Institut für Informatik
Fachbereich Informatik und Mathematik

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Alpha-Renaming of Higher-Order Meta-Expressions

David Sabel

2017

Frankfurter Informatik-Berichte
Institut für Informatik ● Robert-Mayer-Str. 11-15 ● D-60325 Frankfurt am Main, Germany

ISSN 1868-8330

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 [2,1,19]). One technique to establish such proofs for program calculi with small-step operational semantics is the diagram method [19,15] 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[1] to show correctness of the transformation w.r.t. contextual equivalence [9,12]. This diagram method was e.g. used in [19,15] and similar techniques are in [21,8,7], 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 [14].

The language LRSX uses higher-order abstract syntax [10] extended with a letrec-construct $\texttt{letrec}\ \mathsf{x}_1 = \mathsf{s}_1; \ldots; \mathsf{x}_n = \mathsf{s}_n\ \texttt{in}\ \mathsf{s}_{n+1}$ to represent unordered sets of recursive bindings (where the scope of the letrec-bound variables x_i is $\mathsf{s}_1, \ldots, \mathsf{s}_{n+1}$) 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. [2,1,19,18]).

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 (llet):

$$C[\texttt{letrec}\ E_1\ \texttt{in}\ \texttt{letrec}\ E_2\ \texttt{in}\ S] \xrightarrow{llet} C[\texttt{letrec}\ E_1; E_2\ \texttt{in}\ S]$$

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[3] $\mathsf{x} = \mathsf{y}$, meta-variable E_2 by an environment which consists of a single binding $\mathsf{y} = \texttt{True}$, 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.link/LRSXTOOL

[3] Later in this paper these bindings are written as $\mathsf{x}.\texttt{var}\ \mathsf{y}$, since "." is used instead of "=" and the function symbol var is necessary to lift variables to expressions.

$\texttt{letrec}\ \mathsf{x} = \mathsf{y}\ \texttt{in}\ \texttt{letrec}\ \mathsf{y} = \texttt{True}\ \texttt{in}\ \mathsf{x} \to \texttt{letrec}\ \mathsf{x} = \mathsf{y}; \mathsf{y} = \texttt{True}\ \texttt{in}\ \mathsf{x}$ which however should be forbidden, since variable y in $\mathsf{x} = \mathsf{y}$ is a free occurrence in the left expression, but becomes a bound occurrence (captured by the binding $\mathsf{y} = \texttt{True}$) in the right expression. So-called non-capture constraints in constrained expressions enable us to forbid those instantiations. They are pairs (s, d) where s is a meta-expression, d is a meta-context and they are satisfied by a ground instantiation ρ if context $\rho(d)$ does not capture any variable of $\rho(s)$. For our example, the constraint $(s_0, d_0) = (\texttt{letrec}\ E_1\ \texttt{in}\ \texttt{True}, \texttt{letrec}\ E_2\ \texttt{in}\ [\cdot])$ suffices.

In turn, if during computing joins, expressions occur which violate the constraints, then in some cases the diagram calculation fails. For instance, consider the overlap of (llet) with itself and a suggested join (written using dahed arrows):

$$\begin{tikzcd}
\begin{array}{l}\texttt{letrec}\ E_1\ \texttt{in}\\ \quad\texttt{letrec}\ E_2\ \texttt{in}\\ \qquad\texttt{letrec}\ E_2'\ \texttt{in}\ S\end{array} \arrow[r, "llet"] \arrow[d, "llet"'] & \begin{array}{l}\texttt{letrec}\ E_1; E_2\ \texttt{in}\\ \quad\texttt{letrec}\ E_2'\ \texttt{in}\ S\end{array} \arrow[d, dashed, "llet"] \\
\begin{array}{l}\texttt{letrec}\ E_1\ \texttt{in}\\ \quad\texttt{letrec}\ E_2; E_2'\ \texttt{in}\ S\end{array} \arrow[r, dashed, "llet"'] & \texttt{letrec}\ E_1; E_2; E_2'\ \texttt{in}\ S
\end{tikzcd}$$

As explained before, (llet) is constrained by the non-capture constraint (s_0, d_0). For the step from the upper-left expression to the upper-right expression, the constraint ensures that the binders of E_2 do not capture variables of E_1, and for the step from the upper-left expression to the lower-left expression, the constraint ensures that the binders of E_2' do not capture variables of E_2. However, for closing the overlap the step from the lower-left expression to the lower-right expression requires the knowledge that binders of $E_2; E_2'$ do not capture variables of E_1 and the step from the upper-right to the lower-right expression requires the knowledge that binders of E_2' do not capture variables of E_1, E_2. In both cases the required knowledge cannot be inferred from the given knowledge and thus the suggested join cannot be computed. Moreover, there are indeed concrete instances which forbid the suggested join, for example, with $\rho = \{E_1 \mapsto \mathsf{x} = \mathsf{z}, E_2 \mapsto \mathsf{y} = \texttt{True}, E_2' \mapsto \mathsf{z} = \texttt{True}, S \mapsto \mathsf{x}\}$, the suggested join would lead to $\rho(\texttt{letrec}\ E_1; E_2; E_2'\ \texttt{in}\ S) = \texttt{letrec}\ \mathsf{x} = \mathsf{z}; \mathsf{y} = \texttt{True}; \mathsf{z} = \texttt{True}\ \texttt{in}\ \mathsf{x}$ which illegally captures the variable z.

The solution to attack those problems in a pen-and-paper-proof is to rename binders by fresh α-renamings. For the above instance, we may α-rename $\texttt{letrec}\ \mathsf{x} = \mathsf{z}; \mathsf{y} = \texttt{True}\ \texttt{in}\ \texttt{letrec}\ \mathsf{z} = \texttt{True}\ \texttt{in}\ \mathsf{x}$ into $\texttt{letrec}\ \mathsf{x} = \mathsf{z}; \mathsf{y} = \texttt{True}\ \texttt{in}\ \texttt{letrec}\ \mathsf{z}' = \texttt{True}\ \texttt{in}\ \mathsf{x}$ and α-rename $\texttt{letrec}\ \mathsf{x} = \mathsf{z}\ \texttt{in}\ \texttt{letrec}\ \mathsf{y} = \texttt{True}; \mathsf{z} = \texttt{True}\ \texttt{in}\ S$ into $\texttt{letrec}\ \mathsf{x} = \mathsf{z}\ \texttt{in}\ \texttt{letrec}\ \mathsf{y} = \texttt{True}; \mathsf{z}' = \texttt{True}\ \texttt{in}\ S$ and then apply the (llet)-transformations of the suggested join. The goal of this paper is to perform such renamings on the meta-level (and not on the (infinitely many) concrete instances). Thus we want to rename $\texttt{letrec}\ E_1; E_2\ \texttt{in}\ \texttt{letrec}\ E_2'\ \texttt{in}\ S$ to guarantee that for all instantiations ρ the letrec-bound variables of $\rho(E_2')$ do not capture variables of $\rho(E_1; E_2)$. Furthermore, an appropriate mechanism of such a symbolic α-renaming must allow to do further reasoning with the expressions. Our approach attaches symbolic renamings directly to the subexpressions as deep as possible. Atomic symbolic renamings are of the form $\alpha_{U,i}{\cdot}U$ for a meta-variable U (which may be an environment variable, an expression variable, a context variable) with the meaning that instantiations ρ guarantee that $\rho(\alpha_{U,i}{\cdot}U)$ is an α-renamed copy of $\rho(U)$ s.t. the α-renaming is fresh (all introduced variables are new) and s.t. the distinct variable convention (bound variables are pairwise disjoint from free variables, and all binders bind different variables) holds for $\rho(\alpha_{U,i}{\cdot}U)$. Since these renamings effect also other sub-expressions, we have to distribute them along the term and binding structure.

Thus to treat α-renamings in a mathematical clean way, we extend the language LRSX by syntactic constructs to represent the α-renamings. The extended language is called LRSXα. Adding this kind of syntactic support for α-renamings should be possible for any meta-language with variable binders, so the use of language LRSX should be understood as exemplary but not exclusive. Besides the definition of the syntax and the (ground term-) semantics of LRSXα-expressions, further results of this paper target basic reasoning tasks with LRSX- and LRSXα-expressions. A first algorithm performs α-renaming, i.e. it takes an LRSX-meta expression and delivers an LRSXα-expression s.t. on the semantic level the instances are α-renamed by a fresh renaming. A further procedure performs simplification of symbolic α-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 [14] s.t. LRSXα-expressions can be matched against LRSX-expressions which allows to rewrite LRSXα-expressions. However, this may require to adapt the symbolic α-renaming after a reduction or transformation step and thus we present an algorithm for this task. We finally present an algorithm to check α-equivalence of LRSXα-expressions.

Related Work. Syntactic reasoning on expressions with binders w.r.t. α-equivalence can be done by nominal techniques [11], including nominal unification [20,4,6], nominal matching [3], and nominal rewriting [5] where

$$
\begin{array}{llll}
\mathsf{x}, \mathsf{y}, \mathsf{z} & \in \mathbf{Var} & & \\
\mathsf{s}, \mathsf{t} & \in \mathbf{HExpr}^0 & ::= \mathtt{letrec}\,\mathsf{env}\,\mathtt{in}\,\mathsf{s} \mid (f\,\mathsf{r}_1 \ldots \mathsf{r}_{ar(f)}) & \\
 & & \quad \text{where } \mathsf{r}_i \in \mathbf{HExpr}^k \text{ if } oar(f)(i) = k \geq 0, \text{ and} & \\
 & & \quad \mathsf{r}_i \in \mathbf{Var}, \text{ if } oar(f)(i) = \mathbf{Var} & \\
\mathsf{s} & \in \mathbf{HExpr}^n & ::= \mathsf{x}.\mathsf{s}_1 \quad \text{if } \mathsf{s}_1 \in \mathbf{HExpr}^{n-1} \text{ and } n \geq 1 & \\
\mathsf{b} & \in \mathbf{Bind} & ::= \mathsf{x}.\mathsf{s} \quad \text{where } \mathsf{s} \in \mathbf{HExpr}^0 & \\
\mathsf{env} & \in \mathbf{Env} & ::= \emptyset \mid \mathsf{b}; \mathsf{env} &
\end{array}
$$

Fig. 1: Syntax of LRS

recently also nominal terms with letrec were analyzed [16]. The semantics of nominal meta-terms are all α-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 α-renamed meta-expression represents only those α-equivalent expressions which fulfill the distinct variable convention which seems to be an indispensable requirement regarding the example of transformation (llet). 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 letrec-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 *Outline.* In Sect. 2 we introduce the ground language LRS and the meta-language LRSX, which is then extended by symbolic α-renamings in Sect. 3 where we give an algorithm to symbolically α-rename LRSX-expressions. In Sect. 4 we provide an algorithm for simplification of symbolic α-renamings. In Sect. 5 we present further algorithms for symbolically α-renamed expressions, i.e. a matching algorithm, an algorithm to refresh the α-renaming after a rewrite step was applied, and an algorithm to check α-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 letrec-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 letrec-environments, resp. An LRSX-expression is *ground* iff it is an LRS-expression. Both languages are parametrized over a set of function symbols $\mathcal{F}$ and a set $\overline{K}$ 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* **Var** *which is a countably-infinite set of* variables, **HExpr** *which are* higher-order expressions, **Env** *representing* letrec-environments, *and* **Bind** *which are* letrec-bindings. *Elements s of* **HExpr** *have an $order(s) \in \mathbb{N}_0$, where $\mathbf{HExpr}^n$ denotes the elements of* **HExpr** *of order n, and where $\mathbf{HExpr}^0 = \mathbf{Expr}$. Each $f \in \mathcal{F}$ has a syntactic type $f : \tau_1 \to \ldots \to \tau_n \to \mathbf{Expr}$, 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 $(\delta_1, \ldots, \delta_n)$, where $\delta_i = k_i \in \mathbb{N}_0$, or $\delta_i = \mathbf{Var}$, depending on the type of f. We assume that $\{\mathtt{var}, \lambda\} \subseteq \mathcal{F}$ where* `var` *is of type $\mathbf{Var} \to \mathbf{Expr}$ and lifts variables to expressions with $oar(\mathtt{var}) = (\mathbf{Var})$, and $oar(\lambda)=(1)$.*

Note that in a higher-order expression $\mathsf{x}.\mathsf{r}$, the scope of x is r. The scope of x in $\mathtt{letrec}\ \mathsf{x}.\mathsf{s}; \mathsf{env}\ \mathtt{in}\ \mathsf{s}'$ 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* `letrec`*-environment. With $LV(\mathsf{env})$ we denote the* let-bound variables *in an environment* env, *i.e. all* x *s.t.* $\mathsf{env} = \mathsf{env}'; \mathsf{x}.\mathsf{s}$.

For instance, the expression $\mathtt{letrec}\ \mathsf{x}.\mathtt{var}\ \mathsf{x}; \mathsf{x}.\mathtt{var}\ \mathit{true}\ \mathtt{in}\ \mathsf{x}$ does not fulfill the LVC while $\mathtt{letrec}\ \mathsf{x}.\mathtt{var}\ \mathsf{x}; \mathsf{y}.\mathtt{var}\ \mathit{true}\ \mathtt{in}\ \mathsf{x}$ does.

With the next definition we formally define the notion of an α-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 $\lambda\mathsf{x}.\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x}$ into $\lambda\mathsf{x}_1.\lambda\mathsf{x}_2.\mathtt{var}\ \mathsf{x}_2$ 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, where we use Dewey-notation for the sequences. For (a higher-order) expression or a binding* r *that satisfies the LVC, the* positions *of* r, $\mathcal{P}os(\mathsf{r})$, *are inductively defined as follows where w.l.o.g. we assume* $\mathsf{x}_i < \mathsf{x}_j$ *for* $1 \leq i < j \leq n$: $\mathcal{P}os(\mathsf{x}) := \{\varepsilon\}$, $\mathcal{P}os(f\ \mathsf{r}_1 \ldots \mathsf{r}_n) := \{\varepsilon\} \cup \bigcup_{i=1}^{n}\{i.p \mid p \in \mathcal{P}os(\mathsf{r}_i)\}$, $\mathcal{P}os(\mathtt{letrec}\,\mathsf{x}_1.\mathsf{s}_1;\ldots;\mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\,\mathsf{t}) := \{\varepsilon\} \cup \bigcup_{i=1}^{n}\{i.p \mid p \in \mathcal{P}os(\mathsf{x}_i.\mathsf{s}_i)\} \cup \{(n{+}1).p \mid p \in \mathcal{P}os(\mathsf{t})\}$, *and* $\mathcal{P}os(\mathsf{x}.\mathsf{r}) := \{\varepsilon, 1\} \cup \{2.p \mid p \in \mathcal{P}os(\mathsf{r})\}$. *Given a position* $p \in \mathcal{P}os(\mathsf{r})$, *with* $\mathsf{r}|_p$, *we denote the* term at position p *which is inductively defined by* $\mathsf{r}|_\varepsilon := \mathsf{r}$, $\mathsf{x}.\mathsf{r}|_1 := \mathsf{x}$, $\mathsf{x}.\mathsf{r}|_{2.p} := \mathsf{r}|_p$, $(f\ \mathsf{r}_1 \ldots \mathsf{r}_n)|_{i.p} := \mathsf{r}_i|_p$ *for* $1 \leq i \leq n$, $(\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\ldots;\mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{t})|_{i.1} := \mathsf{x}_i$ *for* $1 \leq i \leq n$, $(\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\ldots;\mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{t})|_{i.2.p} := \mathsf{s}_i|_p$ *for* $1 \leq i \leq n$, *and* $(\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\ldots;\mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{t})|_{n+1.p} := \mathsf{t}|_p$. *A position p is a* variable position *of r if $r|_p$ is a variable, and it is a* binder position *iff $p = q.1$, and $r|_q$ is a higher-order expression of order > 0 or a letrec-binding. For a construct* r, *we denote with $\mathcal{BP}os(r)$ the binder positions of* r. *With* $BV(\mathsf{r})$ *we denote the set of bound variables of* r, *i.e.* $BV(\mathsf{r}) = \{\mathsf{r}|_p \mid p \in \mathcal{BP}os(r)\}$.

If $r|_p = \mathsf{x}$ 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* $\mathtt{letrec}$*-expression s.t. $r|_{q.1} = \mathsf{x}$ 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 $q.1$ (written $binder(r,p) = q.1$) where q is maximal. For* r, *the set of* free variables *is* $FV(\mathsf{r}) := \{\mathsf{r}|_p \mid \mathsf{r}_p = \mathsf{x}$ *and* x *at position p is a free occurrence*$\}$. *We set* $Var(\mathsf{r}) := FV(\mathsf{r}) \cup BV(\mathsf{r})$. *For an expression r, an* α-renaming $A : \mathcal{BP}os(r) \to \mathbf{Var}$ *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* $\mathtt{letrec}$*-expression s.t. $A(q.1) = \mathsf{x}$ 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 *which has p as corresponding binder by $A(p)$. An α-renaming A is a* fresh α-renaming *for r if* $\mathtt{Cod}(A) \cap Var(r) = \emptyset$ *and $A(p) \neq A(p')$ whenever $p \neq p'$.*

The condition on α-renamings implies that the renaming cannot capture free variables. For fresh α-renamings, it always holds.

Example 2.4. For expression $\mathsf{s} = \lambda\mathsf{x}.\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x}$, the positions of s are $\mathcal{P}os(\mathsf{s}) = \{\varepsilon, 1, 1.1, 1.2, 1.2.1, 1.2.1.1, 1.2.1.2, 1.2.1.2.1\}$ and $\mathsf{s}|_{1.2.1.1} = (\mathsf{x}.\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x})|_{2.1.1} = (\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x})|_{1.1} = (\mathsf{x}.\mathtt{var}\ \mathsf{x})|_1 = (\mathtt{var}\ \mathsf{x})|_\varepsilon = \mathtt{var}\ \mathsf{x}$. The positions 1.1, 1.2.1.1, 1.2.1.2.1 are variable positions where $\mathcal{BP}os(\mathsf{s}) = \{1.1, 1.2.1.1\}$ 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 α-renaming $A = \{1.1 \mapsto \mathsf{x}_1, 1.2.1.1 \mapsto \mathsf{x}_2\}$ is a fresh α-renaming for s and $A(s) = \lambda\mathsf{x}_1.\lambda\mathsf{x}_2.\mathtt{var}\,\mathsf{x}_2$ while $A' = \{1.1 \mapsto \mathsf{y}, 1.2.1.1 \mapsto \mathsf{y}\}$ is an α-renaming (which is not fresh for s) s.t. $A'(s) = \lambda\mathsf{y}.\lambda\mathsf{y}.\mathtt{var}\,\mathsf{y}$.

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

Applying a fresh α-renaming to an expression ensures that the distinct variable convention[4] holds for the expression.

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

A position $p \in \mathcal{P}os(r)$ is an *expression position* iff $r|_p \in \mathbf{HExpr}^0$. *Contexts* are LRS-expressions where at one such position, the expression is replaced by the context hole $[\cdot]$. We write d[s] for the operation of filling the hole of context d by expression s. With $CV(\mathsf{d})$ 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 $\mathsf{x} \in CV(\mathsf{d})$ iff the occurrence of x at position $p.1$ in d[$\mathtt{var}$ x] is a bound occurrence. A context class $\mathcal{K} \in \overline{K}$ is a set of contexts and a class $\mathcal{K}$ is *non-binding* if for all contexts d of class $\mathcal{K}$, $CV(\mathsf{d}) = \emptyset$.

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

Lemma 2.6. *The following cases show how to construct a fresh α-renaming from fresh α-renamings for the direct subexpressions:*

1. *Let A_i be fresh α-renamings for s_i for $i = 1, \ldots, n$ s.t.* $\mathtt{Cod}(A_i) \cap \mathtt{Cod}(A_j) = \emptyset$ *for all $i \neq j$. Let $A'(i.p) := A_i(p)$ for $p \in \mathtt{Dom}(A_i)$ and $i = 1, \ldots, n$. Then A' is a fresh α-renaming for $(f\ \mathsf{s}_1 \ldots \mathsf{s}_n)$ and $A'(f\ \mathsf{s}_1 \ldots \mathsf{s}_n) = f\ A_1(\mathsf{s}_1) \ldots A_n(\mathsf{s}_n)$.*
2. *Let A be a fresh α-renaming for* s, $\mathsf{y} \notin \{\mathsf{x}\} \cup \mathtt{Cod}(A)$, $\varsigma = \{\mathsf{x} \mapsto \mathsf{y}\}$. *Let $A'(1)$:=y and $A'(2.p)$:=$A(p)$ for all $p \in \mathtt{Dom}(A)$. Then A' is a fresh α-renaming for* x.s *and* $A'(\mathsf{x}.\mathsf{s}) = \mathsf{y}.(\varsigma(A(\mathsf{s})))$.

[4] Sometimes called Barendregt's variable convention.

$$
\begin{aligned}
x, y, z \in \mathbf{Var} &::= X \mid \mathsf{x} \\
s, t \in \mathbf{HExpr}^0 &::= S \mid D[s] \mid \mathtt{letrec}\ env\ \mathtt{in}\ s \mid (f\, r_1 \ldots r_{ar(f)}) \\
&\quad \text{where } r_i \in \mathbf{HExpr}^k \text{ if } oar(f)(i) = k \geq 0, \text{ and} \\
&\quad\quad r_i \in \mathbf{Var}, \text{ if } oar(f)(i) = \mathbf{Var}. \\
s \in \mathbf{HExpr}^n &::= x.s_1 \quad \text{if } s_1 \in \mathbf{HExpr}^{n-1} \text{ and } n \geq 1 \\
b \in \mathbf{Bind} &::= x.s \quad \text{where } s \in \mathbf{HExpr}^0 \\
env \in \mathbf{Env} &::= \emptyset \mid E; env \mid b; env
\end{aligned}
$$

Fig. 2: Syntax of LRSX, where X, S, D, E are meta-variables.

3. *Let A_i be fresh α-renamings for* s_i *for* $i = 1, \ldots, n{+}1$, *s.t.* $\mathtt{Cod}(A_i) \cap \mathtt{Cod}(A_j) = \emptyset$ *for all* $i \neq j$ $\{\mathsf{y}_1, \ldots, \mathsf{y}_n\} \cap (\bigcup \mathtt{Cod}(A_i) \cup \bigcup Var(\mathsf{s}_i)) = \emptyset$, *and* $\varsigma = \bigcup_i^n \{\mathsf{x}_i \mapsto \mathsf{y}_i\}$. *Let* $A'(i.1) := \mathsf{y}_i$ *for* $i = 1, \ldots, n$, $A'(i.2.p) := A_i(p)$ *for all* $p \in \mathtt{Dom}(A_i)$ *and* $i = 1, \ldots, n$, $A'(n{+}1.p) := A_{n+1}(p)$ *for all* $p \in \mathtt{Dom}(A_{n+1})$. *Then A' is a fresh α-renaming for* $\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1; \ldots; \mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{s}_{n+1}$, *and* $A'(\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1; \ldots; \mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{s}_{n+1}) = \mathtt{letrec}\ \mathsf{y}_1.\varsigma(A_1(\mathsf{s}_1)); \ldots; \mathsf{y}_n.\varsigma(A_n(\mathsf{s}_n))\ \mathtt{in}\ \varsigma(A_{n+1}(\mathsf{s}_{n+1}))$.
4. *Let A be a fresh α-renaming for* s *and A' be a fresh α-renaming for* d *s.t.* $\mathtt{Cod}(A) \cap \mathtt{Cod}(A') = \emptyset$, *and p be the position of the hole in* d. *Let* $A''(p) := A(p)$ *for* $p \in \mathtt{Dom}(A)$ *and* $A''(p.q) := A'(q)$ *for* $q \in \mathtt{Dom}(A')$, *and let* $\varsigma = \{\mathsf{x} \mapsto \mathsf{y} \mid \mathsf{x} \in CV(\mathsf{d}), binder(\mathsf{d}[\mathsf{x}], p) = q.1 \text{ and } A'(q.1) = \mathsf{y}\}$. *Then A'' is a fresh α-renaming for* $\mathsf{d}[\mathsf{s}]$ *and* $A''(\mathsf{d}[\mathsf{s}]) = A(\mathsf{d})[\varsigma(A'(\mathsf{s}))]$.

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

Definition 2.7. LRS*-expressions* $\mathsf{s}_1, \mathsf{s}_2$ *are* α-equivalent, *if there exist fresh α-renamings A_i for* s_i, *s.t.* $A_1(\mathsf{s}_1) = A_2(\mathsf{s}_2)$. *Let $\sim_{let}$ be the reflexive-transitive closure of permuting bindings in a* `letrec`*-environment and $\sim_\alpha$ (*extended α-equivalence*) be the reflexive-transitive closure of combining $\sim_{let}$ and α-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 $cl(D) \in \overline{K}$ denotes the context class of D. The *semantics of meta-variables* X, Y are all concrete variables of type **Var**, expression variables S represent any ground expression of type **Expr**, environment variables E represent all ground environments of type **Env**, and a context variable D with $cl(D) = \mathcal{K}$ represents all contexts of class $\mathcal{K}$.

Definition 2.8. *A* meta-variable substitution ρ *maps a finite set of meta-variables to variables, expressions, environments, and contexts respecting their types and classes. With* $\mathtt{Dom}(\rho)$ *(*$\mathtt{Cod}(\rho)$*, resp.) we denote the domain (co-domain, resp.) of ρ and ρ is* ground *iff it maps all variables in* $\mathtt{Dom}(\rho)$ *to* LRS*-expressions.*

We use the LVC and DVC as well as $\sim_{let}$ also for LRSX-expressions where the sets of variables include concrete variables as well as meta-variables representing concrete variables. We also use $Var(\cdot)$, $BV(\cdot)$, $FV(\cdot)$, $LV(\cdot)$ on the extended syntax. With $MV(s)$ 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, Δ) where s is an* LRSX*-expression, and* $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ *is a* constraint tuple *s.t. Δ_1 is a finite set of context variables, called* non-empty context constraints*; Δ_2 is a finite set of environment variables, called* non-empty environment constraints*; and Δ_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 ρ* satisfies *Δ iff $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$; $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$; and $Var(\rho(t)) \cap CV(\rho(d)) = \emptyset$ for all $(t, d) \in \Delta_3$. If there exists a ground substitution ρ that satisfies Δ, then we say Δ* is satisfiable*. The set of* concretizations *of a constrained* LRSX*-expression (s, Δ) is: $\gamma(s, \Delta) := \{\rho(s) \mid \rho$ is ground, $\rho(s)$ fulfills the LVC, ρ 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 $\Delta_3 = \{(\mathtt{letrec}\ E_1\ \mathtt{in}\ c, \mathtt{letrec}\ E_2\ \mathtt{in}\ [\cdot])\}$, the constrained expression $(\mathtt{letrec}\ E_1\ \mathtt{in}\ \mathtt{letrec}\ E_2\ \mathtt{in}\ S, \Delta)$ 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 calculus L_{need} [18] which copies an abstraction into a needed position in a `letrec`-environment: $\mathtt{letrec}\, E; X.\lambda W.S; Y.A_1[\mathtt{var}\, X]\ \mathtt{in}\ A[\mathtt{var}\, Y] \rightarrow \mathtt{letrec}\, E; X.\lambda W.S; Y.A_1[\lambda W.S]\ \mathtt{in}\ A[\mathtt{var}\, Y]$. If A_1 is empty, then the target of the copy operation should be the variable Y in $A[\mathtt{var}\, Y]$. Thus the case $A = [\cdot]$ should be excluded which can be expressed by setting $\Delta_1 = \{A_1\}$.

$$\begin{aligned}\xi_U \in SAR &::= \langle\rangle \mid \alpha_{U,i} : \eta \\ \eta \in RS &::= \langle rc_1, \ldots, rc_n\rangle, n \geq 0 \\ rc \in RC &::= \{arc_1, \ldots, arc_m\}, m \geq 0 \\ arc \in ARC &::= \alpha_{x,i} \mid LV(\alpha_{E,i}) \mid CV(\alpha_{D,i})\end{aligned}$$

Fig. 3: Symbolic α-renamings

$$\begin{aligned}x, y, z \in \mathbf{Var} &::= \eta{\cdot}X \mid \eta{\cdot}\mathsf{x} \\ s, t \in \mathbf{HExpr}^0 &::= \xi_S{\cdot}S \mid \xi_D{\cdot}D[s] \mid \mathtt{letrec}\ env\ \mathtt{in}\ s \mid (f\, r_1 \ldots r_{ar(f)}) \\ &\qquad \text{where } r_i \in \mathbf{HExpr}^k \text{ if } oar(f)(i) = k \geq 0, \text{ and} \\ &\qquad\qquad r_i \in \mathbf{Var}, \text{ if } oar(f)(i) = \mathbf{Var}. \\ s \in \mathbf{HExpr}^n &::= x.s_1 \quad \text{if } s_1 \in \mathbf{HExpr}^{n-1} \text{ and } n \geq 1 \\ b \in \mathbf{Bind} &::= x.s \quad \text{where } s \in \mathbf{HExpr}^0 \\ env \in \mathbf{Env} &::= \emptyset \mid \xi_E{\cdot}E; env \mid b; env\end{aligned}$$

Fig. 4: Syntax of LRSXα

3 α-Renaming of Meta-Expressions

3.1 The Language LRSXα

While for ground expressions, α-renaming is a well-known task, our setting is different. We want to apply α-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 α-renaming, written as $\xi{\cdot}S$, $\xi{\cdot}D$, $\xi{\cdot}E$, $\eta{\cdot}X$, or $\eta{\cdot}\mathsf{x}$, respectively[5]. We now define the syntax of symbolic renamings and renaming sequences.

Definition 3.1. *The syntax of* symbolic α-renamings ξ *and renaming sequences* η *is defined by the grammar given in Fig. 3. A* renaming sequence $\eta \in RS$ *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* η*. A* renaming component $rc \in RC$ *is a set of atomic renaming components. An* atomic renaming component $arc \in ARC$ *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 variables. For expression-, context-, and environment-variables* U*, a symbolic* α*-renaming* $\xi_U \in SAR$ *is either empty or a sequence* $\alpha_{U,i} : \eta$*, and for variables* X *or* x *it is a renaming sequence* η*. As notation, we write* c *instead of* $\langle c\rangle$ *or* $\{c\}$ *and* $\langle c_1, \ldots, c_n\rangle +\!\!+ \langle c_{n+1}, \ldots, c_m\rangle$ *means* $\langle c_1, \ldots, c_m\rangle$*. The* language LRSX$\alpha$ *(see Fig. 4) extends the syntax of* LRSX *by adding symbolic* α*-renamings* ξ *to each occurrence of meta-variable* S, E, D *and renaming sequences* η *to all occurrences of concrete variables* x *or meta-variables* X*. A* constrained LRSXα-expression *is a pair* (s, Δ) *where* s *is an* LRSXα*-expression and* $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ *is a constraint tuple, s.t.* Δ_1 *is a set of context variables,* Δ_2 *is a set of environment variables, and* Δ_3 *is a set of pairs* (t, d) *where* t *is an* LRSXα*-expression and* d *is an* LRSXα*-context.*

We informally explain the meaning of symbolic α-renamings. Let ρ be a ground substitution. Component $\alpha_{U,i}$ represents a fresh α-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 α-renamings induce a notion of equivalence of symbolic α-renamings:

Definition 3.2. *The relation* $\approx$ *is the smallest equivalence relation satisfying* $c \approx c$ *for* $c = \alpha_{U,i}$ *or an atomic renaming component* c*;* $\langle rc_1, \ldots, rc_{i-1}, \{\}, rc_{i+1}, \ldots, rc_n\rangle \approx \langle rc_1, \ldots, rc_{i-1}, rc_{i+1}, \ldots, rc_n\rangle$*; if* $rc_i \approx rc'_i$ *for* $i = 1, \ldots, n$ *then* $\langle rc_1, \ldots, rc_n\rangle \approx \langle rc'_1, \ldots, rc'_n\rangle$*; if there exists a permutation* π *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 α-renamings up to $\approx$. To embed LRSX-expressions into LRSXα, we identify $\langle\rangle{\cdot}U$ with U and let ϵ : LRSX$\alpha \to$ LRSX be the mapping that erases all renamings.

[5] 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 LRSXα-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* LRSXα*-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* LRSXα*-expression* (s, Δ) *is well-formed, iff* s *is well-formed and for all* $(t, d) \in \Delta_3$ *the expression* t *and the context* d *are well-formed.*

We define the formal semantics of symbolic α-renamings.

Definition 3.4. *Let* (s, Δ) *be a well-formed, constrained* LRSXα*-expression and* ρ *be a ground substitution with* $\mathtt{Dom}(\rho) = MV(s) \cup MV(\Delta)$ *s.t.* $\rho(\epsilon(s))$ *fulfills the LVC. With* $\mathtt{VarCod}(\rho)$ *we denote the variables which appear in the co-domain of* ρ*, i.e.* $\mathtt{VarCod}(\rho) = \bigcup\{\mathit{Var}(\rho(U)) \mid U \in \mathtt{Dom}(\rho)\}$*. A* ground and fresh α-renaming *for* s *and* ρ *is a function* τ *s.t. for all* $U \in MV(s)$*,* τ *maps* $\alpha_{U,i}$ *to a fresh* α*-renaming* $\tau(\alpha_{U,i}) = A_{U,i}$ *for* $\rho(U)$*,* $\tau(\alpha_{X,i})$ *is the substitution* $\{\rho(X) \mapsto \mathsf{y}_{X,i}\}$ *and* $\tau(\alpha_{\mathsf{x},i})$ *is the substitution* $\{\mathsf{x} \mapsto \mathsf{y}_{\mathsf{x},i}\}$ *where all co-domains are fresh and disjoint, i.e.* $\mathtt{Cod}(A_{U,i}) \cap \mathtt{Cod}(A_{U',i'}) = \emptyset$ *for* $i \neq i'$ *or* $U \neq U'$*,* $\mathtt{Cod}(\tau(\alpha_{x,i})) \cap \mathtt{Cod}(\tau_{\alpha_{x',i'}}) = \emptyset$ *for* $i \neq i'$ *or* $x \neq x'$*,* $\mathtt{Cod}(A_{U,i}) \cap \mathtt{VarCod}(\rho) = \emptyset$*,* $\mathtt{Cod}(\alpha_{x,i}) \cap \mathtt{VarCod}(\rho) = \emptyset$*,* $\mathtt{Cod}(A_{U,i}) \cap \mathtt{Cod}(\tau_{\alpha_{x,j}}) = \emptyset$*; and for each environment variable* E*, with* $\rho(E) = \mathsf{x}_1.\mathsf{s}_1; \ldots; \mathsf{x}_n.\mathsf{s}_n$*,* $\tau(\alpha_{E,i}) = A_{E,i}$*,* $\tau(LV(\alpha_{E,i}))$ *is the substitution* $\{\mathsf{x}_j \mapsto A_{E,i}(j.1) \mid j = 1, \ldots n\}$*; and for each context variable* D*, with* $\rho(D) = \mathsf{d}$ *where* p *is the position of the hole in* d*,* $\tau(\alpha_{D,i}) = A_{D,i}$*,* $A_{D,i}(\mathsf{d}) = \mathsf{d}'$*,* $\tau(CV(\alpha_{D,i}))$ *is the substitution induced by* τ *between* $CV(\mathsf{d})$ *and* $CV(\mathsf{d}')$*, i.e.* $\{\mathsf{x} \mapsto \mathsf{x}' \mid \mathsf{x} \in CV(\mathsf{d}), \mathit{binder}(\mathsf{d}[\mathsf{x}], p) = q.1 \text{ and } A_{D,i}(q.1) = \mathsf{x}'\}$*; and* $\tau(\{c_1, \ldots, c_n\}) = \tau(c_1) \circ \cdots \circ \tau(c_n)$ *s.t.* $\tau(c_1) \circ \cdots \circ \tau(c_n) = \tau(c_{\pi(1)}) \circ \cdots \circ \tau(c_{\pi(n)})$ *for all permutations* π *on* $\{1, \ldots, n\}$*; and* $\tau(\langle c_1, \ldots, c_n\rangle)$ *is the composition* $\tau(c_1) \circ \cdots \circ \tau(c_n)$*.*

Applying τ *and* ρ *to* s *and* Δ *first replaces every occurrence* $\xi_U \cdot U$ *in* s *by* $\xi_U \cdot \rho(U)$ *and then replaces* ξ_U *by the corresponding substitution or* α*-renaming, i.e. by* $\tau(\xi_U)(\rho(U))$ *or* $\tau(\eta)(\rho(x))$*. We write* $(\tau(\rho(s)), \tau(\rho(\Delta))$ *for this process. For a constrained* LRSXα*-expression* (s, Δ)*, the concretizations are:*

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

For LRSXα*-expressions* s*, we define* $\gamma(s) = \gamma(s, (\emptyset, \emptyset, \emptyset))$*.*

We use $\sim_{let}$ also for LRSXα-expressions where we allow permutation of bindings and environment variables and also allow to apply $\approx$ to α-renamings.

3.2 Performing Symbolic Alpha-Renaming

We describe how to perform symbolic α-renaming, i.e. how to transform an LRSX-expression s into an LRSXα-expression s', s.t. the instances of s' are α-renamed copies of the instances of s (which are LRS-expressions). The algorithm to symbolically α-rename s, first α-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* sift *shown in Fig. 5) computes an* LRSXα*-expression for* s*. For a constrained* LRSX*-expression* (s, Δ)*, we compute a symbolically* α*-renamed expression as* $(AR(s), \Delta)$*.*

Example 3.6. We α-rename the expression $\lambda X.\lambda X.\mathtt{var}\ X$:

$$\begin{aligned} & AR(\lambda X.\lambda X.\mathtt{var}\ X) = \lambda AR(X.\lambda X.\mathtt{var}\ X) \\ = {} & \lambda \alpha_{X,1} \cdot X.\mathit{sift}(\alpha_{X,1}, AR(\lambda X.\mathtt{var}\ X)) \\ = {} & \lambda \alpha_{X,1} \cdot X.\mathit{sift}(\alpha_{X,1}, \lambda \alpha_{X,2} \cdot X.\mathit{sift}(\alpha_{X,2}, \mathtt{var}\ \langle\rangle \cdot X)) \\ = {} & \lambda \alpha_{X,1} \cdot X.\mathit{sift}(\alpha_{X,1}, \lambda \alpha_{X,2} \cdot X.\mathtt{var}\ \alpha_{X,2} \cdot X) \\ = {} & \lambda \alpha_{X,1} \cdot X.\lambda \alpha_{X,2} \cdot X.\mathtt{var}\ \langle \alpha_{X,2}, \alpha_{X,1}\rangle \cdot X) \end{aligned}$$

Note that the renaming component $\alpha_{X,1}$ in $\langle \alpha_{X,2}, \alpha_{X,1}\rangle \cdot X$ 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 α-renamings in the subsequent section.

$$
\begin{array}{ll}
AR(x) & = \langle\rangle{\cdot}x \\
AR(S) & = \alpha_{S,i}{\cdot}S \\
AR(D[s]) & = \alpha_{D,j}{\cdot}D[\mathit{sift}(CV(\alpha_{D,j}), AR(s))] \\
AR(f\ s_1 \dots s_n) & = f\ AR(s_1) \dots AR(s_n) \\
AR(x.s) & = \alpha_{x,i}{\cdot}x.\mathit{sift}(\langle\alpha_{x,i}\rangle, AR(s)) \\
\multicolumn{2}{l}{AR(\texttt{letrec}\ x_1.s_1; \dots; x_m.s_m; E_1; \dots; E_n\ \texttt{in}\ s)} \\
\quad = \texttt{letrec}\ & \alpha_{x_1,i_1}{\cdot}x_1.\mathit{sift}(\eta, AR(s_1)); \\
& \dots; \\
& \alpha_{x_m,i_m}{\cdot}x_m.\mathit{sift}(\eta, AR(s_m)); \\
& \langle\alpha_{E_1,j_1}, \eta_1\rangle{\cdot}E_1; \dots; \langle\alpha_{E_n,j_n}, \eta_n\rangle{\cdot}E_n \\
\quad\quad \texttt{in}\ \mathit{sift}(\eta, AR(s)) & \\
\multicolumn{2}{l}{\text{where } \eta = (\bigcup_{k=1}^{m}\{\alpha_{x_k,i_k}\}) \cup (\bigcup_{k=1}^{n}\{LV(\alpha_{E_k,j_k})\})} \\
\multicolumn{2}{l}{\text{and } \eta_k = \eta \setminus LV(\alpha_{E_k,i_k})} \\
\mathit{sift}(\eta, x.s) & = x.\mathit{sift}(\eta, s) \\
\mathit{sift}(\eta, f\ s_1 \dots s_n) & = f\ \mathit{sift}(\eta, s_1) \dots \mathit{sift}(\eta, s_n) \\
\mathit{sift}(\eta, \eta'{\cdot}S) & = (\eta' +\!\!+\, \eta){\cdot}S \\
\mathit{sift}(\eta, \eta'{\cdot}D[s]) & = (\eta' +\!\!+\, \eta){\cdot}D[\mathit{sift}(\eta, s)] \\
\multicolumn{2}{l}{\mathit{sift}(\eta, \texttt{letrec}\ z_1.s_1; \dots; z_m.s_m; \eta_1{\cdot}E_1; \dots; \eta_n{\cdot}E_n\ \texttt{in}\ s)} \\
\quad = \texttt{letrec}\ & z_1.\mathit{sift}(\eta, s_1); \dots; z_m.\mathit{sift}(\eta, s_m); \\
& (\eta_1 +\!\!+\, \eta){\cdot}E_1; \dots; (\eta_n +\!\!+\, \eta){\cdot}E_n \\
\quad\quad \texttt{in}\ \mathit{sift}'(\eta, s) & \\
\mathit{sift}(\eta, \eta'{\cdot}x) & = (\eta' +\!\!+\, \eta){\cdot}x
\end{array}
$$

Fig. 5: Adding symbolic α-renamings to an LRSX-expression. All $\alpha_{U,i}$ 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 $\texttt{letrec}\ E_1; E_2; E_3\ \texttt{in}\ \texttt{letrec}\ E_4\ \texttt{in}\ S$:

$$
\begin{array}{l}
AR(\texttt{letrec}\ E_1; E_2; E_3\ \texttt{in}\ \texttt{letrec}\ E_4\ \texttt{in}\ S) = \\
\quad \texttt{letrec}\ \langle\alpha_{E_1,1}, \{LV(\alpha_{E_2,1}), LV(\alpha_{E_3,1})\}\rangle{\cdot}E_1; \\
\quad\quad\quad\quad \langle\alpha_{E_2,1}, \{LV(\alpha_{E_1,1}), LV(\alpha_{E_3,1})\}\rangle{\cdot}E_2; \\
\quad\quad\quad\quad \langle\alpha_{E_3,1}, \{LV(\alpha_{E_1,1}), LV(\alpha_{E_2,1})\}\rangle{\cdot}E_3; \\
\quad \texttt{in}\ \texttt{letrec}\ \langle\alpha_{E_4,1}, \{LV(\alpha_{E_1,1}), LV(\alpha_{E_2,1}), LV(\alpha_{E_3,1})\}\rangle{\cdot}E_4; \\
\quad\quad \texttt{in}\ \langle\alpha_{S,1}, LV(\alpha_{E_4,1}), \{LV(\alpha_{E_1,1}), LV(\alpha_{E_2,1}), LV(\alpha_{E_3,1})\}\rangle{\cdot}S
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{letrec}\ \langle\alpha_{E_1,1},\rangle{\cdot}E_1; \langle\alpha_{E_2,1}\rangle{\cdot}E_2; \langle\alpha_{E_3,1}\rangle{\cdot}E_3; \texttt{in} \\
\quad \texttt{letrec}\ \langle\alpha_{E_4,1}\rangle{\cdot}E_4\ \texttt{in} \\
\quad\quad \langle\alpha_{S,1}, LV(\alpha_{E_4,1}), \{LV(\alpha_{E_1,1}), LV(\alpha_{E_2,1}), LV(\alpha_{E_3,1})\}\rangle{\cdot}S
\end{array}
$$

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 2.6 imply:

Proposition 3.8. *Let s be an* LRSX-*expression and $s' = AR(s)$. Then for each $\mathsf{s} \in \gamma(s)$, there exists $\mathsf{s}' \in \gamma(s')$ s.t. $\mathsf{s} \sim_\alpha \mathsf{s}'$ and for each $\mathsf{s}' \in \gamma(s')$ there exists $\mathsf{s} \in \gamma(s)$ s.t. $\mathsf{s} \sim_\alpha \mathsf{s}'$. Furthermore all $\mathsf{s}' \in \gamma(s')$ 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 $\xi{\cdot}U$. For a set $\mathcal{S}$ of NCCs, the function $\mathit{split}_{\text{NCC}}$ is defined by

$$\mathit{split}_{\text{NCC}}(\mathcal{S}) := \bigcup\nolimits_{(s,d)\in\mathcal{S}} \{(u, v) \mid u \in \mathit{Var}_M(s), v \in CV_M(d)\}$$

$Var_M(\eta{\cdot}x) = \{\eta{\cdot}x\} \quad Var_M(\eta{\cdot}x.s) = \{\eta{\cdot}x\} \cup Var_M(s)$
$Var_M(\xi{\cdot}S) = \{\xi{\cdot}S\} \quad Var_M(f\ s_1 \dots s_n) = \bigcup_i Var_M(s_i)$
$Var_M(\xi{\cdot}D[s]) = \{\xi{\cdot}D\} \cup Var_M(s)$
$Var_M(\texttt{letrec}\ env\ \texttt{in}\ s) = Var_M(env) \cup Var_M(s)$
$Var_M(env) = \{\xi{\cdot}E \mid \xi{\cdot}E; env' = env\}$
$\qquad \cup \bigcup\{\{\eta{\cdot}z\} \cup Var_M(s) \mid \eta{\cdot}z.s; env' = env\}$
$CV_M(\eta{\cdot}x) = \emptyset \quad CV_M(\xi{\cdot}D[d]) = CV_M(\xi{\cdot}D) \cup CV_M(d)$
$CV_M(\xi{\cdot}S) = \emptyset \quad CV_M(\eta{\cdot}x.d) = \{\eta{\cdot}x\} \cup CV_M(d) \quad CV_M([\cdot]) = \emptyset$
$CV_M(\xi{\cdot}D) = \emptyset$, if $cl(D)$ is non-capturing
$CV_M(\xi{\cdot}D) = \{\xi{\cdot}D\}$, otherwise
$CV_M(f\ s_1 \dots\ d \dots s_n) = CV_M(d)$
$CV_M(\texttt{letrec}\ env\ \texttt{in}\ d) = CV_M(env) \cup CV_M(d)$
$CV_M(\texttt{letrec}\ \eta{\cdot}z.d; env\ \texttt{in}\ s) = CV_M(env) \cup \{\eta{\cdot}z\} \cup CV_M(d)$
$CV_M(env) = \{\xi{\cdot}E \mid \xi{\cdot}E; env' = env\} \cup \{\eta{\cdot}z \mid \eta{\cdot}z.s; env' = env\}$

Fig. 6: The functions Var_M and CV_M

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, ρ be a ground substitution, and τ be a ground and fresh α-renaming for s, d, ρ. Then $CV(\tau(\rho(d))) = \{\tau(\rho(\eta{\cdot}x)) \mid x \in CV_M(d)\} \cup \{LV(\tau(\rho(\xi{\cdot}E))) \mid E \in CV_M(d)\} \cup \{CV(\tau(\rho(\xi{\cdot}D))) \mid D \in CV_M(d)\}$ and $Var(\tau(\rho(s))) = \{Var(\tau(\rho(u))) \mid u \in Var_M(s)\}$.*

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 $=_{\mathsf{num}}$ identifies renaming components and sequences up to the number i in $\alpha_{U,i}$, i.e. it is defined by $\alpha_{U,i} =_{\mathsf{num}} \alpha_{U,j}$, where U may be E, D, S, X, x, $CV(\alpha_{D,i}) =_{\mathsf{num}} CV(\alpha_{D,j})$, $LV(\alpha_{E,i}) =_{\mathsf{num}} LV(\alpha_{E,j})$. We extend $=_{\mathsf{num}}$ to renaming sequences ξ_U and η in the obvious way. Compared to $=_{\mathsf{num}}$, the relation $\geq_{\mathsf{num}}$ is defined on atomic renaming components only and it also holds if an $\alpha_{U,i}$ component is replaced by $LV(\alpha_{U,j})$ or $CV(\alpha_{U,j})$, i.e. $\geq_{\mathsf{num}}$ is defined as $arc_1 \geq_{\mathsf{num}} arc_2$ if $arc_1 =_{\mathsf{num}} arc_2$, and $\alpha_{E,i} \geq_{\mathsf{num}} LV(\alpha_{E,j})$, $\alpha_{D,i} \geq_{\mathsf{num}} CV(\alpha_{D,j})$, for all i, j, E, D. A renaming η_1 is* an instance *of a renaming η_2 if $\eta_1 = \eta_2$ or if $\eta_1 = rc_1 : \eta_1'$ and $\eta_2 = rc_2 : \eta_2'$, $rc_1 \subseteq rc_2$, η_1' is an instance of $(rc_2 \setminus rc_1) : \eta_2'$. A renaming η_1 is* a weak instance *of a renaming η_2 if η_1 is an instance of η_2 or $\eta_1 = rc_1 : \eta_1'$ and $\eta_2 = rc_2 : \eta_2'$, $rc_1 \subseteq_{\mathsf{w}} rc_2$, η_1' is a weak instance of $(rc_2 \setminus rc_1) : \eta_2'$. Here $rc_1 \subseteq_{\mathsf{w}} rc_2$ holds if for all $arc \in rc_1$ there exists an $arc' \in rc_2$ with $arc =_{\mathsf{num}} arc'$.*

Example 4.3. The instance relation allows to (partially) order sets of renamings, for example the renaming $\langle \alpha_{S,1}, CV(\alpha_{D_2,1}), \{CV(\alpha_{D_1,1}), LV(\alpha_{E_1,1})\}, LV(\alpha_{E_2,1})\rangle$ is an instance of $\langle \alpha_{S,1}, \{CV(\alpha_{D_1,1}), CV(\alpha_{D_2,1}), LV(\alpha_{E_1,1})\}, LV(\alpha_{E_2,1})\rangle$. The weak instance relation additionally allows to switch between the copies of atomic renaming components, and thus e.g. $\langle \alpha_{S,1}, CV(\alpha_{D_2,2}), \{CV(\alpha_{D_1,1}), LV(\alpha_{E_1,2})\}, LV(\alpha_{E_2,1})\rangle$ is not an instance but a weak instance of $\langle \alpha_{S,1}, \{CV(\alpha_{D_1,1}), CV(\alpha_{D_2,1}), LV(\alpha_{E_1,1})\}, LV(\alpha_{E_2,1})\rangle$.

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

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 Δ_3.

Definition 4.5 (Simplification Algorithm). *Let (s, Δ) be a constrained LRSXα-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 $arc \not\asymp_\Delta v$ 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

$$\text{(IdU)}\frac{}{\xi{\cdot}U \models_\Delta \xi{\cdot}U} \quad \text{(TrU)}\frac{\xi_1{\cdot}U \models_\Delta \xi_2{\cdot}U \text{ and } \xi_2{\cdot}U \models_\Delta \xi_3{\cdot}U}{\xi_1{\cdot}U \models_\Delta \xi_3{\cdot}U} \quad \text{(SimU)}\frac{\{\mathsf{VAR}(U), \mathsf{Cod}(\alpha_{U,i})\}, \eta \vdash_\Delta \eta'}{\alpha_{U,i} : \eta{\cdot}U \models_\Delta \alpha_{U,i} : \eta'{\cdot}U}\ U \neq x$$

$$\text{(IdX)}\frac{}{\eta{\cdot}x \models_\Delta \eta{\cdot}x} \quad \text{(TrX)}\frac{\eta_1{\cdot}x \models_\Delta \eta_2{\cdot}x \text{ and } \eta_2{\cdot}x \models_\Delta \eta_3{\cdot}x}{\eta_1{\cdot}x \models_\Delta \eta_3{\cdot}x} \quad \text{(SimX)}\frac{\{x\}, \eta \vdash_\Delta \eta'}{\eta{\cdot}x \models_\Delta \eta'{\cdot}x} \quad \text{(SubstX)}\frac{\xi{\cdot}x \models_\Delta ((\{\alpha_{x,i}\} \uplus rc) : \eta){\cdot}x}{\xi{\cdot}x \models_\Delta \langle \alpha_{x,i} \rangle{\cdot}x}$$

$$\text{(RemDup)}\frac{arc \geq_{\mathsf{num}} arc'}{\eta_1 +\!\!+ (arc \uplus rc){:}\eta_2 +\!\!+ (arc' \uplus rc'){:}\eta_3{\cdot}U \models_\Delta \eta_1 +\!\!+ (arc \uplus rc){:}\eta_2 +\!\!+ rc'{:}\eta_3{\cdot}U}$$

$$\text{(SimNCCU)}\frac{(\xi_U{\cdot}U, x) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3), \xi'_U \text{ is a weak instance of } \xi_U}{\xi'_U +\!\!+ \{\alpha_{x,i}\} \uplus rc : \eta_2{\cdot}U \models_\Delta \xi'_U +\!\!+ rc : \eta_2{\cdot}U}\ U \neq y \quad \text{(SimNCCX)}\frac{(\eta{\cdot}y, x) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3), \eta' \text{ is a weak instance of } \eta}{\eta' +\!\!+ \{\alpha_{x,i}\} \uplus rc : \eta_2{\cdot}y \models_\Delta \eta' +\!\!+ rc : \eta_2{\cdot}y}$$

(a) Judgments $\xi{\cdot}U \models_\Delta \xi'{\cdot}U$ and $\eta{\cdot}x \models_\Delta \eta'{\cdot}x$ mean that LRSXα-expression $\xi{\cdot}U$ ($\eta{\cdot}x$, resp.) can be simplified to $\xi'{\cdot}U$ ($\eta'{\cdot}x$, resp.).

$$\text{(RMarc)}\frac{\forall v \in V{:}arc \not\sim_\Delta v \quad V, rc{:}\eta \vdash_\Delta rc{:}\eta'}{V, (\{arc\} \uplus rc){:}\eta \vdash_\Delta rc{:}\eta'} \quad \text{(IdEta)}\frac{}{V, \eta \vdash_\Delta \eta} \quad \text{(Order)}\frac{V, \langle rc_i : \{arc_1, \ldots, arc_{i-1}, arc_{i+1}, \ldots, arc_n\} : \eta \vdash_\Delta \eta'}{V, \{arc_1, \ldots, arc_n\} : \eta \vdash_\Delta \eta'}$$

$$\text{(Parc)}\frac{V \cup \{\mathsf{Cod}(arc) \mid arc \in rc\}, \eta \vdash_\Delta \eta'}{V, rc : \eta \vdash_\Delta rc : \eta'} \quad \text{(MSet)}\frac{\forall i \neq j : x_i \neq x_j, \forall i \neq j : \alpha_{x_i,k_i} \not\sim_\Delta x_j, V, \eta_1 +\!\!+ \{\alpha_{x_1,k_1}, \ldots, \alpha_{x_n,k_n}\}{:}\eta_2 \vdash_\Delta \eta_3}{V, \eta_1 +\!\!+ \langle \alpha_{x_1,k_1}, \ldots, \alpha_{x_n,k_n} \rangle +\!\!+ \eta_2 \vdash_\Delta \eta_3}$$

(b) Judgment $V, \eta \vdash_\Delta \eta'$ means that for the variables represented by V, η can be simplified to η'

$$\text{(Cod)}\frac{}{arc_1 \not\sim_\Delta \mathsf{Cod}(arc_2)} \quad \text{(EmCV)}\frac{cl(D) \text{ is non-binding}}{CV(\alpha_{D,i}) \not\sim_\Delta v} \quad \text{(NccDU)}\frac{(U, D) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3)}{CV(\alpha_{D,i}) \not\sim_\Delta \mathsf{VAR}(U)} \quad \text{(NccDX)}\frac{(x, D) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3)}{CV(\alpha_{D,i}) \not\sim_\Delta x}$$

$$\text{(NccEU)}\frac{(U, E) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3)}{LV(\alpha_{E,i}) \not\sim_\Delta \mathsf{VAR}(U)} \quad \text{(NccEX)}\frac{(x, E) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3)}{LV(\alpha_{E,i}) \not\sim_\Delta x} \quad \text{(NccUX)}\frac{(U, x) \in \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3)}{\alpha_{x,i} \not\sim_\Delta \mathsf{VAR}(U)}$$

$$\text{(NccXX)}\frac{(x{=}\mathsf{x}{\neq}\mathsf{x}'{=}x') \vee \{(x', x), (x, x')\} \cap \mathit{split}_{\scriptscriptstyle NCC}(\Delta_3) \neq \emptyset}{\alpha_{x,i} \not\sim_\Delta x'}$$

(c) The predicate $arc \not\sim_\Delta v$ holds iff arc cannot rename the variables represented by v

Fig. 7: Simplification of symbolic α-renamings

$$
\text{(SimU)}\,\dfrac{\text{(Order)}\,\dfrac{\text{(RMarc)}\,\dfrac{\text{(NccEU)}\,\dfrac{(E_i,E_j)\in split_{\scriptscriptstyle NCC}(\Delta_3)}{LV(\alpha_{E_j,1})\not\asymp_\Delta \mathsf{VAR}(E_i)}\quad \text{(Cod)}\,\dfrac{}{LV(\alpha_{E_j,1})\not\asymp_\Delta \mathsf{Cod}(\alpha_{E_i,1})}\quad \text{(RMarc)}\,\dfrac{\text{(NccEU)}\,\dfrac{(E_i,E_k)\in split_{\scriptscriptstyle NCC}(\Delta_3)}{LV(\alpha_{E_k,1})\not\asymp_\Delta \mathsf{VAR}(E_i)}\quad \text{(Cod)}\,\dfrac{}{LV(\alpha_{E_k,1})\not\asymp_\Delta \mathsf{Cod}(\alpha_{E_i,1})}\quad \text{(IdEta)}\,\dfrac{}{\{\mathsf{VAR}(E_i),\mathsf{Cod}(\alpha_{E_i,1})\},\langle\rangle\vdash_\Delta \langle\rangle}}{\{\mathsf{VAR}(E_i),\mathsf{Cod}(\alpha_{E_i,1})\},\langle LV(\alpha_{E_k,1})\rangle\vdash_\Delta \langle\rangle}}{\{\mathsf{VAR}(E_i),\mathsf{Cod}(\alpha_{E_i,1})\},\langle LV(\alpha_{E_j,1}),LV(\alpha_{E_k,1})\rangle\vdash_\Delta \langle\rangle}}{\{\mathsf{VAR}(E_i),\mathsf{Cod}(\alpha_{E_i,1})\},\{LV(\alpha_{E_j,1}),LV(\alpha_{E_k,1})\}\vdash_\Delta \langle\rangle}}{\langle\alpha_{E_i,1},\{LV(\alpha_{E_j,1}),LV(\alpha_{E_k,1})\}\rangle\cdot E_i \models_\Delta \langle\alpha_{E_i,1}\rangle\cdot E_i}
$$

Fig. 8: Derivation for Example 4.6

further renaming components for a renaming for x if the first component is $\alpha_{x,i}$. Rule (SimX) performs simplification of symbolic α-renamings applied to x- 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 α-renaming starts with $\alpha_{U,i}$ and the symbolic set of variables consists of $\mathsf{VAR}(U)$ and the co-domain of $\alpha_{U,i}$, Rules (SimNCCU) and (SimNCCX) allow to remove a component $\alpha_{x,i}$ if an NCC ensures that x cannot occur in $\xi'_U\cdot U$ or $\eta'\cdot y$, 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 α_{x_i,j_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 $\not\asymp_\Delta$ is defined in Fig. 7 (c) where $arc \not\asymp_\Delta v$ 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 $(\lambda\alpha_{X,1}\cdot X.\lambda\alpha_{X,2}\cdot X.\mathtt{var}\ \langle\alpha_{X,2},\alpha_{X,1}\rangle\cdot X,(\emptyset,\emptyset,\emptyset))$ results in $(\lambda\alpha_{X,1}\cdot X.\lambda\alpha_{X,2}\cdot X.\mathtt{var}\ \langle\alpha_{X,2}\rangle\cdot X,(\emptyset,\emptyset,\emptyset))$ since

$$
\text{(SubstX)}\,\dfrac{\text{(IdX)}\,\dfrac{}{\langle\alpha_{X,2},\alpha_{X,1}\rangle\cdot X \models_\Delta \langle\alpha_{X,2},\alpha_{X,1}\rangle\cdot X}}{\langle\alpha_{X,2},\alpha_{X,1}\rangle\cdot X \models_\Delta \langle\alpha_{X,2}\rangle\cdot X}
$$

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

$$
\begin{aligned}
s = \mathtt{letrec}\ &\langle\alpha_{E_1,1},\{LV(\alpha_{E_2,1}),LV(\alpha_{E_3,1})\}\rangle\cdot E_1;\\
&\langle\alpha_{E_2,1},\{LV(\alpha_{E_1,1}),LV(\alpha_{E_3,1})\}\rangle\cdot E_2;\\
&\langle\alpha_{E_3,1},\{LV(\alpha_{E_1,1}),LV(\alpha_{E_2,1})\}\rangle\cdot E_3;\\
\mathtt{in}\ \mathtt{letrec}\ &\langle\alpha_{E_4,1},\{LV(\alpha_{E_1,1}),LV(\alpha_{E_2,1}),LV(\alpha_{E_3,1})\}\rangle\cdot E_4;\\
\mathtt{in}\ &\langle\alpha_{S,1},LV(\alpha_{E_4,1}),\{LV(\alpha_{E_1,1}),LV(\alpha_{E_2,1}),LV(\alpha_{E_3,1})\}\rangle\cdot S
\end{aligned}
$$

$$
\begin{aligned}
\Delta_2 &= \{E_1,E_2,E_3,E_4\}\\
\Delta_3 &= \{(\mathtt{letrec}\ E_i\ \mathtt{in}\ c,\mathtt{letrec}\ E_j\ \mathtt{in}\ [\cdot])\mid i,j\in\{1,2,3,4\},i\neq j\}
\end{aligned}
$$

Applying the simplification algorithm results in (s',Δ) with

$$
\begin{aligned}
s' = \mathtt{letrec}\ &\langle\alpha_{E_1,1},\rangle\cdot E_1;\langle\alpha_{E_2,1}\rangle\cdot E_2;\langle\alpha_{E_3,1}\rangle\cdot E_3;\mathtt{in}\\
&\mathtt{letrec}\ \langle\alpha_{E_4,1}\rangle\cdot E_4\ \mathtt{in}\\
&\quad\langle\alpha_{S,1},LV(\alpha_{E_4,1}),\{LV(\alpha_{E_1,1}),LV(\alpha_{E_2,1}),LV(\alpha_{E_3,1})\}\rangle\cdot S
\end{aligned}
$$

since $\langle\alpha_{E_i,1},\{LV(\alpha_{E_j,1}),LV(\alpha_{E_k,1})\}\rangle\cdot E_i \models_\Delta \langle\alpha_{E_i,1}\rangle\cdot E_i$ can be derived for all i,j,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 Δ be a constraint tuple where $MV(\Delta)\subseteq M$. Let ρ be ground substitution for M and τ be a ground α-renaming for ρ and M, s.t. ρ and τ satisfy Δ.*

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

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

Proof. For part (1), we inspect all rules in Fig. 7 (c). For rule (Cod), the claim holds, since $\tau(\rho(v)) = \tau(\rho(\mathsf{Cod}(arc'))) = \mathtt{Cod}(\tau(arc))$ is a set of fresh variables which cannot be renamed by $\tau(arc)$. For (EmCV), the claim holds, since $\mathtt{Dom}(\tau(CV(\alpha_{D,i}))) = \emptyset$ if D is non-binding. For rule (NccDU), the premise ensures that $CV(\rho(D)) \cap Var(\rho(U)) = \emptyset$ and with $\mathtt{Dom}(\tau(CV(\alpha_{D,i}))) = CV(\rho(D))$ this implies that the equation $\mathtt{Dom}(\tau(CV(\alpha_{D,i}))) \cap Var(\rho(U)) = \emptyset$ holds. For (NccDX), the premise ensures that $CV(\rho(D)) \cap \{\rho(x)\} = \emptyset$ and since $\mathtt{Dom}(\tau(CV(\alpha_{D,i}))) = CV(\rho(D))$ this shows that the equation $\mathtt{Dom}(\tau(CV(\alpha_{D,i}))) \cap \{\rho(x)\} = \emptyset$ holds. For rule (NccEU), the premise ensures that $LV(\rho(E)) \cap Var(\rho(U)) = \emptyset$ and since $\mathtt{Dom}(\tau(LV(\alpha_{E,i}))) = LV(\rho(E))$ this shows that the equation $\mathtt{Dom}(\tau(LV(\alpha_{E,i}))) \cap Var(\rho(U)) = \emptyset$ holds. For (NccEX), we have $LV(\rho(E)) \cap \{\rho(x)\} = \emptyset$ by the premise and with $\mathtt{Dom}(\tau(LV(\alpha_{E,i}))) = LV(\rho(E))$ this shows that the equation $\mathtt{Dom}(\tau(LV(\alpha_{E,i}))) \cap \{\rho(x)\} = \emptyset$ holds. For (NccUX), the premise ensures that $\{\rho(x)\} \cap Var(\rho(U)) = \emptyset$ and since $\mathtt{Dom}(\tau(\alpha_{x,i})) = \{\rho(x)\}$ this shows $\mathtt{Dom}(\tau(\alpha_{x,i})) \cap Var(\rho(U)) = \emptyset$. For rule (NccXX), the premise ensures that $\rho(x) \neq \rho(x')$ and since $\mathtt{Dom}(\tau(\alpha_{x,i})) = \{\rho(x)\}$ this shows $\mathtt{Dom}(\tau(\alpha_{x,i})) \cap \{\rho(x')\} = \emptyset$.

For part (2), 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 $\forall v \in V : arc \not\sim_\Delta v$ and part 1 ensure that $\tau(\{arc\} \cup rc{:}\eta)(\mathsf{x}) = \tau(rc{:}\eta)(\mathsf{x})$ for $\mathsf{x} \in \tau(\rho(V))$. Applying the induction hypothesis to the second part of the premise shows $\tau(rc{:}\eta)(\mathsf{x}) = \tau(rc{:}\eta')(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(V))$ which shows the claim.

For rule (Parc), the induction hypothesis shows that $\tau(\eta)(\mathsf{x}) = \tau(\eta')(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(V \cup V'))$ with $V' = \{Cod(arc) \mid arc \in rc\}$. Since $\tau(arc{:}\eta)(\mathsf{x}) = \tau(\eta)(\tau(arc)(\mathsf{x}))$, $\tau(arc{:}\eta')(\mathsf{x}) = \tau(\eta')(\tau(arc)(\mathsf{x}))$ and $\tau(arc)(\mathsf{x}) \in \tau(\rho(V'))$ for all $\mathsf{x} \in \tau(\rho(V))$ this implies $\tau(arc{:}\eta)(\mathsf{x}) = \tau(arc{:}\eta')(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(V))$.

For rule (Order), we have $\tau(\{arc_1, \ldots, arc_n\}{:}\eta)(\mathsf{x}) = \tau(arc_i{:}\{arc_1, \ldots, arc_{i-1}, arc_{i+1}, \ldots arc_n\}{:}\eta)(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(V))$ by the definition of ground α-renamings.

For rule (MSet), the premise ensures that $\tau(\langle \alpha_{x_1,k_1}, \ldots, \alpha_{x_n,k_n} \rangle)(\mathsf{x}) = \tau(\langle \alpha_{x_{\pi(1)},k_{\pi(1)}}, \ldots, \alpha_{x_{\pi(n)},k_{\pi(n)}} \rangle)(\mathsf{x})$ for all permutations π on $\{1, \ldots, n\}$: all variables x_i are pairwise different, all variables $\alpha_{x_i,k_i} \not\sim_\Delta x_j$ for all $i \neq j$ ensures that $\tau(\alpha_{x_i,k_i})(\rho(x_j)) = \rho(x_j)$ for all $i \neq j$ (see part 1). Since $\tau(arc{:}\eta)(\mathsf{x}) = \tau(\eta)(\tau(arc)(\mathsf{x}))$ for all η, we thus have $\tau(\eta_1 +\!\!+ \langle \alpha_{x_1,k_1}, \ldots, \alpha_{x_n,k_n} \rangle +\!\!+ \eta_2)(\mathsf{x}) = \tau(\eta_1 +\!\!+ \{\alpha_{x_1,k_1}, \ldots, \alpha_{x_n,k_n}\}{:}\eta_2)(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(V))$. Now the induction hypothesis shows the claim.

For proving part (3a), we use induction on the height of the derivation tree for $\eta{\cdot}x \vdash_\Delta \eta'{\cdot}y$. If the height is 1, then one of the rules (IdX), (RemDup), or (SimNCCX) is applied: For rule (IdX), the claim obviously holds, for rule (RemDup), the claim holds, since $arc \geq_{\mathsf{num}} arc'$ ensures that $\mathtt{Dom}(arc') \subseteq \mathtt{Dom}(arc)$ and thus applying arc' has no effect. For rule (SimNCCX), the premise $(\eta{\cdot}y, x) \in split_{\text{NCC}}(\Delta_3)$ ensures that $\rho(x) \notin \tau(\eta)(\rho(y))$, and the second condition of the premise (i.e. η' is a weak instance of η) also ensures that $\rho(x) \notin \tau(\eta')(\rho(y))$, since the difference between η' and η is that some sets of renaming components in η may be sequences of components in η' and that a different variant $\alpha_{U,j}$ of a renaming $\alpha_{U,i}$ is used. However, since the co-domain of $\tau(\alpha_{U,i'})$ is always a set of fresh variables, we have $\rho(x) \in \tau(\rho(\eta{\cdot}y)) \iff \rho(x) \in \tau(\rho(\eta'{\cdot}y))$.

Since $\mathtt{Dom}(\tau(\alpha_{x,i})) = \{\rho(x)\}$, we thus have for renaming $\tau(\alpha_{x,i})$, $\tau(\eta'{:}\{\alpha_{x,i}\})(\rho(y)) = \tau(\eta'\rho(y))$ and thus also $\tau(\rho(\eta'{:}\{\alpha_{x,i}\} \cup rc{:}\eta_2{\cdot}y)) = \tau(\rho(\eta'{:}\{rc{:}\eta_2{\cdot}y))$ which shows the claim.

For the induction step, we consider the last rule application in the derivation tree. If (TrX) is applied, then the induction hypothesis shows $\tau(\eta_1)(\rho(x)) = \tau(\eta_2)(\rho(x))$ and $\tau(\eta_2)(\rho(x)) = \tau(\eta_3)(\rho(x))$ and thus $\tau(\eta_1)(\rho(x)) = \tau(\eta_4)(\rho(x))$. If (SimX) is applied, then part 2 shows $\tau(\eta)(\rho(x)) = \tau(\eta')(\rho(x))$. If (SubstX) is applied, then the induction hypothesis shows $\tau(\eta_1)(\rho(x)) = \tau((\{y/x\} \cup rc){:}\eta_2)(\rho(x))$ Since $\tau(rc{:}\eta_2)(\tau(\{y/x\})(\rho(x))) = \tau(rc{:}\eta_2)((\rho(x) \mapsto \mathsf{y})\rho(x)) = \tau(rc{:}\eta_2)(\mathsf{y}) = \mathsf{y}$ where the last equation holds, since y must be fresh, and thus $\mathsf{y} \notin \mathtt{Dom}(\tau(rc{:}\eta_2))$, we have $\tau(\eta_1)(\rho(x)) = \mathsf{y}$.

For part (3b), we use induction on the height of the derivation tree. For the induction base, the rules (IdU), (RemDup), (SimNCCU) have to be considered. For rule (IdU), the claim holds, for rule (RemDup) the reasoning is analogous to part (3a) of this proof, and for (SimNCCU), the premise $(\xi_U{\cdot}U, x) \in split_{\text{NCC}}(\Delta_3)$ ensures that $\rho(x) \notin Var(\tau(\rho(\xi_U{\cdot}U)))$ and the second premise ensures that also $\rho(x) \notin Var(\tau(\rho(\xi'_U{\cdot}U)))$, since $Var(\tau(\rho(\xi'_U{\cdot}U))) \cap Var(\tau(\rho(\xi_U{\cdot}U)))$ is a set of fresh variables different from $\rho(x)$. Thus, the renaming $\tau(\alpha_{x,i})$ has no effect for $\tau(\rho(\xi'_U{\cdot}U))$ and the claim holds. If rule (TrU) is used, then the claim holds by the induction hypothesis and transitivity of $=$. If $\xi{\cdot}U \vdash_\Delta \xi'{\cdot}U$ is derived by rule (SimU) where $\xi = \alpha_{U,i}{:}\eta$ and $\xi' = \alpha_{U,i}{:}\eta'$, then let $\tau(\alpha_{U,i}) = A_{U,i}$. By part 2 we have $\tau(\eta)(\mathsf{x}) = \tau(\eta')(\mathsf{x})$ for all $\mathsf{x} \in \tau(\rho(\{\mathsf{VAR}(U), \mathsf{Cod}(\alpha_{U,i})\})) = Var(\rho(U)) \cup \mathtt{Cod}(A_{U,i})$. This shows $\tau(\alpha_{U,i}{:}\eta)(\rho(U)) = \tau(\eta)(A_{U,i}(\rho(U))) = \tau(\eta')(A_{U,i}(\rho(U))) = \tau(\alpha_{U,i}{:}\eta')(\rho(U))$. $\square$

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* LRSXα*-expression* (s, Δ) *s.t.* s *fulfills the LVC and* s *does not contain an environment variable twice in the same environment, the simplified expression* (s', Δ)*, we have* $\gamma(s, \Delta) = \gamma(s', \Delta)$*.*

5 Algorithms for LRSXα-Expressions

In this section we show how to perform rewriting of LRSXα-expressions by matching LRSXα-expressions to apply rewrite steps, and by refreshing the α-renaming to guarantee that the distinct variable convention holds after applying a rewrite step. We finally present an algorithm to test extended α-equivalence of LRSXα-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 $\ell \rightarrow_\Delta r$ where ℓ and r are LRSX-expressions and Δ is a constraint tuple. Applying a rewrite rule to a constrained expression (s, ∇) consists of matching ℓ against s s.t. the constraints in ∇ imply the constraints in Δ. Given a matcher (i.e. a substitution σ with $\sigma(\ell) \sim_{let} s$) the reduction is $s \rightarrow \sigma(r)$ (or more precisely $(s, \nabla) \rightarrow (\sigma(r), \nabla \cup \sigma(\Delta))$). In [14] the letrec matching problem was defined and analyzed for LRSX-expressions. However, as argued before, often transformations are not applicable, since ∇ does not imply Δ (see the example for an (llet) overlap in Sect. 1). Here α-renaming of s often helps to satisfy the constraints. Hence, we formulate an adapted form of a letrec matching problem where (s, ∇) is a constrained LRSXα-expression. Our matching equations are of the form $\ell \trianglelefteq s$ where s is a meta-expression with *instantiable meta-variables* and ℓ is meta-expression with meta-variables that act like constants. In addition ℓ may contain symbolic α-renamings (i.e. ℓ is an LRSXα-expression), but s is an LRSX-expression. To distinguish the meta-variables we use blue font for instantiable meta-variables and red font and underlining for fixed meta-variables. With $MV_I(\cdot)$ and $\underline{MV_F}(\cdot)$ we denote functions to compute the sets.

Definition 5.1. *A* letrec matching problem with α-renamed expressions *is a tuple* $P = (\Gamma, \Delta, \nabla)$ *where* Γ *is a set of matching equations* $s \trianglelefteq t$ *s.t.* s *is an* LRSX*-expression,* t *is an* LRSXα*-expression,* $MV_I(t) = \emptyset$*;* $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ *is a constraint tuple over* LRSX*, called* needed constraints*;* $\nabla = (\nabla_1, \nabla_2, \nabla_3)$ *is a constraint tuple over* LRSXα*, called* given constraints*, where* $MV_I(\nabla_i) = \emptyset$ *for* $i = 1, 2, 3$ *and* ∇ *is satisfiable; and for all expressions in* Γ*, the LVC must hold. The following occurrence restrictions must hold: every variable of kind* S *occurs at most twice in* Γ*; every variable of kind* E *or* D *occurs at most once in* Γ*. A matcher* σ *of* P *is a substitution s.t. for any ground substitution* ρ *together with a ground renaming* τ *with* $\mathtt{Dom}(\rho) = \underline{MV_F}(P)$ *s.t.* $\tau \circ \rho$ *satisfies* ∇*,* $\tau(\rho(\sigma(s)))$*,* $\tau(\rho(t))$ *fulfill the LVC for all* $s \trianglelefteq t \in \Gamma$*, we have* $\tau(\rho(\sigma(s))) \sim_{let} \tau(\rho(t))$ *for all* $s \trianglelefteq t \in \Gamma$*, and there exists a ground substitution* ρ_0 *with* $\mathtt{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ *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 τ is involved. The additional substitution ρ_0 in the definition of a matcher is used for the case that rewrite rules $\ell \rightarrow_\Delta r$ introduce meta-variables, i.e. if there are meta-variables which occur in r but not in ℓ. Then the existence of ρ_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: $(f\ s_1 \ldots s_n) \rightarrow_\Delta$ $\mathtt{letrec}\ X_1.s_1; \ldots; X_n.s_n\ \mathtt{in}\ (f\ (\mathtt{var} X_1) \ldots (\mathtt{var} X_n))$ where Δ 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 Sol and in a final step it checks whether the given constraints in ∇_3 imply the required constraints in Δ_3. Except for this final step the algorithm can be reused to solve the letrec matching problem for LRSXα-expressions and computing matchers as follows: Let (Γ, Δ, ∇) be a letrec matching problem with α-renamed expression. Transform the LRSXα-expressions on right-hand sides of Γ and in ∇ into LRSX-expressions by replacing all occurrences $\xi \cdot \underline{U}$, $\xi' \cdot \underline{U}$ with $\xi \approx \xi'$ by a single fresh fixed meta-variable $\underline{U'}$ (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 $(Sol_F, \Delta_F, \nabla_F)$ is produced. Then construct $(Sol_O, \Delta_O, \nabla_O)$ by replacing $\underline{U'}$ by $\xi \cdot \underline{U}$ and x' by $\eta{\cdot}x$ in $(Sol_F, \Delta_F, \nabla_F)$. Now the following check whether Δ_O implies ∇_O is performed. If it succeeds, then Sol_O is delivered as a matcher.

$$
\begin{aligned}
&Var_{sym}(\eta{\cdot}x) = \begin{cases}\{\mathsf{Cod}(\alpha_{x,i})\} & \text{if } \eta = \alpha_{x,i} \cup rc : \eta' \\ \{x\} \cup SV_{sym}(\eta), & \text{otherwise}\end{cases}\\
&Var_{sym}(\xi{\cdot}U) = \{\mathsf{VAR}(U)\} \cup SV_{sym}(\xi)\\
&CV_{sym}(\xi{\cdot}x) = \begin{cases}\{\mathsf{Cod}(\alpha_{x,i})\} & \text{if } \xi = \alpha_{x,i} \cup rc : \eta \\ \{x\} \cup SV_{sym}(\eta)\end{cases}\\
&CV_{sym}(\eta{\cdot}U) = \begin{cases}SV_{sym}(\eta), & \text{if } \eta = \alpha_{U,i} \cup rc : \eta' \\ \{\mathsf{VAR}(U)\} \cup SV_{sym}(\eta) & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
SV_{sym}(\langle rc_1,\ldots,rc_n\rangle) &= \textstyle\bigcup_i SV_{sym}(rc_i)\\
SV_{sym}(\{arc_1,\ldots,arc_n\}) &= \textstyle\bigcup_i SV_{sym}(arc_i)\\
SV_{sym}(\alpha_{U,i}) &= \{\mathsf{Cod}(\alpha_{U,i})\}\\
SV_{sym}(LV(\alpha_{U,i})) &= \{\mathsf{Cod}(\alpha_{U,i})\}\\
SV_{sym}(CV(\alpha_{U,i})) &= \{\mathsf{Cod}(\alpha_{U,i})\}
\end{aligned}
$$

The relation $\bowtie$ is the symmetric closure of the axioms:

$\mathsf{x} \bowtie \mathsf{y}$ if $\mathsf{x} \neq \mathsf{y}$ $\quad x \bowtie \mathsf{Cod}(\alpha_{U,i})$ $\quad \mathsf{VAR}(U) \bowtie \mathsf{Cod}(\alpha_{U',i})$
$\mathsf{Cod}(\alpha_{U,i}) \bowtie \mathsf{Cod}(\alpha_{U',i'})$ if $U \neq U'$ or $i \neq i'$.

Fig. 9: The functions Var_{sym} and CV_{sym} and the relation $\bowtie$

Definition 5.2. *Let $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ and $\nabla = (\nabla_1, \nabla_2, \nabla_3)$ be constraint tuples over* LRSXα *s.t. $MV_I(\nabla) = \emptyset$ and $MV_F(\Delta) \subseteq MV_F(\nabla)$. Then Δ* implies *∇ if $D \in \Delta_1 \implies D \in \nabla_1$, $E \in \Delta_2 \implies E \in \nabla_2$, and for all $(\xi{\cdot}u, \xi'{\cdot}v) \in split_{NCC}(\Delta_3)$ one of the following cases applies:*

1. *$\xi{\cdot}u = \langle\rangle \cdot \mathsf{x}$ and $\xi'{\cdot}v = \langle\rangle \cdot \mathsf{y}$ where $\mathsf{x} \neq \mathsf{y}$.*
2. *$(\xi{\cdot}u, \xi'{\cdot}v) \in split_{NCC}(\nabla_3)$.*
3. *$u = v$ and $u = D$ or $u = E$ with $E \notin \Delta_2$.*
4. *$u \neq v$ and $u = S$, or $u = D$ or $u = E$, or $u = X$.*
5. *$u \neq v$ or $v = D$, or $v = E$, or $v = X$.*
6. *$\xi' = \langle\rangle$ and $v = E$ or $v = D$ and $(v, v) \in split_{NCC}(\nabla_3)$.*
7. *$\xi = \langle\rangle$, $\xi' = \langle\rangle$, and (u, v) is of the form (X, y), (x, Y), (X, Y), (x, D), (X, D), (x, E), (X, E) where in all cases $(v, u) \in split_{NCC}(\nabla_3)$.*
8. *$v_1 \bowtie v_2$ for all $(v_1, v_2) \in (Var_{sym}(\xi{\cdot}u) \times CV_{sym}(\xi'{\cdot}v))$, where $Var_{sym}(\xi{\cdot}u)$ computes symbolic variables which represent the set of free and bound variables that may occur in concretizations of $\xi{\cdot}u$ and $CV_{sym}(\eta{\cdot}v)$ computes symbolic variables which may capture variables in the concretizations of $\xi'{\cdot}v$ and the relation $v_1 \bowtie v_2$ symbolically checks whether the sets of variables represented by v_1 and v_2 are disjoint (see Fig. 9).*

For a ground substitution ρ and a ground α-renaming τ for ρ, let $CV_A(\tau(\rho(\eta{\cdot}x))) := \{\tau(\eta)(\rho(x))\}$, $CV_A(\tau(\rho(\xi{\cdot}D))) := CV(\tau(\xi)(\rho(D)))$, and $CV_A(\tau(\rho(\xi{\cdot}E))) := LV(\tau(\xi)(\rho(E)))$. Note that for an NCC (s, d), $Var(\tau(\rho(s))) = \{Var(\tau(\rho(\xi{\cdot}u))) \mid \xi{\cdot}u \in Var_M(s)\}$ and $CV(\tau(\rho(d))) = \{CV_A(\tau(\rho(\xi{\cdot}u))) \mid \xi{\cdot}u \in CV_M(s)\}$ which justifies to work with the split NCCs.

Lemma 5.3. *Assume that Δ implies ∇. Let ρ be a ground substitution for $MV_F(\nabla)$ and τ be a ground renaming for ρ, s.t. $\tau \circ \rho$ satisfies ∇. Then there exists a ground substitution ρ_0 with $\mathtt{Dom}(\rho_0) = MV_I(\rho(\Delta))$ s.t. $\tau(\rho_0(\rho(\Delta)))$ is satisfied.*

Proof. Let $(\xi{\cdot}u, \xi'{\cdot}v) \in split_{NCC}(\Delta_3)$ s.t. one of the cases of the implication check applies. We consider the different cases and use the following instantiation ρ_0 for instantiable meta-variables: $\rho_0(S) = \lambda \mathsf{x}_S.\mathsf{x}_S$ for a fresh variable x_S; $\rho_0(D) = [\cdot]$ if $D \notin \Delta_1$, and $\rho_0(D) = \mathsf{d}$ where d is a context with $CV(\mathsf{d}) = \emptyset$; $\rho_0(E) = \emptyset$ if $E \notin \Delta_2$ and $\rho_0(E) = \mathsf{x}_E.\mathtt{var}\ \mathsf{x}_E$, otherwise where x_E is a fresh variable; $\rho_0(X) = \mathsf{x}_X$ for a fresh variable x_X.

In case (1), $(\xi{\cdot}u, \xi'{\cdot}v) = (\langle\rangle{\cdot}\mathsf{x}, \langle\rangle{\cdot}\mathsf{y})$ and the constraint is satisfied. In case (2), $(\xi{\cdot}u, \xi'{\cdot}v) \in split_{NCC}(\nabla_3)$ implies $Var(\tau(\xi)\rho(u)) \cap CV_A(\tau(\xi')\rho(v)) = \emptyset$. For case (3), assume that $u = v$ and $u = D$ or $u = E$ with $E \notin \Delta_2$. Then $\tau(\rho(\xi{\cdot}u)) = \tau(\rho(\xi'{\cdot}v)) = u$, and $Var(\tau(\rho_0(\rho(\xi{\cdot}u)))) = CV_A(\tau(\rho_0(\rho(\xi'{\cdot}u)))) = \emptyset$.

For case (4), assume that $u \neq v$ and $u = S$, or $u = D$ or $u = E$, or $u = X$. then $Var(\tau(\rho_0(\rho(\xi{\cdot}u))) = \{\tau(\xi){\cdot}\rho_0(u)\}$ contains only fresh variables and these variables must be disjoint from $CV_A(\tau(\xi'){\cdot}\rho_0(\rho(v))))$, since the variables in $\rho_0(\rho(u))$ must be pairwise distinct from the variables in $\rho_0(\rho(v))$.

For case (5), assume that $u \neq v$, $v = D$, or $v = E$, or $v = X$. Then $\rho_0(\rho(v)) = \rho_0(v)$ and $CV_A(\rho_0(v))$ contains only fresh variables which cannot occur in $\rho_0(\rho(u))$. and thus $CV_A(\tau(\xi')\rho_0(\rho(v))) \cap Var(\tau(\xi)(\rho_0(\rho(u)))) = \emptyset$.

For case (6), let $\xi' = \langle\rangle$, $v = \underline{E}$ or $v = \underline{D}$, and $(v, v) \in split_{NCC}(\nabla_3)$. Then $Var(\rho(v)) \cap CV_A(\rho(v)) = \emptyset$ must hold, which requires that $\rho(v) = \emptyset$ (for $v = \underline{E}$), $\rho(v) = \mathsf{d}$ with $CV_A(\mathsf{d}) = \emptyset$ (for $v = \underline{D}$). In all cases $CV_A(\rho(v)) = \emptyset$ and thus $Var(\tau(\rho_0(\rho(\xi{\cdot}u)))) \cap CV_A(\rho_0(\rho(v))) = \emptyset$.

For case (7), $\xi = \xi' = \langle\rangle$, $(v, u) \in split_{NCC}(\nabla_3) \cup NCC_{dvc}$, and (u, v) is of the form $(\underline{X}, \mathsf{y})$, $(\mathsf{x}, \underline{Y})$, $(\underline{X}, \underline{Y})$, $(\mathsf{x}, \underline{D})$, $(\underline{X}, \underline{D})$, $(\mathsf{x}, \underline{E})$, or $(\underline{X}, \underline{E})$. It suffices to show that $Var(\rho(v)) \cap CV_A(\rho(u)) = \emptyset$ implies $Var(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$. For $(u, v) \in \{(\underline{X}, \mathsf{y}), (\mathsf{x}, \underline{Y}), (\underline{X}, \underline{Y})\}$ this holds since $Var(\mathsf{y}) = CV_A(\mathsf{y})$ for every variable y. For $(u, v) = (\mathsf{x}, \underline{U})$ or $(\underline{X}, \underline{U})$ where $\underline{U}$ is an $\underline{D}$ or $\underline{E}$-variable, $CV_A(\rho(v)) \subseteq Var(\rho(v))$ and $Var(\rho(u)) = CV_A(\rho(u))$ and thus $Var(\rho(v)) \cap CV_A(\rho(u)) = \emptyset$ implies $Var(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$.

For case (8), assume that $v_1 \bowtie v_2$ for all $(v_1, v_2) \in Var_{sym}(\xi{\cdot}u) \times CV_{sym}(\xi'{\cdot}v)$. Then also $Var(\tau(\rho(\xi{\cdot}u))) \cap CV_A(\tau(\rho(\xi'{\cdot}v))) = \emptyset$ holds, which can be verified using the definition of $\bowtie$, Var_{sym}, and CV_{sym} and the interpretation of symbolic set variables in Definition 4.4. ◻

Soundness of the matching algorithm for LRSX [14] implies:

Theorem 5.4. *The matching algorithm for* LRSXα *is sound.*

Example 5.5. As an example for rewriting which also illustrates the necessity of simplification, consider the transformation `letrec` $X.S$ `in var` X $\rightarrow_{\emptyset,\emptyset,(S,\lambda X.[\cdot])}$ S, called (ucp), which inlines the binding $X.S$ where the NCC $(S, \lambda X.[\cdot])$ guarantees that X does not occur in S. For the constrained expression (`letrec` $Y.S_0$ `in var` $Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$, α-renaming results in (`letrec` $\alpha_{Y,1}{\cdot}Y.\langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0$ `in var` $\alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$.

Matching this expression against the left hand side of the transformation (ucp) fails for the substitution $\sigma = \{X \mapsto \alpha_{Y,1}{\cdot}Y, S \mapsto \langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0\}$, since validity of the NCC $\sigma(S, \lambda X.[\cdot]) = (\langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0, \lambda\alpha_{Y,1}{\cdot}Y.[\cdot])$ cannot be inferred. If simplification is applied before the matching, then simplification of (`letrec` $\alpha_{Y,1}{\cdot}Y.\langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0$ `in var` $\alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$ results in (`letrec` $\alpha_{Y,1}{\cdot}Y.\alpha_{S_0,1}{\cdot}S_0$ `in var` $\alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$ and matching this expression against the left hand side of (ucp) delivers the matcher $\sigma = \{X \mapsto \alpha_{Y,1}{\cdot}Y, S \mapsto \alpha_{S_0,1}{\cdot}S_0\}$ and validity of the NCC $\sigma(S, \lambda X.[\cdot]) = (\alpha_{S_0,1}{\cdot}S_0, \lambda\alpha_{Y,1}{\cdot}Y.[\cdot])$ can be inferred since $split_{NCC}(\{(\alpha_{S_0,1}{\cdot}S_0, \lambda\alpha_{Y,1}{\cdot}Y.[\cdot])\} = \{(\alpha_{S_0,1}{\cdot}S_0, \alpha_{Y,1}{\cdot}Y)\}$ and $\mathsf{VAR}(S_0) \bowtie Cod(\alpha_{Y,1})$ and $\mathsf{Cod}(\alpha_{S_0,1}) \bowtie \mathsf{Cod}(\alpha_{Y,1})$.

5.2 Refreshing α-Renamings

Matching can be used to rewrite constrained symbolically α-renamed expressions. However, after applying such a rewrite step, the concretizations may no longer fulfill the strong DVC. For instance, consider a meta letrec rewrite rule that copies a subexpression: `letrec` $X.S$ `in` C[`var` X] $\rightarrow_\Delta$ `letrec` $X.S$ `in` $C[S]$. Applying the rule to `letrec` $\alpha_{X,1}{\cdot}X.\alpha_{S_0,1} \cdot S_0$ `in var` $\alpha_{X,1}{\cdot}X$ results in `letrec` $\alpha_{X,1}{\cdot}X.\alpha_{S_0,1} \cdot S_0$ `in` $\alpha_{S_0,1} \cdot S_0$. The same α-renaming $\alpha_{S_0,1}$ is used for both occurrences of S_0 which violates the DVC for instances of the expression. An approach to deal with this problem could be to generalize the symbolic α-renamings to again symbolically α-rename the expressions. However, this approach seems to be not easily tractable (for instance, this would require to introduce renaming components of the form $\alpha_{\xi \cdot S,i}$ which represents an α-renaming of already α-renamed expressions). We choose a simpler approach that uses the existing α-renamings and refreshes them:

Definition 5.6 (Refreshing Alpha-Renamings). *A* renumbering *of a symbolic α-renaming modifies $\alpha_{U,i}$ components by replacing $\alpha_{U,i}$ (or $\alpha_{x,i}$ resp.) with $\alpha_{U,j}$ ($\alpha_{x,j}$,resp.) where j is a fresh number. Let (s, Δ) be a constrained* LRSXα*-expression. The function $refresh(s, \nabla)$ renumbers all occurrences of $\alpha_{U,i}$ and replaces $CV(\alpha_{U,i})$ with $CV(\alpha_{U,j})$ and $LV(\alpha_{U,i})$ with $LV(\alpha_{U,j})$ respecting the scopes. For bound variables $\langle\rangle{\cdot}x$ or meta-variables $\langle\rangle{\cdot}U$ it introduces a fresh α-renaming $\alpha_{x,i}$ or $\alpha_{U,i}$ 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, Δ) be a constrained* LRSXα*-expression and $(s', \Delta) = refresh(s, \Delta)$. Then for each* $\mathsf{s} \in \gamma(s, \Delta)$ *there exists* $\mathsf{s}' \in \gamma(s', \Delta)$ *with* $\mathsf{s} \sim_\alpha \mathsf{s}'$ *and for each* $\mathsf{s}' \in \gamma(s', \Delta)$ *there exists* $\mathsf{s} \in \gamma(s, \Delta)$ *with* $\mathsf{s} \sim_\alpha \mathsf{s}'$

Proof. Replacing $\alpha_{U,i}$- and $\alpha_{x,i}$-renamings by fresh copies implies that the corresponding ground α-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 α-Equivalence

We finally provide a test for checking extend α-equivalence.

Definition 5.8. *Let s and s' be* LRSXα*-expressions. The extended α-equivalence check for s and s' first guesses an order of the environment variables and bindings in all* `letrec`*-environments in s and in s' and then recursively works along the structure of s and s' in parallel and tries to renumber all symbolic α-renamings s.t.*

- *for each binder $\alpha_{x,i}{\cdot}x$ in s at position p and $\alpha_{x',i'}{\cdot}x'$ in s' at position p, replace $\alpha_{x,i}$ by $\alpha_{x,k}$ in s and replace $\alpha_{x',i'}$ by $\alpha_{x',k}$ in s', where k is a fresh number. Perform these replacements for all occurrences of $\alpha_{x,i}$ and $\alpha_{x',i'}$, resp. in the scope of the binder at position p.*
- *for each occurrence of $\alpha_{U,i} : \eta{\cdot}U$ in s at position p and $\alpha_{U,i'} : \eta'{\cdot}U$ at position p in s', replace $\alpha_{U,i}$ by $\alpha_{U,k}$ in s and $\alpha_{U,i'}$ by $\alpha_{U,k'}$ in s'. Perform the replacements for all occurrences of $\alpha_{U,i}$ in s and $\alpha_{U,i'}$ in s'.*

If the structures of s and s' are different or if position p in s' does not exist or is not of the demanded form, then fail. Otherwise, let the modified expressions be s_0 and s'_0. Replace $\alpha_{\mathsf{x},k}$ by the substitution $\{\mathsf{x} \mapsto \mathsf{y}_k\}$ (where y_k is a fresh variable) and replace $\alpha_{X,k}$ by the substitution $\{X \mapsto Y_k\}$ where Y_k is a fresh meta-variable. Let s_1 and s'_1 be the resulting expressions. Check whether the expressions s_1 and s'_1 are equivalent w.r.t. $\sim_{let}$. If this check succeeds, then deliver success else fail.

Given two constrained LRSXα*-expressions (s, Δ) and (s', Δ'), the extended α-equivalence check is valid, if the extended α-equivalence check for s and s' is valid, and Δ implies Δ' as well as Δ' implies Δ using the implication check from Definition 5.2, where all meta-variables are treated as fixed meta-variables.*

Proposition 5.9. *Let s and s' be* LRSXα*-expressions and let ρ be a ground substitution for $\epsilon(s)$ and $\epsilon(s')$ and τ be a ground renaming for ρ. Then the extended α-equivalence $\tau(\rho(s)) \sim_\alpha \tau(\rho(s'))$ holds.*

Proof. Let s_0, s'_0, s_1, s'_1 be the modified expressions of the α-equivalence check. First observe that with the extension ρ_0 of ρ s.t. $\rho_0(Y_k) = \tau(\rho(\alpha_{X,k}{\cdot}X_k))$ and $\rho_0(U) = \rho(U)$ for all meta-variables which are not replaced by the modification from s_0 to s_1 and s'_0 to s'_1, we have $\tau(\rho(s_0)) \sim_\alpha \tau(\rho_0(s_1))$ and $\tau(\rho(s'_0)) \sim_\alpha \tau(\rho_0(s'_1))$. Clearly, we also have $\tau(\rho(s_0)) \sim_\alpha \tau(\rho(s))$ and $\tau(\rho(s'_0)) \sim_\alpha \tau(\rho(s'))$, since only the co-domains of α-renamings are modified. Since s_1 and s'_1 are equivalent w.r.t. $\sim_{let}$, this also holds for $\tau(\rho_0(s_1))$ and $\tau(\rho_0(s'_1))$ and thus we have $\tau(\rho(s)) \sim_\alpha \tau(\rho(s'))$.

Soundness of the implication check and the previous proposition imply correctness of the extended α-equivalence check:

Theorem 5.10. *Let (s, Δ), (s', Δ') be constrained* LRSXα*-expressions which pass the extended α-equivalence check. Let ρ be a ground substitution with $\mathtt{Dom}(\rho) = MV(s) \cup MV(s') \cup MV(\Delta) \cup MV(\Delta')$ and let τ be a ground renaming for ρ s.t. $\tau \circ \rho$ satisfies $\Delta_1, \Delta_2, \Delta'_1, \Delta'_2$. Then $\tau \circ \rho$ satisfies Δ iff $\tau \circ \rho$ satisfies Δ' 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 α-renaming. We tested the LRSX Tool with the calculus L_{need} [18], and the calculus LR [19] (which extends L_{need} 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 α-renamings, and thus α-renaming is applied only, if no join was found before without performing renaming. The results show that α-renaming is necessary in about 20 percent of the cases (except for overlaps of left hand sides in the calculus L_{need}). With the help of α-renaming all computed overlaps could be closed and the correctness of program transformations (16 transformations for L_{need} and 43 transformation for LR) could be shown automatically.

7 Conclusion

We presented an extension of the meta-language LRSX by symbolic α-renamings. We introduced algorithms for simplification of renamings, matching, reduction, and checking extended α-equivalence. The algorithms are implemented and used in the LRSX Tool, and our experiments show that the approach for α-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_{need}			
$\rightarrow$	2242	5425	93
$\leftarrow$	3001	7273	1402
Calculus LR			
$\rightarrow$	87041	391264	73601
$\leftarrow$	107333	429104	93230

Table 1: Statistics of executing the LRSX Tool

References

1. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.
2. Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL 1995*, pages 233–246. ACM, 1995.
3. Christophe Calvès and Maribel Fernández. Nominal matching and alpha-equivalence. In *WoLLIC 2008*, volume 5110 of *LNCS*, pages 111–122. Springer, 2008.
4. Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.
5. Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Inf. Comput.*, 205(6):917–965, 2007.
6. Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *RTA 2008*, volume 5117 of *LNCS*, pages 246–260. Springer, 2008.
7. Elena Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. In *WRS 2007*, ENTCS, 2007.
8. Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In *ESOP 2000*, volume 1782 of *LNCS*, pages 260–274. Springer, 2000.
9. James Hiram Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.
10. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI 1988*, pages 199–208. ACM, 1988.
11. Andrew Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, 2016.
12. Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.
13. Conrad Rau, David Sabel, and Manfred Schmidt-Schauß. Correctness of program transformations as a termination problem. In *IJCAR 2012*, volume 7364 of *LNCS*, pages 462–476. Springer, 2012.
14. David Sabel. Rewriting of higher-order-meta-expressions with recursive bindings. Frankfurter Informatik-Berichte 2017-1, Goethe-University Frankfurt am Main, 2017. http://goethe.link/fib-2017-1.
15. David Sabel and Manfred 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.
16. Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In *LOPSTR 2016*, 2016. informal proceedings.
17. Manfred Schmidt-Schauß and David Sabel. Unification of program expressions with recursive bindings. In *PPDP 2016*, pages 160–173. ACM, 2016.
18. Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *RTA 2010*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl, 2010.
19. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
20. Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *CSL 2003*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.
21. Joe B. Wells, Detlef Plump, and Fairouz Kamareddine. Diagrams for meaning preservation. In *RTA 2003*, volume 2706 of *LNCS*, pages 88 –106. Springer, 2003.