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

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 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 Tool2 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 has a letrec-construct letrec \( E_1 = s_1; \ldots; x_n = s_n \) where \( x_1 = s_1; \ldots; x_n = s_n \) is an unordered sets of
recursive bindings (the scope of the letrec-bound variables \( x_1 \) is \( s_1; \ldots; s_n + 1 \)). 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):

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

which joins two nested letrec-environments and where \( S \) is a meta-variable for an arbitrary expression, \( C \) is a meta-variable for an
arbitrary context, and \( E_1, E_2 \) are meta-variables for arbitrary letrec-environments. Using this rule without constraints would allow
one to instantiate the meta-variable \( E_1 \) by the environment which consists of a single binding \( x = y \), meta-variable \( E_2 \) by an
environment which consists of a single binding \( y = \text{True} \), meta-variable \( S \) by \( x \), and meta-variable \( C \) by the empty context resulting
in the instantiated rule letrec \( x = y \) in letrec \( y = \text{True} \) in \( x \) \xrightarrow{\text{llet}} \text{letrec } x = y; y = \text{True} \text{ in } x \) which however should be forbidden,
since variable \( y \) in \( x = y \) is a free occurrence in the left expression, but becomes a bound occurrence (captured by the binding \( y = \text{True} \) ) 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

\footnotesize{\begin{itemize}
  \item This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1
  \item See [24] for an automation of this step using automated termination provers.
\end{itemize}}
instantiation $\rho$ if context $\rho(d)$ does not capture any variable of $\rho(s)$. For our example, $(s_0, d_0) = (\text{lletrec } E_1 \text{ in } \text{True}, \text{lletrec } E_2 \text{ in } [\cdot])$ 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):

$$
\begin{align*}
\text{lletrec } E_1 \text{ in } \\
\text{lletrec } E_2 \text{ in } S \\
\text{llet } \\
\text{lletrec } E_1; E_2 \text{ in } S \\
\text{lletrec } E_1; E_2 \text{ in } S \\
\text{llet } \\
\text{lletrec } E_1; E_2 ; E_1; E_2 \text{ in } S \\
\text{lletrec } E_1; E_2; E_1; E_2 \text{ in } S
\end{align*}
$$

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 upper-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 $\rho = (E_1 \mapsto x = z, E_2 \mapsto y = \text{True}, E'_2 \mapsto z = \text{True}, S \mapsto x)$, the suggested join would lead to $\rho(\text{lletrec } E_1; E_2; E'_2 \text{ in } S) = \text{lletrec } x = z; y = \text{True}; z = \text{True} \text{ in } x$ which illegally captures the variable $z$.

The solution to attack those problems in a pen-and-paper-proof is to rename binders by fresh $\alpha$-renamings. For the above instance, we may $\alpha$-rename letrec $x = z; y = \text{True}$ in letrec $z = \text{True} \text{ in } x$ into letrec $x = z; y = \text{True}$ in letrec $z' = \text{True} \text{ in } x$ and $\alpha$-rename letrec $x = z$ in letrec $y = \text{True}$; $z = \text{True} \text{ in } S$ into letrec $x = z$ in letrec $y = \text{True}$; $z' = \text{True} \text{ in } S$ 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 letrec $E_1; E_2 \text{ in } S$ to guarantee that for all instantiations $\rho$ the letrec-bound variables of $\rho(E'_1)$ do not capture variables of $\rho(E_1; E_2)$. Furthermore, an appropriate mechanism of such a symbolic $\alpha$-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 $\alpha U \cdot U$ for a meta-variable $U$ (which may be an environment variable, an expression variable, a context variable) with the meaning that instantiations $\rho$ guarantee that $\rho(\alpha U \cdot U)$ is an $\alpha$-renamed copy of $\rho(U)$ where the $\alpha$-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 $\rho(\alpha U \cdot U)$. Since these renamings affect also other subexpressions, we have to distribute them along the term and binding structure.

Thus to treat $\alpha$-renamings, we extend the LRSX by syntactic constructs to represent the $\alpha$-renamings. The extended language is called LRSX$\alpha$. Adding such a syntactic support for $\alpha$-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 LRSX$\alpha$-expressions, further results of this paper target basic reasoning tasks with LRSX- and LRSX$\alpha$-expressions. A first algorithm performs $\alpha$-renaming, i.e. it takes an LRSX-meta expression and delivers an LRSX$\alpha$-meta expression such that on the semantic level the instances are $\alpha$-renamed by a fresh renaming. A further procedure performs simplification of symbolic $\alpha$-renamings, i.e. it deduces that parts of the symbolic renamings can be removed. This procedure is important for our automated tool, since in the tool equivalence of expressions has to be detected and without simplification of renamings this is impossible in many cases. We provide an adaptation of the matching algorithm from [26] such that LRSX$\alpha$-expressions can be matched against LRSX-expressions which allows one to rewrite LRSX$\alpha$-expressions. However, this may require one to adapt the symbolic $\alpha$-renaming after a rewrite step and thus we present an algorithm for this task. We finally present a test to check $\alpha$-equivalence of LRSX$\alpha$-expressions.

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

A further possibility to avoid explicit $\alpha$-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 (let).

Using freshness constraints, instances of nominal meta-terms can be restricted to ensure that the distinct variable convention holds. However, this requires knowledge about the binders (to form freshness constraints). Our approach is more general since it includes meta-syntax with meta-variables representing contexts and parts of 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-renamed 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 F and a set K of context classes. A context class K ∈ K is a set of contexts which is provided as a part of the input and defined by a grammar4.

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 variables, HExpr which are higher-order expressions, Env representing letrec-environments, and Bind representing letrec-bindings. Elements s of HExpr have an order(s) ∈ N₀, where HExpr₀ denotes the elements of HExpr of order n, and where HExpr = Expr.

Each f ∈ F has a syntactic type f : i₁ → · · · → in → Expr, where i₁ may be Var, or HExpr; n is called the arity of f, denoted ar(f); and the order arity oar(f) is the n-tuple (δ₁, . . . , δ)n, where δi ∈ N₀, or δi = Var, depending on the type of f. We assume that (var, λ) ∈ F where var : Var → Expr lifts variables to expressions with oar(var) = (λ), and oar(λ) = (1).

Example 2.2. The identity is written as λ(x.var x). Applications can be represented by a symbol app with oar(app) = (0, 0).

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

Definition 2.3. 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(env) we denote the let-bound variables of env, i.e. all x with env = env′; x.s.

For instance, the expression letrec x.var x;x.var true in x does not fulfill the LVC while letrec x.var x;y.var true in 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 λx.λx.var x into λx₁,λx₂.var x₂ 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, Pos(r), are inductively defined as follows where w.l.o.g. we assume xi < xj for 1 ≤ i < j ≤ n:

Pos(x) := {x} 
Pos(f r₁ . . . rn) := {x} ∪ ∪n
i=1 {i.p | p ∈ Pos(r₁)}
Pos(letrec x₁,s₁; . . . ; xn,sn in t) := 
{var} ∪ ∪n
i=1 {i.p | p ∈ Pos(s₁)} ∪ {(n + 1).p | p ∈ Pos(t)}
Pos(x.r) := {1, 1} ∪ {2.p | p ∈ Pos(t)}

For a position p ∈ Pos(r), we denote with ri,p the term at position p, inductively defined by ri,p := r. r₁,x₁,r₂,x₂.p := ri,p, (letrec x₁,s₁; . . . ; xn,sn in t)||₁.i := xi for 1 ≤ i ≤ n, and letrec x₁,s₁; . . . ; xn,sn in t)|₂,i.p := s₁|p for 1 ≤ i ≤ n, and (letrec x₁,s₁; . . . ; xn,sn in t)|n+1.p := t|p, and (f r₁ . . . rn)|i.p := x.
r|_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 BPos(r) the binder positions of r. With BV(r) we denote the set of bound variables of r, i.e. \( BV(r) = \{ r|_p \mid p \in BPos(r) \} \).

If r|_p = x and p is not a binder position of r, the occurrence of x at position 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 r|_q is a letrec-expression such that r|_q,i\_1 = x and q.1 is a binder position, then x at position p is a bound occurrence, otherwise it is a free occurrence. For a bound occurrence of x at p, its corresponding binder is q.1 (written binder(r, p) = q.1) where q is maximal. The set of free variables of r is \( FV(r) := \{ r|_p \mid r_p = x \text{ and x at position p is a free occurrence} \} \). We set \( Var(r) := FV(r) \cup BV(r) \). For functions h, we denote by Dom(h) its domain, by Cod(h) its codomain, and (if the co-domain of h consists of expressions), with \( Var(Cod(h)) \) the variables appearing in its co-domain, i.e. \( Var(Cod(h)) = \bigcup \{ Var(bh(U)) \mid U \in Dom(h) \} \).

For an expression r, an α-renaming \( A : BPos(r) \rightarrow Var \) computes a variable for each binder position where the following condition must hold: For each free occurrence 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) = 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 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 \( s = \lambda x.\lambda x.\text{var} \ x \), the positions of s are \( Pos(s) = \{ 1, 1.1, 1.2, 1.2.1, 1.2.1.1, 1.2.1.2, 1.2.1.2.1 \} \) and \( s|_{1.2.1.2.1} = (x.\lambda x.\text{var} \ x) \). 1.1 = (\( \lambda x.\text{var} \ x \)) \_(1.1) = (x.\text{var} \ x) \_1 = x_{1.1} = x \). The positions 1.1, 1.2, 1.1.1, 1.2.1.2 are variable positions where \( BPos(s) = \{ 1.1, 1.2, 1.2.1, 1.2.1.1 \} \) are binder positions, the occurrence of x at position 1.2.1.2 is a bound occurrence where the corresponding binder is 1.2.1.1. The α-renaming \( A = \{ 1.1 \rightarrow x_1, 1.2.1.1 \rightarrow x_2 \} \) is a fresh α-renaming for s and \( A(s) = \lambda x_1.\lambda x_2.\text{var} \ x_2 \) while A′ = \( \{ 1.1 \rightarrow y, 1.2.1.1 \rightarrow y \} \) is an α-renaming (which is not fresh for s) such that A′(s) = \( \lambda x.\lambda y.\text{var} \ y \). For \( s = \lambda x.\text{var} \ y \), the mapping 1.1 \( \rightarrow \) 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 holds for the expression.

Definition 2.6. An expression s satisfies the distinct variable convention (DVC) iff \( BV(s) \cap FV(s) = \emptyset \) and all binders bind different variables.

A position p \( \in \) Pos(r) is an expression position iff r|_p \( \in \) HExpr. 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(d) we denote the set of variables x which are captured if they are plugged into the hole of d; i.e. if the hole of d is at position p then x \( \in \) CV(d) iff the occurrence of x at position p.1 in d[\text{var} x]

\( r|_p \) is a bound occurrence. A context class \( \mathcal{K} \) is non-binding if for all contexts d of class \( \mathcal{K} \), CV(d) = \{ \}.

The following lemma expresses how to iteratively construct a fresh α-renaming. In the lemma, \( \zeta \) represents a substitution that maps variables to variables and applying \( \zeta \) to an LRS-expression means to apply \( \zeta \) 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 \( Cod(A_i) \cap Cod(A_j) = \emptyset \) for all \( i \neq j \). Let \( A'(i,p) := A_i(p) \) for \( p \in Dom(A_i) \) and \( 1 \leq i \leq n \). Then \( A' \) is a fresh α-renaming for \( (f \ s_1 \ldots s_n) \) and \( A'(f \ s_1 \ldots s_n) = f \ A_1(s_1) \ldots A_n(s_n) \).

2. Let A be a fresh α-renaming for s, \( \forall \xi \in \{ x \} \cup Cod(A) \), and \( \zeta = \langle x \mapsto y \rangle \). Let \( A'(1) := \gamma \) and \( A'(2.p) := \lambda p \) for all p \( \in \) Dom(A). Then \( A' \) is a fresh α-renaming for x.s such that the equation \( A'(x.s) = (\gamma \zeta(A(s))) \) holds.

3. Let \( A_i \) be fresh α-renamings for \( s_i \) for \( i = 1, \ldots, n \) such that \( Cod(A_i) \cap Cod(A_j) = \emptyset \) for all \( i \neq j \), and such that \( (\{ Cod(A_i) \cup \{ Var(s_i) \}) \cap \{ y_1, \ldots, y_n \} = \emptyset \).

Let \( \zeta = \bigcup \{ x_1 \mapsto y_1 \} \), for \( 1 \leq i \leq n \) let \( A'(i.1) := y_i \), for all \( p \in Dom(A_i) \) and \( 1 \leq i \leq n \) let \( A'(i.2.p) := A_i(p) \), and for all p \( \in \) Dom(A\(n+1\)) let \( A'(n+1.p) := A_{n+1}(p) \). Then \( A' \) is a fresh α-renaming for letrec \( x_1.s_1; \ldots; x_n.s_n \in s_{n+1} \), and additionally we have \( A'(letrec \ x_1.s_1; \ldots; x_n.s_n \in s_{n+1}) = \lambda s_{n+1} \zeta(A(A_i(s_i))) \ldots \gamma_n \zeta(A_n(s_n)) \in \zeta(A_{n+1}(s_{n+1})) \).

4. Let A be a fresh α-renaming for s and \( A' \) be a fresh α-renaming for d such that \( Cod(A) \cap Cod(A') = \emptyset \), and p be the position of the hole in d. Let \( A''(p) := A(p) \) for \( p \in Dom(A) \) and \( A''(q,g) := A'(q,g) \) for \( q \in Dom(A') \), and let \( \zeta = \{ x \mapsto y \mid x \in CV(d) \} \), binder[d[x],p] = q.1 and A′′(q.1) = y. Then \( A''(p) \) is a fresh α-renaming for d[s] and \( A''(d[s]) = A(d)[\zeta(A''(p))]. \)

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 \( s_1, s_2 \) are α-equivalent, if there exist fresh α-renamings \( A_1, A_2 \) for \( s_i \), such that \( A_1(s_1) = A_2(s_2) \). 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 \( \sim_{\alpha} \).

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 \( cl(D) \in \mathcal{K} \) denotes the context class of \( D \). The semantics of meta-variables \( X, Y \) are all concrete variables of type \( \text{Var} \), expression variables \( S \) represent any ground expression of type \( \text{Expr} \), environment variables \( E \) represent all ground environments of type \( \text{Env} \), and a context variable \( D \) with \( cl(D) = \mathcal{K} \) represents all contexts of class \( \mathcal{K} \).

Definition 2.9. A meta-variable substitution \( \rho \) maps a finite set of meta-variables to variables, expressions, environments, and contexts respecting their types and classes. We say \( \rho \) is ground if it maps all variables in \( \text{Dom}(\rho) \) to LRS-expressions.
While for ground expressions, \( \alpha \)-renaming of higher-order meta-expressions is different. We want to apply \( \alpha \)-renamings to the meta-expressions of LR\( \alpha \), 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 the symbolic \( \alpha \)-renaming. Thus, we extend LR\( \alpha \) such that meta-variables \( S, D, E, X \) and variables \( x \) come with an additional symbolic \( \alpha \)-renaming, written as \( \xi, \xi, \xi, \xi, \eta, \eta, \eta, \eta \), respectively.

Definition 3.1. The syntax of symbolic \( \alpha \)-renamings \( \xi \) and renaming sequences \( \eta \) is defined by the grammar given in Fig. 3.

A renaming sequence \( \eta \in \text{RS} \) is a sequence of renaming components. We use list notation and write both \( (r_1, \ldots, r_n) \) representing all elements of the sequence in order as \( r_\ell \) and \( r_n \) for an additional symbolic \( \alpha \)-renaming, written as \( \xi, \xi, \xi, \xi, \eta, \eta, \eta, \eta \), respectively.

\[ x, y, z \in \text{Var} \implies X | x \]

\( s, t \in \text{HEexp}^0 \implies S | D[s] | \text{letrec env in } s | (f r_1 \cdots r_{nf}) \]

where \( r_j \in \text{HEexp}^0 \) if \( \text{oar}(f)(i) = k \geq 0 \), and \( r_j \in \text{Var} \), if \( \text{oar}(f)(i) = \text{Var} \).

\( s \in \text{HEexp}^\alpha \implies x.s_1 \) if \( s_1 \in \text{HEexp}^\alpha \) and \( n \geq 1 \)

\( \text{env} \in \text{Env} \implies \emptyset | E; \text{env} | b; \text{env} \)

where \( ; \) is associative and commutative

\( b \in \text{Bind} \implies x.s \) where \( s \in \text{HEexp}^0 \)

Figure 2: Syntax of LR\( \alpha \), where \( X, S, D, E \) are meta-variables.

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

Definition 3.10. A constrained LR\( \alpha \)-expression \( (s, \Delta) \) consists of an LR\( \alpha \)-expression \( s \) and a constraint tuple \( \Delta = (\Delta_1, \Delta_2, \Delta_3) \) such that \( \Delta_1 \) is a finite set of context variables, called non-empty context constraints; \( \Delta_2 \) is a finite set of environment variables, called non-empty environment constraints, and \( \Delta_3 \) is a finite set of pairs \( (d, t) \) where \( t \) is an LR\( \alpha \)-expression and \( d \) is an LR\( \alpha \)-context, called non-empty context constraints (NCC\( s \), for short). A ground substitution \( \rho \) satisfies \( \Delta \) iff \( \rho(D) \neq \emptyset \) for all \( D \in \Delta_1 \); \( \rho(E) \neq \emptyset \) for all \( E \in \Delta_2 \); and \( \text{Var}(\rho(t)) \cap \text{CV}(\rho(d)) = \emptyset \) for all \( t, d \in \Delta_3 \). If there exists a ground substitution \( \rho \) that satisfies \( \Delta \), then we say \( \Delta \) is satisfiable. The set of concretizations of a constrained LR\( \alpha \)-expression \( (s, \Delta) \) is:

\[ \gamma(s, \Delta) := \{ \rho(s) | \rho \text{ is ground, } \rho \text{ satisfies } \Delta \} \]

For an LR\( \alpha \)-expression \( s \), we define \( \gamma(s) = \gamma(s, (\emptyset, \emptyset, \emptyset)) \).

Example 3.11. For \( \Delta = (\emptyset, \Delta_2, \Delta_3) \) with \( \Delta_2 = (E_1, E_2) \), and \( \Delta_3 = \{ (\text{letrec } E_1 \in c, \text{letrec } E_2 \in c) \} \), the constrained expression \( \text{letrec } E_1 \in c \text{letrec } E_2 \in c \) in \( s, \Delta \) represents all LR\( \alpha \)-expressions that are nested \( \text{letrec} \)-expressions where both \( \text{letrec} \)-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 \( L_{\text{need}} \) [31] which copies an abstraction into a needed position in a \( \text{letrec} \)-environment:

\[ \text{letrec } E; X.AW.S; Y.D_1[\text{var } X] \in D[\text{var } Y] \]

\[ \Rightarrow \text{letrec } E; X.AW.S; Y.D_1[\text{var } X] \in D[\text{var } Y] \]

If \( D_1 \) is empty, then the target of the copy operation should be the variable \( Y \) in \( D[\text{var } Y] \). Thus the case \( D_1 = \{ \} \) should be excluded which can be expressed by setting \( \Delta_3 = \{ D_1 \} \).

3 \( \alpha \)-RENAMEING OF META-EXPRESSIONS

3.1 The Language LR\( \alpha \)

While for ground expressions, \( \alpha \)-renaming is a well-known task, our setting is different. We want to apply \( \alpha \)-renamings to the meta-expressions of LR\( \alpha \), 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

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

Note that this notation is similar and also related to the notation of suspensions \( \pi X \) in nominal syntax (see e.g. [33]).
the meta-variable $U$. Note that $all_{U,i}$ can only occur as the first component of a sequence of renamings applied to $U$. Components $\alpha_{x,i}$ represent fresh renamings of variable $p(x)$, and Component $CV(\alpha_{x,i})$ represents the restriction of $\alpha_{x,i}$ to those bound variables of $\rho(D)$ which affect the context hole. Component $LV(\alpha_{x,i})$ represents the restriction of $\alpha_{x,i}$ to the let-variables of $\rho(D)$. 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 $\alpha$-renamings induce a notion of equivalence of symbolic $\alpha$-renamings:

**Definition 3.1.** The relation $\equiv$ is the smallest equivalence relation satisfying: $c \equiv c$ for $c = all_{U,i}$, or an atomic renaming component $c; (r_{c1}, \ldots, r_{cn}) \equiv (r_{c1}', \ldots, r_{cn}')$ if $r_{ci} \equiv r_{ci}'$ for $i = 1, \ldots, n$; $\{r_{c1}, \ldots, r_{ci}, \ldots, r_{cn}\} \equiv \{r_{c1}, \ldots, r_{ci}'', \ldots, r_{cn}\}$; and if there exists a permutation $\pi$ on $\{1, \ldots, n\}$ such that $arc_{i} \equiv arc'_{i}(\pi)$, then $\{arc_{1}, \ldots, arc_{n}\} \equiv \{arc'_{1}, \ldots, arc'_{n}\}$.

We do not distinguish symbolic $\alpha$-renamings up to $\equiv$. To embed LRSX-expressions into LRSX$\alpha$, we identify $\langle \cdot \rangle U$ with $U$ and let $\epsilon: LRSX \rightarrow LRSX$ be the mapping that erases all renamings.

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

**Definition 3.2.** An LRSX$\alpha$-expression $s$ is well-formed iff $\rho(s)$ 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 $LV(\alpha_{x,i}), LV(\alpha_{x,j}) \in rc$ for some $x$ and some $i \neq j$, or $CV(\alpha_{x,i}), CV(\alpha_{x,j}) \in rc$ for some $x$ and some $i \neq j$. A constrained LRSX$\alpha$-expression $(s, \Delta)$ is well-formed, iff $s$ is well-formed and for all $(t, d) \in \Delta$, the expression $t$ and the context $d$ are well-formed.

We define the formal semantics of symbolic $\alpha$-renamings.

**Definition 3.3.** Let $(s, \Delta)$ be a well-formed, constrained LRSX$\alpha$-expression and $\rho$ be a ground substitution with $\text{Dom}(\rho) = MV(s) \cup MV(\Delta)$ such that $\rho(s)$ fulfills the LVC. A ground and fresh $\alpha$-renaming for $s$ and $\rho$ is a function $\tau$ such that

- for all expression, context, and environment meta-variables $U$ with $U \in MV(s)$, $\tau$ maps $all_{U,i}$ to a fresh $\alpha$-renaming $\tau(all_{U,i}) = all_{U,i}(\rho(U))$;
- for all variables $X$, $\tau(\alpha_{X,i})$ is the substitution $\{\rho(X) \mapsto \gamma Y_X,i\}$ and $\tau(\alpha_{X,i})$ is the substitution $\{X \mapsto \gamma X,i\}$;
- for each environment meta-variable $E_i$, with $\tau(\alpha_{E,i}) = A_{E,i}$ and $\rho(E) = x_1; x_2; \ldots; x_n$, $\tau(LV(\alpha_{E,i}))$ is the substitution $\{x_j \mapsto A_{E,i}(j,1)\}$ for $1 \leq j \leq n$;
- for each context variable $D$ with $\tau(\alpha_{D,i}) = A_{D,i}$ and $\rho(D) = \varnothing$, $\tau(CV(\alpha_{D,i}))$ is the substitution induced by $\tau$ between $CV(d)$ and $CV(d')$, i.e. $\{\tau(CV(\alpha_{D,i})) = \{x \mapsto x'\mid x \in CV(d)\}$,
- $\tau(\{c_1, \ldots, c_n\}) = \tau(c_1) \circ \cdots \circ \tau(c_n)$ such that for any permutation $\{1, \ldots, n\}$ the equation $\tau(c_1) \circ \cdots \circ \tau(c_n) = \tau(c_{\pi(i)})$ holds.

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

\*\*Note that this excludes such $\tau$ which are not invariant w.r.t. permutations.

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

Applying $\tau$ and $\rho$ to $s$ and $\Delta$ first replaces every occurrence $\xi_U U$ in $s$ by $\xi_U \rho(U)$ and then replaces $\xi_U U$ by the corresponding substitution or $\alpha$-renaming, i.e. by $\tau(\xi_U \rho(U))$ or $\tau(\rho(x))$.

For a constrained LRSX$\alpha$-expression $(s, \Delta)$, the concretizations are:

$\gamma(s, \Delta) \equiv \begin{cases} \tau(\rho(s)) \text{ if } \rho(s) \text{ fulfills the LVC, } \tau \text{ is a ground and fresh } \alpha \text{-renaming for } s, \rho, \text{ and } \tau \circ \rho \text{ satisfies } \Delta \end{cases}$

For LRSX$\alpha$-expressions $s$, we define $\gamma(s) = \gamma(s, (0, \emptyset, \emptyset))$.

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

### 3.2 Performing Symbolic Alpha-Renaming

We define introduction of symbolic $\alpha$-renamings, i.e. how to transform an LRSX-expression $s$ into an LRSX$\alpha$-expression $s'$, such that the instances of $s'$ are $\alpha$-renamed copies of the instances of $s$ which are LRS-expressions. The algorithm to symbolically $\alpha$-rename, first $\alpha$-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 $\text{sift}$ shown in Fig. 5) computes an LRSX$\alpha$-expression for $s$. For a constrained LRSX-expression $(s, \Delta)$, we compute a symbolically $\alpha$-renamed expression as $(AR(s), \Delta)$.

**Example 3.4.** We $\alpha$-rename the expression $\lambda \lambda X. \lambda X. \var X$.

$AR(\lambda \lambda X. \lambda X. \var X) = AR(\lambda \lambda X. \lambda X. \var X) = \lambda \lambda X, X. \xi(\lambda X. \lambda X. \var X) = \lambda \lambda X, X. \xi(\lambda X. \lambda X. \var X) = \lambda \lambda X. \lambda X, X. \var X, \xi(\lambda X, X. \var X)$

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

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

$AR(\text{letrec } E_1; E_2; E_3 \text{ in letrec } E_4 \text{ in } S) =$

$\text{letrec } (\alpha_{E_1,1}[LV(\alpha_{E_1,1}), LV(\alpha_{E_1,1})]); E_1;$

$\langle E_2, \mid LV(\alpha_{E_2,1}), LV(\alpha_{E_2,1}) \rangle; E_2;$

$\langle E_3, \mid LV(\alpha_{E_3,1}), LV(\alpha_{E_3,1}) \rangle; E_3;$

$\langle E_4, \mid LV(\alpha_{E_4,1}), LV(\alpha_{E_4,1}) \rangle; E_4;$

$\langle (\alpha_{E_1,1}[LV(\alpha_{E_1,1}), LV(\alpha_{E_1,1})]), LV(\alpha_{E_4,1}), LV(\alpha_{E_4,1}) \rangle; S$.

In this example no further simplification of the symbolic renamings is possible. However, if we assume that there are non-capture constraints (letrec $E_i$ in c, letrec $E_j$ in [] 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
The goal of this section is to define a sound mechanism to simplify 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 \( t \in \gamma(s) \), there exists \( t' \in \gamma(s') \) such that \( t \sim_\alpha t' \) and for each \( t'' \in \gamma(s'') \) there exists \( t \in \gamma(s) \) such that \( t \sim_\alpha t'' \). Furthermore all \( t' \in \gamma(s') \) fulfill the DVC.

4 SIMPLIFICATION OF \( \alpha \)-RENAIMINGS

The goal of this section is to define a sound mechanism to simplify symbolic \( \alpha \)-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 \( \alpha \)-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

\[ Var_M(\eta \cdot x) = \{ \eta \cdot x \} \]

and

\[ Var_M(\eta \cdot x \cdot s) = \{ \eta \cdot x \} \cup Var_M(s) \]

\[ Var_M(\xi \cdot s) = \{ \xi \} \cup Var_M(s) \]

\[ Var_M(\xi \cdot D[s]) = \{ \xi \} \cup Var_M(s) \]

\[ Var_M(\text{letrec en} \; in \; s) = Var_M(\text{env} \cup \text{Var}(s)) \]

\[ Var_M(\text{env}) = \{ [\xi \cdot E \mid \xi \cdot \text{env}^\prime = \text{env}] \}

\[ \cup \{ [\eta \cdot z \mid \text{Var}(s) \cup \eta \cdot z \cdot s; \text{env}^\prime = \text{env}] \}

\[ CV_M(\eta \cdot x) = 0 \]

\[ CV_M(\xi \cdot D[s]) = CV_M(\xi \cdot D) \cup CV_M(d) \]

\[ CV_M(\xi \cdot s) = \{ \xi \} \cup CV_M(s) \]

\[ CV_M([\xi \cdot D] = \emptyset) \]

\[ CV_M(\xi \cdot D) \]

\[ CV_M(\text{letrec en} \; in \; d) = CV_M(\text{env}) \cup CV_M(d) \]

\[ CV_M(\text{env}) = \{ [\xi \cdot E \mid \xi \cdot \text{env}^\prime = \text{env}] \}

\[ \cup \{ [\eta \cdot z \mid \text{Var}(s) \cup \eta \cdot z \cdot s; \text{env}^\prime = \text{env}] \}

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 \( S \) of NCCs, the function \( split_\text{env} \) is defined by

\[ split_\text{env}(S) = \bigcup \{(u, v) \mid u \in Var_M(s), v \in CV_M(d)\} \]

where the functions \( Var_M \) and \( CV_M \) are shown in Fig. 6. \( Var_M \) 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

\[ Var_M(\lambda x. \text{app}(\gamma) S) = \{ X, \gamma, S \} \]

and

\[ CV_M(D[2][\lambda x. \text{app}(\gamma) S]) = \{ D, X \} \]

Computation of \( Var_M \) and \( CV_M \) implies:

Lemma 4.1. Let \( (s, d) \) be an NCC, \( \rho \) be a ground substitution, and \( \tau \) be a ground and fresh \( \alpha \)-renaming for \( s, d, \rho \). Then the following equations hold:

\[ Var(\tau(\rho(s))) = \{ Var(\tau(\rho(u))) \mid u \in Var_M(s) \} \]

\[ CV(\tau(\rho(d))) = \{ [\tau(\rho(x)) \mid x \in CV_M(d)] \}

\[ \cup \{ [\tau(\rho(\xi) \mid \xi \in CV_M(d)] \}

\[ \cup \{ [\tau(\rho(\xi) \mid \xi \in CV_M(d)] \}

As a further preparation for simplification, we define two kinds of relationships between symbolic renamings. Roughly speaking, a renaming sequence \( \eta_1 \) is an instance of \( \eta_2 \) if in \( \eta_1 \) compared to \( \eta_2 \) some sets of renaming components \( [arc_1, \ldots, arc_n] \) are partly ordered, e.g. replaced by sequences \( (rc_1, \ldots, rc_n) \) such that \( \bigcup i rc_i = [arc_1, \ldots, arc_n] \) and \( rc_i \cap rc_j = \emptyset \) for \( i \neq j \). Furthermore, \( \eta_1 \) is a weak instance of \( \eta_2 \) if it is an instance after forgetting about the concrete indexes \( i \) in \( \alpha(t, \text{Var}(d), \text{LV}(\alpha(t, i))) \).

Definition 4.2. The relation \( \alpha \text{num} \) identifies renaming components and sequences up to the number \( i \) in \( \alpha(t, i) \), it is defined by

\[ \alpha(t, i) = \alpha(t, j) \]

where \( U \) may be \( E, D, S, X, x \), \( CV(\alpha(t, i)) \) = num \( CV(\alpha(t, j)) \)

\[ \text{LV}(\alpha(t, i)) = \text{num} \text{LV}(\alpha(t, j)) \]

\[ \text{Var}(\alpha(t, i)) = \text{num} \text{Var}(\alpha(t, j)) \]

\[ \text{Env}(\alpha(t, i)) = \text{num} \text{Env}(\alpha(t, j)) \]

We extend \( \alpha \text{num} \) to renaming sequences \( \xi U \) and \( \eta \) in the obvious way. A renaming \( \eta_1 \) is an instance of a renaming \( \eta_2 \) if

- \( \eta_1 \) is \( \eta_2 \), or
- \( \eta_1 = rc_1 : \eta'_1, \eta_2 = rc_2 : \eta'_2, rc_1 \subseteq rc_2, \) and \( \eta'_1 \) is an instance of \( \{rc_2 \setminus rc_1\} : \eta'_2 \).
(1) $\xi_1 U \vdash_{\Delta} \xi_2 U \text{ and } \xi_2 U \vdash_{\Delta} \xi_3 U \quad (\text{TrU})$

(2) $\xi_1 U \vdash_{\Delta} \xi_3 U \quad (\text{SkU})$

(3) $\eta_1 x \vdash_{\Delta} \eta_2 x \text{ and } \eta_2 x \vdash_{\Delta} \eta_3 x \quad (\text{SkX})$

(4) $\eta_1 x \vdash_{\Delta} \eta_3 x \quad (\text{SubU})$

(5) $(\text{RC})$

(6) $\eta \vdash_{\Delta} \eta' \quad (\text{RC})$

(7) $\eta \vdash_{\Delta} \eta' \quad (\text{RC})$

(8) $\eta \vdash_{\Delta} \eta' \quad (\text{RC})$

(9) $\eta \vdash_{\Delta} \eta' \quad (\text{RC})$

A renaming $\eta_1$ is a weak instance of a renaming $\eta_2$ if

- $\eta_1$ is an instance of $\eta_2$, or
- $\eta_1 = rc_1 \eta_2' \text{ and } \eta_2' \vdash_{\Delta} \eta_2, \text{ where } rc_1 \subseteq_{w} rc_2$, and $\eta_1'$ is a weak instance of $(rc_2 \setminus rc_1) \eta_2$. Here $rc_1 \subseteq_{w} rc_2$ holds if for all $\eta \in rc_1$ there exists an $\eta' \in rc_2$ with $\eta = \text{num } \eta'$.

Example 4.3. As an example we consider the renaming $\eta(\alpha_{S,1}, CV(a_{D,1}), CV(a_{D,3}), \alpha_{S,1}, CV(a_{D,1}), \alpha_{S,2}, CV(a_{D,3}), \alpha_{S,2})$. It is an instance of $(\alpha_{S,1}, CV(a_{D,1}), CV(a_{D,3}), \alpha_{S,1}, CV(a_{D,1}), \alpha_{S,2}, CV(a_{D,3}), \alpha_{S,2})$. The weak instance relation additionally allows one to switch between the copies of atomic renaming components, and thus e.g. the renaming $\eta(\alpha_{S,1}, CV(a_{D,1}), CV(a_{D,3}), \alpha_{S,1}, CV(a_{D,1}), \alpha_{S,2}, CV(a_{D,3}), \alpha_{S,2})$ is not an instance but a weak instance of the symbolic renaming $\eta(\alpha_{S,1}, CV(a_{D,1}), CV(a_{D,3}), \alpha_{S,1}, CV(a_{D,1}), \alpha_{S,2}, CV(a_{D,3}), \alpha_{S,2})$.

Definition 4.4. We consider formal expressions of the form $x, \text{VAR}(U), \text{COD}(arc)$, which we call symbolic set-variables, and let $V$ be a set of such formal expressions. With $MV(U)$ we denote the meta-variables occurring in $V$ (i.e. $U \in \text{VAR}(U)$ and all meta-variables occurring as index of some $arc$ in $\text{COD}(arc)$). For a set $MV$ of meta-variables with $MV \in MV(U)$, a ground substitution $\rho$ for $MV$ and a ground $\alpha$-renaming $\tau$ for $\rho$ and $MV$, we define $\tau(\rho(V)) = \bigcup_{\eta \in MV} \tau(\rho(\eta))$ where $\tau(\rho(\text{VAR}(U))) = \text{VAR}(\rho(U))$, $\tau(\rho(x)) = (\rho(x))$, and $\tau(\rho(\text{COD}(arc))) = \text{COD}(\tau(\rho))$.

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 $\Delta_3$.

Definition 4.5 (Simplification). The simplification relation $\vdash_{\Delta}$ 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 $\eta \vdash_{\Delta} \eta'$ which are defined by the rules shown in Fig. 7 (b) and the predicate $\eta \vdash_{\Delta} \eta'$ which is defined in Fig. 7 (c).

Note that in the present 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 $(s, \Delta)$ be a constrained LRS$\alpha$-expression. The simplification algorithm replaces occurrences $\xi U (\eta x, \text{resp.})$ in $s$ by $\xi' U (\eta' x, \text{resp.})$ if $\xi U (\eta x) \vdash_{\Delta} \xi' U (\eta' x, \text{resp.})$ 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 \(\alpha\)-renamings applied to \(\alpha\)- or \(X\)-variables, where the symbolic set of variables in the premise is the singleton containing the to-be-simplified variable. Rule (SimU) performs simplification for meta-variables \(U\) which are not \(X\)-variables. Hence the \(\alpha\)-renaming starts with \(\alpha_{U,i}\) and the symbolic set of variables consists of \(\text{VAR}(U)\) 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 \(\xi^\Delta_\alpha U\) or \(\eta^\Delta_\alpha y\), 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 applying 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 \(\alpha_{x,i,j}\) into a set of components provided that it is guaranteed that the ground instances of all variables \(x_i\) are pairwise different. The predicate \(\phi_\Delta\) is defined in Fig. 7 (c) where arc \(\phi_\Delta v\) expresses that atomic renaming component \(\alpha_{x,i}\) 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 \((\lambda x_1, X.\lambda x_2, X.\var{\alpha_{x_2, x_1}} X, \var{0, 0, 0})\) then it results in \((\lambda x_1, X.\lambda x_2, X.\var{\alpha_{x_2, x_1}} X, \var{0, 0, 0})\), since

\[
\begin{align*}
&\text{(SubstX)} & (\lambda x_1, X.\lambda x_2, X.\var{\alpha_{x_2, x_1}} X, \var{0, 0, 0}) \\
&= (\lambda x_1, X.\lambda x_2, X.\var{\alpha_{x_2, x_1}} X, \var{0, 0, 0})
\end{align*}
\]

As a further example, consider \((s, \Delta) = (s, \var{0, 0, 0, \Delta})\) with

\[
\begin{align*}
&\text{letrec } \var{s_1} = \var{\alpha_{E_1, E_1}}, \var{\alpha_{E_2, E_2}}; \\
&\var{s_2} = \var{\alpha_{E_1, E_1}}; 
\end{align*}
\]

\[
\begin{align*}
&\var{s_3} = \var{\alpha_{E_2, E_2}}; \\
&\var{s_4} = \var{\alpha_{E_1, E_1}}; \\
&\var{s_5} = \var{\alpha_{E_2, E_2}}.
\end{align*}
\]

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

\[
\begin{align*}
&\text{letrec } \var{s'_1} = \var{\alpha_{E_1, E_1}}; \\
&\var{s'_2} = \var{\alpha_{E_1, E_1}}; \\
&\var{s'_3} = \var{\alpha_{E_2, E_2}}; \\
&\var{s'_4} = \var{\alpha_{E_1, E_1}}; \\
&\var{s'_5} = \var{\alpha_{E_2, E_2}}.
\end{align*}
\]

(2) (Correctness of \(\phi_\Delta\)) Let \(V\) be a set of symbolic set-variables and \(\eta\) be a sequence of renaming components with over \(M\), such that \(V, \eta \vdash \eta^\prime\). Then for each \(x \in \tau(\rho(V))\), we have \(\tau(\eta(x)) = \tau(\eta^\prime(x))\).

(3) (Correctness of \(\vdash\))

(a) Let \(\eta\) and \(\eta^\prime\) be symbolic \(\alpha\)-renamings with components over \(M\), such that \(\eta \vdash \eta^\prime\). Then the equation \(\tau(\eta(x)) = \tau(\eta^\prime(x))\) holds.

(b) Let \(\xi\) and \(\xi^\prime\) be symbolic \(\alpha\)-renamings with components over \(M\), and let \(U \in M\) such that \(\xi^\prime U \vdash \Delta\). Then the equation \(\tau(\xi(U)) = \tau(\xi^\prime(U))\) holds.

Applying the previous proposition for all occurrences \(\eta x\) and \(\xi U\) 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 \(\text{LRSX}_{\alpha}\)-expression \((s, \Delta)\) such that \(s\) fulfills the LVC and \(s\) does not contain an environment variable twice in the same environment, the simplified expression \((s', \Delta)\), we have \(\gamma(s, \Delta) = \gamma(s', \Delta)\).

5 ALGORITHMS FOR \(\text{LRSX}_{\alpha}\)-EXPRESSIONS

We show how to rewrite \(\text{LRSX}_{\alpha}\)-expressions by matching \(\text{LRSX}_{\alpha}\)-expressions and by refreshing the \(\alpha\)-renaming to guarantee that the distinct variable convention holds after applying a rewrite step. We finally present an algorithm to test extended \(\alpha\)-equivalence of \(\text{LRSX}_{\alpha}\)-expressions which, for instance, is necessary during diagram computation to check whether a diagram is closed.

5.1 Rewriting Meta-Expressions

Meta letrec rewrite rules (see [26]) are rewrite rules of the form \(\ell 

Theorem 4.9. The simplification algorithm does not change the set of concretizations, i.e. for a constrained \(\text{LRSX}_{\alpha}\)-expression \((s, \Delta)\) such that \(s\) fulfills the LVC and \(s\) does not contain an environment variable twice in the same environment, the simplified expression \((s', \Delta)\), we have \(\gamma(s, \Delta) = \gamma(s', \Delta)\).

Theorem 4.9. The simplification algorithm does not change the set of concretizations, i.e. for a constrained \(\text{LRSX}_{\alpha}\)-expression \((s, \Delta)\) such that \(s\) fulfills the LVC and \(s\) does not contain an environment variable twice in the same environment, the simplified expression \((s', \Delta)\), we have \(\gamma(s, \Delta) = \gamma(s', \Delta)\).

5 ALGORITHMS FOR \(\text{LRSX}_{\alpha}\)-EXPRESSIONS

We show how to rewrite \(\text{LRSX}_{\alpha}\)-expressions by matching \(\text{LRSX}_{\alpha}\)-expressions and by refreshing the \(\alpha\)-renaming to guarantee that the distinct variable convention holds after applying a rewrite step. We finally present an algorithm to test extended \(\alpha\)-equivalence of \(\text{LRSX}_{\alpha}\)-expressions which, for instance, is necessary during diagram computation to check whether a diagram is closed.

5.1 Rewriting Meta-Expressions

Meta letrec rewrite rules (see [26]) are rewrite rules of the form \(\ell \rightarrow_{\alpha} r\) where \(\ell\) and \(r\) are \(\text{LRSX}_{\alpha}\)-expressions and \(\Delta\) is a constraint tuple. Applying a rewrite rule to a constrained expression \((s, \Delta)\) consists of matching \(\ell\) against \(s\) such that the constraints in \(\Delta\) imply the constraints in \(\Delta\). Given a matcher (i.e. a substitution \(\sigma\) with \(\sigma(\ell) \rightarrow_{\alpha} s\)) the reduction is \(s \rightarrow_{\alpha} (\sigma(r), \Delta \cup \{\sigma(\ell), \Delta\})\). In [26] the letrec matching problem was defined and analyzed for \(\text{LRSX}_{\alpha}\)-expressions. However, as argued before, often transformations are not applicable, since \(\Delta\) does not imply \(\Delta\) (see the example for an (let) overlap in Sect. 1). Here \(\alpha\)-renaming of \(s\) often helps to satisfy the constraints. Hence, we formulate an adapted form of a letrec matching problem where \((s, \Delta)\) is a constrained \(\text{LRSX}_{\alpha}\)-expression.

Our matching equations are of the form \(s \equiv_\Delta s\) where \(s\) is a meta-expression with instanciable meta-variables and \(\ell\) is meta-expression with meta-variables that act like constants. In addition \(\ell\) may contain symbolic \(\alpha\)-renamings (i.e. if \(\ell\) is an \(\text{LRSX}_{\alpha}\)-expression), but \(s\) is an \(\text{LRSX}_{\alpha}\)-expression. To distinguish the meta-variables we use blue font for instanciable meta-variables and red font and underlining for fixed meta-variables. With \(\text{MV}_j(\cdot)\) and \(\text{MV}_F(\cdot)\) we denote functions to compute the sets of instanciable and fixed meta-variables.

Definition 5.1. A letrec matching problem with \(\alpha\)-renamed expressions is a tuple \(P = (\Gamma, \Delta, \psi)\) where \(\Gamma\) is a set of matching equations \(s \equiv_\Delta t\) such that \(s\) is an \(\text{LRSX}_{\alpha}\)-expression, \(t\) is an \(\text{LRSX}_{\alpha}\)-expression, \(\text{MV}_j(t) = \emptyset, \Delta = (\Delta_1, \Delta_2, \Delta_3)\) is a constraint tuple over \(\text{LRSX}_{\alpha}\), called needed constraints; \(\psi = (\psi_1, \psi_2, \psi_3)\) is a constraint tuple over \(\text{LRSX}_{\alpha}\), called given constraints, where \(\text{MV}_j(\psi_1) = \emptyset\) for
$i = 1, 2, 3$ and $\nabla$ is satisfiable; and for all expressions in $\Gamma$, the LVC must hold. The following occurrence restrictions must hold: every variable of kind $S$ occurs at most twice in $\Gamma$; every variable of kind $E$ or $D$ occurs at most once in $\Gamma$. A matcher $\sigma$ of $P$ is a substitution such that for any ground substitution $\rho$ together with a ground renaming $\tau$ with $\text{Dom}(\rho) = \text{MV}_\tau(P)$ such that $\tau \circ \rho$ satisfies $\nabla$, and $\tau(\rho(\sigma(s)))$, $\tau(\rho(t))$ fulfill the LVC for all $s \leq t \in \Gamma$, we have $\tau(\rho(\sigma(s))) \rightarrow_{\text{let}} \tau(\rho(t))$ for all $s \leq t \in \Gamma$, and there exists a ground substitution $\rho_0$ with $\text{Dom}(\rho_0) = \text{MV}_\rho(\rho(\sigma(\Delta)))$ such that $\tau(\rho(\sigma(\Delta)))$ is satisfied.

The letrec matching problem with LRSX-expressions, only and corresponding matchers for LRSX-expressions are defined analogously but all expressions are LRSX-expressions, and no ground renaming $\tau$ is involved. The additional substitution $\rho_0$ in the definition of a matcher is used for the case that rewrite rules $\ell \rightarrow_{\text{let}} \tau$ introduce meta-variables, i.e. if there are meta-variables which occur in $\tau$ but not in $\ell$. Then the existence of $\rho_0$ 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: $(f \; s_1 \ldots s_n) \rightarrow_{\text{Lambda}} \text{letrec} \; x_1 \cdot s_1 \ldots x_n \cdot s_n$ in $(f \; (\text{var} \; x_1) \ldots (\text{var} \; x_n))$ where $\Delta$ contains NCCs that ensure that $x_1, \ldots, x_n$ are fresh w.r.t. $s_1, \ldots, s_n$.

In [26] a sound and complete matching algorithm takes a letrec matching problem as input and computes symbolic variables that represent the set of free and bound variables that may occur in concretizations of $\xi \cdot u$ and $\text{CVsym}(\eta \cdot v)$ computes symbolic variables which may capture variables in the concretizations of $\xi' \cdot u$ and the relation $\xi' \equiv \eta'$ symbolically checks whether the sets of variables represented by $\xi' \equiv \eta'$ are disjoint (see Fig. 9).

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

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

Theorem 5.4. The matching algorithm for LRSKx is sound.

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

\begin{align*}
&\forall i = 1, 2, 3 \quad \text{and}\quad \nabla \text{ is satisfiable;} \quad \text{and for all expressions in } \Gamma, \text{ the LVC must hold.} \\
&\text{The following occurrence restrictions must hold: every variable of kind } S \text{ occurs at most twice in } \Gamma; \quad \text{every variable of kind } E \text{ or } D \text{ occurs at most once in } \Gamma. \\
&\text{A matcher } \sigma \text{ of } P \text{ is a substitution such that for any ground substitution } \rho \text{ together with a ground} \\
&\text{renaming } \tau \text{ with } \text{Dom}(\rho) = \text{MV}_\tau(P) \text{ such that } \tau \circ \rho \text{ satisfies } \nabla, \quad \text{and } \tau(\rho(\sigma(s))), \tau(\rho(t)) \text{ fulfill the LVC for all } s \leq t \in \Gamma, \\
&\text{we have } \tau(\rho(\sigma(s))) \rightarrow_{\text{let}} \tau(\rho(t)) \text{ for all } s \leq t \in \Gamma, \quad \text{and there exists a ground substitution } \rho_0 \text{ with } \text{Dom}(\rho_0) = \text{MV}_\rho(\rho(\sigma(\Delta))) \text{ such that } \\
&\tau(\rho(\sigma(\Delta))) \text{ is satisfied.}
\end{align*}
ensures that $X$ does not occur in $S$. For the constrained expression
\[(\text{letrec } Y.S_0 \text{ in } \var{Y}, \var{0}, \var{(S_0, \lambda Y.[])})\], $\alpha$-renaming results in
\[(\text{letrec } \alpha_{Y.1}.1 \cdot Y.(\text{letrec } \alpha_{Y.1}.1.S_1 \text{ in } \var{Y}, \var{0}, \var{(S_1, \lambda Y.[])})\].

Matching the left hand side of the transformation (ucp) against this constrained LRSXa-expression fails, since for the substitution $\alpha = \{X \mapsto \alpha_{Y.1}.1.Y, S \mapsto (\alpha_{S_1.1}.1.S_0)\}$ the validity of the NCC $\sigma(S, \lambda X.[])) = (\alpha_{S_1.1.0} S_0, \lambda Y.1.H.[])$ cannot be inferred. If simplification is applied before the matching, then simplification of
\[(\text{letrec } \alpha_{Y.1}.1.Y.(\text{letrec } \alpha_{Y.1}.1.S_0 \text{ in } \var{Y}, \var{0}, \var{(S_0, \lambda Y.[])})\] leads to
\[(\text{letrec } \alpha_{Y.1}.1.Y.(\text{letrec } \alpha_{Y.1}.1.S_1 \text{ in } \var{Y}, \var{0}, \var{(S_1, \lambda Y.[])})\] and matching the left hand side of (ucp) against it delivers the matcher $\sigma = \{X \mapsto \alpha_{Y.1}.1.Y, S \mapsto (\alpha_{S_1.1}.1.S_0)\}$ where validity of the NCC $\sigma(S, \lambda X.[]) = (\alpha_{S_1.1.0} S_0, \lambda Y.1.H.[])$ can be inferred since
\[\text{split}_i((\alpha_{S_1.1.0} S_0, \lambda Y.1.H.[])) = (\alpha_{S_1.1.0} S_0, \alpha_{Y.1}.1.Y)\] as well as since $\text{VAR}(S_0) \models \text{COD}(\alpha_{Y.1}.1)$ and $\text{COD}((\alpha_{S_1.1}.1.S_0)) \models \text{COD}(\alpha_{Y.1}.1)$.

5.2 Refreshing $\alpha$-Renamings

Matching can be applied to rewrite constrained LRSXa-expressions. If the constrained expression stems from symbolically $\alpha$-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:

\[
\text{letrec } X.S \text{ in } C[\var{X}] \rightarrow X.S \text{ in } C[S].
\]

Applying the rule to letrec $\alpha_{X.1.1}.X.(\text{letrec } \alpha_{X.1.1}.S_0 \text{ in } \var{X}, \var{0}, \var{(S_0, \lambda X.[])})$ results in
\[
\text{letrec } \alpha_{X.1.1}.X.(\text{letrec } \alpha_{X.1.1}.S_1 \text{ in } \var{X}, \var{0}, \var{(S_1, \lambda X.[])})\] and the same $\alpha$-renaming $\alpha_{X.1.1}$ 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 $\alpha$-renamings to again symbolically $\alpha$-rename the expressions. However, this approach seems to be not easily tractable (for instance, this one needs to introduce renaming components of the form $\alpha_{X.1.1}$ representing an $\alpha$-renaming of already $\alpha$-renamed expressions). We choose a simpler approach. It uses the existing $\alpha$-renamings and refreshes them:

**Definition 5.6 (Refreshing Alpha-Renamings).** A $\alpha$-renaming of a symbolic $\alpha$-renaming modifies $\alpha_{X,i}$ components by replacing $\alpha_{U,i}$ (or $\alpha_{S,i}$, resp.) with $\alpha_{U,i}$, resp.) where $j$ is a fresh number. For a constrained LRSXa-expression $(s, \Delta)$, refresh $(s, \Delta)$ renumbers all occurrences of $\alpha_{U,i}$ and replaces $CV(\alpha_{U,i})$ with $CV(\alpha_{U,i})$ and $LV(\alpha_{U,i})$ with $LV(\alpha_{U,i})$ respecting the scopes. For bound variables $\var{}$ or meta-variables $\var{}$ it introduces a fresh $\alpha$-renaming $\alpha_{X,i}$ and adds it to the meta-variable and sifts the corresponding renaming downwards, analogous to AR and sift shown in Fig. 5.

**Proposition 5.7.** Let $(s, \Delta)$ be a constrained LRSXa-expression and $(s', \Delta') = $ refresh $(s, \Delta)$. Then for each $t \in \text{gamma}(s, \Delta)$ there exists $t' \in \text{gamma}(s', \Delta')$ with $t \sim_\alpha t'$ and for each $t' \in \text{gamma}(s', \Delta')$ there exists $t \in \text{gamma}(s, \Delta)$ with $t \sim_\alpha t'$.

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

5.3 Checking $\alpha$-Equivalence

We finally provide a test for checking extended $\alpha$-equivalence.
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 $\rightarrow$ represents the overlaps between left hand sides of transformations and standard reductions, while $\leftarrow$ represent the overlaps between right hand sides of transformations and standard reductions. Due to branching in unjoinable cases, the number of joins is higher than the number of overlaps. Note that the strategy of the LRSX Tool is to avoid $\alpha$-renamings, and thus $\alpha$-renaming is applied only, if no join was found before without performing renaming. The results show that $\alpha$-renaming is necessary in about 20 percent of the cases (except for overlaps of left hand sides in the calculus $L_{\text{need}}$). With the help of $\alpha$-renaming all computed overlaps could be closed and the correctness of program transformations (16 transformations for $L_{\text{need}}$ and 45 transformation for LR) could be shown automatically.

7 CONCLUSION

We presented an extension of the meta-language LRSX by symbolic $\alpha$-renamings. We introduced an algorithm for simplification of renamings, matching and rewriting of LRSX-expressions, refreshing of symbolic $\alpha$-renamings and checking extended $\alpha$-equivalence of LRSX-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 $\alpha$-renaming is successful in automatically proving correctness of program transformations. Further work is to use the approach in other inference procedures and to investigate whether it can be adapted for nominal techniques.

ACKNOWLEDGMENTS

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

REFERENCES