A Unification Algorithm to Compute Overlaps in a Call-by-Need Lambda-Calculus with Variable-Binding Chains

Conrad Rau and Manfred Schmidt-Schauß

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

Abstract. We extend the unification algorithm from previous work of the authors to cover the call-by-need λ-calculus LR. The main task of the unification algorithm is to compute all possible overlaps (also called forks) between the reduction rules of LR and a set of program transformations. The new contribution is that the variable-binding chains (a form of indirections) that occur in the rules and the transformations are in the scope of the unification method. This is achieved through the use of additional term syntax to treat variable-binding chains of any length. The result is a unification algorithm that terminates and computes a finite and complete set of overlaps (i.e. critical pairs) between all rules and given transformations.

1 Introduction and Motivation

Proving correctness of program transformations in call-by-need λ-calculi can be done using a method that heavily relies on sets of reduction diagrams, which can be interpreted as a description of local confluence between program transformations and the reduction rules of the λ-calculus. The material step to generate such diagram sets is the determination of overlaps (also called forks) between reductions and transformations. The work in [5] presented a terminating and complete unification algorithm that computes all forks and thus the critical pairs between the reduction rules and a set of transformation rules in the calculus $L_{\text{need}}$, a call-by-need lambda calculus with a recursive let. In this paper we extend this work to the more expressive call-by-need lambda calculus LR which was used in [7] to model the functional part of the programming language Haskell. The calculus LR extends $L_{\text{need}}$ in several aspects: (i) There are data types in the form of data constructors and a syntax for case analysis, (ii) The reduction rules are defined using variable-binding chains, and (iii) there is a seq-construct for enforcing sequential reduction. The variable chains are required in this calculus to enable correctness proofs of the reduction rules seen as transformations.

The main motivation for this work is to make a step forward towards the automatic verification of the correctness of program transformations in the LR-calculus. Therefore, as an intermediate goal we are interested in the automatic...
verification of the reduction diagrams given in [7]. Another motivation is to develop tools that are also applicable to other, non-deterministic and concurrent calculi aiming at automatically detecting program transformation and proving them to be correct.

The central idea for computing all overlaps is to use (a variant of) first-order unification. It works as follows: The expressions in the reduction rules (of the \(\lambda\)-calculus LR), which are in fact rule schemes, are translated into many sorted first-order terms. These terms are extended with several special syntactic constructs to finitely capture the infiniteness of reduction rule sets, the letrec-construct, the binding primitives and the different classes of context variables. The letrec-construct is translated using the equational theory of left-commutativity. The higher-order feature of bound variables is translated such that bound variables are terms (i.e. variables) of an empty sort, i.e. without ground terms. Moreover, syntactical restrictions like enforcing different variable names in letrecs together with the distinct variable convention provide a means that can be checked in the first-order translation to avoid illegal or unsound unifiers that otherwise would equate variables that must be kept different, like in the first-order translation of \(\lambda x.x\) and \(\lambda x.y\). Context names in rules are encoded as context variables at the term level. All these aspects of the first-order encoding of a higher order calculus are treated as in [5].

The novelty of our unification algorithm is that it can unify terms \(s_1, s_2\), where both \(s_1\) and \(s_2\) may contain variable chains. Variable chains are the first-order encodings of environments like \((\text{letrec } x_1 = v, x_2 = x_1, x_3 = x_2, \ldots, x_n = x_{n-1} \text{ in } A[x_n])\). There are also the binding chains with bindings \(x = A[ y]\), where \(A\) is a context variable. The additional complication is that it is unavoidable that these binding chains occur on both sides of equations in unification problems, in contrast to the (proper) binding chains in [5] that only appear on one side of equations. We devise a unification procedure that can treat binding chains of any length, first-order encoded as \(\text{VCh}(x, y, n)\) for variable-only chains and \(\text{NCh}(x, y, n')\) for other binding chains. The unification rules exploit the equational theory of left-commutativity as well as the distinct variable convention-restrictions and so can enforce termination without losing completeness.

The main result of this work is a translation of the higher-order overlapping problems into extended first-order unification problems (called initial problems) and a complete and terminating unification algorithm for these initial problems. This enables the automation of the computation of complete diagram sets for the calculus LR, in particular for the transformations that are derived from the reduction rules of the calculus.

2 Motivating Example for the Unification Algorithm

We demonstrate the main ideas and effects of the encoding and the unification rules by an extended example, since a formal development would exceed the space limit. Therefore we illustrate how the overlap of the reduction rule (no, cp-e)
Unification in Lambda-Calculi with Variable Chains

with the transformation rule (cp-in) in the calculus LR can be computed. The respective rules are in Fig. [1] where $C, A$ denote contexts.

\[
\begin{align*}
\text{(no, cp-e)} & \quad \text{letrec } x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = A_1[x_m], \\
& \quad \quad \quad y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \text{Env in } A[y_n] \\
& \quad \rightarrow \text{letrec } x_1 = v, x_2 = x_1, \ldots, x_m = x_{m-1}, y_1 = A_1[v], \\
& \quad \quad \quad y_2 = A_2[y_1], \ldots, y_n = A_n[y_{n-1}], \text{Env in } A[y_n] \\
\text{(cp-in)} & \quad \text{letrec } x_1 = v, x_2 = x_1, \ldots, x_k = x_{k-1}, \text{Env in } C[x_k] \\
& \quad \rightarrow \text{letrec } x_1 = v, x_2 = x_1, x_k = x_{k-1}, \text{Env in } C[v]
\end{align*}
\]

Where $v$ is an abstraction and $A_1$ is a non-empty context.

**Fig. 1.** Reduction rule and Transformation rule of the calculus LR

The initial unification problem corresponding to the overlap of (no, cp-e) with the transformation (cp-in) is an equation between the first-order encoded left hand sides of the rules (with a context variable $S$ indicating that the (cp-in) transformation may not occur at top position). It looks as follows:

\[
S(\text{let}(\text{env}^* \left(\{\text{bind}(x_1, \text{lam}(v, s))\} \cup \text{VCh}(x_1, x, m) \cup \{\text{bind}(y_1, A_1(\text{var}(x)))\} \cup \text{NCh}(y_1, y, n) \cup \text{Env}\right), A(\text{var}(y)))) = (1)
\]

We briefly elaborate on the translation used to arrive at the above unification problem: the letrec-environment (that is the set of LR-expression of form $x = s$) is encoded by a nesting of a binary function symbol $\text{env}$, similar to a list or set representation. The irrelevance of the order of elements in letrec-environments is achieved by the equality axiom $\text{LC}_{\text{env}} := \{\text{env}(x, \text{env}(y, z)) = \text{env}(y, \text{env}(x, z))\}$, i.e. $\text{env}$ is a left-commutative function symbol (For the $\text{LC}$-theory and unification modulo $\text{LC}$ see [21]). We use the abbreviation $\text{env}^*\{t_1, \ldots, t_m \cup r\}$ to denote the term $\text{env}(t_1, \text{env}(t_2, \ldots, \text{env}(t_m, r) \ldots))$, and usually use union-notation within the first argument of $\text{env}^*$. A binding $x = y$ is encoded as $\text{bind}(x, \text{var}(y))$. Some terms in the first order encoding may not possess syntactical admissible counterparts in LR: for example $\text{env}^*\{\text{bind}(x, \text{var}(y)), \text{bind}(x, \text{var}(z))\} \cup \text{Env}$, because $x$ occurs twice at a binder position. Such terms are called syntactically incorrect w.r.t. LR.

The binding chains of variable length in the left hand sides of the LR-reductions (i.e. $x_2 = x_1, \ldots, x_m = x_{m-1}$ etc.) are encoded as $\text{VCh}(x, y, n)$, and $\text{NCh}(x, y, n)$, respectively, where $x, y$ are the variables starting and ending the chain; they may also occur in the environment outside of the chain (which is the case for the encoding of the LR reductions and transformations). The integer variable $n$ stands for the length, and it is assumed that the (implicit) intermediate variables and context variables are fresh. The meaning is that such a chain-construct expands to $n$ bindings, for example $\text{VCh}(x, y, 2)$ represents $z = x, y = z$, and in the first-order encoding:
where \( z \) is a fresh variable. Similarly, \( \text{NCh}(x, y, 2) \) represents \( z = A_1[x], y = A_2[z], \) and its first-order encoding is:

\[
\text{env}(\text{bind}(z, A_1(\text{var}(x))), \text{env}(\text{bind}(y, A_2(\text{var}(z))), [[]])),
\]

where \( A_1, A_2 \) are fresh context variables. The constructs for binding chains describe (possibly infinite) sets of terms. They bear some similarities to term schematizations used in [6][3][4].

We proceed to solve equation (1). Therefore we first use the unification rule that achieves this goal is empty.

After that we decompose the \( \lambda \)-terms. It remains to solve the equation \( \text{A}(\text{var}(y)) = C(\text{var}(x')) \) and

\[
\text{env}^* \left( \begin{array}{l}
\text{bind}(x_1, \text{lam}(v, s)) \\
\text{bind}(y_1, A_1(\text{var}(x)))
\end{array} \right) \cup \text{VCh}(x_1, x, m) \cup \text{NCh}(y_1, y, n) \cup \text{Env}
\]

\[
\text{env}^* \left( \text{bind}(x_1', \text{lam}(v', s')) \right) \cup \text{VCh}(x_1', x', k) \cup \text{Env}'
\]

Again the first equation can be treated using similar methods as in [5], taking care of the extended signature. The second equation (2), however, requires special treatment because of the variable chains occurring in both terms. We have to treat the integer variables \( m, n, k \) (for the lengths of chains) in a general way avoiding guessing natural numbers, since this would lead to an infinite number of solutions. Our unification method employs some crucial properties of the to-be-solved problems, such that infinite solution sets can be avoided, therefore leading to a terminating procedure without sacrificing completeness of the algorithm.

The new unification rule that achieves this goal is \( \text{U-Chain} \) in Fig. 2, which states that there are two possibilities for unifying chains \( \text{Ch}_1 \) and \( \text{Ch}_2 \): They are either identical (described by case 2, where the length of the chains and their

\[
\text{U-Chain:}
\{ \text{env}^* (\text{Ch}(x_1, y_1, l_1) \cup M_1 \cup r_1) = \text{env}^* (\text{Ch}(x_2, y_2, l_2) \cup M_2 \cup r_2) \} \uplus P
\]

choose one of the following possibilities

1) \( \{ l_1 = l_1' + l + l_1'' , l_2 = l + l_2', \text{env}^* \left( \text{Ch}(x_1, x_2, l_1''), \text{Ch}(z, y_1, l_1'') \cup M_1 \cup r_1 \right) = \text{env}^* \left( \text{Ch}(z, y_2, l_1'') \cup M_2 \cup r_2 \right) \} \uplus P
\]

2) \( \{ l_1 = l_2, x_1 = x_2, y_1 = y_2, \text{env}^* (M_1 \cup r_1) = \text{env}^* (M_2 \cup r_2) \} \uplus P
\]

Where \( z \) is a fresh variable of sort \( \text{BV} \) and \( l, l_1', l_1'', l_2' \) are fresh integer variables.

\[
\text{Dec-Chain}
\{ \text{env}^* (s_1) \cup M_1 \cup r_1) = \text{env}^* (\text{Ch}(x, y, l) \cup M_2 \cup r_2) \} \uplus P
\]

select one of the following possibilities

1) \( \{ l = 1, s_1 = \text{bind}(y, A(\text{var}(x))), \text{env}^* (M_1 \cup r_1) = \text{env}^* (M_2 \cup r_2) \} \uplus P
\]

2) \( \{ l = 1 + l_1, s_1 = \text{bind}(z, A(\text{var}(x))), \text{env}^* (M_1 \cup r_1) = \text{env}^* (\text{Ch}(z, y_1, l_1) \cup M_2 \cup r_2) \} \uplus P
\]

3) \( \{ l = 1 + l_1, s_1 = \text{bind}(y, A(\text{var}(z))), \text{env}^* (M_1 \cup r_1) = \text{env}^* (\text{Ch}(x, z, l_1) \cup M_2 \cup r_2) \} \uplus P
\]

4) \( \{ l = 1 + l_1 + l_2, s_1 = \text{bind}(z_2, A(\text{var}(z_1))), \text{env}^* (M_1 \cup r_1) = \text{env}^* (\text{Ch}(x, z_1, l_1) \cup \text{Ch}(z_2, y_1, l_2) \cup M_2 \cup r_2) \} \uplus P
\]

Where \( s_1 \) is a binding expression. \( z, z_1, z_2 \) are fresh variables of sort \( \text{BV} \) and \( A \) is either a fresh context variable of class \( \mathcal{A} \) if \( \text{Ch} = \text{NCh} \)

Fig. 2. Two unification rules dealing with variable chains. (Here we use the symbol \( \text{Ch} \) to denote either chain construct.)
start- and end-points are equated), or the initial part of Ch₂ is equal to some intermediate part of Ch₁ and both tails of Ch₂ and Ch₁ and the initial sequence of Ch₁ are disjoint (case 1). These (and the symmetrical case, where Ch₁ and Ch₂ are swapped) are the sole possibilities of equating bindings from chains: All other unification schemes of chain-bindings would result in terms that do not represent syntactically admissible LR-expressions. We will elaborate on this by applying the rule $\text{U-Chain}$ to the equation (2) from our continued example, where one possible transformation (i.e. case 1) of $\text{U-Chain}$ applying the rule $\text{U-Chain}$ represent syntactically admissible LR-expressions. We will elaborate on this by applying the rule $\text{U-Chain}$ to the equation (2) from our continued example, where one possible transformation (i.e. case 1) of $\text{U-Chain}$ yields:

\[
m \doteq m_1 + l + m_2, \quad k \doteq l + k_1,
\]

\[
env^* \left( \left\{ \text{bind}(x_1, \text{lam}(v, s)) \right\} \cup \text{VCh}(x_1, x'_1, m_1) \cup \text{VCh}(z, x, m_2) \cup \text{NCh}(y_1, y, n) \cup Env \right) \doteq env^* \left( \left\{ \text{bind}(x'_1, \text{lam}(v', s')) \right\} \cup \text{VCh}(z, x', k_1) \cup Env' \right)
\]

Now we can solve the last equation from (3) by setting $Env \doteq (a)$ and $Env' \doteq (a)$ where in the equations (a) and (b) the environment variables $Env$ and $Env'$ are replaced by $Env''$. Now the system is in solved form. Applying the resulting unifier to one term of the original problem (1) yields:

\[
\ldots \text{let}(env^* \left( \left\{ \text{bind}(x'_1, \text{lam}(v', s')) \right\} \cup \ldots \text{VCh}(x_1, x'_1, m_1) \cup \ldots \right)) \ldots
\]

Instantiating $m_1$ with 1 and back-translating the term into LR results in $(\text{letrec } x'_1 = \lambda v'.s', \ldots, x'_1 = x_1 \text{ in } \ldots)$; a expression that is syntactically not admissible because the variable $x'_1$ occurs twice at a binder position. Hence in the context of the specific unification problems (initial problems) we want to solve, case 1) of the rule $\text{U-Chain}$ can be simplified to

\[
\{ l_1 \doteq l + l'_1, \quad l_2 \doteq l + l'_2, \quad x_1 \doteq x_2, \}
\]

\[
env^* (\text{Ch}(z, y_1, l'_1) \cup M_1 \cup r_1) \doteq env^* (\text{Ch}(z, y_2, l'_2) \cup M_2 \cup r_2) \}
\]

I.e. two chains are equated beginning from their starting point up to some point from where they are disjoint. Here the justification is that initial problems have an additional property: bindings in variable chains are equipped with a (strict partial) order (like a linear list) with a least element called anchor-binding. In the case of $\text{VCh}$-constructs these bindings are always of the form $\text{bind}(x, v)$ where $v$ is either an abstraction or a constructor application. The partial order in conjunction with the syntactical correctness criterion ensures the following: If some bindings of two chains are equated then all chain bindings that are smaller are also equated, until one anchor-binding is reached. The equation between such an anchor-binding and a non-anchor chain-binding (e.g. $\text{bind}(y, A(\text{var}(z)))$) can never hold, i.e. $\text{bind}(x, v) \doteq \text{bind}(y, A(\text{var}(z)))$ has no solution. Therefore our algorithm avoids the derivation of such unsolvable equations by equating initial parts of variable chains starting from their anchor-bindings. This strategy is crucial for the termination of the algorithm but it is only complete for initial problems (where each variable chain comes equipped with an anchor). Together with some additional unification rules and conditions that control the application of rules, termination of the algorithm can be assured.
One of those other rules, also concerned with solving equations with binding chains, is the rule Dec-Chain from Fig. 2. It covers the cases where a non-chain binding $s_1$ is equated with a chain binding. The possibilities are: 1) The chain consists only of one binding which is equated with $s_1$, or 2) the first binding of the chain is equated with binding $s_1$, or 3) the last chain binding is equated with $s_1$, or 4) a binding from the middle of the chain is equated with $s_1$ and the original chain is split around this externalized binding. All of these cases require that some of the internal chain variables (context and $BV$-sorted) are made explicit. These variables are always chosen as fresh (i.e. not occurring anywhere else in the unification problem).

Note that syntactical correctness and the distinct variable convention of the re-translated terms are enforced by the unification rules and by failure rules, and that without these precautions our rule-based unification algorithm would not terminate.

3 Overview of the Algorithm and Results

The unification algorithm for computing the overlaps in LR is applied to (initial) unification problems of the form \{ $S[l_{T,i}] = l_{no,j}$ \} where $l_{T,i}$ and $l_{no,j}$ are encoded left hand sides of LR reduction rules. Initial problems are restricted: They are linear in the variables and context variables, with the exception of variables of sort $Bind$, which is an empty sort. The occurrence of chains in initial problems is also restricted. These restrictions stem from the syntactical form of the reduction rules and the transformations of the LR calculus.

The unification rules of our algorithm consist of (i) rules from [5] that are adapted to the extended signature, and (ii) rules for dealing with equations $env^*(\ldots) = env^*(\ldots)$, where both sides contain binding chains.

The following holds:

- The (nondeterministic) algorithm terminates on initial unification problems.
- The algorithm is sound and complete on initial unification problems, under the sensible restriction that only solutions are permitted that lead to syntactically correct expressions after translating them back to LR.
- The result of all nondeterministic executions is a finite set of final representations. These can be re-translated and lead to a finite set of overlaps of reduction rules and transformations.

4 Conclusion and Further Work

We devised an algorithm that computes complete sets of forks for the calculus $LR$ from [7]. Therefore we first encode left hand sides of reduction rules into a term representation and use it to generate initial unification problems that describe all overlaps. Then we solve those unification problems using the sketched unification algorithm. After these steps we eventually instantiate the unification problems
that describe the forks with the computed solutions and translate them back to yield all forks in the LR calculus.

We plan to implement the computation and thus the verification of most of the diagrams of LR as presented in [7]. The core will be the unification algorithm as sketched above. This requires in addition closing the critical pairs using the normal-order reduction.

References