Alpha-Renaming of Higher-Order Meta-Expressions

David Sabel*
Goethe-University
Frankfurt am Main, Germany
sabel@ki.cs.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 higher-order meta-syntax which allows one 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 operators and 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 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.

CCS CONCEPTS

• **Theory of computation** → *Rewrite systems*; *Logic and verification*; *Functional constructs*; *Program specifications*; *Program verification*; • **Software and its engineering** → *Functional languages*;

KEYWORDS

semantics, verification, functional programming, α-renaming

ACM Reference Format:
David Sabel. 2017. Alpha-Renaming of Higher-Order Meta-Expressions. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages. https://doi.org/10.1145/3131851.3131866

1 INTRODUCTION

We focus on automatically proving correctness of program transformations for higher-order programming languages with recursive bindings as they occur in functional programming languages with call-by-need semantics like Haskell (see [1, 2, 32]). One technique to establish such proofs for program calculi with small-step operational semantics is the diagram method [27, 32] 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 [14, 22]. This diagram method was e.g. used in [27, 32] and similar techniques are in [11, 12, 34], where the overlaps and the joins are computed manually by a case-analysis. In our recently developed LRSX Tool[2] we try to automate 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 [30] and reductions and transformations to join the overlaps are applied using a matching algorithm [26].

To represent different (untyped) program calculi, the language `LRSX` is parametric over a set of higher-order function symbols and over a set of context classes. The latter can for instance be used to describe the appropriate class of evaluation contexts of the represented programming language. To represent call-by-need functional programming languages, `LRSX` has a letrec-construct $\mathtt{letrec}\ \mathtt{x}_1 = \mathtt{s}_1; \ldots; \mathtt{x}_n = \mathtt{s}_n\ \mathtt{in}\ \mathtt{s}_{n+1}$ where $\mathtt{x}_1 = \mathtt{s}_1; \ldots; \mathtt{x}_n = \mathtt{s}_n$ is an unordered sets of recursive bindings (the scope of the letrec-bound variables $\mathtt{x}_i$ is $\mathtt{s}_1, \ldots, \mathtt{s}_{n+1}$). To model small-step reduction rules of call-by-need program calculi (see e.g. [1, 2, 31, 32]), the language `LRSX` provides meta-variables for expressions, variables, parts of letrec-environments, and contexts of different classes.

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 in the LRSX Tool are so-called constrained expressions which are meta-expressions augmented by constraints that restrict the instances. For example, consider the transformation (llet):

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

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 allow one to instantiate the meta-variable E_1 by the environment which consists of a single binding[3] $\mathtt{x} = \mathtt{y}$, meta-variable E_2 by an environment which consists of a single binding $\mathtt{y} = \mathtt{True}$, meta-variable S by $\mathtt{x}$, and meta-variable C by the empty context resulting in the instantiated rule $\mathtt{letrec}\ \mathtt{x} = \mathtt{y}\ \mathtt{in}\ \mathtt{letrec}\ \mathtt{y} = \mathtt{True}\ \mathtt{in}\ \mathtt{x} \rightarrow \mathtt{letrec}\ \mathtt{x} = \mathtt{y}; \mathtt{y} = \mathtt{True}\ \mathtt{in}\ \mathtt{x}$ which however should be forbidden, since variable $\mathtt{y}$ in $\mathtt{x} = \mathtt{y}$ is a free occurrence in the left expression, but becomes a bound occurrence (captured by the binding $\mathtt{y} = \mathtt{True}$) in the right expression. So-called non-capture constraints 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

*This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1

[1] See [24] 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 `x.var y`, since "." is used instead of "=" and the function symbol `var` is necessary to lift variables to expressions.

PPDP'17, October 9–11, 2017, Namur, Belgium

instantiation ρ if context $\rho(d)$ does not capture any variable of $\rho(s)$. For our example, $(s_0, d_0) = (\texttt{letrec}\ E_1\ \texttt{in True}, \texttt{letrec}\ E_2\ \texttt{in}\ [\cdot])$ guarantees that no variable of E_1 is captured by the binders of E_2.

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 dashed arrows):

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

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 instances which forbid the suggested join, for example, with $\rho = \{E_1 \mapsto \texttt{x = z}, E_2 \mapsto \texttt{y = True}, E_2' \mapsto \texttt{z = True}, S \mapsto \texttt{x}\}$, the suggested join would lead to $\rho(\texttt{letrec}\ E_1; E_2; E_2'\ \texttt{in}\ S) = \texttt{letrec x = z; y = True; z = True in x}$ 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 `letrec x = z; y = True in letrec z = True in x` into `letrec x = z; y = True in letrec z′ = True in x` and α-rename `letrec x = z in letrec y = True; z = True in` S into `letrec x = z in letrec y = True; z′ = True 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 $\texttt{letrec}\ E_1; E_2\ \texttt{in letrec}\ E_2'\ \texttt{in}\ S$ 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 one to do further reasoning with the expressions. Our approach attaches symbolic renamings directly to the subexpressions as deeply 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 $\rho(U)$ where the α-renaming is fresh (all introduced variables are new) and 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 affect also other subexpressions, we have to distribute them along the term and binding structure.

Thus to treat α-renamings, we extend the LRSX by syntactic constructs to represent the α-renamings. The extended language is called LRSXα. Adding such a 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-meta expression such that 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 adaptation of the matching algorithm from [26] such that LRSXα-expressions can be matched against LRSX-expressions which allows one to rewrite LRSXα-expressions. However, this may require one to adapt the symbolic α-renaming after a rewrite step and thus we present an algorithm for this task. We finally present a test to check α-equivalence of LRSXα-expressions.

Related Work. We discuss approaches to represent higher-order languages with binders and their treatment of α-renaming.

A general approach to represent higher-order languages with binders is higher-order abstract syntax [16] where binders of the object language are represented by binders of the meta-language. For instance, the Twelf system [17] uses this approach for implementing the logical framework LF [9]. More recent work extends this approach also to contextual modal type theory [15, 18, 20] which allows one to represent and reason about contexts. The approach is implemented in Beluga [19] and also allows one to reason with context variables but in contrast to LRSX there seems to be no easy mechanism to express the syntactic structure of contexts and context variables (as LRSX' context classes do) and as a further difference the language LF and its extensions do not provide a syntactic letrec-construct as it is available in LRSX. In general the approaches using higher-order abstract syntax are used to represent and implement logical frameworks, which require a quite complicated mathematical machinery with very sophisticated techniques (like dependent type theory). This is not our focus, since the targeted diagram method is a syntactic method. Thus, in comparison to higher-order abstract syntax, our approach uses a first-order representation and is light-weight and syntax-oriented. On the one hand, algorithms for first-order syntax (like unification and matching, even with meta-variables for contexts and letrec-expressions) can be adapted for our representation, on the other hand, our approach requires to take care about low-level details like explicitly performing α-renaming.

A further possibility to avoid explicit α-renaming would be to use a canonical representation of bindings such as de Bruijn indices [7] and locally nameless approaches [3, 6, 13], or a canonical choice of names [23]. One hurdle in using such an approach is to combine it with our requirements to have meta-variables for contexts and environments, but a more important problem is that it is unclear how to define a canonical representation for letrec-expressions and their unordered set of bindings. Since deciding α-equivalence of ground letrec-expressions is GI-complete (see [29]) there does not seem to be an efficiently computable canonical representation. For these reasons, LRSX uses a 'nameful' approach.

Another approach for syntactic reasoning on expressions with binders w.r.t. α-equivalence are nominal techniques [21], including nominal unification [5, 10, 33], nominal matching [4], and nominal rewriting [8] where recently also nominal terms with letrec were analyzed [28]. The semantics of nominal meta-terms are all α-equivalent expressions of all instances. Similarly to our constrained expressions, nominal terms allow one 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 for 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 considered in this work.

Outline. In Sect. 2 we introduce the languages LRS and LRSX, and in Sect. 3 we extend them by symbolic α-renamings and give an algorithm to symbolically α-rename LRSX-expressions. In Sect. 4 we consider 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. Experimental results are discussed in Sect. 6. In Sect. 7 we conclude. Due to space constraints some proofs are omitted, but can be found in the technical report [25].

2 LANGUAGES LRS AND LRSX

We introduce two languages. The language LRS is a functional language with higher-order operators (like lambda-abstractions) and letrec-expressions which represent shared and recursive bindings. The meta-language LRSX extends LRS by meta-variables for variables, expressions, contexts, and (parts of) 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. A context class $\mathcal{K} \in \overline{K}$ is a set of contexts which is provided as a part of the input and defined by a grammar[4].

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** for a countably-infinite set of

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

Figure 1: Syntax of LRS

variables, **HExpr** which are *higher-order expressions*, **Env** representing *letrec-environments*, and **Bind** representing *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 $f : \tau_1 \to \dots \to \tau_n \to \mathbf{Expr}$, 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 $(\delta_1, \dots, \delta_n)$, 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 $\mathtt{var} : \mathbf{Var} \to \mathbf{Expr}$ lifts variables to expressions with $oar(\mathtt{var}) = (\mathbf{Var})$, and $oar(\lambda)=(1)$.

Example 2.2. The identity is written as $\lambda(\mathsf{x}.\mathtt{var}\ \mathsf{x})$. Applications can be represented by a symbol app with $oar(\mathsf{app}) = (0, 0)$.

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

Definition 2.3. An LRS-expression 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* of env. i.e. all x with $\mathsf{env} = \mathsf{env}';\mathsf{x}.\mathsf{s}$.

For instance, the expression $\texttt{letrec}\ \mathsf{x}.\mathtt{var}\ \mathsf{x};\mathsf{x}.\mathtt{var}\ \mathit{true}\ \texttt{in}\ \mathsf{x}$ does not fulfill the LVC while $\texttt{letrec}\ \mathsf{x}.\mathtt{var}\ \mathsf{x};\mathsf{y}.\mathtt{var}\ \mathit{true}\ \texttt{in}\ \mathsf{x}$ 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 2.4. 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 $1 \leq i < j \leq n$:

$$
\begin{array}{ll}
\mathcal{P}os(\mathsf{x}) & := \{\varepsilon\} \\
\mathcal{P}os(f\ \mathsf{r}_1 \dots \mathsf{r}_n) & := \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in \mathcal{P}os(\mathsf{r}_i)\} \\
\multicolumn{2}{l}{\mathcal{P}os(\texttt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\dots;\mathsf{x}_n.\mathsf{s}_n\ \texttt{in}\ \mathsf{t}) :=} \\
\multicolumn{2}{l}{\quad \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i.p \mid p \in \mathcal{P}os(\mathsf{x}_i.\mathsf{s}_i)\} \cup \{(n+1).p \mid p \in \mathcal{P}os(\mathsf{t})\}} \\
\mathcal{P}os(\mathsf{x}.\mathsf{r}) & := \{\varepsilon, 1\} \cup \{2.p \mid p \in \mathcal{P}os(\mathsf{r})\}
\end{array}
$$

For a position $p \in \mathcal{P}os(\mathsf{r})$, we denote with $\mathsf{r}|_p$ the *term at position* p, inductively defined by $\mathsf{r}|_\varepsilon := \mathsf{r}$, $\mathsf{x}.\mathsf{r}|_1 := \mathsf{x}$, $\mathsf{x}.\mathsf{r}|_{2.p} := \mathsf{r}|_p$, $(\texttt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\dots;\mathsf{x}_n.\mathsf{s}_n\ \texttt{in}\ \mathsf{t})|_{i.1} := \mathsf{x}_i$ for $1 \leq i \leq n$, and $\texttt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\dots;\mathsf{x}_n.\mathsf{s}_n\ \texttt{in}\ \mathsf{t})|_{i.2.p} := \mathsf{s}_i|_p$ for $1 \leq i \leq n$, and $(\texttt{letrec}\ \mathsf{x}_1.\mathsf{s}_1;\dots;\mathsf{x}_n.\mathsf{s}_n\ \texttt{in}\ \mathsf{t})|_{n+1.p} := \mathsf{t}|_p$, and $(f\ \mathsf{r}_1 \dots \mathsf{r}_n)|_{i.p} :=$

[4] For instance, the class $\mathcal{A}$ of call-by-name evaluation contexts of the pure lambda calculus can be described with the grammar (using LRSX-expressions, see Sect. 2.2) $A ::= [\cdot] \mid \mathsf{app}\ A\ S$ where $[\cdot]$ is the empty context and S is an expression meta-variable.

$\mathsf{r}_i|_p$ for $1 \leq i \leq n$. 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 $\mathcal{BPos}(r)$ the binder positions of r. With $BV(\mathsf{r})$ we denote the set of bound variables of r, i.e. $BV(\mathsf{r}) = \{\mathsf{r}|_p \mid p \in \mathcal{BPos}(r)\}$.

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 such that either $q = q'$ or $q = q'.i$ and $\mathsf{r}|_q$ is a `letrec`-expression such that $r|_{q.1} = \mathsf{x}$ 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. The set of *free variables* of r 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 functions h, we denote by $\mathsf{Dom}(h)$ its domain, by $\mathsf{Cod}(h)$ its co-domain, and (if the co-domain of h consists of expressions), with $\mathsf{VarCod}(h)$ the variables appearing in its co-domain, i.e. $\mathsf{VarCod}(h) = \bigcup\{Var(h(U)) \mid U \in \mathsf{Dom}(\rho)\}$.

For an expression r, an *α-renaming* $A : \mathcal{BPos}(r) \to \mathbf{Var}$ 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 such that either $q = q'$ or $q = q'.i$ and $r|_q$ is a `letrec`-expression such that $A(q.1) = \mathsf{x}$ 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 $\mathsf{Cod}(A) \cap Var(r) = \emptyset$ 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 2.5. For expression $\mathsf{s} = \lambda\mathsf{x}.\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x}$, the positions of s are $\mathcal{Pos}(\mathsf{s}) = \{\varepsilon,\ 1,\ 1.1,\ 1.2,\ 1.2.1,\ 1.2.1.1,\ 1.2.1.2,\ 1.2.1.2.1\}$ and $\mathsf{s}|_{1.2.1.1} = (\mathsf{x}.\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x})|_{2.1.1} = (\lambda\mathsf{x}.\mathtt{var}\ \mathsf{x})|_{1.1} = (\mathsf{x}.\mathtt{var}\ \mathsf{x})|_1 = \mathsf{x}|_\varepsilon = \mathsf{x}$. The positions 1.1, 1.2.1.1, 1.2.1.2.1 are variable positions where $\mathcal{BPos}(\mathsf{s}) = \{1.1, 1.2.1.1\}$ 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) such that $A'(s) = \lambda\mathsf{y}.\lambda\mathsf{y}.\mathtt{var}\ \mathsf{y}$. For $\mathsf{s} = \lambda\mathsf{x}.\mathtt{var}\ \mathsf{y}$, 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[5] holds for the expression.

Definition 2.6. 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 $p \in \mathcal{Pos}(r)$ is an *expression position* iff $\mathsf{r}|_p \in \mathbf{HExpr}^0$. *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[var x] is a bound occurrence. A context class $\mathcal{K}$ is *non-binding* if for all contexts d of class $\mathcal{K}$, $CV(\mathsf{d}) = \emptyset$.

[5]Sometimes called Barendregt's variable convention.

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 2.7. *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 $i = 1, \ldots, n$ such that $\mathsf{Cod}(A_i) \cap \mathsf{Cod}(A_j) = \emptyset$ for all $i \neq j$. Let $A'(i.p) := A_i(p)$ for $p \in \mathsf{Dom}(A_i)$ 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, $\mathsf{y} \notin \{\mathsf{x}\} \cup \mathsf{Cod}(A)$*, and* $\varsigma = \{\mathsf{x} \mapsto \mathsf{y}\}$. *Let $A'(1)$:=y and $A'(2.p)$:=$A(p)$ for all $p \in \mathsf{Dom}(A)$. Then A' is a fresh α-renaming for* x.s *such that the equation* $A'(\mathsf{x.s})$=$\mathsf{y}.(\varsigma(A(\mathsf{s}))$ *holds.*
3. *Let A_i be fresh α-renamings for s_i for $i = 1, \ldots, n{+}1$, such that $\mathsf{Cod}(A_i) \cap \mathsf{Cod}(A_j) = \emptyset$ for all $i \neq j$, and such that $(\bigcup \mathsf{Cod}(A_i) \cup \bigcup Var(\mathsf{s}_i)) \cap \{\mathsf{y}_1, \ldots, \mathsf{y}_n\} = \emptyset$.*
   Let $\varsigma = \bigcup_i^n \{\mathsf{x}_i \mapsto \mathsf{y}_i\}$, for $1 \leq i \leq n$ let $A'(i.1) := \mathsf{y}_i$, for all $p \in \mathsf{Dom}(A_i)$ and $1 \leq i \leq n$ let $A'(i.2.p) := A_i(p)$, and for all $p \in \mathsf{Dom}(A_{n+1})$ let $A'(n{+}1.p) := A_{n+1}(p)$. 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 additionally we have* $A'(\mathtt{letrec}\ \mathsf{x}_1.\mathsf{s}_1; \ldots; \mathsf{x}_n.\mathsf{s}_n\ \mathtt{in}\ \mathsf{s}_{n+1}) = \mathtt{letrec}\ \mathsf{y}_1.\varsigma(A_1(\mathsf{s}_1)); \ldots; \mathsf{y}_n.\varsigma(A_n(\mathsf{s}_n))\ \mathtt{in}\ \varsigma(A_{n+1}(\mathsf{s}_{n+1}))$.
4. *Let A be a fresh α-renaming for* s *and A' be a fresh α-renaming for* d *such that $\mathsf{Cod}(A) \cap \mathsf{Cod}(A') = \emptyset$, and p be the position of the hole in* d. *Let $A''(p) := A(p)$ for $p \in \mathsf{Dom}(A)$ and $A''(p.q) := A'(q)$ for $q \in \mathsf{Dom}(A')$, and let $\varsigma = \{\mathsf{x} \mapsto \mathsf{y} \mid \mathsf{x} \in CV(\mathsf{d}),\ binder(\mathsf{d}[\mathsf{x}], p) = q.1$ and $A'(q.1) = \mathsf{y}\}$. Then A'' is a fresh α-renaming for* d[s] *and* $A''(\mathsf{d}[\mathsf{s}]) = A(\mathsf{d})[\varsigma(A'(\mathsf{s}))]$.

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

Definition 2.8. LRS-expressions $\mathsf{s}_1, \mathsf{s}_2$ are *α-equivalent*, if there exist fresh α-renamings A_i for s_i, such that $A_1(\mathsf{s}_1) = A_2(\mathsf{s}_2)$. 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 2.9. A *meta-variable substitution* ρ maps a finite set of meta-variables to variables, expressions, environments, and contexts respecting their types and classes. We say ρ is *ground* iff it maps all variables in $\mathsf{Dom}(\rho)$ to LRS-expressions.

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

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

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

Definition 2.10. A *constrained* LRSX*-expression* (s, Δ) consists of an LRSX-expression s and a *constraint tuple* $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ such that 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: $\gamma(s, \Delta) := \{\rho(s) \mid \rho \text{ is ground}, \rho(s) \text{ fulfills the LVC}, \rho \text{ satisfies } \Delta\}$ For an LRSX-expression s, we define $\gamma(s) = \gamma(s, (\emptyset, \emptyset, \emptyset))$.

Example 2.11. 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 where 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 that requires a non-empty context constraint is the following rule from the calculus L_{need} [31] which copies an abstraction into a needed position in a letrec-environment:

$$
\begin{array}{l}
\mathtt{letrec}\ E; X.\lambda W.S; Y.D_1[\mathtt{var}\ X]\ \mathtt{in}\ D[\mathtt{var}\ Y] \\
\quad \to \mathtt{letrec}\ E; X.\lambda W.S; Y.D_1[\lambda W.S]\ \mathtt{in}\ D[\mathtt{var}\ Y].
\end{array}
$$

If D_1 is empty, then the target of the copy operation should be the variable Y in $D[\mathtt{var}\ Y]$. Thus the case $D_1 = [\cdot]$ should be excluded which can be expressed by setting $\Delta_1 = \{D_1\}$.

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 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

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

For renaming sequences $\eta \in RS$ we also use list-notation, writing $rc : \eta'$ where rc is the head and η' the tail. The index i in atomic renaming components $arc \in$ ARC is used for uniquification.

Figure 3: Symbolic α-renamings

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

Figure 4: Syntax of LRSXα

the symbolic renaming. Thus, we extend LRSX such that 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[6].

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 and write both $\langle rc_1, \ldots, rc_n \rangle$ representing all elements of the sequence in order as well as $rc : \eta$ 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 $CV(\alpha_{D,i})$ for a context meta-variable D, or a symbol $LV(\alpha_{E,i})$ 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 meta-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 abbreviation, we sometimes 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* LRSXα (see Fig. 4) extends the syntax of LRSX by adding symbolic α-renamings ξ to each occurrence of expression, environment and context meta-variables and renaming sequences η to all occurrences of concrete variables x or variable 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, such that 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 for uniquification, since there may be several fresh renamings for

[6]Note that this notation is similar and also related to the notation of *suspensions* $\pi{\cdot}X$ in nominal syntax (see e.g. [33]).

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 are 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; $\langle rc_1, \dots, rc_n\rangle \approx \langle rc'_1, \dots, rc'_n\rangle$ if $rc_i \approx rc'_i$ for $i = 1, \dots, n$; $\langle rc_1, \dots, rc_{i-1}, \{\}, rc_{i+1}, \dots, rc_n\rangle \approx \langle rc_1, \dots, rc_{i-1}, rc_{i+1}, \dots, rc_n\rangle$; and if there exists a permutation π on $\{1, \dots, n\}$ such that $arc_i \approx arc'_{\pi(i)}$ then $\{arc_1, \dots, arc_n\} \approx \{arc'_1, \dots, arc'_n\}$.

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.

We introduce well-formedness of LRSXα-expressions, which requires that in sets of renaming components there is at most one renaming component for each meta-variable or variable:

Definition 3.3. An LRSXα-expression s is *well-formed* iff s does not have a renaming sequence which contains a set rc of atomic renaming components, such that $\alpha_{x,i}, \alpha_{x,j} \in rc$ 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 $\mathsf{Dom}(\rho) = MV(s) \cup MV(\Delta)$ such that $\rho(\epsilon(s))$ fulfills the LVC. A *ground and fresh α-renaming* for s and ρ is a function τ such that

- for all expression, context, and environment meta-variables U with $U \in MV(s)$, τ maps $\alpha_{U,i}$ to a fresh α-renaming $\tau(\alpha_{U,i}) = A_{U,i}$ for $\rho(U)$;
- for all variables X, $\tau(\alpha_{X,i})$ is the substitution $\{\rho(X) \mapsto \mathsf{y}_{X,i}\}$ and $\tau(\alpha_{\mathsf{x},i})$ is the substitution $\{\mathsf{x} \mapsto \mathsf{y}_{\mathsf{x},i}\}$;
- for each environment meta-variable E, with $\tau(\alpha_{E,i}) = A_{E,i}$ and $\rho(E) = \mathsf{x}_1.\mathsf{s}_1; \dots; \mathsf{x}_n.\mathsf{s}_n$, $\tau(LV(\alpha_{E,i}))$ is the substitution $\{\mathsf{x}_j \mapsto A_{E,i}(j.1) \mid j = 1, \dots n\}$;
- for each context variable D, with $\tau(\alpha_{D,i}) = A_{D,i}$ and $\rho(D) = \mathsf{d}$ where p is the position of the hole in d, and $A_{D,i}(\mathsf{d}) = \mathsf{d}'$, $\tau(CV(\alpha_{D,i}))$ is the substitution induced by τ between $CV(\mathsf{d})$ and $CV(\mathsf{d}')$, i.e. $\tau(CV(\alpha_{D,i})) = \{\mathsf{x} \mapsto \mathsf{x}' \mid \mathsf{x} \in CV(\mathsf{d}), \mathit{binder}(\mathsf{d}[\mathsf{x}], p) = q.1 \text{ and } A_{D,i}(q.1) = \mathsf{x}'\}$;
- $\tau(\langle c_1, \dots, c_n\rangle)$ is the composition[7] $\tau(c_n) \circ \dots \circ \tau(c_1)$;
- $\tau(\{c_1, \dots, c_n\}) = \tau(c_n) \circ \dots \circ \tau(c_1)$ such that for any permutation on $\{1, \dots, n\}$ the equation $\tau(c_n) \circ \dots \circ \tau(c_1) = \tau(c_{\pi(n)}) \circ \dots \circ \tau(c_{\pi(1)})$ holds[8]

[7] We write $f \circ g$ for the composition defined by $(f \circ g)(x) = f(g(x))$

[8] Note that this excludes such τ which are not invariant w.r.t. permutations.

and such that all co-domains are fresh and pairwise disjoint, that is $\mathsf{Cod}(A_{U,i}) \cap \mathsf{Cod}(A_{U',i'}) = \emptyset$ for $i \neq i'$ or $U \neq U'$, $\mathsf{Cod}(A_{U,i}) \cap \mathsf{Cod}(\tau(\alpha_{x,j})) = \emptyset$, $\mathsf{Cod}(\tau(\alpha_{x,i})) \cap \mathsf{Cod}(\tau(\alpha_{x',i'})) = \emptyset$ for $i \neq i'$ or $x \neq x'$, $\mathsf{Cod}(A_{U,i}) \cap \mathsf{VarCod}(\rho) = \emptyset$, $\mathsf{Cod}(\alpha_{x,i}) \cap \mathsf{VarCod}(\rho) = \emptyset$, $\mathsf{Cod}(A_{U,i}) \cap \mathsf{Cod}(\tau(\alpha_{x,j})) = \emptyset$;

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 $\tau(\eta)(\rho(x))$. For a constrained LRSXα-expression (s, Δ), the concretizations are:

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

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 define introduction of symbolic α-renamings, i.e. how to transform an LRSX-expression s into an LRSXα-expression s', such that 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 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 $\lambda X.\lambda X.\mathsf{var}\ X$:

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

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.

As a further example, we consider the symbolic α-renaming of the expression $\mathtt{letrec}\ E_1; E_2; E_3\ \mathtt{in}\ \mathtt{letrec}\ E_4\ \mathtt{in}\ S$:

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

In this example no further simplification of the symbolic renamings is possible. However, if we assume that there are non-capture constraints $(\mathtt{letrec}\ E_i\ \mathtt{in}\ c, \mathtt{letrec}\ E_j\ \mathtt{in}\ [\cdot])$ 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

$$
\begin{array}{lcl}
AR(x) & = & \langle\rangle{\cdot}x\\
AR(S) & = & \alpha_{S,i}{\cdot}S\\
AR(D[s]) & = & \alpha_{D,j}{\cdot}D[\mathit{sift}(CV(\alpha_{D,j}), AR(s))]\\
AR(f\ s_1 \ldots s_n) & = & f\ AR(s_1) \ldots AR(s_n)\\
AR(x.s) & = & \alpha_{x,i}{\cdot}x.\mathit{sift}(\langle\alpha_{x,i}\rangle, AR(s))
\end{array}
$$

$$
\begin{array}{l}
AR(\texttt{letrec}\ x_1.s_1; \ldots; x_m.s_m; E_1; \ldots; E_n\ \texttt{in}\ s)\\
\quad = \texttt{letrec}\ \alpha_{x_1,i_1}{\cdot}x_1.\mathit{sift}(\eta, AR(s_1));\\
\qquad\qquad \ldots;\\
\qquad\qquad \alpha_{x_m,i_m}{\cdot}x_m.\mathit{sift}(\eta, AR(s_m));\\
\qquad\qquad \langle\alpha_{E_1,j_1},\eta_1\rangle{\cdot}E_1; \ldots; \langle\alpha_{E_n,j_n},\eta_n\rangle{\cdot}E_n\\
\qquad \texttt{in}\ \mathit{sift}(\eta, AR(s))
\end{array}
$$

where $\eta = (\bigcup_{k=1}^{m}\{\alpha_{x_k,i_k}\}) \cup (\bigcup_{k=1}^{n}\{LV(\alpha_{E_k,j_k})\})$
and $\eta_k = \eta \setminus LV(\alpha_{E_k,i_k})$

$$
\begin{array}{lcl}
\mathit{sift}(\eta, x.s) & = & x.\mathit{sift}(\eta, s)\\
\mathit{sift}(\eta, f\ s_1 \ldots s_n) & = & f\ \mathit{sift}(\eta, s_1) \ldots \mathit{sift}(\eta, s_n)\\
\mathit{sift}(\eta, \eta'{\cdot}S) & = & (\eta' +\!\!+\ \eta){\cdot}S\\
\mathit{sift}(\eta, \eta'{\cdot}D[s]) & = & (\eta' +\!\!+\ \eta){\cdot}D[\mathit{sift}(\eta, s)]
\end{array}
$$

$$
\begin{array}{l}
\mathit{sift}(\eta, \texttt{letrec}\ z_1.s_1;\ldots;z_m.s_m;\eta_1{\cdot}E_1;\ldots;\eta_n{\cdot}E_n\ \texttt{in}\ s)\\
\quad = \texttt{letrec}\ \ z_1.\mathit{sift}(\eta, s_1); \ldots; z_m.\mathit{sift}(\eta, s_m);\\
\qquad\qquad (\eta_1 +\!\!+\ \eta){\cdot}E_1; \ldots; (\eta_n +\!\!+\ \eta){\cdot}E_n\\
\qquad \texttt{in}\ \mathit{sift}(\eta, s)\\
\mathit{sift}(\eta, \eta'{\cdot}x) \quad = \quad (\eta' +\!\!+\ \eta){\cdot}x
\end{array}
$$

For LRSX-expression s, $AR(s)$ computes a symbolically α-renamed LRSXα-expression. Here all $\alpha_{U,i}$ on right hand sides of equations are assumed to be fresh and pairwise distinct, which can be guaranteed for instance, by using a global counter.

Figure 5: Adding symbolic α-renamings

E_j and thus the LRSXα-expression could be simplified to

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

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 2.7 imply:

Proposition 3.8. *Let s be an* LRSX*-expression and $s' = AR(s)$. Then for each $\mathsf{t} \in \gamma(s)$, there exists $\mathsf{t}' \in \gamma(s')$ such that $\mathsf{t} \sim_\alpha \mathsf{t}'$ and for each $\mathsf{t}' \in \gamma(s')$ there exists $\mathsf{t} \in \gamma(s)$ such that $\mathsf{t} \sim_\alpha \mathsf{t}'$. Furthermore all $\mathsf{t}' \in \gamma(s')$ fulfill the DVC.*

4 SIMPLIFICATION OF α-RENAMINGS

The goal of this section is to define a sound mechanism to simplify symbolic α-renamings. Hence, we present an inference system which performs such a simplification and subsequently we show correctness of the system, i.e. simplification of symbolic α-renamings does not change the set of concretizations.

As a preparation we first consider a preprocessing step of non-capture constraints, i.e. we compute so-called atomic NCCs which

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

Figure 6: The functions Var_M and CV_M

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

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

where the functions Var_M, and CV_M are shown in Fig. 6. Var_M computes the variables and meta-variables (together with their symbolic alpha-renaming) of a meta-expression and CV_M collects all variables and meta-variables (together with their symbolic alpha-renaming) which may capture variables if plugged into the context hole. E.g., we have $\mathit{Var}_M(\lambda X.\mathit{app}\ (\texttt{var}\ \mathsf{y})\ S) = \{X, \mathsf{y}, S\}$ and $CV_M(D_1[D_2[\lambda X.\mathit{app}\ [\cdot]\ \mathsf{y}]]) = \{D_1, X\}$ provided that $cl(D_2)$ is non-capturing, while $cl(D_1)$ is capturing. 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 the following equations hold:

$$
\begin{array}{rl}
\mathit{Var}(\tau(\rho(s))) = & \{\mathit{Var}(\tau(\rho(u))) \mid u \in \mathit{Var}_M(s)\}\\
CV(\tau(\rho(d))) = & \{\tau(\rho(\eta{\cdot}x)) \mid \eta{\cdot}x \in CV_M(d)\}\\
& \cup\{LV(\tau(\rho(\xi{\cdot}E))) \mid \xi{\cdot}E \in CV_M(d)\}\\
& \cup\{CV(\tau(\rho(\xi{\cdot}D))) \mid \xi{\cdot}D \in CV_M(d)\}
\end{array}
$$

As a further preparation for simplification, we define two kinds of relationships between symbolic renamings. Roughly speaking, a renaming sequence η_1 is an *instance* of η_2 if in η_1 compared to η_2 some sets of renaming components $\{arc_1, \ldots, arc_n\}$ are partly ordered, e.g. replaced by sequences $\langle rc_1, \ldots, rc_m\rangle$ such that $\bigcup_i rc_i = \{arc_1, \ldots, arc_n\}$ and $rc_i \cap rc_j = \emptyset$ for $i \neq j$. Furthermore, η_1 is a weak instance of η_2 if it is an instance after forgetting about the concrete indexes i in $\alpha_{U,i}$, $CV(\alpha_{D,i})$, and $LV(\alpha_{E,i})$.

Definition 4.2. The relation $=_{\mathsf{num}}$ identifies renaming components and sequences up to the number i in $\alpha_{U,i}$, 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. A renaming η_1 is *an instance* of a renaming η_2 if

- $\eta_1 = \eta_2$, or
- $\eta_1 = rc_1 : \eta_1'$, $\eta_2 = rc_2 : \eta_2'$, $rc_1 \subseteq rc_2$, and η_1' is an instance of $(rc_2 \setminus rc_1) : \eta_2'$.

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

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

$$\text{(RemDup)}\ \frac{arc =_{\mathsf{num}} arc' \text{ or } (arc = \alpha_{U,i} \text{ and } arc' \in \{LV(\alpha_{U,j}), CV(\alpha_{U,j})\})}{\eta_1 +\!\!+ (arc \mathbin{\dot\cup} rc):\eta_2 +\!\!+ (arc' \mathbin{\dot\cup} rc'):\eta_3\cdot U \models_\Delta \eta_1 +\!\!+ (arc \mathbin{\dot\cup} rc):\eta_2 +\!\!+ rc':\eta_3\cdot U}$$

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

(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.).

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

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

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

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

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

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

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

Figure 7: Simplification of symbolic α-renamings

A renaming η_1 is *a weak instance* of a renaming η_2 if

- η_1 is an instance of η_2, or
- $\eta_1 = rc_1 : \eta'_1$, $\eta_2 = rc_2 : \eta'_2$, $rc_1 \subseteq_{\mathsf{w}} rc_2$, and 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. As an example we consider the symbolic renaming $\langle\alpha_{S,1}, CV(\alpha_{D_2,1}), \{CV(\alpha_{D_1,1}), LV(\alpha_{E_1,1})\}, LV(\alpha_{E_2,1})\rangle$. It 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 one to switch between the copies of atomic renaming components, and thus e.g. the renaming $\langle\alpha_{S,1}, CV(\alpha_{D_2,2}), \{CV(\alpha_{D_1,1}), LV(\alpha_{E_1,2})\}, LV(\alpha_{E_2,1})\rangle$ is not an instance but a weak instance of the symbolic renaming $\langle\alpha_{S,1}, \{CV(\alpha_{D_1,1}), CV(\alpha_{D_2,1}), LV(\alpha_{E_1,1})\}, LV(\alpha_{E_2,1})\rangle$.

Definition 4.4. We consider formal expressions of the form x, $\mathsf{VAR}(U)$, and $\mathsf{COD}(arc)$, which we call *symbolic set-variables*, and let V be a set of such formal expressions. 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 $\tau(\rho(\mathsf{VAR}(U)))) := \mathit{Var}(\rho(U))$, $\tau(\rho(x)) = \{\rho(x)\}$, and $\tau(\rho(\mathsf{COD}(arc))) = \mathtt{Cod}(\tau(arc))$.

Simplification removes renaming components if they cannot affect (instances of) the corresponding meta symbol. Information is gathered from the renamings and from the NCCs in Δ_3.

Definition 4.5 (Simplification). The simplification relation $\models_\Delta$ is defined by the inference rules 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 $arc \not\sim_\Delta v$ which is defined in Fig. 7 (c).

Note that in the presented form, the inference system is non-deterministic and does not necessarily have unique normal forms. Our implementation (see Section 6) uses the following strategy: It applies rules (Order) and (MSet) as late as possible, and for instance for rule (Order) it tries all possible orderings and heuristically chooses the most-simplified result. We leave the development of a normalizing system as future work.

Definition 4.6. 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 (see Definition 4.5).

Axioms (IdU), (IdX), and (IdEta) allow one to keep the renaming and rules (TrU) and (TrX) enable transitivity of simplification. Rule (RemDup) removes a duplicated renaming component in a sequence. Rule (SubstX) removes further renaming components for a renaming for x if the first component includes $\alpha_{x,i}$. 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 one 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 one to order a set of atomic renaming components for further simplification, rule (MSet) allows one 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 $\not\approx_\Delta$ is defined in Fig. 7 (c) where $arc \not\approx_\Delta v$ 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 4.7. We reconsider the expressions from Example 3.6. If we apply the simplification algorithm to the constrained expression $(\lambda\alpha_{X,1}{\cdot}X.\lambda\alpha_{X,2}{\cdot}X.\mathtt{var}\ \langle\alpha_{X,2},\alpha_{X,1}\rangle{\cdot}X,(\emptyset,\emptyset,\emptyset))$ then it 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

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

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

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,k with $\{i,j,k\} = \{1,2,3\}$ (see Fig. 8).

Proposition 4.8. *Let M be a set of meta-variables, Δ be a constraint tuple with $MV(\Delta) \subseteq M$, ρ be ground substitution for M, and τ be a ground α-renaming for ρ and M, such that ρ and τ satisfy Δ.*

(1) *(Correctness of $\not\approx_\Delta$) Let v be a symbolic set-variable and arc be an atomic renaming component over M, such that $arc \not\approx_\Delta v$. Then for all $\mathsf{x} \in \tau(\rho(v))$, 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, such that $V,\eta \vdash_\Delta \eta'$. 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 and η' be symbolic α-renamings with components over M, such that $\eta{\cdot}x \models_\Delta \eta'{\cdot}x$. Then the equation $\tau(\eta)(\rho(x)) = \tau(\eta')(\rho(x))$ holds.*
   (b) *Let ξ and ξ' be symbolic α-renamings with components over M, and let $U \in M$ such that $\xi{\cdot}U \models_\Delta \xi'{\cdot}U$. Then the equation $\tau(\xi)(\rho(U)) = \tau(\xi')(\rho(U))$ holds.*

Applying the previous proposition for all occurrences $\eta{\cdot}x$ and $\xi{\cdot}U$ which are transformed by the simplification algorithm shows:

Theorem 4.9. *The simplification algorithm does not change the set of concretizations, i.e. for a constrained* LRSXα*-expression (s,Δ) such that 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

We show how to rewrite LRSXα-expressions by matching LRSXα-expressions 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 [26]) are rewrite rules of the form $\ell \to_\Delta r$ 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 such that 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 $s \to \sigma(r)$ (or more precisely $(s,\nabla) \to (\sigma(r),\nabla \cup \sigma(\Delta))$). In [26] 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 $MV_F(\cdot)$ we denote functions to compute the sets of instantiable and fixed meta-variables.

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 $s \trianglelefteq t$ such that 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

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

Figure 8: Derivation for Example 4.7

$i = 1, 2, 3$ 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 such that for any ground substitution ρ together with a ground renaming τ with $\mathsf{Dom}(\rho) = MV_F(P)$ such that $\tau \circ \rho$ satisfies ∇, and $\tau(\rho(\sigma(s))), \tau(\rho(t))$ 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 $\mathsf{Dom}(\rho_0) = MV_I(\rho(\sigma(\Delta)))$ such that $\tau(\rho_0(\rho(\sigma(\Delta))))$ 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 $\ell \to_\Delta r$ 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 of a rewrite rule which introduces meta-variables is the rule (abs) which shares the arguments of a function symbol application: $(f\ s_1 \ldots s_n) \to_\Delta$ $\mathtt{letrec}\, X_1.s_1; \ldots; X_n.s_n\ \mathtt{in}\, (f\ (\mathtt{var}\, X_1) \ldots (\mathtt{var}\, X_n))$ where Δ contains NCCs that ensure that $X_1, \ldots, X_n$ are fresh w.r.t. $s_1, \ldots, s_n$.

In [26] 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 expressions. Transform the LRSXα-expressions on right-hand sides of Γ and in ∇ into LRSX-expressions by replacing all occurrences $\xi \cdot U, \xi' \cdot U$ with $\xi \approx \xi'$ by a single fresh fixed meta-variable U' (of the same kind as U) and by replacing $\eta{\cdot}x, \eta'{\cdot}x$ with $\eta \approx \eta'$ by a fresh variable x'. Now apply the matching algorithm for LRSX of [26] until a solution $(Sol_F, \Delta_F, \nabla_F)$ is produced. Then construct $(Sol_O, \Delta_O, \nabla_O)$ by replacing U' by $\xi \cdot U$ 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.

Definition 5.2. Let $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ and $\nabla = (\nabla_1, \nabla_2, \nabla_3)$ be constraint tuples over LRSXα such that $MV_I(\nabla) = \emptyset$ 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) $(\xi{\cdot}u, \xi'{\cdot}v) \in split_{NCC}(\nabla_3)$.

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

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

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 $i \neq i'$.

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

(3) $u = v$ and ($u = D$ or $u = E$ with $E \notin \Delta_2$).
(4) $u \neq v$ and ($u \in \{S, D, E, X\}$ or $v \in \{D, E, X\}$).
(5) $\xi' = \langle\rangle$ and $v = E$ or $v = D$ and $(v, v) \in split_{NCC}(\nabla_3)$.
(6) $\xi = \langle\rangle$, $\xi' = \langle\rangle$, 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 the membership $(v, u) \in split_{NCC}(\nabla_3)$ holds.
(7) $v_1 \bowtie v_2$ 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 that 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).

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, such that $\tau \circ \rho$ satisfies ∇. Then there exists a ground substitution ρ_0 with* $\mathsf{Dom}(\rho_0) = MV_I(\rho(\Delta))$ *such that $\tau(\rho_0(\rho(\Delta)))$ is satisfied.*

Soundness of the matching algorithm for LRSX [26] implies:

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

Example 5.5. As an example for rewriting of LRSXα-expression, which also illustrates the necessity of simplification, consider the transformation (ucp) which inlines a binding that is used only once. The transformation can be expressed as the meta letrec rewrite rule $\mathtt{letrec}\ X.S\ \mathtt{in\ var}\ X \to_{\emptyset,\emptyset,(S,\lambda X.[\cdot])} S$ where the NCC $(S, \lambda X.[\cdot])$

ensures that X does not occur in S. For the constrained expression ($\texttt{letrec}\ Y.S_0\ \texttt{in var}\ Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$, α-renaming results in ($\texttt{letrec}\ \alpha_{Y,1}{\cdot}Y.\langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0\ \texttt{in var}\ \alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$.

Matching the left hand side of the transformation (ucp) against this constrained LRSXα-expression fails, since for the substitution $\sigma = \{X \mapsto \alpha_{Y,1}{\cdot}Y, S \mapsto \langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0\}$ the 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 ($\texttt{letrec}\ \alpha_{Y,1}{\cdot}Y.\langle\alpha_{S_0,1}, \alpha_{Y,1}\rangle{\cdot}S_0\ \texttt{in var}\ \alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$ leads to ($\texttt{letrec}\ \alpha_{Y,1}{\cdot}Y.\alpha_{S_0,1}{\cdot}S_0\ \texttt{in var}\ \alpha_{Y,1}{\cdot}Y, (\emptyset, \emptyset, (S_0, \lambda Y.[\cdot])))$ and matching the left hand side of (ucp) against it delivers the matcher $\sigma = \{X \mapsto \alpha_{Y,1}{\cdot}Y, S \mapsto \alpha_{S_0,1}{\cdot}S_0\}$ where 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 $split_{_{NCC}}(\{(\alpha_{S_0,1}{\cdot}S_0, \lambda\alpha_{Y,1}{\cdot}Y.[\cdot])\}) = \{(\alpha_{S_0,1}{\cdot}S_0, \alpha_{Y,1}{\cdot}Y)\}$ as well as since $\mathsf{VAR}(S_0) \bowtie \mathsf{COD}(\alpha_{Y,1})$ and $\mathsf{COD}(\alpha_{S_0,1}) \bowtie \mathsf{COD}(\alpha_{Y,1})$.

5.2 Refreshing α-Renamings

Matching can be applied to rewrite constrained LRSXα-expressions. If the constrained expression stems from symbolically α-renaming an LRSX-expression, then by Proposition 3.8 it concretizations fulfill the DVC. However, after applying such a rewrite step, the concretizations may no longer fulfill the DVC. For instance, consider a meta letrec rewrite rule that copies a subexpression:

$$\texttt{letrec}\ X.S\ \texttt{in}\ C[\texttt{var}\ X] \to_\Delta \texttt{letrec}\ X.S\ \texttt{in}\ C[S].$$

Applying the rule to $\texttt{letrec}\ \alpha_{X,1}{\cdot}X.\alpha_{S_0,1}{\cdot}S_0\ \texttt{in var}\ \alpha_{X,1}{\cdot}X$ results in $\texttt{letrec}\ \alpha_{X,1}{\cdot}X.\alpha_{S_0,1}{\cdot}S_0\ \texttt{in}\ \alpha_{S_0,1}{\cdot}S_0$. 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 one needs to introduce renaming components of the form $\alpha_{\xi{\cdot}S,i}$ representing an α-renaming of already α-renamed expressions). We choose a simpler approach. It 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. For a constrained LRSXα-expression (s, Δ), $refresh(s, \Delta)$ 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 $\mathsf{t} \in \gamma(s, \Delta)$ there exists $\mathsf{t}' \in \gamma(s', \Delta)$ with $\mathsf{t} \sim_\alpha \mathsf{t}'$ and for each $\mathsf{t}' \in \gamma(s', \Delta)$ there exists $\mathsf{t} \in \gamma(s, \Delta)$ with $\mathsf{t} \sim_\alpha \mathsf{t}'$.*

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 extended 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 s' and then recursively works along the structure of s and s' in parallel and tries to renumber all symbolic α-renamings such that

- 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 fresh) 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 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 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, and the extended α-equivalence check for s and s' is valid.

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 such that $\rho_0(Y_k) = \tau(\rho(\alpha_{X,k}{\cdot}X_k))$ 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 $\tau(\rho(s'_0)) \sim_\alpha \tau(\rho_0(s'_1))$. We 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. *Assume that the constrained LRSXα-expression (s, Δ) and (s', Δ') pass the extended α-equivalence check. Let ρ be a ground substitution with $\mathsf{Dom}(\rho) = MV(s) \cup MV(s') \cup MV(\Delta) \cup MV(\Delta')$ and let τ be a ground renaming for ρ such that $\tau \circ \rho$ satisfies $\Delta_1, \Delta_2, \Delta'_1, \Delta'_2$. Then i) $\tau \circ \rho$ satisfies Δ iff $\tau \circ \rho$ satisfies Δ' and ii) $\tau(\rho(s)) \sim_\alpha \tau(\rho(s'))$.*

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} [31], and the calculus LR [32] (which extends L_{need} by data constructors for lists,

Table 1: Statistics of executing the LRSX Tool

	# 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

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 and rewriting of LRSXα-expressions, refreshing of symbolic α-renamings and checking extended α-equivalence of LRSXα-expressions. While we have shown soundness of the algorithms, we did not consider completeness which is left for further work. The algorithms are 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of PPDP 2017 for their very valuable comments and suggestions to improve the paper.

REFERENCES

[1] Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need lambda Calculus. *J. Funct. Program.* 7, 3 (1997), 265–301.
[2] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. A call-by-need lambda calculus. In *POPL 1995*. ACM, 233–246.
[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *POPL 2008*. ACM, 3–15.
[4] Christophe Calvès and Maribel Fernández. 2008. Nominal Matching and Alpha-Equivalence. In *WoLLIC 2008 (LNCS)*, Vol. 5110. Springer, 111–122.
[5] Christophe Calvès and Maribel Fernández. 2008. A polynomial nominal unification algorithm. *Theor. Comput. Sci.* 403, 2-3 (2008), 285–306.
[6] Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reasoning* 49, 3 (2012), 363–408.
[7] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.
[8] Maribel Fernández and Murdoch Gabbay. 2007. Nominal rewriting. *Inf. Comput.* 205, 6 (2007), 917–965.
[9] Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.
[10] Jordi Levy and Mateu Villaret. 2008. Nominal Unification from a Higher-Order Perspective. In *RTA 2008 (LNCS)*, Vol. 5117. Springer, 246–260.
[11] Elena Machkasova. 2007. Computational Soundness of a Call by Name Calculus of Recursively-scoped Records. In *WRS 2007 (ENTCS)*.
[12] Elena Machkasova and Franklyn A. Turbak. 2000. A Calculus for Link-Time Compilation. In *ESOP 2000 (LNCS)*, Vol. 1782. Springer, 260–274.
[13] Conor McBride and James McKinna. 2004. Functional pearl: i am not a number–i am a free variable. In *Haskell 2004*. ACM, 1–9.
[14] James Hiram Morris. 1968. *Lambda-Calculus Models of Programming Languages.* Ph.D. Dissertation. MIT.
[15] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49.
[16] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *PLDI 1988*. ACM, 199–208.
[17] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *CADE 1999 (LNCS)*, Vol. 1632. Springer, 202–206.
[18] Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL 2008*. ACM, 371–382.
[19] Brigitte Pientka and Andrew Cave. 2015. Inductive Beluga: Programming Proofs. In *CADE 2015 (LNCS)*, Vol. 9195. Springer, 272–281.
[20] Brigitte Pientka and Joshua Dunfield. 2008. Programming with proofs and explicit contexts. In *PPDP 2008*. ACM, 163–173.
[21] Andrew Pitts. 2016. Nominal Techniques. *ACM SIGLOG News* 3, 1 (2016), 57–72.
[22] Gordon D. Plotkin. 1975. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.* 1 (1975), 125–159.
[23] Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. 2012. A Canonical Locally Named Representation of Binding. *J. Autom. Reasoning* 49, 2 (2012), 185–207.
[24] Conrad Rau, David Sabel, and Manfred Schmidt-Schauß. 2012. Correctness of Program Transformations as a Termination Problem. In *IJCAR 2012 (LNCS)*, Vol. 7364. Springer, 462–476.
[25] David Sabel. 2017. *Alpha-Renaming of Higher-Order Meta-Expressions.* Frankfurter Informatik-Berichte 2017-2. Goethe-University Frankfurt am Main. http://goethe.link/fib-2017-2
[26] David Sabel. 2017. *Rewriting of Higher-Order Meta-Expressions with Recursive Bindings.* Frankfurter Informatik-Berichte 2017-1. Goethe-University Frankfurt am Main. http://goethe.link/fib-2017-1
[27] David Sabel and Manfred Schmidt-Schauß. 2008. A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations. *Math. Structures Comput. Sci.* 18, 03 (2008), 501–553.
[28] Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. 2017. Nominal Unification of Higher Order Expressions with Recursive Let. In *LOPSTR 2016, Revised Selected Papers (LNCS)*, Vol. 10184. Springer, 328–344.
[29] Manfred Schmidt-Schauß, Conrad Rau, and David Sabel. 2013. Algorithms for Extended Alpha-Equivalence and Complexity. In *RTA 2013 (LIPIcs)*, Vol. 21. Schloss Dagstuhl, 255–270.
[30] Manfred Schmidt-Schauß and David Sabel. 2016. Unification of Program Expressions with Recursive Bindings. In *PPDP 2016*. ACM, 160–173.
[31] Manfred Schmidt-Schauß, David Sabel, and Elena Machkasova. 2010. Simulation in the Call-by-Need Lambda-Calculus with letrec. In *RTA 2010 (LIPIcs)*, Vol. 6. Schloss Dagstuhl, 295–310.
[32] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. 2008. Safety of Nöcker's Strictness Analysis. *J. Funct. Programming* 18, 04 (2008), 503–551.
[33] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. 2003. Nominal Unification. In *CSL 2003 (LNCS)*, Vol. 2803. Springer, 513–527.
[34] Joe B. Wells, Detlef Plump, and Fairouz Kamareddine. 2003. Diagrams for Meaning Preservation. In *RTA 2003 (LNCS)*, Vol. 2706. Springer, 88 –106.