_1 \doteq env_2\}, \Delta[y.D^{\mathcal{K}}[s]/Ch^{\mathcal{K}}[y,s]]\}) \\
\mid \Big|_{\forall Ch_1^{\mathcal{K}}[y,s]; env_2 = env} (Sol \circ \sigma, \Gamma \dot\cup \{b_1 \doteq y.D^{\mathcal{K}}[\texttt{var}\ Y], env_1 \doteq Ch_2^{\mathcal{K}}[Y,s]; env_2\}, \Delta\sigma) \text{ where } \sigma = \{Ch_1^{\mathcal{K}}[y,s] \mapsto y.D^{\mathcal{K}}[\texttt{var}\ Y]; Ch_2^{\mathcal{K}}[Y,s]\} \\
\mid \Big|_{\forall Ch^{\mathcal{K}}[y,s]; env_2 = env} (Sol \circ \sigma, \Gamma \dot\cup \{b_1 \doteq Y_1.D^{\mathcal{K}}[\texttt{var}\ Y_2], env_1 \doteq Ch_1^{\mathcal{K}}[y, \texttt{var}\ Y_1]; Ch_2^{\mathcal{K}}[Y_2, s]; env_2\}, \Delta\sigma) \\
\qquad\qquad \text{where } \sigma = \{Ch^{\mathcal{K}}[y,s] \mapsto Ch_1^{\mathcal{K}}[y, (\texttt{var}\ Y_1)]; Y_1.D^{\mathcal{K}}[\texttt{var}\ Y_2]; Ch_2^{\mathcal{K}}[Y_2, s]\} \\
\mid \Big|_{\forall Ch_1^{\mathcal{K}}[y,s]; env_2 = env} (Sol \circ \sigma, \Gamma \dot\cup \{b_1 \doteq Y_1.D^{\mathcal{K}}[s], env_1 \doteq Ch_2^{\mathcal{K}}[y, \texttt{var}\ Y_1]; env_2, \Delta\sigma\}) \text{ where } \sigma = \{Ch_1^{\mathcal{K}}[y,s] \mapsto Ch_2^{\mathcal{K}}[y, \texttt{var}\ Y_1]; Y_1.D^{\mathcal{K}}[s]\}
\end{array}}$$

where in all branches, for $\mathcal{K} = \mathit{Triv}$, the notation $D^{\mathcal{K}}[s]$ simply stands for s.

$$\textsf{(env-E1)}\ \frac{(Sol, \Gamma \dot\cup \{E \doteq env\}, \Delta)}{(Sol \circ \{E \mapsto env\}, \Gamma, \Delta[env/E])}$$

if $E \notin \Delta_2$ or if $E \in \Delta_2$ and env contains a binding, or a $Ch^{\mathcal{K}}$-variable, or an $E' \in \Delta_2$

$$\textsf{(env-E2)}\ \frac{(Sol, \Gamma \dot\cup \{E_1; \ldots; E_n \doteq \emptyset\}, \Delta)}{(Sol \circ \sigma, \Gamma, \Delta\sigma)} \text{ if } \forall i : E_i \notin \Delta_2$$

where $\sigma = \{E_1 \mapsto \emptyset, \ldots, E_n \mapsto \emptyset\}$

$$\textsf{(env-E3)}\ \frac{(Sol, \Gamma \dot\cup \{E \doteq E_1; \ldots; E_n\}, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))}{\Big|_{j \in \{1,\ldots,n\}} (Sol, \Gamma, \dot\cup \{E \doteq E_1; \ldots; E_n\}, (\Delta_1, \Delta_2 \cup \{E_j\}, \Delta_3, \Delta_4))} \begin{array}{l} E \in \Delta_2, \\ \forall i : E_i \notin \Delta_2 \end{array}$$

$$\textsf{(env-Ch)}\ \frac{(Sol, \Gamma \dot\cup \{E_1; \ldots; E_n \doteq Ch_1^{\mathcal{K}_1}[y_1, s_1]; \ldots; Ch_k^{\mathcal{K}_k}[y_k, s_k]; E'_1; \ldots; E'_m\}, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))}{\Big|_{\forall \Delta'_2} (Sol \circ \sigma, \Gamma, (\Delta_1, \Delta_2 \cup \Delta'_2, \Delta_3, \Delta_4 \cup \bigcup_{j \in \{1,\ldots,k\}} \{E_{1,j}; \ldots; E_{n,j} \doteq Ch_j^{\mathcal{K}_j}[y_j, s_j]\})\sigma)} \text{if } n > 1, k \geq 1 \text{ or } m, n > 1$$

where $\sigma = \sigma_1 \cup \sigma_2$, $\sigma_1 = \bigcup_{i=1}^{n} \{E_i \mapsto E_{i,1}; \ldots; E_{i,k+m}\}$, $\sigma_2 = \bigcup_{j=1}^{m} \{E'_j \mapsto E_{1,k+j}; \ldots; E_{n,k+j}\}$, Δ'_2 is a minimal subset of $\bigcup_{i,j} \{E_{i,j}\}$, s.t. $E_i \in \Delta_2$ implies $\exists h : E_{i,h} \in \Delta'_2$, $E'_j \in \Delta_2$ implies $\exists h : E_{h,k+j} \in \Delta'_2$, and $\forall 1 \leq j \leq k : \exists h : E_{h,j} \in \Delta'_2$

Figure 4: Rules for environments

$$\textsf{(c1)}\ \frac{(Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))}{(Sol, \Gamma, (\Delta_1 \cup \{D\}, \Delta_2, \Delta_3, \Delta_4)) \mid (Sol \circ \{D \mapsto [\cdot]\}, \Gamma[[\cdot]/D], (\Delta_1, \Delta_2, \Delta_3, \Delta_4)[[\cdot]/D])}$$

if D occurs at top in an equation in Γ, $D \notin \Delta_1$

$$\textsf{(c2)}\ \frac{(Sol, \Gamma \dot\cup \{D_1^{\mathcal{K}_1}[s] \doteq f\ r_1 \ldots r_n\}, \Delta)}{\Big|_{\forall (i, \mathcal{K}_2) \in CCSv_{Df}(\mathcal{K}_1, f)} (Sol \circ \sigma, \Gamma \dot\cup \{D_2^{\mathcal{K}_2}[s] \doteq s_i\}, \Delta\sigma)} \text{ if } D_1^{\mathcal{K}_1} \in \Delta_1$$

where $\sigma = \{D_1^{\mathcal{K}_1} \mapsto f\ r_1 \ldots r_{i-1}\ x_1 \ldots . x_k . D_2^{\mathcal{K}_2}\ r_{i+1} \ldots r_n\}$, $oar(f)(i) = k$, and $r_i = x_1 \ldots x_k . s_i$

$$\textsf{(c3)}\ \frac{(Sol, \Gamma \dot\cup \{D_1^{\mathcal{K}_1}[s] \doteq \texttt{letrec}\ env\ \texttt{in}\ s', \Delta)}{\begin{array}{l}
\Big|_{\forall (\texttt{in}, \mathcal{K}_2) \in CCSv_{Dlet}(\mathcal{K}_1)} (Sol \circ \sigma, \Gamma \dot\cup \{D_2^{\mathcal{K}_2}[s] \doteq s'\}, \Delta\sigma) \text{ where } \sigma = \{D_1^{\mathcal{K}_1} \mapsto \texttt{letrec}\ env\ \texttt{in}\ D_2^{\mathcal{K}_2}\} \\
\mid \Big|_{\forall (env, \mathcal{K}_2) \in CCSv_{Dlet}(\mathcal{K}_1)} (Sol \circ \sigma, \Gamma \dot\cup \{Y.D_2^{\mathcal{K}_2}[s]; E \doteq env\}, \Delta\sigma) \text{ where } \sigma = \{D_1^{\mathcal{K}_1} \mapsto \texttt{letrec}\ Y.D_2^{\mathcal{K}_2}; E\ \texttt{in}\ s'\}
\end{array}} \text{ if } D_1^{\mathcal{K}_1} \in \Delta_1$$

$$\textsf{(c4)}\ \frac{(Sol, \Gamma \dot\cup \{D_1^{\mathcal{K}_1}[s] \doteq D_2^{\mathcal{K}_2}[t]\}, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))}{\begin{array}{l}
\Big|_{\forall (\mathcal{K}_3, \mathcal{K}_4) \in CCSv_{DDpref}(\mathcal{K}_1, \mathcal{K}_2)} (Sol \circ \sigma, \Gamma \dot\cup \{s \doteq D_4^{\mathcal{K}_4}[t]\}, (\Delta_1 \cup \{D_3^{\mathcal{K}_3}\}, \Delta_2, \Delta_3, \Delta_4)\sigma) \text{ where } \sigma = \{D_1^{\mathcal{K}_1} \mapsto D_3^{\mathcal{K}_3}, D_2^{\mathcal{K}_2} \mapsto D_3^{\mathcal{K}_3} D_4^{\mathcal{K}_4}\} \\
\mid \Big|_{\forall (\mathcal{K}_3, \mathcal{K}_4) \in CCSv_{DDpref}(\mathcal{K}_2, \mathcal{K}_1)} (Sol \circ \sigma, \Gamma \dot\cup \{D_4^{\mathcal{K}_4}[s] \doteq t\}, (\Delta_1 \cup \{D_3^{\mathcal{K}_3}\}, \Delta_2, \Delta_3, \Delta_4)\sigma) \text{ where } \sigma = \{D_1^{\mathcal{K}_1} \mapsto D_3^{\mathcal{K}_3} D_4^{\mathcal{K}_4}, D_2^{\mathcal{K}_2} \mapsto D_3^{\mathcal{K}_3}\} \\
\mid \Big|_{\forall (d[\cdot_1, \cdot_2], \mathcal{K}_3, \mathcal{K}_4, \mathcal{K}_5) \in CCSv_{DDfork}(\mathcal{K}_1, \mathcal{K}_2)} (Sol \circ \sigma_d, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \Delta_4)\sigma_d) \text{ where } \sigma_d = \{D_1^{\mathcal{K}_1} \mapsto D_3^{\mathcal{K}_3}[d[D_4^{\mathcal{K}_4}[s], D_5^{\mathcal{K}_5}]], D_2^{\mathcal{K}_2} \mapsto D_3^{\mathcal{K}_3}[d[D_4^{\mathcal{K}_4}, D_5^{\mathcal{K}_5}[t]]]\}
\end{array}} \text{ if } D_1^{\mathcal{K}_1}, D_2^{\mathcal{K}_2} \in \Delta_1$$

Figure 5: Rules (context) for solving equations $D[s] \doteq t$

$$\mathbb{C} ::= [\cdot] \mid \lambda x.\mathbb{C} \mid (f\, r_1 \dots r_{i-1}\, (x_1 \dots x_k.\mathbb{C})\, r_{i+1} \dots r_{ar(f)}) \text{ where } oar(f)(i) = k \text{ and } f \in \mathcal{F}.$$
$$\mid (\texttt{letrec}\ env_1\ \texttt{in}\ \mathbb{C}) \mid (\texttt{letrec}\ x.\mathbb{C}; env\ \texttt{in}\ s) \qquad \text{where } env_1 \neq \emptyset$$
$$\mathbb{T} ::= [\cdot] \mid (\texttt{letrec}\ x.\mathbb{T}; env\ \texttt{in}\ s) \mid (\texttt{letrec}\ env'\ \texttt{in}\ \mathbb{T}) \mid (f\, r_1 \dots r_{i-1}\ \mathbb{T}\, r_{i+1} \dots r_{ar(f)}) \text{ where } oar(f)(i) = 0, f \in \mathcal{F}, env' \neq \emptyset.$$
$$\mathbb{A} ::= [\cdot] \mid (f\, r_1 \dots r_{i-1}\ \mathbb{A}\, r_{i+1}\ \dots r_{ar(f)}), \text{ where } i \in SPOS(f) \text{ and } f \in \mathcal{F}.$$

Figure 6: The context classes of our intended application: General contexts $\mathbb{C}$, top contexts $\mathbb{T}$, and application contexts $\mathbb{A}$ of context classes $\mathcal{C}$, $\mathcal{T}$, or $\mathcal{A}$, respectively.

The *rules for equations with context variables* are in Fig. 5. Either context variable $D^{\mathcal{K}}$ is guessed as empty or non-empty, and solving proceeds by guessing the top symbol of (the instance of) $D^{\mathcal{K}}$, or in case $D_1^{\mathcal{K}_1}[s] \doteq D_2^{\mathcal{K}_2}[t]$ either by guessing $D_1^{\mathcal{K}_1}$ as a prefix of $D_2^{\mathcal{K}_2}$ or by guessing the forking positions of the holes.

The *rule for satisfiability testing of Δ_4-constraints* is:

Definition 4.6 (Δ_4-satisfiability check). *This rule is applied only once to the final state ($\Gamma = \emptyset$): For $m \geq 1$ and $\Delta_4 = \bigcup_{j=1}^{m}\{E_{j,1}; \dots; E_{j,n_j} \doteq Ch_j^{\mathcal{K}}[y_j, s_j]\}$, guess a common instantiation σ for Δ_4 as follows: $Ch_j^{\mathcal{K}}[y_j, s_j]$ is instantiated by an environment $y_j.D_{j,1}^{\mathcal{K}}[Y_{j,1}]$; $Y_{j,1}.D_{j,2}^{\mathcal{K}}[Y_{j,2}]; \dots$; $Y_{j,k_j}.D_{j,k_j+1}^{\mathcal{K}}[s_j]$ where $k_j+1 \leq {M_{j,1}}^2 * (M_{j,2}+1) + M_{j,2}$ with $M_{j,1} = |\Delta_2 \cap \{E_{j,1}; \dots; E_{j,n_j}\}|$ and $M_{j,2} = n_j - M_{j,2}$ (see Theorem 5.5) and each $E_{j,i}$ is instantiated by a subset of $\{y_j.D_{j,1}^{\mathcal{K}}[Y_{j,1}], Y_{j,1}.D_{j,2}^{\mathcal{K}}[Y_{j,2}], \dots, Y_{j,k_j}.D_{j,k_j+1}^{\mathcal{K}}[s_j]\}$ s.t. every $E_{j,i} \in \Delta_2$ is instantiated with a non-empty subset and for all j: $(E_{j,1}; \dots; E_{j,n_j})\sigma \sim_{let} Ch_j^{\mathcal{K}}[y_j, s_j]\sigma$. The $D_{j,i}^{\mathcal{K}}$-variables are immediately eliminated, if $\mathcal{K} = Triv$. If σ satisfies the NCCs in Δ_3, then (Sol, Δ) is accepted, otherwise a FAIL occurs.*

We now provide an algorithm to test satisfiability of the NCCs in Δ_3.

Definition 4.7 (NCC-test). *Let Δ_1 be a set of non-empty context constraints, Δ_2 be a set of non-empty environment constraints, and Δ_3 be a set non-capture constraints.*

The NCC-test *fails iff there exists a pair $(s, d) \in \Delta_3$ s.t. $Var_M(s) \cap CV_M(d) \neq \emptyset$. Otherwise, the* NCC-test *is valid.*

The sets $Var_M(s)$ and $CV_M(d)$ consist of concrete expression variables and meta-variables for variables, expressions, contexts, and environments, and they are inductively defined in Fig. 7.

Let $\Delta_1, \Delta_2, \Delta_3$ be sets of non-empty context constraints, non-empty environment constraints and non-capture constraints. For a ground substitution ρ, we say that ρ satisfies Δ_3 w.r.t. Δ_1, Δ_2 iff $\rho(D) \neq \emptyset$ for all $D \in \Delta_1$, $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$, and $Var(\rho(s)) \cap CV(\rho(d)) = \emptyset$ for all $(s, d) \in \Delta_3$

Proposition 4.8. *Let $\Delta_1, \Delta_2, \Delta_3$ be sets of non-empty context constraints, non-empty environment constraints, and non-capture constraints. The NCC-test is valid iff there exists a ground substitution ρ that satisfies Δ_3 w.r.t. Δ_1, Δ_2.*

Proof. We distinguish the cases whether the NCC-test is valid or fails.

First, consider the case that the NCC-test fails. Then there exists some $(s, d) \in \Delta_3$ s.t. there exists some $U \in Var_M(s) \cap CV_M(d)$. If $U = \mathsf{x}$ for some concrete variable x, then clearly for any ground instantiation ρ we have $\mathsf{x} \in Var(\rho(s)) \cap CV(\rho(d))$. If $U = X$, then any ground instantiation ρ must instantiate X with some concrete variable x, i.e. $\rho(X) = \mathsf{x}$. Clearly, we also have $\mathsf{x} \in Var(\rho(s)) \cap CV(\rho(d))$ in this case. If $U = E$ with $E \in \Delta_2$, then any ground instantiation ρ must instantiate E by a non-empty environment $\mathsf{x}_1 = s_1, \dots \mathsf{x}_n = s_n$ and clearly $\mathsf{x}_1 \in Var(\rho(s))$ and $\mathsf{x}_1 \in CV(\rho(s))$. If $U = D^{\mathcal{K}}$ where $D^{\mathcal{K}} \in \Delta_1$ and $\mathcal{K}$ does not have a non-binding concretization, then any ground instantiation must instantiate $D^{\mathcal{K}}$ with a concrete context $\mathbb{K}$ of class $\mathcal{K}$ s.t. there exists some variable $\mathsf{x} \in CV(\mathbb{K})$. Clearly, we have $\mathsf{x} \in Var(\rho(s)) \cap CV(\rho(d))$ in this case.

Now consider the case that NCC-test is valid. Then the following ground subsitution ρ satisfies Δ_3 w.r.t. Δ_1, Δ_2: Let UV be the unification variables occuring in Δ_3.

- For all $X \in UV$: $\rho(X) = \mathsf{x}_X$ for a fresh variable x_X;
- For all $D^{\mathcal{K}} \in UV$: if $D^{\mathcal{K}} \notin \Delta_1$ then $\rho(D^{\mathcal{K}}) = [\cdot]$, and $\mathcal{K}$ has a non-binding concretization, then $\rho(D^{\mathcal{K}}) = \mathbb{K}$ for a non-binding context $\mathbb{K}$ of class $\mathcal{K}$ if $D^{\mathcal{K}} \in \Delta_1$ and $\rho(D^{\mathcal{K}})$ is context of class $\mathcal{K}$ otherwise.
- For all $E \in UV$: if $E \notin \Delta_2$ then $\rho(E) = \emptyset$ iff $E \notin \Delta_2$, and otherwise $\rho(E) = \mathsf{x}_E.\texttt{var}\ \mathsf{x}_E$ for a fresh variable x_X.
- For all $Ch^{\mathcal{K}} \in UV$: $\rho(Ch^{\mathcal{K}}) = [\cdot_1].[\cdot_2]$.
- $\rho(S) = c$ for some constant c (if available), or $\rho(S) = \lambda \mathsf{x}_S.\mathsf{x}_S$ for some fresh variable x_S otherwise.

□

We finally define the other failure rules of the algorithm:

Definition 4.9 (Failure rules). *Let the input problem of* UnifLRS *be $(\Gamma_{inp}, \Delta_{inp,1}, \Delta_{inp,2}, \Delta_{inp,3})$ and let (Sol, Γ, Δ) be the current state of* UnifLRS. *The unification algorithm delivers a FAIL, if one of the following conditions holds:*

1. *$(\mathsf{x}_1 \doteq \mathsf{x}_2) \in \Gamma$ with $\mathsf{x}_1 \neq \mathsf{x}_2$; or $(s_1 \doteq s_2) \in \Gamma$ and s_1, s_2 have different top symbols $f_1 \neq f_2 \in (\mathcal{F} \cup \{\texttt{letrec}\})$.*
2. *$(S \doteq s) \in \Gamma$ and S is a proper subterm of s, and the top symbol of s is not a context variable.*
3. *$(S \doteq D[s]) \in \Gamma$, $D \in \Delta_1$, and S is a proper subterm of s.*
4. *There is an equation $D[s] \doteq s'$ in Γ that is in the scope of the rules (c2) – (c4), but the algorithms in Definition 4.5 deliver empty sets of possibilities.*
5. *$(env \doteq \emptyset) \in \Gamma$, and $env = b; env'$, $env = Ch^{\mathcal{K}}[y, s]; env'$, or $env = E; env'$ with $E \in \Delta_2$.*
6. *There is an equation $(env_1 \doteq Ch_1^{\mathcal{K}}[y_1, s_1]; env_2)$ or an equation or $(env_1 \doteq y_1.s_1; env_2)$ in Γ and env_2 contains a second let-binder for y_1, i.e. $env_2 = Ch_2^{\mathcal{K}}[y_1, s_2]; env_2'$ or $env_2 = y_1.s_2; env_2'$.*
7. *The NCC-test (see Definition 4.7) fails.*
8. *There is an equation $(s \doteq s') \in \Gamma_{inp}$, s.t. the weak DVC is violated for $s\sigma$, $s'\sigma$ where $\sigma = Sol$.*

$$\begin{array}{lcl}
\mathit{Var}_M(x) & = & \{x\} \\
\mathit{Var}_M(x_1.\ldots.x_n.s) & = & \{x_1,\ldots,x_n\} \cup \mathit{Var}_M(s) \\
\mathit{Var}_M(f\ s_1 \ldots s_n) & = & \bigcup_i \mathit{Var}_M(s_i) \\
\mathit{Var}_M(\texttt{letrec}\ \mathit{env}\ \texttt{in}\ s) & = & \mathit{Var}_M(\mathit{env}) \cup \mathit{Var}_M(s) \\
\mathit{Var}_M(S) & = & \{S\} \\
\mathit{Var}_M(D^{\mathcal{K}}[s]) & = & \mathit{Var}_M(s) \text{, if } \mathcal{K} \text{ has a non-binding concretization, or } D^{\mathcal{K}} \notin \Delta_1 \\
\mathit{Var}_M(D^{\mathcal{K}}[s]) & = & \{D^{\mathcal{K}}\} \cup \mathit{Var}_M(s) \text{, otherwise} \\
\mathit{Var}_M(\mathit{env}) & = & \bigcup \{\{z\} \cup \mathit{Var}_M(s) \mid \mathit{Ch}^{\mathcal{K}}[z,s]; \mathit{env}' = \mathit{env}\} \\
 & & \cup \{E \mid E; \mathit{env}' = \mathit{env} \wedge E \in \Delta_2\} \\
 & & \cup \bigcup \{\{\{z\} \cup \mathit{Var}_M(s)\} \mid z.s; \mathit{env}' = \mathit{env}\} \\
\mathit{CV}_M(x) & = & \emptyset \\
\mathit{CV}_M(x_1.\ldots.x_n.d) & = & \{x_1,\ldots,x_n\} \cup \mathit{CV}(d) \\
\mathit{CV}_M(f\ s_1 \ldots s_{i-1}\ d\ s_{i+1} \ldots s_n) & = & \mathit{CV}_M(d) \\
\mathit{CV}_M(\texttt{letrec}\ \mathit{env}\ \texttt{in}\ d) & = & \bigcup \{\{z\} \mid \mathit{Ch}^{\mathcal{K}}[z,s]; \mathit{env}' = \mathit{env}\} \\
 & & \cup \{E \mid E; \mathit{env}' = \mathit{env} \wedge E \in \Delta_2\} \\
 & & \cup \{z \mid z.s; \mathit{env}' = \mathit{env}\} \\
 & & \cup \mathit{CV}_M(d) \\
\mathit{CV}_M(\texttt{letrec}\ z.d; \mathit{env}\ \texttt{in}\ s) & = & \mathit{CV}_M(\mathit{env}) \cup \{z\} \cup \mathit{CV}_M(d) \\
\mathit{CV}_M(\texttt{letrec}\ \mathit{Ch}^{\mathcal{K}}[z,d]; \mathit{env}\ \texttt{in}\ s) & = & \mathit{CV}_M(\mathit{env}) \cup \{z\} \cup \mathit{CV}_M(d) \\
\mathit{CV}_M(S) & = & \emptyset \\
\mathit{CV}_M(D^{\mathcal{K}}) & = & \emptyset \text{, if } \mathcal{K} \text{ has a non-binding concretization, or } D^{\mathcal{K}} \notin \Delta_1 \\
\mathit{CV}_M(D^{\mathcal{K}}) & = & \{D^{\mathcal{K}}\} \text{, otherwise} \\
\mathit{CV}_M(D^{\mathcal{K}}[d]) & = & \mathit{CV}_M(D^{\mathcal{K}}) \cup \mathit{CV}_M(d) \\
\mathit{CV}_M([\cdot]) & = & \emptyset \\
\mathit{CV}_M(\mathit{env}) & = & \bigcup \{\{z\} \mid \mathit{Ch}^{\mathcal{K}}[z,s]; \mathit{env}' = \mathit{env}\} \\
 & & \cup \{E \mid E; \mathit{env}' = \mathit{env} \wedge E \in \Delta_2\} \\
 & & \cup \{z \mid z.s; \mathit{env}' = \mathit{env}\}
\end{array}$$

Figure 7: Definition of Var_M and CV_M

4.5 Specialization to a Call-by-Need Calculus

For the call-by-need calculi L_{need} (see Fig. 1) and LR [20], we can apply the unification algorithm UNIFLRS, after specifying the context classes and the algorithms according to Definition 4.5, which is possible including the efficiency requirements. This is an instantiation of the language `LRSX` that supports our intended application, where there are only four context classes: $\mathcal{C}$, $\mathcal{T}$, $\mathcal{A}$, and $\emptyset$ (see Fig. 6). The notation for context variables in this specialization are C, T, A standing for contexts of (general) context class $\mathcal{C}$, top-contexts of context class $\mathcal{T}$, and application contexts of context class $\mathcal{A}$, respectively. The application to L_{need} requires chains $\mathit{Ch}^{\mathcal{A}}[x,s]$ in the representation of reduction contexts e.g. in the LR calculus or in L_{need} (see Fig. 1), which are covered by A or `letrec` E `in` A, or `letrec` $E; \mathit{Ch}^{\mathcal{A}}[x,s]$ `in` $A[x]$.

Some reduction rules also require chains $\mathit{Ch}^{\mathcal{A}}[x,s]$ and the reduction rules (no,cp-in) and (no,cp-e) require variable-to-variable chains $\mathit{Ch}^{\mathit{Triv}}[x,s]$.

Definition 4.5 for the context classes $\mathcal{C}, \mathcal{T}, \mathcal{A}$ in L_{need} requires explanation: The strict positions $\mathit{SPOS}(.)$ are the first argument for `app`, (and of `case`, `seq`, and $\emptyset$ for data constructors in the LR-calculus). The decomposition algorithms for context variables are easy to specify exploiting $\mathcal{A} < \mathcal{T} < \mathcal{C}$ and the grammars in Fig. 6. For example, decomposing $D_1^{\mathcal{A}}[s] \doteq D_2^{\mathcal{T}}[s']$, may give rise to two possibilities in the prefix case: $D_2^{\mathcal{T}} \mapsto D_1^{\mathcal{A}} D_3^{\mathcal{T}}$, and $s \doteq D_3^{\mathcal{T}}[s']$; or $D_2^{\mathcal{T}} \mapsto D_3^{\mathcal{A}}; D_2^{\mathcal{A}} \mapsto D_3^{\mathcal{A}}; D_4^{\mathcal{A}}$, and $D_4^{\mathcal{A}}[s] \doteq s'$. In the forking case, the common prefix will be of context class $\mathcal{A}$. Similar for the other cases.

Example 4.10. *As an example of executing the unification algorithm, we consider an instance of overlapping the left hand side of an (no,cp-e)-reduction with the right hand side of an (llet-in)-transformation in the calculus L_{need}.*

Let $\mathcal{A}$ be the context class corresponding to A-contexts in the calculus L_{need} (see Fig. 1), where we write the variables $D^{\mathcal{A}}$ using the letter A. We want to unify the expressions `letrec` $X_1.(\lambda X.S); \mathit{Ch}^{\mathcal{A}}[Y, A_1[$`var` $X_1]]$ `in` $A[$`var` $Y])$ *and* `letrec` $E_1; E_2$ `in` S_1 *where the following constraints must hold: $\Delta_1 = \{A_1\}$ (since otherwise, the normal order reduction would copy to a target different from $A_1[$`var` $X_1])$, $\Delta_2 = \{E_1, E_2\}$ (since there are no empty* `letrec`*-environments in L_{need}), and the non-capture constraint that ensures that the binders of E_2 do not capture the variables of E_1, that is $\Delta_4 = \{($*`letrec` E_1 `in` c, `letrec` E_2 `in` $[\cdot])\}$ *where c is a constant. We abbreviate the NCC by $(E_1,$* `letrec` E_2 `in` $[\cdot])$*, since this does not change its meaning.*

In Fig. 8 the execution of UNIFLRS *is shown. The first step applies the first-order rule* (f4) *to decompose the* `letrec`*-expression, the second step moves the equation for variable S_1 into the solution (by rule* (f5)*). Now the remaining equation in Γ is $X_1.(\lambda X.S); \mathit{Ch}^{\mathcal{A}}[Y, A_1[$*`var` $X_1]] \doteq E_1; E_2$ *which makes rule* (env-b) *applicable. Here the unification algorithm branches, since the binding $X_1.(\lambda X.S)$ may be part of the environment E_1 or it may be a part of the environment E_2. Let us first consider the left branch: The remaining equation in Γ is $\mathit{Ch}^{\mathcal{A}}[Y, A_1[$*`var` $X_1]] \doteq E_1'; E_2$ *and thus rule* (env-Ch) *has to be applied. The equation is moved into Δ_4, where the environment variables E_1' and E_2 are replaced by fresh ones and there is only one minimal set Δ_2' (see rule*

$$\begin{array}{ll}
\mathit{Sol} & : \emptyset \\
\Gamma & : \{\mathtt{letrec}\ X_1.(\lambda X.S);\ \mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]]\ \mathtt{in}\ A[\mathtt{var}\ Y]) \doteq \mathtt{letrec}\ E_1; E_2\ \mathtt{in}\ S_1\} \\
\Delta & : (\Delta_1, \Delta_2, \Delta_3, \Delta_4) = (\{A_1\}, \{E_1, E_2\}, \{(E_1, \mathtt{letrec}\ E_2\ \mathtt{in}\ [\cdot])\}, \emptyset) \\
\hline
\multicolumn{2}{l}{\text{(f4)}} \\
\mathit{Sol} & : \emptyset \\
\Gamma & : \{X_1.(\lambda X.S);\ \mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_1; E_2,\quad A[\mathtt{var}\ Y] \doteq S_1\} \\
\Delta & : (\Delta_1, \Delta_2, \Delta_3, \Delta_4) = (\{A_1\}, \{E_1, E_2\}, \{(E_1, \mathtt{letrec}\ E_2\ \mathtt{in}\ [\cdot])\}, \emptyset) \\
\hline
\multicolumn{2}{l}{\text{(f5)}} \\
\mathit{Sol} & : \{S_1 \mapsto A[\mathtt{var}\ Y]]\} \\
\Gamma & : \{X_1.(\lambda X.S);\ \mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_1; E_2\} \\
\Delta & : (\Delta_1, \Delta_2, \Delta_3, \Delta_4) = (\{A_1\}, \{E_1, E_2\}, \{(E_1, \mathtt{letrec}\ E_2\ \mathtt{in}\ [\cdot])\}, \emptyset) \\
\hline
\multicolumn{2}{l}{\text{(env-b)}}
\end{array}$$

$$\begin{array}{ll|ll}
\mathit{Sol} & : \{S_1 \mapsto A[\mathtt{var}\ Y]], E_1 \mapsto X_1.(\lambda X.S); E_1'\} & \mathit{Sol} & : \{S_1 \mapsto A[\mathtt{var}\ Y]]E_2 \mapsto X_1.(\lambda X.S); E_2'\} \\
\Gamma & : \{\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_1'; E_2\} & \Gamma & : \{\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_1; E_2'\} \\
\Delta_1 & : \{A_1\} & \Delta_1 & : \{A_1\} \\
\Delta_2 & : \{E_1, E_2\} & \Delta_2 & : \{E_1, E_2\} \\
\Delta_3 & : \{(X_1.(\lambda X.S); E_1', \mathtt{letrec}\ E_2\ \mathtt{in}\ [\cdot])\} & \Delta_3 & : \{(E_1, \mathtt{letrec}\ X_1.(\lambda X.S); E_2'\ \mathtt{in}\ [\cdot])\} \\
\Delta_4 & : \emptyset & \Delta_4 & : \emptyset \\
\hline
\multicolumn{2}{l|}{\text{(env-Ch)}} & \multicolumn{2}{l}{\text{(env-Ch)}} \\
\mathit{Sol} & : \{S_1 \mapsto A[\mathtt{var}\ Y]], E_1 \mapsto X_1.(\lambda X.S); E_{1,1}', & \mathit{Sol} & : \{S_1 \mapsto A[\mathtt{var}\ Y]], E_2 \mapsto X_1.(\lambda X.S); E_2', \\
 & \quad E_1' \mapsto E_{1,1}', E_2 \mapsto E_{2,1}\} & & \quad E_1 \mapsto E_{1,1}, E_2' \mapsto E_{2,1}'\} \\
\Gamma & : \emptyset & \Gamma & : \emptyset \\
\Delta_1 & : \{A_1\} & \Delta_1 & : \{A_1\} \\
\Delta_2 & : \{E_1, E_2, E_{2,1}\} & \Delta_2 & : \{E_1, E_2, E_{1,1}\} \\
\Delta_3 & : \{(X_1.(\lambda X.S); E_{1,1}', \mathtt{letrec}\ E_{2,1}\ \mathtt{in}\ [\cdot])\} & \Delta_3 & : \{(E_{1,1}, \mathtt{letrec}\ X_1.(\lambda X.S); E_{2,1}'\ \mathtt{in}\ [\cdot])\} \\
\Delta_4 & : \{\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_{1,1}'; E_{2,1}\} & \Delta_4 & : \{\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_{1,1}; E_{2,1}'\} \\
\cline{3-4}
 & & \multicolumn{2}{l}{\text{(Def. 4.6)}} \\
 & & \multicolumn{2}{c}{\mathit{FAIL}}
\end{array}$$

Figure 8: Exemplary execution of UnifLRS (see Example 4.10)

(env-Ch)*) which is $\Delta_2' = \{E_{2,1}\}$. Thereafter, Γ is empty and no failure is signaled by the failure rules. The final Δ_4-satisfiability check (Definition 4.6) finds the following solution of the equation in Δ_4: $\sigma = \{\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \mapsto Y.A'[A_1[\mathtt{var}\ X_1]], E_{2,1} \mapsto Y.A'[A_1[\mathtt{var}\ X_1]], E_{1,1}' \mapsto \emptyset\}$, and the algorithm delivers the output $(\mathit{Sol}, \emptyset, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))$ with*

$$\begin{aligned}
\mathit{Sol} &= \{S_1 \mapsto A[\mathtt{var}\ Y]], E_1 \mapsto X_1.(\lambda X.S); E_{1,1}', \\
&\quad E_1' \mapsto E_{1,1}', E_2 \mapsto E_{2,1}\} \\
\Delta_1 &= \{\, A_1\} \\
\Delta_2 &= \{\, E_1, E_2, E_{2,1}\} \\
\Delta_3 &= \{\, (X_1.(\lambda X.S); E_{1,1}', \mathtt{letrec}\ E_{2,1}\ \mathtt{in}\ [\cdot])\} \\
\Delta_4 &= \{\, \mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]] \doteq E_{1,1}'; E_{2,1}\}
\end{aligned}$$

Now let us consider the right branch. Again rule (env-Ch) *is applied, where E_1 and E_2' are replaced by fresh environment variables, and there is only one minimal set $\Delta_2' = \{E_{1,1}\}$. However, the Δ_4-satisfiability check (Definition 4.6) detects that the Δ_4-constraint is unsolvable: The algorithm has to check chains of length up to 4. For length 1, the only possible assignment is $\sigma = \{E_{1,1} \mapsto Y.A'[A_1[\mathtt{var}\ X_1]], E_{2,1}' \mapsto \emptyset\}$. Since $\Delta_3\sigma = \{(Y.A'[A_1[\mathtt{var}\ X_1], \mathtt{letrec}\ X_1.(\lambda X.S)\ \mathtt{in}\ [\cdot])\}$, variable X_1 is captured in the hole, and the NCC is violated. For longer chains, all tests again violate the NCC of Δ_3 which can be seen by the following argument: Suppose that σ instantiates $\mathit{Ch}^{\mathcal{A}}$ by more than one binding, i.e. $\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]]\sigma = Y.A_1'[Y_1']; Y_1'.A_2'[Y_2']; \ldots; Y_{k-1}'.[A_k'[A_1\mathtt{var}\ X_1]]$. Since $X_1 \in CV((\mathtt{letrec}\ X_1.(\lambda X.S); E_{2,1}'\ \mathtt{in}\ [\cdot])\sigma)$ holds, the last binding $Y_{k-1}'.A_k'[A_1[\mathtt{var}\ X_1]]$ must be part of $E_{2,1}'\sigma$. However, as a consequence $Y_{k-1}' \in CV(\ (\mathtt{letrec}\ X_1.(\lambda X.S);\ E_{2,1}'\mathtt{in}\ [\cdot])\sigma)$ and thus the binding for Y_{k-2}' must be part of $E_{2,1}'\sigma$. This can be iterated which shows that no binding of instantiating $\mathit{Ch}^{\mathcal{A}}[Y, A_1[\mathtt{var}\ X_1]]\sigma$ can be part of $E_{1,1}\sigma$, which violates the non-emptiness constraint $E_{1,1} \in \Delta_2$.*

5. CORRECTNESS

We show that UNIFLRS is terminating, sound, complete, and hence a decision procedure. Let us define appropriate notions of soundness and completeness.

Definition 5.1. *Let $P = (\Gamma_P, \Delta_{P,1},\ \Delta_{P,2}, \Delta_{P,3})$ be a letrec unification problem, and $\mathcal{S} = (\mathit{Sol}, \emptyset,\ (\Delta_1, \Delta_2, \Delta_3, \Delta_4))$ be an intermediate* UNIFLRS*-state that passes the failure tests of Definition 4.9.A ground substitution σ that is an instance of Sol is a* solution of $\mathcal{S}$*, if $s\sigma \sim_{let} s'\sigma$ for all equations $(s \doteq s') \in \Gamma_P$, $D\sigma \neq [\cdot]$ for all $D \in \Delta_1$, $E\sigma \neq \emptyset$ for all $E \in \Delta_2$, and for all NCCs $(t, d) \in \Delta_3$: the constraint $(t\sigma, d\sigma)$ holds, for all $(s \doteq s') \in \Gamma_P$: $s\sigma$, $s'\sigma$ satisfy the weak DVC, and for all equations $env_1 \doteq env_2 \in \Delta_4$: $env_1\sigma \sim_{let} env_2\sigma$.*

Definition 5.2. *Let $P = (\Gamma_P, \Delta_{P,1},\ \Delta_{P,2}, \Delta_{P,3})$ be a letrec unification problem. We define the (perhaps infinite) set of* output-unifiers $\Omega(P)$ *as follows: $\sigma \in \Omega(P)$ iff there is an execution of* UNIFLRS *that results in an accepted result $\mathcal{S} = (\mathit{Sol}, \emptyset, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))$ (in particular, also the Δ_4-satisfiability check (Definition 4.6) accepts $\mathcal{S}$) and σ solves $\mathcal{S}$.*

We will show for every letrec-unification problem P, that i) $\rho \in \Omega(P)$ implies that ρ is a solution of P (soundness), and ii) if ρ is a solution of P then $\rho \in \Omega(P)$ (completeness).

Lemma 5.3.

1. *The following occurrence restrictions are an invariant of the unification algorithm: Every variable S of type* **Expr** *occurs at most twice in Γ; every variable of kind*

$E, Ch^{\mathcal{K}}, D^{\mathcal{K}}$ occurs at most once in Γ; and the $Ch^{\mathcal{K}}$-variables occur in at most one expression equation of the form $s \doteq d[\texttt{letrec}\ Ch_1^{\mathcal{K}_1}[y_1, s_1]; \ldots; Ch_m^{\mathcal{K}_m}[y_m, s_m]; env\ \texttt{in}\ t]$ where d, t, s, env, and all s_i do not contain $Ch^{\mathcal{K}}$-variables, and $Ch^{\mathcal{K}}$-variables in environment equations occur only on one side of the equation in the top-level environment.

2. *Every instantiation of E and of D by a substitution, triggered by a rule, makes E and D, respectively, be non-empty by explicitly instantiating or by inheriting the non-emptiness constraints. Hence Δ_1, Δ_2 need not be instantiated by the substitution.*

Proof. The occurrence restrictions hold due to the occurrence restrictions for the input (Definition 4.2) and by scanning the rules, and checking where the $Ch^{\mathcal{K}}$-variables may occur. □

Theorem 5.4. *The unification rules terminate in (non-deterministic) polynomial time.*

Proof. We show that the well-founded lexicographic measure $\mu = (\mu_1, \mu_2, \mu_3, \mu_4, \mu_5)$ is strictly decreased by every rule application, where μ_1 is the number of `letrec`-(sub)-expressions in Γ; μ_2 is the number of bindings in environment equations in Γ; μ_3 is the size of Γ; μ_4 is the number of context variables occurring in Γ that are not in Δ_1; μ_5 is the number of E-variables occurring in Γ that are not in Δ_2. The first-order rules strictly decrease μ_3 and do not increase μ_1, μ_2. The substitution $[s/S]$ does not increase the size, since S occurs at most twice in Γ. The rule (env-b) does not change μ_1 but strictly decreases μ_2. The rules (env-E1), (env-E2), and (env-Ch) do not increase μ_1, μ_2, but strictly decrease μ_3, the rule (env-E3) strictly decreases μ_5 by leaving all other μ_i unchanged. The rule (c1) does not increase μ_1, μ_2, μ_3 and strictly decreases μ_4. The rules (c2), (c4) strictly decrease μ_3 (by not increasing μ_1, μ_2) and rule (c3) strictly decreases μ_1. Due to the occurrence restrictions on unification variables, there is no proper duplication that can generate exponentially many symbols. Thus the intermediate growth of the data structure is polynomial in the input. □

We prove several properties of constraint equations and their solvability, which will lead to a bound on the length of to-be guessed chains in the Δ_4-satisfiability check (Definition 4.6).

Now we prove that our final constraints have a decision algorithm for solvability, independent of the context class, where the complexity is also independent of the context class.

Theorem 5.5. *If there is a solution σ of the current state (Sol, Γ, Δ) and an equation $E_1; \ldots; E_n \doteq Ch^{\mathcal{K}}[y, s] \in \Delta_4$ with $n > 1$, $n_1 := |\Delta_2 \cap \{E_1, \ldots, E_n\}|$, and $n_2 := n - n_1$, then there is also a solution σ' with $Z\sigma' = Z\sigma$ for all $Z \in UV(\Gamma) \setminus \{E_1, \ldots, E_n, Ch^{\mathcal{K}}[.,.]\}$, and that expands $Ch^{\mathcal{K}}[.,.]$ to at most $n_1^2 * (n_2 + 1) + n_2$ bindings.*

Proof. Assume that there is a solution σ of (Γ, Δ) that instantiates $Ch^{\mathcal{K}}[y, s]$ by a partial environment with m bindings where m is minimal. W.l.o.g, we can assume that $Ch^{\mathcal{K}}[y, s]\sigma = b_1; \ldots; b_m$ where $b_i = y_i.\mathbb{K}_i[\texttt{var}\ y_{i+1}]$ for $i = 1, \ldots, m-1$ and $b_m = y_m.\mathbb{K}_m[s_1]$, $y_1 = y\sigma, s_1 = s\sigma$, and y_i for $i \geq 2$ are fresh variables, and where $\mathbb{K}$ are ground contexts of context class $\mathcal{K}$. Note that the assumption that σ is a solution implies that the variables y_{i+1} are not bound in the hole of $\mathbb{K}_i[.]$. W.l.o.g. $E_k\sigma \subseteq \bigcup_{i=1}^m \{b_i\}$, s.t. every b_i occurs in exactly one $E_k\sigma$. First we describe the operations that may shorten the chain of bindings, if $m \geq 3$. We will show that the NCCs are not violated by the operations. In the moment we do not take care of nonemptyness of $E_k\sigma$, which will be done later in the proof.

1. If there are two indices $i < j$ with $j - i \geq 2$, and an index k_0, s.t. b_i, b_j are bindings in (the same) $E_{k_0}\sigma$, then let σ' be the modification of σ by removing the bindings i' with $i < i' < j$ as follows: $Ch^{\mathcal{K}}[y, s]\sigma' := b_1; \ldots; b_i'; b_j; b_{j+1}; \ldots; b_m$ where $b_i' := y_i.\mathbb{K}_i'[\texttt{var}\ y_j]$, s.t. $\mathbb{K}_i'$ is a renamed version of $\mathbb{K}_i$ where all bound variables are renamed by fresh variables; and where for all k: $E_k\sigma'$ is adapted from $E_k\sigma$ by eliminating the bindings $b_{i+1}, \ldots, b_{j-1}$, and b_i' is used instead of b_i. This solves the equation and the instantiation of $Ch^{\mathcal{K}}[y, s]$ by σ' is valid by construction. Note that the variables $y_{i+1}, \ldots, y_{j-1}$ may become unbound, which is not a problem, since the variables are chosen fresh.

   Now we show that the NCCs remain satisfied. (For satisfaction of Δ_2 and the conditions, see below).

   Let $(t, d) \in \Delta_3$. By assumption that σ is a solution, $Var(t\sigma) \cap CV(d\sigma) = \emptyset$. We have to show $Var(t\sigma') \cap CV(d\sigma') = \emptyset$. The inclusion $CV(d\sigma') \subseteq CV(d\sigma)$ holds, since σ' may remove binders (w.r.t. σ), but does not introduce new binder that may govern a hole of any context. This also holds for the renaming of $\mathbb{K}_i$, since the fresh binders cannot occur such that they govern a hole in context d. Thus it remains to show that $Var(t\sigma') \cap CV(d\sigma') \subseteq Var(t\sigma)$. The removal of bindings cannot add variables, and the renaming of $\mathbb{K}_i$ cannot add variables that are in $CV(d\sigma')$, hence we only have to look at the effects of the replacement $[y_j/y_{i+1}]$. For y_j there is no change of containment in $Var(t\sigma')$ compared to $Var(t\sigma)$. Hence $Var(t\sigma') \cap CV(d\sigma') \subseteq Var(t\sigma)$, and σ' does not provoke a new capture violation.

2. The second case is that $i+1 = j$, i.e. two adjacent bindings are in the instance of E_{k_0}. Again σ is changed into σ'. The bindings before the modification for i, j are either $y_i.\mathbb{K}_i[\texttt{var}\ y_{i+1}]; y_{i+1}.\mathbb{K}_{i+1}[\texttt{var}\ y_{i+2}]$, or the bindings are $y_{m-1}.\mathbb{K}_{m-1}[\texttt{var}\ y_m]; y_m.\mathbb{K}_m[s_1]$. They are replaced by $y_i.\mathbb{K}_{i+1}[\texttt{var}\ y_{i+2}]$, or $y_{m-1}.\mathbb{K}_m[s_1]$, respectively. Again, a NCC (t, d) satisfied under σ remains satisfied under the modification, since the inclusions $CV(d\sigma') \subseteq CV(d\sigma)$ and $Var(t\sigma') \subseteq Var(t\sigma)$ hold and since no new binders are introduced.

Now we take care of Δ_2: Let us assume that there are strictly more than n_1^2 bindings in $Ch^{\mathcal{K}}[y, s]\sigma$ that are covered by some $E_i\sigma$ with $E_i \in \Delta_2$. By a simple counting argument we see that there is some $E_j \in \Delta_2$ that covers at least $n_1 + 1$ bindings. If we write only the covering E_i for every binding, then $E_j; M_1; E_j; M_2 \ldots; E_j; M_{n_1}; E_j$ represents such a subchain, where $M_k \subset \{E_1, \ldots, E_n\} \cap \Delta_2$ with $E_j \notin M_k$ for all k. Now we can apply the removal operations above. By minimality of the number m of bindings, we see that $M_1 \neq \emptyset$, since otherwise, the second removal operation can be applied. Furthermore, $M_k' := \bigcup_{h \leq k} M_h$ must be strictly increasing w.r.t. $\subset$, since otherwise, the chain can be shortened using $E_j; M_k; E_j \to E_j$ without violating any constraint. Strictly increasing chains of M_k w.r.t. $\subset$ starting with a non-empty set can only have length $n_1 - 1$ since there are at most $n_1 - 1$ elements, hence we have reached a contradiction. Now there are at most n_1^2 occurrences of bindings that are covered by E_i in $\Delta_2 \cap \{E_1; \ldots; E_n\}$, and in every gap there are at most n_2 other bindings (otherwise, we could remove a binding), which leads to the bound $n_1^2 * (n_2 + 1) + n_2$. $\square$

Remark 5.6. *The quadratic upper bound on the number of required bindings in the situation of Theorem 5.5, cannot be improved by our proof techniques, which means that based on the current knowledge, a quadrat number of bindings has to be tried to check solvability of the constraints.*
Let $E_1; \ldots; E_n; \ldots E_{2n} \doteq Ch^{\mathcal{K}}[y, s]$ be the equation, where $E_1, \ldots, E_n \in \Delta_2$ and $\{E_{n+1}, \ldots, E_{2n}\} \cap \Delta_2 = \emptyset$. Let us denote the sequence of instances of bindings and the instantiation of E_i by the indices. Suppose the sequence is $1, n+1, \ldots 2n, 2, n+1 \ldots 2n, 3, \ldots, n+1 \ldots 2n$. It has length n^2, and an easy check exhibits, that it cannot be shortened by the techniques given in the proofs of Theorem 5.5.

Lemma 5.7 (Lifting chains)**.** *Let σ be a solution of the state (Sol, Γ, Δ), $\{E_1; \ldots; E_n \doteq Ch^{\mathcal{K}}[y, s]\} \in \Delta_4$ with $n > 1$, and $Ch^{\mathcal{K}}[y, s]\sigma = y_1.\mathbb{K}_1[\texttt{var}\ y_2]; y_2.\mathbb{K}_2[\texttt{var}\ y_3]; \ldots; y_m.\mathbb{K}_m[s_1]$, where $y_1 = y\sigma, s_1 = s\sigma$, and y_i for $i \geq 2$ are fresh.*
Then the state $\mathcal{S}' := (Sol\rho, \Gamma\rho, \Delta'\rho)$ has a solution, where $\Delta' = (\Delta_1, \Delta_2, \Delta_3 \cup \Delta_3', \Delta_4 \setminus \{E_1; \ldots; E_n \doteq Ch^{\mathcal{K}}[y, s]\})$ and ρ only modifies E_i, and $Ch^{\mathcal{K}}$ as follows:

$$Ch^{\mathcal{K}}[y, s]\rho = y.D_1^{\mathcal{K}}[\texttt{var}\ Y_2]; Y_2.D_2^{\mathcal{K}}[\texttt{var}\ Y_3]; \ldots; Y_m.D_m^{\mathcal{K}}[s]$$

where Y_i and $D_i^{\mathcal{K}}$ are fresh unification variables, $E_i\rho$ is $E_i\sigma$ modified by replacing $y_1.\mathbb{K}_1[\texttt{var}\ y_2]$ by $y.D_1^{\mathcal{K}}[\texttt{var}\ Y_2]$ and all $y_j.\mathbb{K}_j[\texttt{var}\ y_{j+1}]$ by $Y_j.D_j^{\mathcal{K}}[\texttt{var}\ Y_{j+1}]$, and where Δ_3' is the set $\{(\texttt{var}\ Y_{i+1}, D_i^{\mathcal{K}}) \mid i = 1, \ldots, m-1\}$.

Proof. Clearly, the equation $E_1\rho; \ldots; E_n\rho \sim_{let} Ch^{\mathcal{K}}[y, s]\rho$, and Δ_2 are satisfied. Also, from σ we can derive a solution σ' of $\mathcal{S}'$. Also the added NCCs in Δ_3' are satisfied. Hence the state $\mathcal{S}'$ cannot lead to a *FAIL*. $\square$

Thus, the final test of a state for solvability of the constraint equations as introduced in Definition 4.6 is correct. Indeed, a quadratic number of bindings is required for the satisfiability test of constraint equations (see Remark 5.6). Simple counting and exploiting the removal operations in the proof of Theorem 5.5 shows that the upper bound of Theorem 5.5 can be improved for (small) numbers $n_1 \in \{1, 2, 3, 4\}$ to $2n_2 + 1$; $4n_2 + 3$; $6n_2 + 5$; and $9n_2 + 8$, resp.

Lemma 5.8. *Let $P = (\Gamma_P, \Delta_{P,1}, \Delta_{P,2}, \Delta_{P,3})$ be a letrec unification problem, $\mathcal{S} = (Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))$ be an intermediate state of* UnifLRS*, s.t. only the failure rule* (7) *is applicable. Then there is no solution σ of $\mathcal{S}$.*

Proof. Let $\mathcal{S}$ be the intermediate state of UnifLRS, $(t, d) \in \Delta_4$ and σ be a solution. Since (t, d) is violated, we have $(Var(t) \cap CV(d)) \neq \emptyset$. Let $x \in (Var(t) \cap CV(d))$ and let $x\sigma = \mathsf{x}_0$. It is easy to verify that $\{x'\sigma \mid x' \in CV(d)\} \subseteq CV(d\sigma)$ and thus $\mathsf{x}_0 \in CV(d\sigma)$. Since $x \in Var(t)$ also $\mathsf{x}_0 \in Var(t\sigma)$ must hold, even though it may happen that $x \in FV(t)$ and $\mathsf{x}_0 \in BV(t\sigma)$ if t has a binder X s.t. $X\sigma = \mathsf{x}_0$. However, the NCC $(t\sigma, d\sigma)$ is violated. $\square$

By inspecting all rules and the possibilities for Γ, Δ the following lemma holds:

Lemma 5.9. UnifLRS *does not get stuck, i.e. unless $\Gamma = \emptyset$, there is an applicable rule.*

Proposition 5.10. *The unification algorithm is sound and complete*

Proof. For proving soundness, it suffices to show that a solution of the result of a unification step is also a solution of the start. This is standard for almost all rules. Special attention is required for the non-capture constraints: All rules properly inherit the non-capture constraints, thus the failure rules (4.9) will detect every such violation.

For proving completeness, let σ be a solution of the input problem P. We have to show that $\sigma \in \Omega(P)$. It suffices to show that the unification rules do not lose any solutions. For the first-order rules, this is standard. The guessing rules for non-empty environments or contexts guess according to σ. For the environment rules, if there is an explicit binding b, then the six possibilities are all covered: b must be equal to something on the other side of the equation: (i) an explicit binding, (ii) part of an instance of an environment variable, (iii) the (only binding of an) $Ch^{\mathcal{K}}$-chain, (iv) the prefix of an $Ch^{\mathcal{K}}$-chain, (v) in the middle of an $Ch^{\mathcal{K}}$-chain, (vi) the suffix of an $Ch^{\mathcal{K}}$-chain. If there is no explicit binding in the environment equation, then there are rules to treat all the cases without losing solutions. For the context rules, all possibilities are covered by the non-deterministic rules and by the assumptions in Definition 4.5.

For the failure rules, it is also clear, that σ cannot be lost: Lemma 5.8 ensures that if the NCC is violated, then there is no solution after instantiating the state, and if the weak DVC check is violated, then it is also violated after instantiating the state, since variables Z can only be instantiated by variables. The Δ_4-satisfiability check (Definition 4.6) has an accepting execution in case there is a solution due to Theorem 5.5 and Lemma 5.7. Note that the test in Definition 4.6 does not require to test the NCCs generated in Lemma 5.7. $\square$

Lemma 5.11. *If in an execution path, $\Gamma = \emptyset$ and no Failure rule is applicable, then the final data structure represents at least one solution of at most polynomial size.*

Proof. This follows from the final Δ_4-satisfiability check (see Definition 4.6) checking all constraint equations in Δ_4 (see Theorem 5.5 and Lemma 5.7). Combine *Sol* and the guessed σ for all the constraint equations. Note that instantiating *Sol* does not lead to exponential size due to the occurrence restrictions. Then replace (consistently) unification variables X, Y by fresh concrete variables x, y; unification variables S by a constant (or, if not available, by $\texttt{var}\ \mathsf{x}$, for a fresh variable x) replace the remaining E-variables by a binding $\mathsf{x}.\texttt{var}\ \mathsf{x}$ for a fresh variable x, replace remaining $Ch^{\mathcal{K}}[y, s]$-expressions by the environment containing only the binding $y.\mathbb{K}_0[s]$, where $\mathbb{K}_0$ is a nontrivial context of $\mathcal{K}$, where all bound variables in it are fresh ones; and replace the remaining context variables $D^{\mathcal{K}_1}$ by $[\cdot]$ if $D^{\mathcal{K}_1} \notin \Delta_1$, otherwise by $\mathbb{K}_1$, a nontrivial context of class $\mathcal{K}_1$ where all variables in it are fresh ones. Then all constraints are satisfied and it is an instance of the representation in the output. $\square$

The non-deterministic algorithm UNIFLRS is sound, complete, terminates, and produces final results that cover all solutions, and does not produce final results without solutions.

Theorem 5.12. *The non-deterministic algorithm* UNIFLRS *is correct and every run requires at most polynomial time. The determinized version is a decision algorithm.*

The letrec-unification problem is Graph-Isomorphism-hard for letrec-expressions that do not contain $Ch^{\mathcal{K}}, E, S$ nor context variables, are closed, satisfy the (strong) DVC, and only a bijection between the bound variables has to be constructed, since extended α-equivalence of letrec-expressions is Graph-Isomorphism-complete [16]. We show that our unification problem is NP-hard:

Proposition 5.13. *The letrec-unification-problem is NP-hard.*

Proof. Let $\{c_{i,1}, c_{i,2}, c_{i,3}\}$, $i = 1, \dots, n$ be an instance of the MONOTONE ONE-IN-THREE-3-SAT-problem, where $c_{i,j}$ are propositional variables. The problem to find a valuation that makes exactly one variable in every clause `True`, is NP-hard [13]. We encode every clause as an equation between two letrec-environments: $\mathsf{y}_1.S_{i,1}; \mathsf{y}_2.S_{i,2}; \mathsf{y}_3.S_{i,3} \doteq Y_1.(\mathtt{var}\ \mathsf{x}_f); Y_2.(\mathtt{var}\ \mathsf{x}_f); Y_3.(\mathtt{var}\ \mathsf{x}_t)$, where $Y_1, Y_2, Y_3, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ are fresh let-variables for every clause. Then the letrec unification problem is solvable iff there exists a solution to the MONOTONE ONE-IN-THREE-3-SAT-instance, where the one-to-one correspondence holds: $S_{i,j} \mapsto \mathtt{var}\ \mathsf{x}_t$ iff $c_{i,j}$ is `True` (and $S_{i,j} \mapsto \mathtt{var}\ \mathsf{x}_f$ iff $c_{i,j}$ is `False`). □

By Theorem 5.4 unification can be performed in non-deterministic polynomial time, hence:

Theorem 5.14. *The letrec unification problem is NP-complete.*

6. IMPLEMENTATION

We implemented the meta language `LRSX` and the unification algorithm UNIFLRS in Haskell[2]. The implementation is a bit more specific than the formal representation given in this paper, since only the context classes C, A, and T are built in (which are constructed by means of the operator arity and the strict positions of the function symbols), and for the environment chains only the context classes *Triv* and $\mathcal{A}$ are built-in as a parameter. However, the implemented meta-language is capable to model the syntax, the reduction rules, and program transformations of L_{need} (see Fig. 1), and also the LR-calculus [20], which extends L_{need} by data constructors, case-expressions, and Haskell's `seq`-operator. NCCs for occurrences of bound variables are added automatically to the input, by the rules shown in Fig. 9, where sometimes new function symbols are introduced to express the capture constraints, however, these new function symbols are hidden in the semantics (and do e.g. not influence the construction of contexts). Our implementation provides an input interface to define the calculus reductions and the program transformations in a separate file. We defined the calculus L_{need} as well as the calculus LR (where only data constructors for Booleans, lists, and pairs are defined). To capture the different kinds of reduction contexts used in normal order reduction, most of the reduction rules have several variants, also non-emptiness constraints and NCCs are included in the input e.g. the (no,lapp)-reduction is represented by the three rules shown in Fig. 10.

[2]The implementation is available from
http://www.ki.cs.uni-frankfurt.de/research/lrsx

For all subexpressions of the following forms

1. $\mathtt{letrec}\ y.d[\mathtt{var}\ y]; env\ \mathtt{in}\ s$
2. $\mathtt{letrec}\ y.s; env\ \mathtt{in}\ d[\mathtt{var}\ y]$
3. $\mathtt{letrec}\ Ch^{\mathcal{K}}[y, d[\mathtt{var}\ y]]; env\ \mathtt{in}\ s$
4. $\mathtt{letrec}\ Ch^{\mathcal{K}}[y, s]; env\ \mathtt{in}\ d[\mathtt{var}\ y]$
5. $\mathtt{letrec}\ y.s; env; Ch^{\mathcal{K}}[y', d[\mathtt{var}\ y]]\ \mathtt{in}\ t$
6. $\mathtt{letrec}\ Ch^{\mathcal{K}}[y, s]; env; y'.d[\mathtt{var}\ y]\ \mathtt{in}\ t$
7. $\mathtt{letrec}\ y.s; env; y'.d[\mathtt{var}\ y]\ \mathtt{in}\ t$
8. $\mathtt{letrec}\ Ch_1^{\mathcal{K}}[y, s]; env; Ch_2^{\mathcal{K}'}[y', d[\mathtt{var}\ y]]\ \mathtt{in}\ t$

(where $\mathcal{K}, \mathcal{K}' \in \{\mathit{Triv}, \mathcal{A}\}$) generate an NCC (y, d), and for each subexpression of the form

$$(f\ r_1\ \dots\ r_{i-1}\ x_1.\dots.x_j.y_1.\dots.y_n.d[\mathtt{var}\ y_1]\ r_{i+1}\dots\ r_m)$$

generate an NCC $(y_1, (g\ y_2 \dots y_n.d))$ where g is a (new) function symbol of the appropriate operator arity.

Figure 9: Automatically computed capture constraints

- $A[\mathtt{app}\,(\mathtt{letrec}\,E\,\mathtt{in}\,S_1)\,S_2] \rightarrow A[\mathtt{letrec}\,E\,\mathtt{in}\,(\mathtt{app}\,S_1\,S_2)]$
  with $\Delta_1 = \emptyset$, $\Delta_2 = \{E\}, \Delta_3 = \{(S_2, \mathtt{letrec}\,E\,\mathtt{in}\,[\cdot])\}$
- $\mathtt{letrec}\,E_1\,\mathtt{in}\,A[\mathtt{app}\,(\mathtt{letrec}\,E_2\,\mathtt{in}\,S_1)\,S_2]$
  $\rightarrow \mathtt{letrec}\,E_1\,\mathtt{in}\,A[\mathtt{letrec}\,E_2\,\mathtt{in}\,(\mathtt{app}\,S_1\,S_2)]$
  with $\Delta_1 = \emptyset$, $\Delta_2 = \{E_1, E_2\}, \Delta_3 = \{(S_2, \mathtt{letrec}\,E_2\,\mathtt{in}\,[\cdot])\}$
- $\mathtt{letrec}\,E_1; Ch^{\mathcal{A}}[Y, \mathtt{app}\,(\mathtt{letrec}\,E_2\,\mathtt{in}\,S_1)\,S_2]\,\mathtt{in}\,A[\mathtt{var}\,Y]$
  $\rightarrow \mathtt{letrec}\,E_1; Ch^{\mathcal{A}}[Y, \mathtt{letrec}\,E_2\,\mathtt{in}\,\mathtt{app}\,S_1\,S_2]\,\mathtt{in}\,A[\mathtt{var}\,Y]$
  with $\Delta_1{=}\emptyset$, $\Delta_2{=}\{E_2\},\Delta_3{=}\{(S_2, \mathtt{letrec}\,E_2\,\mathtt{in}\,[\cdot]), (\mathtt{var}\ Y, A)\}$

Figure 10: Encoding of rule (no,lapp) in `LRSX`-syntax by three rules

Using the implementation, we computed the overlaps between the left hand sides of normal order reductions with left as well as right hand sides of transformations applied in top-contexts (denoted by $\xrightarrow{T,a}$ for transformation a). The overlaps with left hand sides are necessary to compute so-called forking diagrams, while the overlaps with right hand sides are required for so-called commuting diagrams. Overlapping transformations applied in top-contexts (instead of all contexts) is sufficient for a correctness proof, since a context lemma holds in L_{need} and LR. Fig. 11 shows the numbers of (general) unifiers that were obtained for the transformations (where the numbers are the sums of overlaps with all normal order rules) for L_{need} and for LR. The table shows that there are far more overlaps in LR than in L_{need} which stems from the fact that LR has an extended syntax and more reduction rules.

Ongoing work is to extend the implementation by a matching algorithm for `LRSX`-expressions and a treatment of α-renamings to perform normal order reduction and transformation automatically on the meta-expressions. A hurdle is that also these algorithms have to respect the noncapture-constraints as delivered by the solution of the unification algorithm when performing deduction steps. Having such an algorithm would enable us to join the found critical pairs and thus to produce (complete sets of) forking and commut-

	Calculus L_{need}		Calculus LR	
Transformation	forking	commuting	forking	commuting
(lbeta)	102	109	1876	1953
(cp-in)	131	131	2642	2643
(cp-e)	201	197	3616	3616
(llet-in)	102	142	1908	2236
(llet-e)	115	221	1990	2970
(lapp)	102	226	1876	1989
(cpx-in)	168	168	4082	4082
(cpx-e)	421	421	9840	9840
(xch)	181	181	2576	2576
(gc1)	113	113	1988	1988
(gc2)	105	247	1925	3123
$\sum$	1741	2156	34319	37016

Figure 11: Numbers of solutions of unifying $s \doteq t$ (forking) and $s \doteq t'$ (commuting) where $s \xrightarrow{no,a} s'$ and $t \xrightarrow{T,b} t'$, summed over all normal order rules (no, a) per transformation (T, b).

ing diagrams. Combining the diagram computation with the automated induction technique in [10] then would enable the fully automated correctness proof of program transformations.

7. CONCLUSION

We described a higher order unification algorithm on a parameterized description of call-by-need program calculi and demonstrated its use in an application for reasoning in two different calculi. There is a high potential of transferring the method to other higher-order languages. The research will be continued by studying matching and reduction sequences and joining of diagrams. Further research could be applications to other call-by-need calculi and studying extensions.

8. REFERENCES

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

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL 1995*, pp. 233–246. ACM, 1995.

[3] F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pp. 445–532. Elsevier and MIT Press, 2001.

[4] J. Cheney. Toward a general theory of names: Binding and scope. In *MERLIN 2005*, pp. 33–40. ACM, 2005.

[5] G. P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[6] A. Jez. Context unification is in PSPACE. In *ICALP 2014*, *LNCS* 8573, pp. 244–255. Springer, 2014.

[7] E. Machkasova and F. A. Turbak. A calculus for link-time compilation. In *ESOP 2000*, *LNCS* 1782, pp. 260–274. Springer, 2000.

[8] A. K. D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

[9] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI 1988*, pp. 199–208. ACM, 1988.

[10] C. Rau, D. Sabel, and M. Schmidt-Schauß. Correctness of program transformations as a termination problem. In *IJCAR 2012*, *LNCS* 7364, pp. 462–476. Springer, 2012.

[11] C. Rau and M. Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *UNIF 2011*, pp. 35–41, July 2011.

[12] D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

[13] T. J. Schaefer. The complexity of satisfiability problems. In *STOC 1978*, pp. 216–226. ACM, 1978.

[14] M. Schmidt-Schauß. Decidable variants of higher-order unification. In *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, *LNCS* 2605, pp. 154–168. Springer, 2005.

[15] M. Schmidt-Schauß, T. Kutsia, J. Levy, and M. Villaret. Nominal unification of higher order expressions with recursive let, 2016. submitted to a conference.

[16] M. Schmidt-Schauß, C. Rau, and D. Sabel. Algorithms for Extended Alpha-Equivalence and Complexity. In *RTA 2013*, *LIPIcs* 21, pp. 255–270. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.

[17] M. Schmidt-Schauß and D. Sabel. Improvements in a functional core language with call-by-need operational semantics. In *PPDP 2015*, pp. 220–231. ACM, 2015.

[18] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *RTA 2010*, *LIPIcs* 6, pp. 295–310. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[19] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. *Log. Methods Comput. Sci.*, 11(1), 2015.

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

[21] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.

[22] C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in nominal isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.

[23] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. In *CSL 2003*, *LNCS* 2803, pp. 513–527. Springer, 2003.

[24] J. B. Wells, D. Plump, and F. Kamareddine. Diagrams for meaning preservation. In *RTA 2003*, *LNCS* 2706, pp. 88 –106. Springer, 2003.