Alpha-Renaming of Higher-Order Meta-Expressions

David Sabel

2017
Alpha-Renaming of Higher-Order Meta-Expressions

David Sabel*

Goethe-University, Frankfurt am Main, Germany
sabel@ki.informatik.uni-frankfurt.de

Abstract. Motivated by tools for automated deduction on functional programming languages and programs, we propose a formalism to symbolically represent α-renamings for meta-expressions. The formalism is an extension of usual higher-order meta-syntax which allows to α-rename all valid ground instances of a meta-expression to fulfill the distinct variable convention. The renaming mechanism may be helpful for several reasoning tasks in deduction systems. We present our approach for a meta-language which uses higher-order abstract syntax and a meta-notation for recursive let-bindings, contexts, and environments. It is used in the LRSX Tool – a tool to reason on the correctness of program transformations in higher-order program calculi with respect to their operational semantics. Besides introducing a formalism to represent symbolic α-renamings, we present and analyze algorithms for simplification of α-renamings, matching, rewriting, and checking α-equivalence of symbolically α-renamed meta-expressions.

1 Introduction

We focus on automatically proving correctness of program transformations for higher-order programming languages with cyclic bindings as they occur in functional programming languages with call-by-need semantics like Haskell (see 1211). One technique to establish such proofs for program calculi with small-step operational semantics is the diagram method 1215 which can roughly be described as follows: First all overlaps between calculus reductions and a transformation step are computed, then the overlaps are joined by transformation and reduction steps resulting in a complete set of diagrams, which is then used in an inductive proof 9 to show correctness of the transformation w.r.t. contextual equivalence 912. This diagram method was e.g. used in 1915 and similar techniques are in 21167, where the overlaps and the joins are computed manually by a case-analysis. In our recently developed LRSX Tool 2 we try to automatize these computations for a generic meta-language – called LRSX. The input of the tool is a calculus description consisting of the small-step reduction rules and the transformation rules. Overlaps are computed by a unification algorithm 17 and reductions and transformations to join the overlaps are applied using a matching algorithm 13.

The language LRSX uses higher-order abstract syntax 10 extended with a letrec-construct letrec x_1 = s_1;...;x_n = s_n in s_{n+1} to represent unordered sets of recursive bindings (where the scope of the letrec-bound variables x_i is s_1,...,s_{n+1}) and meta-variables for expressions, variables, parts of letrec-environments, and contexts of different classes. These constructs are required to model typical small-step reduction rules of call-by-need program calculi where reduction strategies are expressed by using an appropriate class of evaluation contexts (see e.g. 211918).

Since more sophisticated methods to reason on meta-expressions with binders (e.g. nominal techniques 11) do not support all these constructs, we use a direct approach, where meta-expressions are interpreted in first-order fashion by instantiating them with all possible ground expressions and thus LRSX-expressions represent (potentially infinite) sets of (ground) expressions. However, the main data structure for meta-programs and transformations in the LRSX Tool are so-called constrained expressions that are meta-expressions augmented by constraints which restrict the instances. For example, consider the transformation (let):

\[ C[\text{letrec } E_1 \text{ in } \text{letrec } E_2 \text{ in } S] \xrightarrow{\text{let}} C[\text{letrec } E_1; E_2 \text{ in } S] \]

which joins two nested letrec-environments and where S is a meta-variable for an arbitrary expression, C is a meta-variable for an arbitrary context, and E_1, E_2 are meta-variables for arbitrary letrec-environments. Using this rule without constraints would e.g. allow to instantiate the meta-variable E_1 by the environment which consists of a single binding y = y, meta-variable E_2 by an environment which consists of a single binding x = y, meta-variable S by x, and meta-variable C by the empty context resulting in the instantiated rule

---

* This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1.
1 See 13 for an automation of this step using automated termination provers.
2 http://goethe.de/LRSXTOOL
3 Later in this paper these bindings are written as x var y, since “:” is used instead of “=” and the function symbol var is necessary to lift variables to expressions.
which allows to rewrite (bound variables are pairwise disjoint from free variables, and all binders bind different variables) holds for variable, a context variable) with the meaning that instantiations techniques \cite{11}, including nominal uni/fication \cite{20,4,6}, nominal matching \cite{3}, and nominal rewriting \cite{5} where \(\alpha\) to check reduction or transformation step and thus we present an algorithm for this task. We /f_inally present an algorithm an \(\text{LRSX}\) adaption of the matching algorithm from \cite{14} s.t. can be removed. This procedure is important for our automated tool, since in the tool equivalence of expressions LRSX an \(\text{LRSX}\)-expression s.t. on the semantic level the instances are \(\alpha\) appropriate mechanism of such a symbolic \(\rho\) all instantiations many) concrete instances). Thus we want to rename \(x\) into \(x'\). For the above instance, we may \(x = y\) letrec into \(x\) letrec which illegaly captures the variable \(y\). Thus to treat \(\alpha\)-renamings in a mathematical clean way, we extend the language LRSX\(\alpha\) by syntactic constructs to represent the \(\alpha\)-renamings. The extended language is called LRSX\(\alpha\). Adding this kind of syntactic support for \(\alpha\)-renamings should be possible for any meta-language with variable binders, so the use of language LRSX\(\alpha\) should be understood as exemplary but not exclusive. Besides the definition of the syntax and the (ground term-) semantics of LRSX\(\alpha\)-expressions, further results of this paper target basic reasoning tasks with LRSX\(\alpha\)-expressions. A first algorithm performs \(\alpha\)-renaming, i.e. it takes an LRSX-meta expression and delivers an LRSX\(\alpha\)-expression s.t. on the semantic level the instances are \(\alpha\)-renamed by a fresh renaming. A further procedure performs simplification of symbolic \(\alpha\)-renamings, i.e. it deduces that parts of the symbolic renamings can be removed. This procedure is important for our automated tool, since in the tool equivalence of expressions has to be detected and without simplification of renamings this is impossible in many cases. We provide an adaption of the matching algorithm from \cite{14} s.t. LRSX\(\alpha\)-expressions can be matched against LRSX\(\alpha\)-expressions which allows to rewrite LRSX\(\alpha\)-expressions. However, this may require to adapt the symbolic \(\alpha\)-renaming after a reduction or transformation step and thus we present an algorithm for this task. We finally present an algorithm to check \(\alpha\)-equivalence of LRSX\(\alpha\)-expressions.

\textit{Related Work.} Syntactic reasoning on expressions with binders w.r.t. \(\alpha\)-equivalence can be done by nominal techniques \cite{11}, including nominal unification \cite{20,4,6}, nominal matching \cite{3}, and nominal rewriting \cite{5} where
Recently also nominal terms with letrec were analyzed [16]. The semantics of nominal meta-terms are all $\alpha$-equivalent expressions of all instances. Similarly to our constrained expressions, nominal terms allow to use so-called freshness constraints to forbid unwanted instantiations. In our approach, an $\alpha$-renamed meta-expression represents only those $\alpha$-equivalent expressions which fulfill the distinct variable convention which seems to be an indispensable requirement regarding the example of transformation (let). Using freshness constraints, instances of nominal meta-terms can be restricted to ensure that the distinct variable convention holds. However, this requires knowledge about the binders (to form freshness constraints). Our approach is more general since it includes meta-syntax with meta-variables representing contexts and parts of let-rec-environments. Adding them to nominal techniques seems to be non-trivial and complicated and thus it is not in the scope of the current work.

In Sect. 2 we introduce the ground language LRS and the meta-language LRSX, which is then extended by symbolic $\alpha$-renamings in Sect. 3 where we give an algorithm to symbolically $\alpha$-rename LRSX-expressions. In Sect. 4 we provide an algorithm for simplification of symbolic $\alpha$-renamings. In Sect. 5 we present further algorithms for symbolically $\alpha$-renamed expressions, i.e. a matching algorithm, an algorithm to refresh the $\alpha$-renaming after a rewrite step was applied, and an algorithm to check $\alpha$-equivalence. In Sect. 6 we report on experimental results. In Sect. 7 we conclude.

2 Languages LRS and LRSX

In this section we introduce two languages. First, we introduce the language LRS which is a generic functional language with higher-order operators (e.g. like lambda-abstractions) and letrec-expressions which represent shared and recursive bindings. We then introduce the meta-language LRSX which extends LRS by meta-variables for variables, expressions, contexts, and let-rec environments. An LRSX-expression represents a set of LRS-expressions which can be generated by instantiating the meta-variables with LRS-variables, -expressions, -contexts, or let-rec-environments, resp. An LRSX-expression is ground iff it is an LRS-expression. Both languages are parametrized over a set of function symbols $F$ and a set $K$ of context classes.

2.1 The Language LRS

Definition 2.1. The syntax of LRS is defined in Fig. 1. The four syntactic categories of objects are $\text{Var}$ which is a countably-infinite set of variables, $\text{HExpr}$ which are higher-order expressions, $\text{Env}$ representing let-rec-environments, and $\text{Bind}$ which are let-rec-bindings. Elements $s$ of $\text{HExpr}$ have an order $\alpha(s) \in \mathbb{N}_0$, where $\text{HExpr}^\alpha$ denotes the elements of $\text{HExpr}$ of order $\alpha$, and $\text{HExpr}^0 = \text{Expr}$.

Each $f \in F$ has a syntactic type $f : \tau_1 \to \ldots \to \tau_n \to \text{Expr}$, where $\tau_i$ may be $\text{Var}$, or $\text{HExpr}^\alpha$; $\nu$ is called the arity of $f$, denoted $\nu(f)$; and the order arity $\oar(f)$ is the $n$-tuple $(\delta_1, \ldots, \delta_n)$, where $\delta_i = k_i \in \mathbb{N}_0$, or $\delta_i = \text{Var}$, depending on the type of $f$. We assume that $\{\text{Var}, \lambda \} \subseteq F$ where $\text{Var}$ is of type $\text{Var} \to \text{Expr}$ and lifts variables to expressions with $\oar(\text{Var}) = (\text{Var})$, and $\oar(\lambda) = (1)$.

Note that in a higher-order expression $x.r$, the scope of $x$ is $r$. The scope of $x$ in letrec $x.s$; env in $s'$ is $s$, env and $s'$.

Definition 2.2. An LRS-expression $s$ satisfies the let variable convention (LVC) iff a let-bound variable does not occur twice as a binder in the same let-rec-environment. With $LV(\text{env})$ we denote the let-bound variables in an environment $\text{env}$, i.e. all $x$ s.t. $\text{env} = \text{env'}; x.s$.

For instance, the expression letrec $x$; $x$.var $x.x$.var true in $x$ does not fulfill the LVC while letrec $x$.var $x$; $x$.var true in $x$ does.

With the next definition we formally define the notion of an $\alpha$-renaming of an LRS-expression. It is insufficient to define such a renaming as a mapping from variables to variables (and lifting it to expressions), since for example, we want to rename the expression $\lambda x. \lambda x$.var $x$ into $\lambda x_1. \lambda x_2$.var $x_2$ which shows that the renaming of variable occurrences depends on their positions. For this reason, we use a formal notion of positions of expressions:
Definition 2.3. Let $<$ be a total order on variables. A position is a sequence of natural numbers, which we use Deweyotation for the sequences. For a (higher-order) expression or a binding $t$ that satisfies the LVC, the positions of $t$, $\text{Pos}(t)$, are inductively defined as follows where w.l.o.g. we assume $x_i < x_j$ for $1 \leq i < j \leq n$: \[
\text{Pos}(x) := \{e\}, \quad \text{Pos}(f t_1 \ldots t_n) := \{e\} \cup \bigcup_{i=1}^n \{i.p | p \in \text{Pos}(t_i)\}, \quad \text{Pos}(\text{letrec}\ x_1:s_1;\ldots;x_n:s_n) := \{e\} \cup \bigcup_{i=1}^n \{i.p | p \in \text{Pos}(s_i)\}, \quad \text{Pos}(x.r) := \{e, 1\} \cup \{2.p | p \in \text{Pos}(t)\}. \]

Given a position $p \in \text{Pos}(t)$, with $t_i$, we denote the term at position $p$ which is inductively defined by $t[x_1:=x, x.t[p:=t[p]] := x.t[i.p := t[i.p]]$ for $1 \leq i \leq n$, (letrec $x_1:s_1;\ldots;x_n:s_n$ in $t$[i.p := $t[i.p]] := x_i for 1 \leq i \leq n$, (letrec $x_1:s_1;\ldots;x_n:s_n$ in $t$[i.p := $t[i.p]] := t[i.p]. A position $p$ is a variable position of $r$ if $r[p]$ is a variable, and it is a binder position if $p = q.1$, and $r_q[p]$ is a higher-order expression of order $> 0$ or a let-rebinding. For a construct $r$, we denote with $\beta\text{Pos}(r)$ the binder positions of $r$. With $\beta V(r)$ we denote the set of bound variables of $r$, i.e. $\beta V(r) = \{r[p] | p \in \beta\text{Pos}(r)\}$. If $r[p] = x$ and $p$ is not a binder position of $r$, the occurrence of $x$ at $p$ is a bound or a free occurrence of $x$: if there exists a proper prefix $q'$ of $p$ s.t. either $q = q'$ or $q = q'.i$ and $r[q]$ is a let-rec-expression s.t. $r[q.1] = x$ and $q.1$ is a binder position, then $x$ at position $p$ is a bound occurrence, otherwise it is a free occurrence. For a bound occurrence of $x$ at $p$, its corresponding binder is $q(1)$ where $q$ is maximal. For $r$, the set of free variables is $\text{FV}(r) := \{r[p] | r_q = x$ and $x$ at position $p$ is a free occurrence\}. We set $\text{Var}(r) := \text{FV}(r) \cup \text{BV}(r)$. For an expression $r$, an $\alpha$-renaming $A : \beta\text{Pos}(r) \rightarrow \text{Var}$ computes a variable for each binder position where the following condition must hold: For each free occurrence of $x$ at position $p$ in $r$, there does not exist a prefix $q'$ of $p$ s.t. either $q = q'$ or $q = q'.i$ and $r[q]$ is a let-rec-expression s.t. $A(q.1) = x$ and $q.1$ is a binder position. Application of $A$ to $r$, written $A(r)$, replaces each binder $x$ at binder position $p$ by $A[p]$ and consistently replaces each bound occurrence of $x$ with $y$ as corresponding binder by $yA[p]$. An $\alpha$-renaming $A$ is a fresh $\alpha$-renaming for $r$ if $\text{Cod}(A) \cap \text{Var}(r) = \emptyset$ and $A[p] \neq A[p']$ whenever $p \neq p'$. The condition on $\alpha$-renamings implies that the renaming cannot capture free variables. For fresh $\alpha$-renamings, it always holds.

Example 2.4. For expression $s = \lambda x.\lambda y.\text{var} \ x$, the positions of $s$ are $\text{Pos}(s) = \{e, 1.1, 1.2, 1.2.1, 1.2.1.1, 1.2.1.2, 1.2.2.1\}$ and $s_1[1.2.1] = (x.\lambda y.\text{var} \ x)\|_{1.2.1} = (\lambda y.\text{var} \ x)\|_{1.1} = (\lambda y.\text{var})\|_{1.1} = (\text{var})\|_{1.1} = \text{var} \ x$. The positions $1.1, 1.2.1, 1.2.1.1$ are variable positions where $\beta\text{Pos}(s) = \{1.1, 1.2.1.1\}$ are binder positions, the occurrence of $x$ at position $1.2.1.2.1$ is a bound occurrence where the corresponding binder is $1.2.1.1$. The $\alpha$-renaming $A = (1.1 \mapsto x_1, 1.2.1.1 \mapsto x_2)$ is a fresh $\alpha$-renaming for $s$ and $A(s) = \lambda x_1.\lambda x_2.\text{var} \ x_2$ while $A' = (1.1 \mapsto y, 1.2.1.1 \mapsto y)$ is an $\alpha$-renaming (which is not fresh for $s$) s.t. $A'(s) = \lambda y.\lambda y.\text{var} \ y$.

For expression $s = \lambda y.\text{var} \ y$, the mapping $\{1.1 \mapsto y\}$ is not an $\alpha$-renaming, since the condition on $\alpha$-renamings is violated for the free occurrence of $y$ at position $1.2.1$.

Applying a fresh $\alpha$-renaming to an expression ensures that the distinct variable convention\footnote{Sometimes called Barendregt's variable convention.} holds for the expression.

Definition 2.5. An expression $s$ satisfies the distinct variable convention (DVC) iff $\beta V(s) \cap \text{FV}(s) = \emptyset$ and all binders bind different variables.

A position $p \in \text{Pos}(r)$ is an expression position iff $r[p] \in \text{HEExpr}$. Contexts are LRS-expressions where at one such position, the expression is replaced by the context hole $[\ ]$. We write $d[s]$ for the operation of filling the hole of context $d$ by expression $s$. With $\text{CV}(d)$ we denote the set of variables $x$ which are captured if they are plugged into the hole of $d$, i.e. if the hole of $d$ is at position $p$ then $x \notin \text{CV}(d)$ iff the occurrence of $x$ at position $p.1$ in $d[\text{var} \ x]$ is a bound occurrence. A context class $K \in \overline{K}$ is a set of contexts and a class $K$ is non-binding if for all contexts $d$ of class $K$, $\text{CV}(d) = \emptyset$.

The following lemma expresses how to iteratively construct a fresh $\alpha$-renaming. In the lemma, $\varsigma$ represents a substitution that maps variables to variables and applying $\varsigma$ to an LRS-expression means to apply $\varsigma$ to all free variable occurrences.

Lemma 2.6. The following cases show how to construct a fresh $\alpha$-renaming for $\alpha$-renamings for the direct subexpressions:

1. Let $A_i$ be fresh $\alpha$-renamings for $s_i$ for $i = 1, \ldots, n$ s.t. $\text{Cod}(A_i) \cap \text{Cod}(A_j) = \emptyset$ for all $i \neq j$. Let $A'(i.p) := A_i(p)$ for $p \in \text{Dom}(A_i)$ and $i = 1, \ldots, n$. Then $A'$ is a fresh $\alpha$-renaming for $(f \ s_1 \ldots s_n)$ and $A'(f \ s_1 \ldots s_n) = f \ A_1(s_1) \ldots A_n(s_n)$.

2. Let $A$ be a fresh $\alpha$-renaming for $s, y \notin \{x\} \cup \text{Cod}(A), s = \{x \mapsto y\}$. Let $A'(1) := y$ and $A'(2.p) := A(p)$ for all $p \in \text{Dom}(A)$. Then $A'$ is a fresh $\alpha$-renaming for $x.s$ and $A'(x.s) = y.\varsigma(A(s))$. 

3. Let $A_i$ be fresh $\alpha$-renamings for $s_i$ for $i = 1, \ldots, n+1$, s.t. $\text{Cod}(A_i) \cap \text{Cod}(A_j) = \emptyset$ for all $i \neq j \{y_1, \ldots, y_n\} \cap (\bigcup \text{Cod}(A_1) \cup \bigcup \text{Var}(s_i)) = \emptyset$, and $\varsigma = \bigcup_i^n \{x_i \mapsto y_i\}$. Let $A'(i.1) := y_i$ for $i = 1, \ldots, n$, $A'(i.2.p) := A_i(p)$ for all $p \in \text{Dom}(A_i)$ and $i = 1, \ldots, n$, $A'(n+1.p) := A_{n+1}(p)$ for all $p \in \text{Dom}(A_{n+1})$. Then $A'$ is a fresh $\alpha$-renaming for letrec $x_1.s_1; \ldots; x_n.s_n$ in $s_{n+1}$, and $A'(\text{letrec } x_1.s_1; \ldots; x_n.s_n$ in $s_{n+1}) = \text{letrec } y_1.\varsigma(A_1(s_1)); \ldots; y_n.\varsigma(A_n(s_n))$ in $\varsigma(A_{n+1}(s_{n+1}))$.

4. Let $A$ be a fresh $\alpha$-renaming for $s$ and $A'$ be a fresh $\alpha$-renaming for $d$ s.t. $\text{Cod}(A) \cap \text{Cod}(A') = \emptyset$, and $p$ be the position of the hole in $d$. Let $A'(p) := A(p)$ for $p \in \text{Dom}(A)$ and $A'(p.q) := A'(q)$ for $q \in \text{Dom}(A')$, and let $\varsigma = \{x \mapsto y \mid x \in CV(d), \text{ binder}(d[x], p) = q.1$ and $A'(q.1) = y\}$. Then $A''$ is a fresh $\alpha$-renaming for $d[s]$ and $A''(d[s]) = A(d)[\varsigma(A'(s))].$

We define two notions of equivalence. While $\sim_{\text{let}}$ extends syntactic equivalence by treating letrec-environments as sets of bindings, the relation $\sim_{\alpha}$ extends $\sim_{\text{let}}$ by allowing $\alpha$-renaming:

**Definition 2.7.** LRS-expressions $s_1, s_2$ are $\alpha$-equivalent, if there exist fresh $\alpha$-renamings $A_i$ for $s_i$, s.t. $A_1(s_1) = A_2(s_2)$. Let $\sim_{\text{let}}$ be the reflexive-transitive closure of permuting bindings in a letrec-environment and $\sim_{\alpha}$ (extended $\alpha$-equivalence) be the reflexive-transitive closure of combining $\sim_{\text{let}}$ and $\sim_{\alpha}$-equivalence.

### 2.2 The Meta-Language LRSX

The language LRSX (see Fig. 2) extends LRS by meta-variables $X$ for variables, $S$ for expressions, $E$ for environments, and $D$ for contexts where $cl(D) \in K$ denotes the context class of $D$. The semantics of meta-variables $X, Y$ are all concrete variables of type $\text{Var}$, $s$ for expressions, $E$ for environments, and $D$ for contexts respecting their types and classes. With $\text{Dom}(\rho)$ $\text{Cod}(\rho)$, resp. we denote the domain (co-domain, resp.) of $\rho$ and $\rho$ is ground iff it maps all variables in $\text{Dom}(\rho)$ to LRS-expressions.

We use the LVC and DVC as well as $\sim_{\text{let}}$ also for LRSX-expressions where the sets of variables include concrete variables as well as meta-variables representing concrete variables. We also use $\text{Var}(\cdot), BV(\cdot), \text{FV}(\cdot), LV(\cdot)$ on the extended syntax. With $MVS$ we denote the set of meta-variables occurring in $s$.

We define constraint tuples and constrained expressions:

**Definition 2.9.** A constrained LRSX-expression is a pair $(s, \Delta)$ where $s$ is an LRSX-expression, and $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ is a constraint tuple s.t. $\Delta_1$ is a finite set of context variables, called non-empty context constraints; $\Delta_2$ is a finite set of environment variables, called non-empty environment constraints; and $\Delta_3$ is a finite set of pairs $(t, d)$ where $t$ is an LRSX-expression and $d$ is an LRSX-context, called non-capture constraints (NCCs, for short). A ground substitution $\rho$ satisfies $\Delta$ (iff $\rho(D) \neq \emptyset$) for all $D \in \Delta_1$; $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$; and $\rho(D) \cap CV(\rho(d)) = \emptyset$ for all $(t, d) \in \Delta_3$. If there exists a ground substitution $\rho$ that satisfies $\Delta$, then we say $\Delta$ is satisfiable. The set of concretizations of a constrained LRSX-expression $(s, \Delta)$ is $\gamma(s, \Delta) := \{\rho(s) \mid \rho$ is ground, $\rho(s)$ fulfills the LVC, $\rho(s)$ satisfies $\Delta\}$. For an LRSX-expression $s$, we define $\gamma(s) = \gamma(s, (\emptyset, \emptyset, \emptyset))$.

**Example 2.10.** For $\Delta = (\emptyset, \Delta_2, \Delta_3)$ with $\Delta_2 = \{E_1, E_2\}$, and $\Delta_3 = \{(\text{letrec } E_1 \text{ in } c, \text{letrec } E_2 \text{ in } [\ ]\}$, the constrained expression $(\text{letrec } E_1 \text{ in } \text{letrec } E_2 \text{ in } S, \Delta)$ represents all LRS-expressions that are nested letrec-expressions s.t. both letrec-environments are non-empty and the let-variables of the inner environment are distinct from all variables occurring in the outer environment.

An example where a non-empty context constraint is required is the following reduction rule from the calculi $L_{\text{need}}$ [13] which copies an abstraction into a needed position in a letrec-environment: $\text{letrec } E; X.\lambda W.S; Y.A_1[\text{var } X] \in A[\text{var } Y] \rightarrow \text{letrec } E; X.\lambda W.S; Y.A_1[\lambda W.S] \in A[\text{var } Y]$. If $A_1$ is empty, then the target of the copy operation should be the variable $Y$ in $A[\text{var } Y]$. Thus the case $A = [\ ]$ should be excluded which can be expressed by setting $\Delta_1 = \{A_1\}$.  

\[
\xi_{U} \in \text{SAR} ::= \emptyset \mid \alpha_{U,i} : \eta \\
\eta \in \text{RS} ::= (r_{c1}, \ldots, r_{cn}), n \geq 0 \\
rc \in \text{RC} ::= \{arc_{1}, \ldots, arc_{m}\}, m \geq 0 \\
arc \in \text{ARC} ::= \alpha_{e,i} \mid LV(\alpha_{E,i}) \mid CV(\alpha_{D,i})
\]

Fig. 3: Symbolic \(\alpha\)-renamings

\[x, y, z \in \text{Var} ::= \eta \cdot X \mid \eta \cdot x\]

\[s, t \in \text{HExpr} ::= \xi_{S} \cdot S \mid \xi_{D} \cdot D[s] \mid \text{letrec env in } s \mid \{f r_{1} \ldots r_{w(f)}\} \]

where \(r_{i} \in \text{HExpr}^{\alpha}\) if \(\alpha(f)(i) = k \geq 0\), and \(r_{i} \in \text{Var}\) if \(\alpha(f)(i) = \text{Var}\).

\[s \in \text{HExpr}^{\alpha} ::= x.s_{i} \quad \text{if } s_{i} \in \text{HExpr}^{\alpha-1} \text{ and } n \geq 1\]

\[b \in \text{Bind} ::= x.s \quad \text{where } s \in \text{HExpr}^{\alpha}\]

\[\text{env} \in \text{Env} ::= \emptyset \mid \xi_{E} \cdot E; \text{env} \mid b; \text{env}\]

Fig. 4: Syntax of LRSX\(\alpha\)

3 \(\alpha\)-Renaming of Meta-Expressions

3.1 The Language LRSX\(\alpha\)

While for ground expressions, \(\alpha\)-renaming is a well-known task, our setting is different. We want to apply \(\alpha\)-renaming to the meta-expressions of LRSX, which of course cannot be computed for meta-variables until they are instantiated and become concrete expressions. Hence we have to introduce extra symbols and constructs to represent the symbolic renaming. Thus, we extend the syntax of LRSX where meta-variables \(S, D, E, X\) and variables \(x\) come with an additional symbolic \(\alpha\)-renaming, written as \(\xi \cdot S, \xi \cdot D, \xi \cdot E, \eta \cdot X, \text{ or } \eta \cdot x\), respectively.

We now define the syntax of symbolic renamings and renaming sequences.

Definition 3.1. The syntax of symbolic \(\alpha\)-renamings \(\xi\) and renaming sequences \(\eta\) is defined by the grammar given in Fig. 3. A renaming sequence \(\eta \in \text{RS}\) is a sequence of renaming components. We use list notation for sequences and write \(rc : \eta\) to split a sequence into its head \(rc\) and tail \(\eta\). A renaming component \(rc \in \text{RC}\) is a set of atomic renaming components. An atomic renaming component \(arc \in \text{ARC}\) is a symbol \(CV(\alpha_{D,i})\), or a symbol \(LV(\alpha_{E,i})\) for a context meta-variable \(D\) and an environment meta-variable \(E\), or a symbol \(\alpha_{x,i}\) where \(x\) is a concrete variable \(x\) or a meta-variable \(X\) for expression, context-, and environment-variables \(U\), a symbolic \(\alpha\)-renaming \(\xi_{U} \in \text{SAR}\) is either empty or a sequence \(\alpha_{U,i} : \eta\), and for variables \(X\) or \(x\) it is a renaming sequence \(\eta\). As notation, we write \(c\) instead of \(\{c\}\) or \(\{c_{1}, \ldots, c_{n}\}\) and \(\{c_{1}, \ldots, c_{n}\} = \{c\}\). The language LRSX\(\alpha\) (see Fig. 4) extends the syntax of LRSX by adding symbolic \(\alpha\)-renamings \(\xi\) to each occurrence of meta-variable \(S, E, D\) and renaming sequences \(\eta\) to all occurrences of concrete variables \(x\) or meta-variables \(X\). A constrained LRSX\(\alpha\)-expression is a pair \((s, \Delta)\) where \(s\) is an LRSX\(\alpha\)-expression and \(\Delta = (\Delta_{1}, \Delta_{2}, \Delta_{3})\) is a constraint tuple, \(s.t.\ \Delta_{1}\) is a set of context variables, \(\Delta_{2}\) is a set of environment variables, and \(\Delta_{3}\) is a set of pairs \((l, d)\) where \(l\) is an LRSX\(\alpha\)-expression and \(d\) is an LRSX\(\alpha\)-context.

We informally explain the meaning of symbolic \(\alpha\)-renamings. Let \(\rho\) be a ground substitution. Component \(\alpha_{U,i}\) represents a fresh \(\alpha\)-renaming of expression \(\rho(U)\) where the parameter \(i\) is required, since there may be several fresh renamings for the meta-variable \(U\). Note that \(\alpha_{U,i}\) can only occur as the first component of a sequence of renamings applied to \(U\). Components \(\alpha_{x,i}\) represent fresh renamings of variable \(\rho(x)\). Component \(CV(\alpha_{D,i})\) represents the restriction of \(\alpha_{D,i}\) to those bound variables of \(\rho(D)\) which affect the context hole. Component \(LV(\alpha_{E,i})\) represents the restriction of \(\alpha_{E,i}\) to the let-variables of \(\rho(E)\). Sets of renamings represent composed renamings where the order is irrelevant, while in sequences of renamings, the order is relevant (they have to be applied from left to right). Sets and sequences of symbolic \(\alpha\)-renamings induce a notion of equivalence of symbolic \(\alpha\)-renamings:

Definition 3.2. The relation \(\approx\) is the smallest equivalence relation satisfying \(c \approx c = \epsilon\) or an atomic renaming component \(c; (r_{c1}, \ldots, r_{cn}, \{\}, r_{c+1}, \ldots, r_{cn}) \approx (r_{c1}, \ldots, r_{ci-1}, r_{ci}, r_{ci+1}, \ldots, r_{cn})\) if \(r_{ci} \approx r_{ci}'\) for \(i = 1, \ldots, n\) then \(rc_{1}, \ldots, rc_{n} \approx (rc_{1}', \ldots, rc_{n}')\); if there exists a permutation \(\pi\) on \(\{1, \ldots, n\}\) s.t. \(arc_{i} \approx arc_{\pi(i)}\) then \(\{arc_{1}, \ldots, arc_{n}\} \approx \{arc_{1}', \ldots, arc_{n}'\}\). We do not distinguish symbolic \(\alpha\)-renamings up to \(\approx\). To embed LRSX-expressions into LRSX\(\alpha\), we identify \(\{\} \cdot U\) with \(U\) and let \(\epsilon : \text{LRSX} \rightarrow \text{LRSX}\) be the mapping that erases all renamings.

\footnote{Note that this notation is similar and also related to the notation of suspensions \(\pi \cdot X\) in nominal syntax (see e.g. \[20\]).}
We formulate the notion of well-formedness for LRSX_α-expressions which can be viewed as the side condition that in sets of renaming components there is at most one renaming component for each meta-variable or variable:

**Definition 3.3.** We say an LRSX_α-expression s is well-formed iff s does not have a renaming sequence which contains a set rc of atomic renaming components, s.t. \( \alpha_{x,i}, \alpha_{x,j} \in rc \) for some x and some \( i \neq j \), or \( LV(\alpha_{E,i}), LV(\alpha_{E,j}) \in rc \) for some E and some \( i \neq j \), or \( CV(\alpha_{D,i}), CV(\alpha_{D,j}) \in rc \) for some D and some \( i \neq j \). A constrained LRSX_α-expression \((s, \Delta)\) is well-formed, iff s is well-formed and for all \( (t, d) \in \Delta \) the expression \( t \) and the context d are well-formed.

We define the formal semantics of symbolic α-renamings.

**Definition 3.4.** Let \((s, \Delta)\) be a well-formed, constrained LRSX_α-expression and \( \rho \) be a ground substitution with \( \text{Dom}(\rho) = MV(s) \cup MV(\Delta) \) s.t. \( \rho(\epsilon(s)) \) fulfills the LVC. With \( \text{VarCod}(\rho) \) we denote the variables which appear in the co-domain of \( \rho \), i.e. \( \text{VarCod}(\rho) = \bigcup \{ \text{Var}(\rho(U)) \mid U \in \text{Dom}(\rho) \} \). A ground and fresh \( \alpha \)-renaming for \( s \) and \( \rho \) is a function \( \tau \) s.t. for all \( U \in MV(s) \), \( \tau \) maps \( \alpha_{U,i} \) to a fresh \( \alpha \)-renaming \( \tau(\alpha_{U,i}) = A_{U,j} \) for \( U \), \( \tau(\alpha_{X,i}) \) is the substitution \( \{ \rho(X) \mapsto y_{X,i} \} \) and \( \tau(\alpha_{s,i}) \) is the substitution \( \{ x \mapsto y_{s,i} \} \) where all co-domains are fresh and disjoint, i.e. \( \text{Cod}(\alpha_{U,i}) \cap \text{Cod}(\alpha_{V,j}) = \emptyset \) for \( i \neq j \) or \( U \neq V \), \( \text{Cod}(\tau(\alpha_{s,i})) \cap \text{Cod}(\tau(\alpha_{s,j})) = \emptyset \) for \( i \neq j \) or \( x \neq x' \), \( \text{Cod}(\alpha_{U,i}) \cap \text{VarCod}(\rho) = \emptyset \), \( \text{Cod}(\alpha_{s,i}) \cap \text{VarCod}(\rho) = \emptyset \), and for each environment variable E, with \( \rho(E) = x_1, s_1; \ldots; x_n, s_n \), \( \tau(\alpha_{E,i}) = A_{E,j} \), \( \tau(LV(\alpha_{E,i})) \) is the substitution \( \{ x_i \mapsto A_{E,i}(x) \mid j = 1, \ldots, n \} \) and for each context variable \( t \) with \( \rho(D) = d \) where \( p \) is the position of the hole in \( d \), \( \tau(\alpha_{D,i}) = A_{D,i}(d') \), \( \tau(CV(\alpha_{D,i})) \) is the substitution induced by \( \tau \) between \( CV(d) \) and \( CV(d') \), i.e. \( \{ x \mapsto x' \mid x \in CV(d) \} \), binder \( (d[x], p) = q \) and \( A_{D,i}(q, 1) \), and \( \tau(\{ x_1, \ldots, c_n \}) = \tau(c_1) \circ \cdots \circ \tau(c_n) \) s.t. \( \tau(c_1) \circ \cdots \circ \tau(c_n) = \tau(\epsilon(\tau(c_1) \circ \cdots \circ \tau(c_n))) \) for all permutations \( \pi \) on \( \{1, \ldots, n\} \) and \( \tau(\{ c_1, \ldots, c_n \}) \) is the composition \( \tau(c_1) \circ \cdots \circ \tau(c_n) \).

Applying \( \tau \) and \( \rho \) to \( s \) and \( \Delta \) first replaces every occurrence \( \xi_U \cdot U \) in \( s \) by \( \xi_U \cdot U(\rho) \) and then replaces \( \xi_U \) by the corresponding substitution or \( \alpha \)-renaming, i.e. by \( \tau(\xi_U)(\rho(\xi_U)) \) or \( \tau(\eta)(\rho(\eta)) \). We write \( \tau(\rho(s)), \tau(\rho(\Delta)) \) for this process. For a constrained LRSX_α-expression \((s, \Delta)\), the concretizations are:

\[
\gamma(s, \Delta) := \left\{ \begin{array}{ll}
\rho & \text{is a ground substitution for } s, \Delta \text{ s.t. } \rho(s) \text{ fulfills the LVC, } \\
\tau & \text{is a ground and fresh } \\
\alpha \text{-renaming for } s, \Delta, \rho \text{ and } \tau \circ \rho \text{ satisfies } \Delta
\end{array} \right.
\]

For LRSX_α-expressions \( s \), we define \( \gamma(s) = \gamma(s, (\emptyset, \emptyset, \emptyset)) \).

We use \( \sim_{let} \) also for LRSX_α-expressions where we allow permutation of bindings and environment variables and also allow to apply \( \approx \) to \( \alpha \)-renamings.

### 3.2 Performing Symbolic Alpha-Renaming

We describe how to perform symbolic \( \alpha \)-renaming, i.e. how to transform an LRSX-expression \( s \) into an LRSX_α-expression \( s' \), s.t. the instances of \( s' \) are \( \alpha \)-renamed copies of the instances of \( s \) (which are LRS-expressions). The algorithm to symbolically \( \alpha \)-rename \( s \), first \( \alpha \)-renames all proper subexpressions of \( s \) and then introduces a renaming for \( s \), which is then moved downwards, since it may affect occurrences of free variables in the subexpressions.

**Definition 3.5.** Let \( s \) be an LRSX-expression. The function \( AR(s) \) (using the auxiliary function \( lift \) shown in Fig. 3) computes an LRSX_α-expression for \( s \). For a constrained LRSX-expression \((s, \Delta)\), we compute a symbolically \( \alpha \)-renamed expression as \((AR(s), \Delta)\).

**Example 3.6.** We \( \alpha \)-rename the expression \( \lambda X.\lambda X.\text{var X} \):

\[
AR(\lambda X.\lambda X.\text{var X}) = \lambda AR(\lambda X.\lambda X.\text{var X}) \\
= \lambda AR(\lambda X.1.X.\text{shift} (\alpha_{X,1}, 1) \cdot AR(\lambda X.\text{var X})) \\
= \lambda AR(\lambda X.1.X.\text{shift} (\alpha_{X,1}, 1) \cdot \lambda AR(\lambda X.\text{var X} \cdot \alpha_{X,2} \cdot X)) \\
= \lambda AR(\lambda X.1.X.\text{shift} (\alpha_{X,1}, 1) \cdot \lambda AR(\lambda X.\text{var X} \cdot \alpha_{X,2} \cdot X \cdot \alpha_{X,1} \cdot X))
\]

Note that the renaming component \( \alpha_{X,1} \) in \((\alpha_{X,2} \cdot X \cdot \alpha_{X,1}) \cdot X \) can be omitted, since the renaming component \( \alpha_{X,2} \) is applied first and renames all occurrences of (instances of) \( X \). We will focus on such simplifications of symbolic \( \alpha \)-renamings in the subsequent section.
where u

If a preprocessing step of non-capture constraints, i.e. we compute so-called atomic NCCs which are pairs (αs, AR(s))

Proposition 3.8. imply:
The construction of the symbolic

same environment, then

In this example no further simplification of the symbolic renamings is possible. However, if we assume that there

et-variables of

AR(x) = (\cdot) · x

AR(S) = αS · S

AR(D[s]) = αD · D[sift(CV(αD), AR(s))]

AR(f s1 ... sn) = f AR(s1) ... AR(sn)

AR(x.s) = αx · x.s f(αx), AR(s))

AR(letrec x1,s1;...;xm,sm;E1;...;En in s) = letrec αx1,s1;...;xm,sm;E1;...;En in s

αx1,s1;...;xm,sm · x.s f(αx1), AR(s1));

... 

αxs,sm · x.s f(αxs), AR(sm));

⟨αE1,j1,η1;E1;...;αEn,jn,ηn;En⟩

in sift(η, AR(s))

where η = (\bigcup_{k=1}^{n} \{αEk,jk\}) ∪ (∪_{k=1}^{n} \{LV(αEk,jk)\})

and ηk = η \setminus LV(αEk,jk)

sift(η, x.s) = x.s f(η, s)

sift(η, f s1 ... sn) = f sift(η, s1) ... sift(η, sn)

sift(η, η′ · S) = (η′ + η) · S

sift(η, η′ · D[s]) = (η′ + η) · D[sift(η, s)]

sift(η, letrec z1,s1;...;zm,sm;η1;E1;...;ηn;En in s) = letrec z1,s1;...;zm,sm,η1;E1;...;ηn;En in s

(η + η) · E1;...;(η + η) · En

in sift′(η, s)

sift(η, x) = (η + η) · x

Fig 5: Adding symbolic α-renamings to an LRSX-expression. All αuij on right hand sides of equations are assumed to be fresh and pairwise distinct. For LRSX-expression s, AR(s) computes a symbolically α-renamed LRSXα-expression.

As a further example, we consider the symbolic α-renaming of the expression

letrec E1;E2;E3 in letrec E4 in S:

| AR(letrec E1;E2;E3 in letrec E4 in S) | = | letrec ⟨αE1,1, {LV(αE1,1), LV(αE1,1)}⟩ · E1;
| ⟨αE2,1, {LV(αE2,1), LV(αE2,1)}⟩ · E2;
| ⟨αE3,1, {LV(αE3,1), LV(αE3,1)}⟩ · E3;
| in letrec ⟨αE4,1, {LV(αE4,1), LV(αE4,1), LV(αE4,1)}⟩ · E4;
| in ⟨αS,1, {LV(αE4,1), LV(αE4,1), LV(αE4,1), LV(αE4,1)}⟩ · S |

In this example no further simplification of the symbolic renamings is possible. However, if we assume that there are non-capture constraints (letrec Ei in c, letrec Ej in []) for all i ≠ j ∈ {1, 2, 3, 4}, then in any instance the let-variables of Ei do not bind variables of Ej and thus the LRSXα-expression could be simplified to

| letrec ⟨αE1,1⟩ · E1; ⟨αE2,1⟩ · E2; ⟨αE3,1⟩ · E3; in
| letrec ⟨αE4,1⟩ · E4 in
| ⟨αS,1, {LV(αE4,1), LV(αE4,1), LV(αE4,1), LV(αE4,1)}⟩ · S |

The simplification algorithm in the subsequent section will infer those simplifications.

Lemma 3.7. If LRSX-expression s fulfills the LVC and it does not contain an environment variable E twice in the same environment, then AR(s) is well-formed.

The construction of the symbolic α-renaming and the semantics of symbolic α-renamings together with Lemma 5.6 imply:

Proposition 3.8. Let s be an LRSX-expression and s′ = AR(s). Then for each s ∈ γ(s), there exists s′ ∈ γ(s′) s.t. s ∼α s′ and for each s′ ∈ γ(s′) there exists s ∈ γ(s) s.t. s ∼α s′. Furthermore all s′ ∈ γ(s′) fulfill the strong DVC.

4 Simplification of α-Renamings

In this section we present an algorithm to simplify symbolic α-renamings. As a preparation we first consider a preprocessing step of non-capture constraints, i.e. we compute so-called atomic NCCs which are pairs (u, v) where u and v are of the form ξ · U. For a set S of NCCs, the function split<sub>α</sub> is defined by

split<sub>α</sub>(S) := \bigcup_{(u, v) \in S} \{(u, v) \mid u \in Var_M(s), v \in CV_M(d)\}
where the functions $Var_M$ and $CV_M$ are shown in Fig. 6.

Computation of $Var_M$ and $CV_M$ implies:

**Lemma 4.1.** Let $(s, d)$ be an NCC, $\rho$ be a ground substitution, and $\tau$ be a ground and fresh $\alpha$-renaming for $s, d, \rho$. Then $CV(\tau(\rho(d))) = \{\tau(\rho(x)) | x \in CV_M(d) \cup LV(\tau(\rho(\xi_E))) | E \in CV_M(d) \cup \{CV(\tau(\rho(\xi_d))) | D \in CV_M(d)\} \} \cup Var(\tau(\rho(s))) = \{Var(\tau(\rho(u))) | u \in Var_M(s)\}$.  

As a further preparation for simplification, we define notions of equivalence and subsumption for symbolic renamings and also a notion for a symbolic representation of the variables of instances.

**Definition 4.2.** The relation $=_{\text{num}}$ identifies renaming components and sequences up to the number $i$ in $\alpha_{U,i}$, i.e., it is defined by $\alpha_{U,i} =_{\text{num}} \alpha_{U,j}$, where $U$ may be $E, D, S, X, x, CV(\alpha_{D,i}) =_{\text{num}} CV(\alpha_{D,j}), LV(\alpha_{E,i}) =_{\text{num}} LV(\alpha_{E,j})$. We extend $=_{\text{num}}$ to renaming sequences $\xi_U$ and $\eta$ in the obvious way. Compared to $=_{\text{num}}$, the relation $\geq_{\text{num}}$ is defined on atomic renaming components only and it also holds if an $\alpha_{U,i}$ component is replaced by $LV(\alpha_{U,i})$ or $CV(\alpha_{U,j})$, i.e., $\geq_{\text{num}}$ is defined as $arc_1 \geq_{\text{num}} arc_2$ if $arc_1 =_{\text{num}} arc_2$, and $\alpha_{E,i} \geq_{\text{num}} LV(\alpha_{E,j})$, $\alpha_{D,i} \geq_{\text{num}} CV(\alpha_{D,j})$, for all $i, j, E, D$. A renaming $\eta_1$ is an instance of a renaming $\eta_2$ if $\eta_1 = \eta_2$ or if $\eta_1 = rc_1 : \eta'_1$ and $\eta_2 = rc_2 : \eta'_2$ if $rc_1 \subseteq w rc_2$. $\eta_1$ is an instance of $\eta_2 : \eta'_2$ if $rc_1 \subseteq rc_2$. A renaming $\eta_1$ is a weak instance of a renaming $\eta_2$ if $\eta_1 = \eta_2$ or $\eta_1 = rc_1 : \eta'_1$ and $\eta_2 = rc_2 : \eta'_2$ if $rc_1 \subseteq w rc_2$. $\eta'_1$ is a weak instance of $\eta_2 : \eta'_2$ if $rc_1 \subseteq rce_1$ and $arc =_{\text{num}} arc'$. 

**Example 4.3.** The instance relation allows to (partially) order sets of renamings, for example the renaming $\langle \alpha_{S,1}, CV(\alpha_{D,1}), \{CV(\alpha_{D,1}), LV(\alpha_{E,1})), LV(\alpha_{E,1})\rangle$ is an instance of $\langle \alpha_{S,1}, \{CV(\alpha_{D,1}), CV(\alpha_{D,1}), LV(\alpha_{E,1}), LV(\alpha_{E,1})\rangle$, $LV(\alpha_{E,1})) \rangle$ is not an instance but a weak instance of $\langle \alpha_{S,1}, \{CV(\alpha_{D,1}), CV(\alpha_{D,1}), LV(\alpha_{E,1}), LV(\alpha_{E,1})\rangle, LV(\alpha_{E,1})) \rangle$. 

**Definition 4.4.** A set $V$ of symbolic set-variables is a finite set of elements, $x, VAR(U)$, and $Cod(\text{arc})$. With $MV(V)$ we denote the meta-variables occurring in $V$ (i.e., $U \in \text{VAR}(U)$ and all meta-variables occurring as index of some arc in $\text{Cod}(\text{arc})$). For a set $MV$ of meta-variables with $MV \subseteq MV(V)$, a ground substitution $\rho$ for $MV$ and a ground $\alpha$-renaming $\tau$ for $\rho$ and $MV$, we define $\tau(\rho(V)) := \bigcup_{x \in V} \tau(\rho(x))$ where $\tau(\rho(\text{VAR}(U))) := \text{Var}(\rho(U))$, $\tau(\rho(x)) = \{\rho(x)\}$, and $\tau(\rho(\text{Cod}(\text{arc}))) = \text{Cod}(\tau(\text{arc}))$.

Simplification removes renaming components if they cannot have an effect on the corresponding meta symbol. Information is gathered from the renamings and from the NCCs in $\Delta_3$.

**Definition 4.5 (Simplification Algorithm).** Let $(s, \Delta)$ be a constrained LRSX$\alpha$-expression. The simplification algorithm replaces occurrences $\xi \cdot U$ ($\eta \cdot x$, resp.) in $s$ by $\xi' \cdot U$ ($\eta' \cdot x$, resp.) if $\xi \cdot U \models_{\Delta} \xi' \cdot U$ ($\eta \cdot x \models_{\Delta} \eta' \cdot x$, resp.) can be inferred by the inference rules shown in Fig. 7(a). In the premises some of the rules use sets $V$ of symbolic set-variables occurring in judgments $V, \eta \models_{\Delta} \eta'$ which are defined by the rules shown in Fig. 7(b) and the predicate $\text{arc} \not\models_{\Delta} v$ which is defined in Fig. 7(c).

Axioms (IdX), (IdU), and (IdEta) allow to keep the renaming and rules (TrX) and (TrU) enable transitivity of simplification. Rule (RemDup) removes a duplicated renaming component in a sequence. Rule (SubstX) removes
\begin{align*}
\text{(IdU)} & \quad \xi U \vdash_\Delta \xi U \\
\text{(TrU)} & \quad \xi_1 U \vdash_\Delta \xi_2 U \text{ and } \xi_2 U \vdash_\Delta \xi_3 U \\
\text{(SimU)} & \quad \{ \text{VAR}(U), \text{Cod}(\alpha_U) \}, \eta \vdash_\Delta \eta' U \neq x \\
\text{(RemDup)} & \quad \eta_1 \vdash_\Delta \eta x \\
\text{(SimNCC)} & \quad (\xi U, x) \in \text{split}_\Delta(\Delta_3), \xi_U \text{ is a weak instance of } \xi_U, \xi_U \vdash_\Delta \xi_U' \neq y \\
\text{(Cod)} & \quad \alpha \neq \Delta \text{ and } \text{Cod}(\alpha) \\
\text{(EmCV)} & \quad \text{cl}(D) \text{ is non-binding} \\
\text{(NccEU)} & \quad (U, E) \in \text{split}_\Delta(\Delta_3) \\
\text{(NccEX)} & \quad \text{LV}(\alpha_E) \neq \Delta \text{ and } \text{VAR}(U) \\
\text{(NccUX)} & \quad (U, x) \in \text{split}_\Delta(\Delta_3) \\
\text{(NccDX)} & \quad \text{LV}(\alpha_x) \neq \Delta \text{ and } \text{VAR}(U) \\
\end{align*}

\text{(a) Judgments } \xi U \vdash_\Delta \xi U \text{ and } \eta x \vdash_\Delta \eta' x \text{ mean that LRSMa-expression } \xi U (\eta x, \text{ resp.) can be simplified to } \xi U (\eta' x, \text{ resp.).}

\text{(b) Judgment } V, \eta \vdash_\Delta \eta' \text{ means that for the variables represented by } V, \eta \text{ can be simplified to } \eta'.

\text{(c) The predicate } \alpha \neq \Delta v \text{ holds iff } \alpha \text{ cannot rename the variables represented by } v

Fig. 7: Simplification of symbolic } \alpha \text{-renamings}
further renaming components for a renaming for $x$ if the first component is $\alpha_{x,i}$. Rule (SimX) performs simplification of symbolic $\alpha$-renamings applied to $\times$ or $X$-variables, where the symbolic set of variables in the premise is the singleton containing the to-be-simplified variable. Rule (SimU) perform simplification for meta-variables $U$ which are not $X$-variables. Hence the $\alpha$-renaming starts with $\alpha_{U,j}$ and the symbolic set of variables consists of $\text{VAR}(U)$ and the co-domain of $\alpha_{U,j}$. Rules (SimNCCU) and (SimNCCX) allow to remove a component $\alpha_{x,i}$ if an NCC ensures that $x$ cannot occur in $\xi_j$. $U$ or $\eta'_i$, resp. Rule (RMarc) removes the first atomic renaming component of a sequence of components provided that it cannot rename any variable represented by the symbolic set of variables. Rule (Parc) processes the first renaming component in a sequence, by adding the co-domain of the component to the symbolic set of variables, and then proceeds with the tail of the sequence. Rule (Order) allows to order a set of atomic renaming components for further simplification, rule (MSet) allows to transform a sequence of atomic renaming components $\alpha_{x,i}$ into a set of components provided that it is guaranteed that the ground instances of all variables $x_i$ are pairwise different. The predicate $\neq_\Delta$ is defined in Fig. 7(c) where $arc \neq_\Delta v$ expresses that atomic renaming component $arc$ cannot rename the set of variables represented by $v$. The rules use the NCCs or some other easy fact to ensure that the property holds.

**Example 4.6.** We reconsider the expressions from Example 3.6 Applying the simplification algorithm to the constrained expression $(\lambda x_1 \cdot X. \lambda x_2 \cdot X. \text{var} (\alpha_{x,2} \cdot X, (\emptyset, \emptyset, \emptyset))$ results in $(\lambda x_1 \cdot X. \lambda x_2 \cdot X. \text{var} (\alpha_{x,2} \cdot X, (\emptyset, \emptyset, \emptyset))$ since

As a further example, consider $(s, \Delta) = (s, (\emptyset, \Delta_2, \Delta_3))$ with

$$s = \text{letrec } \langle \alpha_{E,1}, \{LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot E_1;$$
$$\langle \alpha_{E,1}, \{LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot E_2;$$
$$\langle \alpha_{E,1}, \{LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot E_3;$$
$$\text{in letrec } \langle \alpha_{E,1}, \{LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot E_4;$$
$$\text{in } \langle \alpha_{E,1}, LV(\alpha_{E,1}), LV(\alpha_{E,1}), LV(\alpha_{E,1}) \rangle \cdot E_5;$$

$$\Delta_2 = \{E_1, E_2, E_3, E_4\}$$
$$\Delta_3 = \{\text{letrec } E_i \text{ in } c, \text{letrec } E_j \text{ in } [\cdot] \mid i,j \in \{1,2,3,4\}, i \neq j\}$$

Applying the simplification algorithm results in $(s', \Delta)$ with

$$s' = \text{letrec } \langle \alpha_{E,1}, \rangle \cdot E_1; \langle \alpha_{E,1}, \rangle \cdot E_2; \langle \alpha_{E,1}, \rangle \cdot E_3; \text{in }$$
$$\text{letrec } \langle \alpha_{E,1}, \rangle \cdot E_4 \text{ in }$$
$$\langle \alpha_{E,1}, LV(\alpha_{E,1}), \{LV(\alpha_{E,1}), LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot S$$

since $\langle \alpha_{E,1}, \{LV(\alpha_{E,1}), LV(\alpha_{E,1})\} \rangle \cdot E_i \models_\Delta \langle \alpha_{E,1} \rangle \cdot E_i$ can be derived for all $i,j$ with $\{i,j,k\} = \{1,2,3\}$ (see Fig. 8).

We show correctness of the simplification algorithm by proving correctness of the inference rules:

**Proposition 4.7.** Let $M$ be a set of meta-variables and $\Delta$ be a constraint tuple where $MV(\Delta) \subseteq M$. Let $\rho$ be ground substitution for $M$ and $\tau$ be a ground $\alpha$-renaming for $\rho$ and $M$, s.t. $\rho$ and $\tau$ satisfy $\Delta$.

1. (Correctness of $\neq_\Delta$) Let $v$ be a symbolic set-variable and $arc$ be an atomic renaming component (over $M$), s.t. $arc \neq_\Delta v$. Then for each $x \in \tau(\rho(v))$, the identity $\tau(\text{arc}(x)) = x$ holds.
2. (Correctness of $\models_\Delta$) Let $V$ be a set of symbolic set-variables and $\eta$ be a sequence of renaming components with components over $M$, s.t. $V, \eta \models_\Delta \eta'$. Then for each $x \in \tau(\rho(V))$, we have $\tau(\eta)(x) = \tau(\eta')(x)$. 

11
3. (Correctness of $\vdash_{\Delta}$)
   (a) Let $\eta, \eta'$ be symbolic $\alpha$-renamings with components over $M$, s.t. $\eta x \vdash_{\Delta} \eta' x$. Then $\tau(\eta)(\rho(x)) = \tau(\eta')(\rho(x))$.
   (b) Let $\xi, \xi'$ be symbolic $\alpha$-renamings with components over $M$, and let $U \in M$ s.t. $\xi U \vdash_{\Delta} \xi' U$. Then $\tau(\xi)(\rho(U)) = \tau(\xi')(\rho(U))$.

Proof. For part $\Box$, we inspect all rules in Fig. 2 (c). For rule (Cod), the claim holds, since $\tau(\rho(v)) = \tau(\rho(Cod(\alpha c_i')))) = Cod(\tau(\alpha c_i))$ is a set of fresh variables which cannot be renamed by $\tau(\alpha c_i)$.

For rule (EmCV), the claim holds, since $Dom(\tau(CV(\alpha c_i))) = \emptyset$ if $D$ is non-binding.

For rule (NccDU), the premise ensures that $CV(\rho(D)) \cap Var(\rho(U)) = \emptyset$ and with $Dom(\tau(CV(\alpha c_i))) = CV(\rho(D))$ this implies that the equation $Dom(\tau(CV(\alpha c_i))) \cap Var(\rho(U)) = \emptyset$ holds.

For rule (NccEU), the premise ensures that $CV(\rho(D)) \cap \emptyset = \emptyset$ and since $Dom(\tau(CV(\alpha c_i))) = CV(\rho(D))$ this shows that the equation $Dom(\tau(CV(\alpha c_i))) \cap \emptyset = \emptyset$ holds.

For rule (NccEX), we have $LV(\rho(E)) \cap \emptyset = \emptyset$ by the premise and with $Dom(\tau(LV(\alpha c_i))) = LV(\rho(D))$ this shows that the equation $Dom(\tau(LV(\alpha c_i))) \cap \emptyset = \emptyset$ holds.

For rule (NccUX), the premise ensures that $\rho(x) \cap Var(\rho(U)) = \emptyset$ and since $Dom(\tau(\alpha c_i)) = \emptyset$ this shows $Dom(\tau(\alpha c_i)) \cap Var(\rho(U)) = \emptyset$.

For rule (NccXX), the premise ensures that $\rho(x) \cap \rho(x')$ and since $Dom(\tau(\alpha c_i)) = \emptyset$ this shows $Dom(\tau(\alpha c_i)) \cap \emptyset = \emptyset$.

For part $\Box$, we inspect the inference rules and use an induction on the height of the derivation tree. The induction base is covered by rule (IdEta) which is obviously correct. Otherwise, we inspect the final rule which is applied in the derivation:

For rule (RMarc), the condition $\forall v \in V : arc \not\in \Delta v$ and part $\Box$ ensure that $\tau([arc] \cup rcc)(\eta(x)) = \tau(rcc)(\eta(x))$ for $x \in \tau(\rho(V))$.

For rule (Parc), the induction hypothesis shows that $\tau(\eta(x)) = \tau(\eta')(x)$ for all $x \in \tau(\rho(V))$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_i, \ldots, \alpha c_n\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.

For rule (RSet), the premise ensures that $\tau(\{\alpha c_{i}, \ldots, \alpha c_{n}\}) = \tau(\{\alpha c_1, \ldots, \alpha c_n\})$ for all permutations $\pi$ on $\{1, \ldots, n\}$: all variables $x_i$ are pair wise different, all variables $\alpha c_{\pi(i)} \not\in x_j$ for all $i \not= j$.
Applying the previous proposition for all occurrences $\eta \cdot x$ and $\xi \cdot U$ which are transformed by the simplification algorithm shows:

**Theorem 4.8.** The simplification algorithm does not change the set of concretizations, i.e. for a constrained LRSX$\alpha$-expression $(s, \Delta)$ s.t. $s$ fulfills the LVC and $s$ does not contain an environment variable twice in the same environment, the simplified expression $(s', \Delta)$, we have $\gamma(s, \Delta) = \gamma(s', \Delta)$.

5 Algorithms for LRSX$\alpha$-Expressions

In this section we show how to perform rewriting of LRSX$\alpha$-expressions by matching LRSX$\alpha$-expressions to apply rewrite steps, and by refreshing the $\alpha$-renaming to guarantee that the distinct variable convention holds after applying a rewrite step. We finally present an algorithm to test extended $\alpha$-equivalence of LRSX$\alpha$-expressions which, for instance, is necessary during diagram computation to check whether a diagram is closed.

5.1 Rewriting Meta-Expressions

Meta letrec rewrite rules (see [14]) are rewrite rules of the form $\ell \to_\Delta r$ where $\ell$ and $r$ are LRSX-expressions and $\Delta$ is a constraint tuple. Applying a rewrite rule to a constrained expression $(s, \nabla)$ consists of matching $\ell$ against $s$ s.t. the constraints in $\nabla$ imply the constraints in $\Delta$. Given a matcher (i.e. a substitution $\sigma$ with $\sigma(\ell) \sim_{\text{let}} s$) the reduction is $s \to \sigma(r)$ (or more precisely $(s, \nabla \cup \sigma(\Delta))$). In [14] the letrec matching problem was defined and analyzed for LRSX-expressions. However, as argued before, often transformations are not applicable, since $\nabla$ does not imply $\Delta$ (see the example for an (llet) overlap in Sect. [3]). Here $\alpha$-renaming of $s$ often helps to satisfy the constraints. Hence, we formulate an adapted form of a letrec matching problem where $(s, \nabla)$ is a constrained LRSX$\alpha$-expression. Our matching equations are of the form $\ell \leq s$ where $s$ is a meta-expression with instantaneous meta-variables and $\ell$ is meta-expression with meta-variables that act like constants. In addition $\ell$ may contain symbolic $\alpha$-renamings (i.e. $\ell$ is an LRSX$\alpha$-expression), but $s$ is an LRSX-expression. To distinguish the meta-variables we use blue font for instantaneous meta-variables and red font and underlining for fixed meta-variables. With $MV_I(\cdot)$ and $MV_F(\cdot)$ we denote functions to compute the sets.

**Definition 5.1.** A letrec matching problem with $\alpha$-renamed expressions is a tuple $P = (\Gamma, \Delta, \nabla)$ where $\Gamma$ is a set of matching equations $s \leq t$ s.t. $s$ is an LRSX-expression, $t$ is an LRSX$\alpha$-expression, $MV_I(t) = \emptyset; \Delta = (\Delta_1, \Delta_2, \Delta_3)$ is a constraint tuple over LRSX, called given constraints, where $\nabla$ is a constraint tuple over LRSX$\alpha$, called given constraints, where $MV_I(\nabla_i) = \emptyset$ for $i = 1, 2, 3$ and $\nabla$ is satisfiable; and for all expressions in $\Gamma$, the LVC must hold. The following occurrence restrictions must hold: every variable of kind $S$ occurs at most twice in $\Gamma$; every variable of kind $E$ or $D$ occurs at most once in $\Gamma$. A matcher $\sigma$ of $P$ is a substitution s.t. for any ground substitution $\rho$ together with a ground renaming $\tau$ with $\text{Dom}(\rho) = MV_F(P)$ s.t. $\tau \circ \rho$ satisfies $\nabla, \tau(\rho(\sigma(s))) \sim_{\text{let}} \tau(\rho(t))$ fulfill the LVC for all $s \leq t \in \Gamma$, we have $\tau(\rho(\sigma(s))) \sim_{\text{let}} \tau(\rho(t))$ for all $s \leq t \in \Gamma$, and there exists a ground substitution $\rho_0$ with $\text{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ s.t. $\tau(\rho_0(\rho(\sigma(\Delta))))$ is satisfied.

The letrec matching problem (with LRSX-expressions, only) and corresponding matchers for LRSX-expressions are defined analogously but all expressions are LRSX-expressions, and no ground renaming $\tau$ is involved. The additional substitution $\rho_0$ in the definition of a matcher is used for the case that rewrite rules $\ell \to_\Delta r$ introduce meta-variables, i.e. if there are meta-variables which occur in $r$ but not in $\ell$. Then the existence of $\rho_0$ ensures that always a ground instance can be constructed. An example for a rewrite rule which introduces meta-variables is the rule (abs) which shares the arguments of a function symbol application: $(f \ s_1 \ldots s_n) \to_\Delta \text{letrec } X_1.s_1;\ldots;X_n.s_n \in (f \ (\text{var} \ X_1) \ldots (\text{var} \ X_n))$ where $\Delta$ contains NCCs that ensure that $X_1,\ldots,X_n$ are fresh w.r.t. $s_1,\ldots,s_n$.

In [14] a sound and complete matching algorithm for the letrec matching problem (with LRSX-expressions, only) is given. This algorithm takes a letrec matching problem as input and computes a constructed solution $\text{Sol}_F$ and in a final step it checks whether the given constraints in $\nabla_3$ imply the required constraints in $\Delta_3$. Except for this final step the algorithm can be reused to solve the letrec matching problem for LRSX$\alpha$-expressions and computing matchers as follows: Let $(\Gamma', \Delta, \nabla')$ be a letrec matching problem with $\alpha$-renamed expression. Transform the LRSX$\alpha$-expressions on right-hand sides of $\Gamma'$ and in $\nabla'$ into LRSX-expressions by replacing all occurrences $\xi \cdot U, \xi' \cdot U'$ with $\xi \sim \xi'$ by a single fresh fixed meta-variable $U'$ (of the same kind as $U$) and by replacing $\eta \cdot x, \eta' \cdot x$ with $\eta \approx \eta'$ by a fresh variable $x'$. Now apply the matching algorithm for LRSX of [14] until a solution $(\text{Sol}_F, \Delta_F, \nabla_F)$ is produced. Then construct $(\text{Sol}_O, \Delta_O, \nabla_O)$ by replacing $U'$ by $\xi \cdot U$ and $x'$ by $\eta \cdot x$ in $(\text{Sol}_F, \Delta_F, \nabla_F)$. Now the following check whether $\Delta_O$ implies $\nabla_O$ is performed. If it succeeds, then $\text{Sol}_O$ is delivered as a matcher.
For a ground substitution \( \rho \) and a ground \( \alpha \)-renaming \( \tau \) for \( \rho \), let \( CV_A(\tau(p(\rho(x)))) := \{\tau(\eta)(\rho(x))\} \), \( CV_A(\tau(p(\rho(x)))) := CV(\tau(\xi)(\rho(D))) and CV_A(\tau(p(\rho(E)))) := LV(\tau(\zeta)(\rho(E))) \). Note that for an NCC \((s,d)\), \( \text{Var}(\tau(p(s))) = \{\text{Var}(\tau(p(\xi(u)))) | \xi(u) \in \text{Var}_{\text{LV}}(s)\} and CV(\tau(p(d))) = \{CV_A(\tau(p(\xi(u)))) | \xi(u) \in CV_M(s)\} \) which justifies to work with the split NCCs.

Lemma 5.3. Assume that \( \Delta \) implies \( \nabla \). Let \( \rho \) be a ground substitution for \( MV_F(\nabla) \) and \( \tau \) be a ground renaming for \( \rho \), s.t. \( \tau \circ \rho \) satisfies \( \nabla \). Then there exists a ground substitution \( \rho_0 \) with \( \text{Dom}(\rho_0) = MV_F(\rho(\Delta)) \) s.t. \( \tau(\rho_0(\rho(\Delta))) \) is satisfied.

Proof. Let \( (\xi,u,\xi'v) \in \text{split}_{\nabla}(\Delta) \) s.t. one of the cases of the implication check applies. We consider the different cases and use the following instantiation \( \rho_0 \) for instantiable meta-variables: \( \rho_0(S) := \lambda x_S.X_S \) for a fresh variable \( x_S \); \( \rho_0(D) := 0 \) if \( D \not\in \Delta_1 \), and \( \rho_0(D) = d \) where \( d \) is a context with \( CV(d) = 0 \); \( \rho_0(E) := 0 \) if \( E \not\in \Delta_2 \) and \( \rho_0(E) = e \) where \( e \) is a fresh variable; \( \rho_0(X) := x_X \) for a fresh variable \( x_X \).

In case 1, \( \xi,u,\xi'v \in \text{split}_{\nabla}(\Delta) \) satisfies the constraint. In case 2, \( \xi,u,\xi'v \in \text{split}_{\nabla}(\Delta) \) implies \( \text{Var}(\tau(p(\rho(u)))) \cap CV_A(\tau(p(\rho(u)))) = 0 \). For case 3, assume that \( u \neq v \) and \( u = D \) or \( u = E \), or \( u = X \), then \( \text{Var}(\tau(p(\rho(u)))) = \{\tau(\xi)(\rho(\xi(u)))\} \) contains only fresh variables and these variables must be disjoint from \( CV_A(\tau(p(\xi)(\rho(\xi(u))))) \), since the variables in \( \rho_0(\rho(u)) \) must be pairwise distinct from the variables in \( \rho_0(\rho(u)) \).

For case 4, assume that \( u \neq v, v = D \), or \( v = E \), or \( v = X \). Then \( \rho_0(v) = v \) and \( CV_A(\rho_0(v)) \) contains only fresh variables which cannot occur in \( \rho_0(v) \). and thus \( CV_A(\tau(p(\xi)(\rho(\xi(u))))) \cap \text{Var}(\tau(\xi)(\rho_0(u))) = 0. \)
For case \(6\), let \(\xi' = (\cdot), v = F\) or \(v = D\) and \((v, v) \in \text{split}_\varnothing(\nabla_3)\). Then \(\text{Var}(\rho(v)) \cap CV_\alpha(\rho(v)) = \emptyset\) must hold, which requires that \(\rho(v) = \emptyset\) (for \(v = F\), \(\rho(v) = d\) with \(CV_\alpha(d) = \emptyset\) (for \(v = D\)). In all cases \(CV_\alpha(\rho(v)) = \emptyset\) and thus \(\text{Var}(\tau(\rho(\xi'v))) \cap CV_\alpha(\rho(\xi'v)) = \emptyset\).

For case \(7\), \(\xi = (\cdot), (v, u) \in \text{split}_\varnothing(\nabla_3) \cup \text{NCC}_{\text{rec}}\), and \((u, v)\) is of the form \((X, y), (x, Y), (x, D), (X, D), (x, E), (X, E)\). It suffices to show that \(\text{Var}(\rho(u)) \cap CV_\alpha(\rho(u)) = \emptyset\) implies \(\text{Var}(\rho(u)) \cap CV_\alpha(\rho(v)) = \emptyset\).

Matching can be used to rewrite constrained symbolically \(\alpha\)-renamings. We choose a simpler approach that uses the existing \(\alpha\)-renamings and refreshes them: let \(\text{refresh}\) (or \(\text{refresh}_{\alpha\beta}\)) number all occurrences of \(\alpha\) which violates the DVC for instances of the expression. An approach to deal with this problem could be to generalize the symbolic \(\alpha\)-renamings to again symbolically \(\alpha\)-rename the expressions. However, this approach seems to be not easily tractable (for instance, this would require to introduce renaming components of the form \(\alpha_\xi^\varnothing, S\), which represents an \(\alpha\)-renaming of already \(\alpha\)-renamed expressions). We choose a simpler approach that uses the existing \(\alpha\)-renamings and refreshes them:

**Definition 5.6 (Refreshing Alpha-Renamings).** A renumbering of a symbolic \(\alpha\)-renaming modifies \(\alpha_{U,i}\) components by replacing \(\alpha_{U,i}^\varnothing\) (or \(\alpha_{x,i}\), resp.) with \(\alpha_{U,i}^\varnothing\) (or \(\alpha_{x,i}\), resp.) where \(j\) is a fresh number. Let \((s, \Delta)\) be a constrained LRSX-expression. The function \(\text{refresh}(s, \nabla)\) renumbers all occurrences of \(\alpha_{U,i}\) and replaces \(CV(\alpha_{U,i})\) with \(CV(\alpha_{U,i})\) and \(LV(\alpha_{U,i})\) with \(LV(\alpha_{U,i})\) respecting the scopes. For bound variables \(\{x\} \cdot x\) or \(\{\ell\} \cdot \ell\) \(U\) it introduces a fresh \(\alpha\)-renaming \(\alpha_{x,i}\) or \(\alpha_{x,i}\), and adds it to the meta-variable and sifts the corresponding renaming downwards, analogous to \(AR\) and sift shown in Fig. 5.

**Proposition 5.7.** Let \((s, \Delta)\) be a constrained LRSX-expression and \((s', \Delta) = \text{refresh}(s, \Delta)\). Then for each \(s \in \gamma(s, \Delta)\) there exists \(s' \in \gamma(s', \Delta)\) with \(s \sim_\alpha s'\) and for each \(s' \in \gamma(s', \Delta)\) there exists \(s \in \gamma(s, \Delta)\) with \(s \sim_\alpha s'\).

**Proof.** Replacing \(\alpha_{x,i}\) and \(\alpha_{x,i}\)-renamings by fresh copies implies that the corresponding ground \(\alpha\)-renamings use new sets of variables in their co-domain, which is due to the consistent replacement, also consistent for the concretizations.

### 5.3 Checking \(\alpha\)-Equivalence

We finally provide a test for checking extend \(\alpha\)-equivalence.
We presented an extension of the meta-language LRSX transformation for overlap closures and the correctness of program transformations (16 transformations for L). The strategy of the LRSX Tool is to avoid \(\alpha\)-reductions. Due to branching in unjoinable cases, the number of joins is higher than the number of overlaps. Note renaming procedure. The row marked with LR[19] (which extends LRSX) by data constructors for lists, booleans and pairs together with corresponding need renaming for \(\rho\). Let equivalence check:

\[
\tau \circ \rho \text{ satisfies } \Delta \iff \tau \circ \rho \text{ satisfies } \Delta' \text{ and } \tau(\rho(s)) \sim_{\alpha} \tau(\rho(s')).
\]

6 Experiments

The LRSX Tool (available from http://goethe.link/LRSXTool) tries to automatically prove correctness of transformations by the diagram method. After computing the overlaps, it tries to join them by applying letrec rewrite steps and symbolic \(\alpha\)-renaming. We tested the LRSX Tool with the calculus \(L_{\text{needs}}[18]\), and the calculus LR[19] (which extends \(L_{\text{needs}}\) by data constructors for lists, booleans and pairs together with corresponding case-expressions, and seq-expressions and thus represents an untyped core language of Haskell). Table 1 shows the numbers of computed overlaps, corresponding joins, and the number of those joins which use the alpha-renaming procedure. The row marked with \(\rightarrow\) represent the overlaps between left hand sides of transformations and standard reductions, while \(\leftarrow\) represent the overlaps between right hand sides of transformations and standard reductions. Due to branching in unjoinable cases, the number of joins is higher than the number of overlaps. Note that the strategy of the LRSX Tool is to avoid \(\alpha\)-renamings, and thus \(\alpha\)-renaming is applied only, if no join was found before without performing renaming. The results show that \(\alpha\)-renaming is necessary in about 20 percent of the cases (except for overlaps of left hand sides in the calculus \(L_{\text{needs}}\)). With the help of \(\alpha\)-renaming all computed overlaps could be closed and the correctness of program transformations (16 transformations for \(L_{\text{needs}}\) and 43 transformation for LR) could be shown automatically.

7 Conclusion

We presented an extension of the meta-language LRSX by symbolic \(\alpha\)-renamings. We introduced algorithms for simplification of renamings, matching, reduction, and checking extended \(\alpha\)-equivalence. The algorithms are implemented and used in the LRSX Tool, and our experiments show that the approach for \(\alpha\)-renaming is successful in automatically proving correctness of program transformations. Further work is to use the approach in other inference procedures and to investigate whether it can be adapted for nominal techniques.
# overlaps # meta joins # meta joins with α-renaming
Calculus $L_{\text{need}}$
\[\rightarrow\] 2242 3425 93
\[\leftarrow\] 3001 7273 1402
Calculus LR
\[\rightarrow\] 87041 391264 73601
\[\leftarrow\] 107333 429104 93230

Table 1: Statistics of executing the LRSX Tool

References