Nominal Unification of Higher Order Expressions with Recursive Let

Manfred Schmidt-Schauß[1], Temur Kutsia[2], Jordi Levy[3], and Mateu Villaret[4]

1 **GU Frankfurt, Inst. for Computer Science, Germany,** `schauss@ki.cs.uni-frankfurt.de`
2 **RISC Linz, Austria,** `kutsia@risc.jku.at`
3 **IIIA - CSIC, Spain,** `levy@iiia.scic.es`
4 **IMA, Universitat de Girona, Spain,** `villaret@ima.udg.edu`

Abstract

A sound and complete algorithm for nominal unification of higher-order expressions with a recursive let is described, and shown to run in non-deterministic polynomial time. For nominal letrec-matching a more streamlined algorithm with less rules is given that computes a representation of all solutions and requires nondeterministic polynomial time. We show NP-hardness, which implies that nominal letrec-matching as well as nominal letrec-unification is NP-complete. Nominal letrec-matching of garbage-free expressions is graph-isomorphism hard even if there is only a single occurrence of a variable. We describe nondeterministic polynomial algorithms for nominal letrec-matching of dags as well as for an extension that permits environment variables in the match-expressions. We also show that nominal commutative unification without letrecs is NP-complete.

1998 ACM Subject Classification F.4.2 Grammars and other rewriting systems, F.3.2 Semantics of Programming Languages

Keywords and phrases lambda calculus, nominal unification, higher-order unification, operational semantics, program transformations

1 Introduction

Unification [3] is an operation to make two logical expressions equal by finding substitutions into variables. There are numerous applications in computer science, in particular of (efficient) first-order unification, for example in automated reasoning, type checking and verification. Unification algorithms are also extended to higher-order calculi with various equivalence relations. If equality includes α-conversion and β-reduction and perhaps also η-conversion of a (typed or untyped) lambda-calculus, then unification procedures are known (see e.g. [13]), however, unification is undecidable [12, 14].

Our motivation comes from syntactical reasoning on higher-order expressions, with equality being alpha-equivalence of expressions, and where a unification algorithm is demanded as a basic service, preferably a nominal one. Nominal unification is the extension of first-order unification with abstractions. It unifies expressions w.r.t. alpha-equivalence, and employs permutations as a clean treatment of renamings. It is known that nominal unification is decidable in polynomial time [29, 30]. It can be seen also from a higher-order perspective [6, 16], as equivalent to Miller's higher-order pattern unification [20]. For more information and efficient algorithms see [5, 15]. We are interested in unification w.r.t. an additional extension with cyclic let. To the best of our knowledge, there is no nominal unification algorithm for higher-order expressions permitting also general binding structures like a cyclic let.

Our main application scenario and motivation is as follows: constructing syntactic reasoning algorithms on higher-order expressions in call-by-need functional languages that have a letrec-construct [2] as in Haskell [18], (see e.g. [7] for a discussion on reasoning with more general name binders, and [28] for a formalization of general binders in Isabelle). Essentially, overlaps of expressions have to be computed (a variant of critical pairs) and reduction steps (under some strategy) have to be performed. To this end, first an expressive higher-order language is required to represent the meta-notation of expressions. For example, the meta-notation $((\lambda x.e_1)\ e_2)$ for a beta-reduction is made operational by using unification variables X_1, X_2 for e_1, e_2. The scoping of X_1 and X_2 is different, which can be dealt with by nominal techniques. In fact, a more powerful unification algorithm is required for meta-terms employing recursive letrec-environments and corresponding meta-expressions representing these structural schemes.

Showing correctness of transformations in higher-order languages, in particular in call-by-need languages (see for example [22, 26]), can be done by computing critical pairs between the left hand sides of transformation rules and normal-order reduction rules of the calculus. The computations are done w.r.t. a meta-syntax and an appropriate unification algorithm. There are specialized conditions like reduction strategies, position restrictions, and simplifications, for example derived from a context lemma [21, 22, 26]. These computations require an adapted form of higher-order unification of meta-expressions w.r.t. syntactic or even α-equivalence. Joining (or closing) the critical pairs (requiring adapted nominal matching algorithms) leads to diagrams, which can then be used for correctness proofs of the transformations (see e.g. [26]) by constructing (using induction) complete evaluations for the transformed expressions.

Our main algorithm LetrecUnify is derived from first-order unification and nominal unification: From first-order unification we borrowed the decomposition rules, and the sharing method from Martelli-Montanari-style unification algorithms [19]. The adaptations of decomposition for abstractions and the advantageous use of permutations of atoms is derived from nominal unification algorithms. Decomposing letrec-expression requires an extension by a permutation of the bindings in the environment, where, however, one has to take care of scoping. Since in contrast to the basic nominal unification, there are nontrivial fixpoints of permutations (see Remark 3), novel techniques are required: a fixed-point shifting rule (FPS) and a redundancy removing rule (elimFP) that bounds the number of fixpoint equations $X \doteq \pi{\cdot}X$ (where π is a permutation) using techniques from computations in permutation groups. The application of these techniques is indispensable (see Example 3.4) to obtain an algorithm than runs in nondeterministic polynomial time.

Results in this paper are: a nominal letrec unification algorithm LetrecUnify for lambda expressions with constants and a recursive letrec-environment, which runs in nondeterministic polynomial time by (Theorem 4.1). Since we also show that the problem is NP-hard (Theorem 6.1), the nominal letrec-unification problem is NP-complete. We also describe as a specialization a nominal letrec matching algorithm for plain expressions, and show that the problem is NP-complete (Theorem 5.2 and 6.1). Nominal letrec matching algorithm for dags is performed by using LetrecUnify yielding solutions without fixpoint equations and freshness constraints (Theorem 5.4). Nominal letrec matching of garbage-free expressions is graph-isomorphism hard, even if there is only a single occurrence of a variable (Theorem 6.3). Transferring the methods shows that nominal commutative unification is NP-complete (Theorem 7.4).

2 The Language of Expressions

We define the language *LRL* (**L**et**R**ec **L**anguage) of expression. The (infinite) set of atoms $\mathbb{A}$ is a set of (constant) symbols a, b denoted also with indices. There is a set of function symbols with arity $ar(\cdot)$. The syntax of the expressions e of *LRL* is:

$$e \quad ::= \quad a \mid \lambda a.e \mid (f\ e_1\ \ldots e_{ar(f)}) \mid (\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$$

We assume that binding atoms $a_1, \ldots, a_n$ in a letrec-expression $(\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$ are pairwise distinct. Sequences of bindings $a_1.e_1; \ldots; a_n.e_n$ are sometimes abbreviated as *env*. The top symbol is defined as $tops(X) = X$, $tops(f\ s_1 \ldots s_n) = f$, $tops(a) = a$, $tops(\lambda x.s) = \lambda$, $tops(\texttt{letrec}\ env\ \texttt{in}\ s) = \texttt{letrec}$.

The *scope* of atom a in $\lambda a.e$ is standard: a has scope e. The `letrec`-construct has a special scoping rule: in $\texttt{letrec}\ a_1.s_1; \ldots; a_n.s_n\ \texttt{in}\ r$, every free atom a_i in some s_j or r is bound by the environment $a_1.s_1; \ldots; a_n.s_n$. This defines the notion of free atoms $FA(e)$, bound atoms $BA(e)$ in expression e, and all atoms $AT(e)$ in e. For an environment $env = \{a_1.e_1, \ldots, a_n.e_n\}$, we define the set of letrec-atoms as $LA(env) = \{a_1, \ldots, a_n\}$. We say a *is fresh for* e iff $a \notin FA(e)$ (sometimes denoted as $a\#e$).

For example, $(\texttt{letrec}\ foo.(\lambda a.foo\ a)\ \texttt{in}\ foo)$ defines a recursive function *foo*. The expression $(\texttt{letrec}\ f = cons\ s_1\ g; g = cons\ s_2\ f\ \texttt{in}\ f)$ defines an infinite list $(cons\ s_1\ (cons\ s_2\ (cons\ s_1\ (cons\ s_2\ \ldots))))$, where s_1, s_2 are expressions.

▶ **Definition 2.1** (α-Equivalence.)**.** There are two operations on *LRL*-expressions:

1. α-renaming: For abstractions, the definition is: $\lambda a.e \to_\alpha \lambda b.e[b/a]$, where $a \neq b$, and $b \notin FA(e)$ where substitution $e[b/a]$ is defined as replacing free occurrences of atom a in e by atom b. For $(\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ r)$, and index $1 \leq i \leq n$, an atom b, such that $b \notin \{a_1, \ldots, a_n\}$, and $b \notin FA(e_1, \ldots, e_n, r)$, let $(\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ r) \to_\alpha (\texttt{letrec}\ a_1.e_1[b/a_i]; \ldots; b.e_i[b/a_i]; \ldots; a_n.e_n[b/a_i]\ \texttt{in}\ r[b/a_i])$.
2. permutation of bindings in a letrec-environment: for a permutation ρ on $\{1, \ldots, n\}$: $\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e_0 \to_\alpha \texttt{letrec}\ (a_{\rho(1)}).e_{\rho(1)}; \ldots; (a_{\rho(n)}).e_{\rho(n)}\ \texttt{in}\ e_0$.
3. $s \to_\alpha t$ if $s = C[s_1]$, $s_1 \to_\alpha t_1$ and $t = C[t_1]$ for a nontrivial context C.

We call the reflexive-symmetric-transitive closure of $\to_\alpha$ *α-equivalence*, and denote it by $\sim_\alpha$.

This definition is not appropriate for algorithmically comparing two expressions for α-equivalent, or for unification algorithms trying to solve w.r.t. α-equivalence. We prepare now an algorithm that compares two expressions for α-equivalence in a top-down fashion.

The relation of α-equivalence between *LRL*-expressions can be characterized using (adapted) deBruijn indices [8] as follows:

▶ **Definition 2.2** (extended deBruijn index)**.** For an expression s, we add to every (bound) atom occurrence an information on its binding position. This is done by labeling the occurrence with a pair of numbers (n_1, n_2), where n_1 is the number of binding levels between the atom and its corresponding binder and n_2 is 1 if the atom is bound in an abstraction; if it is bound in a letrec, then n_2 is the index of its binder in the letrec environment. For consistency, free atoms are labeled $(0, 0)$. The notation is $(x, (n_1, n_2))$ for labeled atoms. *LRL*-expressions with these labels are called *deBruijn-labeled* expressions.

For example, in $(\texttt{letrec}\ x_1.(\lambda y.x_2); x_2.x_1\ \texttt{in}\ x_1)$, the two bound occurrences of x_1 are all labeled with $(0, 1)$, where the second component 1 indicates that it is the first binding in the environment, and the bound occurrence of x_2 has label $(1, 2)$.

▶ **Definition 2.3.** Let s_1, s_2 be deBruijn-labeled *LRL*-expressions. Then define $s_1 \sim_{lev,\alpha} s_2$ by recursively defining the relation also on subexpressions. Two subexpressions e_1 of s_1 and e_2 of s_2 are $\sim_{lev,\alpha}$*-equivalent*, $e_1 \sim_{lev,\alpha} e_2$, iff

1. $e_1 = (a_1, l_1)$ and $e_2 = (a_2, l_2)$: iff $l_1 = l_2$, or if $l_1 = l_2 = (0, 0)$, (i.e. these are free atoms), then $a_1 = a_2$ must also hold.
2. $e_1 = f\ e_{1,1} \dots e_{1,n}$, $e_2 = f\ e_{2,1} \dots e_{2,n}$, and $e_{1,j} \sim_{lev,\alpha} e_{2,j}$ for all j.
3. $e_1 = \lambda a.e_1'$, $e_2 = \lambda b.e_2'$, and $e_1' \sim_{lev,\alpha} e_2'$ (also for $a = b$).
4. $e_i = \texttt{letrec}\ a_{i,1}.e_{i,1}; \dots ; a_{i,n}.e_{i,n}\ \texttt{in}\ r_i$ for $i = 1, 2$, and there is a permutation ρ of $\{1, \dots, n\}$, such that after applying this permutation to e_2, i.e. permuting the top letrec bindings, and also permuting the second component of all labeling pairs of bound atoms that are bound in the top letrec of e_2, resulting in $e_2' = \texttt{letrec}\ a_{i,j}'.e_{2,1}'; \dots ; a_{i,n}'.e_{2,n}'\ \texttt{in}\ r_i'$, the following holds: $e_{1,j} \sim_{lev,\alpha} e_{2,\rho(j)}'$ for all j and $r_1 \sim_{lev,\alpha} r_2'$.

The nice property is that the deBruijn-levels do not change under arbitrary permutations of the bindings in a `letrec`, and that under a permutation of an environment only the position indexes of the corresponding occurrences of bound atoms have to be changed.

▶ **Lemma 2.4.** *The relation $\sim_{lev,\alpha}$ is identical with $\sim_\alpha$.*

Proof. The observation that is exploited in the proof is that the deBruijn-labels remain invariant under α-renaming and permutation of bindings.

Let s, t be expressions with $s \rightarrow_\alpha t$. Then we show $s \sim_{lev,\alpha} t$.

Assume that s, t are deBruijn-labeled. Let $s = C[s_1], t = C[t_1]$ and $s_1 \rightarrow_\alpha t_1$ by an immediate renaming or permutation. If s_1, t_1 are letrec expressions and $s_1 \rightarrow_\alpha t_1$ is a permutation of the bindings, then it is easy to see using Definition 2.3 that also $s_1 \sim_{lev,\alpha} t_1$ and thus also $s \sim_{lev,\alpha} t$. If s_1, t_1 are lambda expressions and $s_1 \rightarrow_\alpha t_1$ is a renaming, then $s_1 \sim_{lev,\alpha} t_1$ holds. If s_1, t_1 are letrec expressions and $s_1 \rightarrow_\alpha t_1$ is a renaming, then the conditions in Definition 2.3 ensure that the deBruijn labels are unchanged, and hence $s \sim_{lev,\alpha} t$. By standard reasoning $\sim_{lev,\alpha}$ is reflexive and symmetric.

For the other direction, let us assume s, t are *LRL*-expressions, and $s \sim_{lev,\alpha} t$. The simple idea is to show that s as well as t can be reduced to the same expression using perhaps several $\rightarrow_\alpha$-steps. This is done by first proceeding top-down using the definition of $\sim_{lev,\alpha}$, by selecting and applying permutations of all letrec environments in t. This will result in t'. Then we select a sequence $b_1, b_2, \dots$ of fresh atoms for s, t and generate s' from s by renaming every binder in a top-down left-to-right fashion where always the next atom from the sequence is chosen. The renamings are always possible since the atoms b_i are fresh ones. The same for t' by reusing $b_1, b_2, \dots$. This will result in the same final expression. ◀

We will use mappings on atoms from $\mathbb{A}$. A *swapping* $(a\ b)$ is a function that maps an atom a to atom b, atom b to a, and is the identity on other atoms. We will also use finite permutations on atoms from $\mathbb{A}$, which are usually represented as a composition of swappings. Let $dom(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq a\}$. Then every finite permutation can be represented by a composition of at most $|dom(\pi)|$ swappings. Composition $\pi_1 \circ \pi_2$ and inverses π^{-1} can be immediately computed. Note that swapping and permutations also change names of bound atoms. Swappings and permutations π operate on expressions simply by recursing on the structure. For a letrec expression this is $\pi \cdot (\texttt{letrec}\ a_1.s_1; \dots ; a_n.s_n\ \texttt{in}\ e)$ $= (\texttt{letrec}\ \pi \cdot a_1.\pi \cdot s_1; \dots ; \pi \cdot a_n.\pi \cdot s_n;\ \texttt{in}\ \pi \cdot e)$.

A further definition of equivalence is the following:

▶ **Definition 2.5.** The equivalence $\sim$ on expressions $e \in LRL$ is defined as follows:

- $a \sim a$.
- if $e_i \sim e'_i$ for all i, then $f e_1 \ldots e_n \sim f e'_1 \ldots e'_n$ for an n-ary function symbol f.
- If $e \sim e'$, then $\lambda a.e \sim \lambda a.e'$.
- If for $a \neq b$, $a\#e'$, $e \sim (a\ b) \cdot e'$, then $\lambda a.e \sim \lambda b.e'$.
- `letrec` $a_1.e_1; \ldots; a_n.e_n$ `in` e_0 $\sim$ `letrec` $a'_1.e'_1; \ldots; a'_n.e'_n$ `in` e'_0 iff there is some permutation ρ on $\{1, \ldots, n\}$, such that $\lambda a_1. \ldots .\lambda a_n.(e_1, \ldots, e_n, e_0) \sim \lambda a'_{\rho(1)}. \ldots .\lambda a'_{\rho(n)}.(e'_{\rho(1)}, \ldots, e'_{\rho(n)}, e'_0)$. ◀

Lemma 2.4 implies:

▶ **Lemma 2.6.** *$\sim$ is the same as $\sim_{lev,\alpha}$ which is already known as equivalent to $\sim_\alpha$*

In the following we will use the results on complexity of operations in permutation groups, see [17], and [10]. We consider a set $\{a_1, \ldots, a_n\}$ of distinct objects, the symmetric group $\Sigma(\{a_1, \ldots, a_n\})$ (of size $n!$) of permutations of the objects, and consider its elements, subsets and subgroups. Subgroups are always represented by a set of generators. If H is a set of elements (or generators), then $\langle H \rangle$ denotes the generated subgroup. Some facts are:

- Permutations can be represented in space linear in n.
- Every subgroup of $\Sigma(\{a_1, \ldots, a_n\})$ can be represented by an at most quadratic number (in n) of generators.
- However, not every element in a subgroup can be represented by a product of a polynomial number of generators.

The following questions can be answered in polynomial time, where we always assume that the groups are presented by a set of generators:

- The element-question: $\pi \in G$?,
- The subgroup question: $G_1 \subseteq G_2$.

However, intersection of groups and set-stabilizer (i.e. $\{\pi \in G \mid \pi(M) = M\}$) are not known to be computable in polynomial time. Those problems are as hard as graph-isomorphism (see [17]).

3 A Nominal Letrec Unification Algorithm

As an extension of *LRL*, there is also a countably infinite set of variables X, Y also denoted perhaps using indices. Tuples of any arity ≥ 1 are written as $(e_1, \ldots, e_n)$, and are treated as function symbols in the language. The syntax of the language *LRLX* (**L**et**R**ec **L**anguage e**X**tended) is

$$e \quad ::= \quad a \mid X \mid \pi \cdot X \mid \lambda a.e \mid (f\ e_1\ \ldots e_{ar(c)}) \mid (\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$$

Expressions $\pi \cdot X$ are called *suspensions*. There is an extra simplification operation: $\pi_1 \cdot (\pi_2 \cdot X) \to (\pi_1 \circ \pi_2) \cdot X)$. Note that $\pi \cdot e$ for a non-variable e means an operation, but not a language expression. A *freshness constraint* in our unification algorithm is of the form $a\#e$, where e is an *LRLX*-expression. An *atomic* freshness constraint is of the form $a\#X$.

▶ **Definition 3.1** (Simplification of Freshness Constraints).

$$\frac{\{a\#b\} \mathbin{\dot\cup} \nabla\ ,\ a \neq b}{\nabla} \qquad \frac{\{a\#(f\ s_1 \ldots s_n)\} \mathbin{\dot\cup} \nabla}{\{a\#s_1, \ldots, a\#s_n\} \mathbin{\dot\cup} \nabla} \qquad \frac{\{a\#(\lambda a.s)\} \mathbin{\dot\cup} \nabla}{\nabla}$$

$$\frac{\{a\#(\lambda b.s)\}\dot\cup\nabla}{\{a\#s\}\dot\cup\nabla} \text{ if } a \neq b \qquad \frac{\{a\#(\texttt{letrec } a_1.s_1;\ldots,a_n.s_n \texttt{ in } r)\}\dot\cup\nabla}{\nabla} \text{ if } a \in \{a_1,\ldots,a_n\}$$

$$\frac{\{a\#(\texttt{letrec } a_1.s_1;\ldots,a_n.s_n \texttt{ in } r)\}\dot\cup\nabla}{\{a\#s_1,\ldots a\#s_n, a\#r\}\dot\cup\nabla} \text{ if } a \notin \{a_1,\ldots,a_n\} \qquad \frac{\{a\#(\pi\cdot X)\}\dot\cup\nabla}{\pi^{-1}(a)\#X\}\dot\cup\nabla}$$

▶ **Definition 3.2.** A *LRLX*-unification problem is a pair (Γ,∇), where Γ is a set of equations $s_1 \doteq t_1,\ldots,s_n \doteq t_n$ between expressions from *LRLX*, and ∇ is a set of freshness constraints. A *ground solution* of (Γ,∇) is a substitution σ (for variables), such that $\Gamma\sigma$ is an *LRL*-equation system, in $\Gamma\sigma$ all equations hold w.r.t. α-equivalence (see Definition 2.1) and σ is also a solution of ∇. That means, for all $i : s_i\sigma \sim_\alpha t_i\sigma$, and for all $a\#e \in \nabla$: $a \notin FA(e\sigma)$ holds.

Since we want to avoid the exponential size explosion of the Robinson-style unification algorithms, keeping the good properties of Martelli Montanari-style unification algorithms [19], but not their notational overhead, we stick to a set of equations as data structure. As a preparation for the algorithm, all expressions in equations are exhaustively flattened as follows: $(f\ t_1\ldots t_n) \to (f\ X_1\ldots X_n)$ plus the equations $X_1 \doteq t_1,\ldots,X_n \doteq t_n$. Also $\lambda a.s$ is replaced by $\lambda a.X$ with equation $X \doteq s$, and $(\texttt{letrec } a_1.s_1;\ldots,a_n.s_n \texttt{ in } r)$ is replaced by $(\texttt{letrec } a_1.X_1;\ldots,a_n.X_n \texttt{ in } X)$ and the additional equations $X_1 \doteq s_1;\ldots;X_n \doteq s_n; X \doteq r$. The introduced variables are always fresh ones. We may denote the resulting set of equations of flattening an equation *eq* as *flat*(*eq*). Thus, all expressions in equations are of depth at most 1, where we do not count the permutation applications in the suspensions.

▶ Remark. Replacement of the equation $X \doteq \pi\cdot X$ by freshness constraints as in the nominal unification algorithm in [29] is not complete in the presence of `letrec`. As an example, the relation $(a\ b)\cdot(\texttt{letrec } c.a; d.b \texttt{ in } \mathit{True}) \sim_\alpha (\texttt{letrec } c.a; d.b \texttt{ in } \mathit{True})$ holds, which means that there are solutions t in *LRL* of the equation $X \doteq (a\ b)\cdot X$ with $FA(t) = \{a,b\}$.

A variable-dependency quasi-ordering on variables occurring in Γ is required: If $e_1 \doteq e_2$ is in Γ, and e_1 is X or $\pi\cdot X$, and e_2 is Y or $\pi'\cdot Y$, then $X \approx_{dep,imm} Y$, and otherwise, if e_2 contains an occurrence of Y, then $X >_{dep,imm} Y$. Let $\approx_{dep}$ be the equivalence closure of $\approx_{dep,imm}$, and let $>_{dep}$ be the transitive closure of $\approx_{dep} \circ >_{dep,imm} \circ \approx_{dep}$. This quasi-ordering is only used, if no failure rule (see below) applies, hence there are no cycles.

3.1 Rules of the Algorithm LetrecUnify

LetrecUnify operates on a tuple (Γ,∇,θ), where Γ is a set of flattened equations $e_1 \doteq e_2$, where we assume that $\doteq$ is symmetric, ∇ contains freshness constraints, θ represents the already computed solution as a list of replacements of the form $X \mapsto e$. Initially θ is empty. The final state, i.e. the output, is when Γ only contains fixpoint equations of the form $X \doteq \pi\cdot X$ that are non-redundant.

In the notation of the rules, we may omit ∇ or θ if they are not changed. We will use a notation | in the consequence part of the rules, perhaps with a set of possibilities, to denote disjunctive (i.e. don't know) nondeterminism.

Standard and decomposition rules:

$$(1)\ \frac{\Gamma\dot\cup\{e \doteq e\},\nabla}{\Gamma,\nabla} \qquad (2)\ \frac{\Gamma\dot\cup\{\pi\cdot X \doteq s\},\nabla}{\Gamma\dot\cup\{X \doteq \pi^{-1}\cdot s\},\nabla} \text{ if } s \text{ is not a variable.}$$

$$(3)\ \frac{\Gamma\dot\cup\{X \doteq \pi\cdot Y\},\nabla,\theta \qquad X \neq Y}{\Gamma[\pi\cdot Y/X],\nabla[\pi\cdot Y/X],\theta\cup\{X \mapsto \pi\cdot Y\}} \qquad (4)\ \frac{\Gamma\dot\cup(f\ s_1\ldots s_n) \doteq (f\ s'_1\ldots s'_n)\},\nabla}{\Gamma\dot\cup\{s_1 \doteq s'_1,\ldots,s_n \doteq s'_n\},\nabla}$$

$$(5)\ \frac{\Gamma\dot\cup(\lambda a.s \doteq \lambda a.t\},\nabla}{\Gamma\dot\cup\{s \doteq t\},\nabla} \qquad (6)\ \frac{\Gamma\dot\cup(\lambda a.s \doteq \lambda b.t\},\nabla \qquad a \neq b}{\Gamma\dot\cup\{s \doteq (a\ b)\cdot t\},\nabla\cup\{a\#t\}}$$

$$(7)\quad \frac{\Gamma\dot\cup\{\texttt{letrec}\ a_1.s_1;\ldots,a_n.s_n\ \texttt{in}\ r \doteq \texttt{letrec}\ b_1.t_1;\ldots,b_n.t_n\ \texttt{in}\ r'\},\nabla}{\underset{\forall\rho}{|}\ \Gamma\dot\cup \mathit{flat}(\lambda a_1.\ldots\lambda a_n.(s_1,\ldots,s_n,r) \doteq \lambda b_{\rho(1)}.\ldots\lambda b_{\rho(n)}.(t_{\rho(1)},\ldots,t_{\rho(n)},r')),\ \nabla}$$

where ρ is a (mathematical) permutation on $\{1,\ldots,n\}$.

Rules (1), (2), (3) are called standard rules, and rules (4)–(7) are called decomposition rules.

Main Rules: The following rules (MMS) (Martelli-Montanari-Simulation) and (FPS) (Fixpoint-Shift) will always be immediately followed by a decomposition of the resulting set of equations.

$$\text{(MMS)}\ \frac{\Gamma\dot\cup\{X \doteq e_1, X \doteq e_2\},\nabla}{\Gamma\dot\cup\{X \doteq e_1, e_1 \doteq e_2\},\nabla}\ \text{If } e_1, e_2 \text{ are neither variables nor suspensions.}$$

$$\text{(FPS)}\ \frac{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X, X \doteq e\},\theta}{\Gamma\dot\cup\{e \doteq \pi_1{\cdot}e,\ldots,e \doteq \pi_n{\cdot}e\},\theta\cup\{X\mapsto e\}}$$

If X is maximal w.r.t. $>_{dep}$, no variable $Y \neq X$ with $X \approx_{dep} Y$ occurs in Γ, there are no other equations where X occurs, and e is neither a variable nor a suspension, and no failure rule (see below) is applicable.

$$\text{(ElimFP)}\ \frac{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X, X \doteq \pi{\cdot}X\},\theta}{\Gamma\dot\cup\{X \doteq \pi_1{\cdot}X,\ldots,X \doteq \pi_n{\cdot}X\},\theta}\ \text{If } \pi \in \langle\pi_1,\ldots,\pi_n\rangle$$

We assume that the rule (ElimFP) will be applied whenever possible.
Note that the two rules (MMS) and (FPS) have the potential to generate an exponential blow-up in the number of fixpoint-equations. Thus a further rule will limit the number of fixpoint equations by exploiting knowledge on operations on permutation groups.

3.2 Failure Rules of the Algorithm

(Clash Failure:) if $s \doteq t$ is in Γ, where s, t are not variables or suspensions, and $\mathit{tops}(s) \neq \mathit{tops}(t)$.
(Cycle Detection) If there are equations $X_1 \doteq s_1,\ldots,X_n \doteq s_n$ where s_i are neither variables nor suspensions, and X_{i+1} occurs in s_i for $i = 1,\ldots,n-1$ and X_1 occurs in s_n.
(Freshness Fail) If there is a freshness constraint $a\#a$, then FAIL
(Freshness Solution Fail) If there is a freshness constraint $a\#X \in \nabla$, and $a \in \mathit{FA}((X)\theta)$ then FAIL.

The computation of $\mathit{FA}((X)\theta)$ can be done in polynomial time by iterating over the solution components instead of first computing $X\,\theta$ and then taking FA of the result.

▶ **Example 3.3.** We illustrate the letrec-rule by a ground example, however, without flattening.
Let the equation be: $(\texttt{letrec}\ a.(a,b), b.(a,b)\ \texttt{in}\ b \doteq \texttt{letrec}\ b.(b,c), c.(b,c)\ \texttt{in}\ c)$
Select the identical permutation ρ, which results in:
$\lambda a.\lambda b.((a,b),(a,b),b) \doteq \lambda b.\lambda c.((b,c),(b,c),c)$. Then:
$\lambda b.((a,b),(a,b),b) \doteq (a\ b)\lambda c.((b,c),(b,c),c)$. (The freshness constraint $a\#\ldots$ holds)
$\lambda b.((a,b),(a,b),b) \doteq \lambda c.((a,c),(a,c),c)$. apply the λ-rule:
$((a,b),(a,b),b) \doteq (b\ c)((a,c),(a,c),c)$. (The freshness constraint $b\#\ldots$ holds)
The resulting equation is $((a,b),(a,b),b) \doteq ((a,b),(a,b),b)$, which obviously holds.

▶ **Example 3.4.** This example shows that FPS (together with the standard and decomposition rules) may give rise to an exponential number of equations on the size of the original problem. Let there be variables $X_i, i = 0, \dots, n$ and the equations $\Gamma = \{X_n \doteq \pi{\cdot}X_n, X_n \doteq (f\ X_{n-1}\ \rho_n{\cdot}X_{n-1}), \dots, X_2 \doteq (f\ X_1\ \rho_2{\cdot}X_1)\}$ where $\pi, \rho_1, \dots, \rho_n$ are permutations.

We prove that this unification problem may give rise to 2^{n-1} many equations, if the redundancy rule is not there.

The first step is by (FPS): $$\left\{ \begin{array}{rcl} f\ X_{n-1}\ \rho_n{\cdot}X_{n-1} & \doteq & \pi{\cdot}(f\ X_{n-1}\ \rho_n{\cdot}X_{n-1}), \\ X_{n-1} & \doteq & (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \dots \end{array} \right\}$$

Using decomposition and inversion $$\left\{ \begin{array}{rcl} X_{n-1} & \doteq & \pi{\cdot}X_{n-1}, \\ X_{n-1} & \doteq & \rho_n^{-1}{\cdot}\pi{\cdot}\rho_n{\cdot}X_{n-1}, \\ X_{n-1} & \doteq & (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \dots \end{array} \right\}$$

The next step by (FPS) $$\left\{ \begin{array}{rcl} (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}) & \doteq & \pi{\cdot}(f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \\ (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}) & \doteq & \rho_n^{-1}{\cdot}\pi{\cdot}\rho_n{\cdot}(f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \\ X_{n-2} & \doteq & (f\ X_{n-3}\ \rho_{n-2}{\cdot}X_{n-3}), \dots \end{array} \right\}$$

Using decomposition and inversion $$\left\{ \begin{array}{rcl} X_{n-2} & \doteq & \pi{\cdot}X_{n-2}, \\ X_{n-2} & \doteq & \rho_{n-1}^{-1}{\cdot}\pi{\cdot}\rho_{n-1}{\cdot}X_{n-2}, \\ X_{n-2} & \doteq & \rho_n^{-1}{\cdot}\pi{\cdot}\rho_n{\cdot}X_{n-2}, \\ X_{n-2} & \doteq & \rho_{n-1}^{-1}{\cdot}\rho_n^{-1}{\cdot}\pi{\cdot}\rho_n{\cdot}\rho_{n-1}{\cdot}X_{n-2}, \\ X_{n-2} & \doteq & (f\ X_{n-3}\ \rho_{n-2}{\cdot}X_{n-3}), \dots \end{array} \right\}$$

Now it is easy to see that all equations $X_1 \doteq \pi'{\cdot}X_1$ are generated, with $\pi' \in \{\rho^{-1}\pi\rho$ where ρ is a composition of a subsequence of $\rho_n, \rho_{n-1}, \dots, \rho_2\}$, which makes 2^{n-1} equations. The permutations are pairwise different using an appropriate choice of ρ_i and π. The starting equations can be constructed using the decomposition rule of abstractions.

4 Correctness and Complexity of LetrecUnify

▶ **Theorem 4.1.** *The algorithm* LetrecUnify *terminates. The number of steps as well as the size of the intermediate state is at most polynomial. Thus it runs in nondeterministic polynomial time.*
Its determinized version returns at most an exponential number of final representations, which are of at most polynomial size.

Proof. Note that we assume that the input equations are flattened before applying the rules, which can be performed in polynomial time.
Let Γ_0, ∇_0 be the input, and let $S = \mathit{size}(\Gamma_0, \nabla_0)$. The execution of the rules can be done in polynomial time depending on the size of the intermediate state, thus we have only to show that the number of rule applications is at most polynomial.
The termination measure $(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6)$, which is ordered lexicographically, is as follows: μ_1 is the number of letrec expressions in Γ, μ_2 is the number of letrec-, λ-symbols, function-symbols and atoms in Γ, μ_3 is the number of different variables in Γ, μ_4 is the number of occurrences of variables in Γ, μ_5 is the number of equations not of the form $X \doteq e$, and μ_6 is the number of equations.

Since shifting permutations down and simplification of freshness constraints both terminate and do not increase the measures, we only compare states which are normal forms for shifting down permutations and simplifying freshness constraints. We assume that the algorithm stops if a failure rule is applicable, and that the rules (MMS) and (FPS) are immediately followed by a full decomposition of the results (or failure).

Now it is easy to check that the rule applications strictly decrease μ: The rules (MMS) and (FPS) together with the subsequent decomposition strictly decrease (μ_1, μ_2). Since expressions in equations are flat, (MMS) does not increase the size: $X \doteq s_1, X \doteq s_2$ is first replaced by $X \doteq s_1, s_1 \doteq s_2$, and the latter is decomposed, which due to flattening results only in equations containing variables and suspensions. Thus μ_2 is reduced by the size of s_2. In the same way (FPS) strictly decreases (μ_1, μ_2).
The number of fixpoint-equations for every variable is at most S^2, since the number of atoms is not increased and since we assume that (ElimFP) is applied whenever possible. Note that the redundancy of generators can be tested in polynomial time depending on the number of atoms.
Now we prove a (global) upper bound on the number of variables μ_3: An application of (MMS) may increase μ_3 at most by S. There is a constant c, such that an application of (FPS) may increase this number at most by $c * S \log(S) * S$, where the worst case occurs when e is a letrec-expression. This holds, since the size of the permutation group is at most $S!$, and so the length of proper subset-chains of subgroups is at most $\log(S!) = O(S * log(S))$. Since (MMS) and (FPS) can be applied at most S times, the number of variables is not more than $c * S^3 \log(S)$.
The other rules strictly decrease (μ_1, μ_2), or they do not increase (μ_1, μ_2), and strictly decrease $(\mu_3, \mu_4, \mu_5, \mu_6)$. ◀

The problematic rule for complexity is (FPS), which does not increase μ_1 and μ_2, but may increase μ_3, μ_4 and μ_6 (see Example 3.4). This increase is defeated by the rule (ELimFP), which helps to keep the numbers μ_4 and μ_6 low.

▶ **Theorem 4.2.** *The algorithm* LetrecUnify *is sound and complete.*

Proof. Soundness of the algorithm holds, by easy arguments for every rule, similar as in [29], and since the letrec-rule is justified in Lemma 2.6. A further argument is that the failure rules are sufficient to detect final states without solutions.
Completeness requires more arguments. The decomposition and standard rules (with the exception of the letrec-letrec-rule) retain the set of solutions. The same for (MMS), (FPS), and (ElimFp). The letrec-letrec-rule provides all possibilities for solutions. And the failure rules are not applicable to states that are solvable.

A final output of LetrecUnify (which may still contain fixpoint equations) has at least one ground solution as instance: we can instantiate all variables that remain in Γ_{out} by a fresh atom. Then all fixpoint equations are satisfied, since the permutations cannot change this atom, and since the (atomic) freshness constraints hold. This ground solution can be represented in polynomial space by using θ, plus an instance $X \mapsto a$ for all remaining variables X and a fresh atom a, and removing all fixpoint equations and freshness constraints. ◀

▶ **Theorem 4.3.** *The nominal letrec-unification problem is in NP.*

Proof. This follows from Theorem 4.1 and 4.2. ◀

5 Nominal Matching with Letrec: LetrecMatch

Reductions in higher order calculi with letrec, in particular on a meta-notation, always require a form of matching the rules to the current expressions. In particular higher-order reductions may benefit from nominal letrec-matching algorithms.

▶ **Example 5.1.** Consider the (lbeta)-rule, which is the version of (beta) used in call-by-need calculi with sharing [1, 22, 26].
$(lbeta)$ $(\lambda x.e_1)\ e_2 \to \texttt{letrec}\ x.e_2\ \texttt{in}\ e_1$.
An (lbeta) step for example on$(\lambda x.x)\ (\lambda y.y)$ is performed by switching to the language LRL and then matching $(app\ (\lambda a.X_1)\ X_2) \trianglelefteq app\ (\lambda a.a)\ (\lambda b.b)$, where app is the explicit representation of the binary application operator. This results in $\sigma := \{X_1 \mapsto a; X_2 \mapsto (\lambda b.b)\}$, and the reduction result is the σ-instance of $(\texttt{letrec}\ a.X_2\ \texttt{in}\ X_1)$, which is $(\texttt{letrec}\ a.(\lambda b.b)\ \texttt{in}\ a)$. Note that only the sharing power of the recursive environment is used here.

We derive such an algorithm as a specialization of LetrecUnify. Luckily, matching does not require to deal with fixpoint equations.

We use nonsymmetric equations written $s \trianglelefteq t$, where s is a unification expression, and t does not contain variables. Note that neither freshness constraints nor suspensions are necessary. We assume that the input is a set of equations of (plain) expressions.
The rules of the algorithm LetrecMatch are:

$$\frac{\Gamma \uplus \{e \trianglelefteq e\}}{\Gamma} \qquad \frac{\Gamma \uplus \{(f\ s_1 \dots s_n) \trianglelefteq (f\ s'_1 \dots s'_n)\}}{\Gamma \uplus \{s_1 \trianglelefteq s'_1, \dots, s_n \trianglelefteq s'_n\}} \qquad \frac{\Gamma \uplus \{\lambda a.s \trianglelefteq \lambda a.t\}}{\Gamma \uplus \{s \trianglelefteq t\}}$$

$$\frac{\Gamma \uplus \{\lambda a.s \trianglelefteq \lambda b.t\}}{\Gamma \uplus \{s \trianglelefteq (a\ b){\cdot}t\}} \quad \text{if } a\#t\text{, otherwise Fail.}$$

$$\frac{\Gamma \uplus \{\texttt{letrec}\ a_1.s_1; \dots, a_n.s_n\ \texttt{in}\ r \trianglelefteq \texttt{letrec}\ b_1.t_1; \dots, b_n.t_n\ \texttt{in}\ r'\}}{\underset{\forall \rho}{|}\ \Gamma \uplus \{\lambda a_1. \dots \lambda a_n.(s_1, \dots, s_n, r) \trianglelefteq \lambda a_{\rho(1)}. \dots \lambda a_{\rho(n)}.(t_{\rho(1)}, \dots, t_{\rho(n)}, r')\}}$$

where ρ is a (mathematical) permutation on $\{1, \dots, n\}$

$$\frac{\Gamma \uplus \{X \trianglelefteq e_1, X \trianglelefteq e_2\}}{\Gamma \uplus \{X \trianglelefteq e_1\}} \quad \text{If } e_1 \sim_\alpha e_2\text{, otherwise Fail}$$

The test $e_1 \sim_\alpha e_2$ will be performed using the (nondeterministic) matching rules. **(Clash Failure:)** if $s \doteq t$ is in Γ, where s, t are not variables or suspensions, and $tops(s) \neq tops(t)$.

▶ **Theorem 5.2.** *The algorithm* LetrecMatch *is sound and complete for nominal letrec matching. It decides nominal letrec matching in nondeterministic polynomial time. Its determinized version returns a finite complete set of an at most exponential number of matching substitutions, which are of at most polynomial size.*

Proof. This follows by standard arguments. ◀

▶ **Theorem 5.3.** *Nominal letrec matching is NP-complete.*

Proof. The problem is in NP, which follows from Theorem 5.2. It is also NP-hard, which follows from the (independent) Theorem 6.1. ◀

A slightly more general situation for matching occurs, when the matching equations Γ_0 are compressed using a dag. We construct an algorithm LetrecDagMatch as follows. First we generate Γ_1 from Γ_0, which only contains (plain) flattened expressions by encoding the dag-nodes as variables together with an equation. In order to avoid suspension (i.e. to have nicer results), the decomposition rule for λ-expressions with different binder names is modified as follows, such that the right hand side is always a variable-free expression:

$$\frac{\Gamma \dot\cup (\lambda a.s \doteq \lambda b.t\}, \nabla \qquad a \neq b \qquad \lambda b.t \text{ does not reference variables}}{\Gamma \dot\cup \{s \doteq (a\ b) \cdot t\}, \nabla \cup \{a \# t\}}$$

The extra condition can be tested in polynomial time. The equations Γ_1 are processed applying LetrecUnify (with the mentioned modification) with the guidance that the right-hand sides of match-equations are also right-hand sides of equations in the decomposition rules. The resulting matching substitutions can be interpreted as the instantiations into the variables of Γ_0. Since Γ_0 is a matching problem, the result will be free of fixpoint equations, and there will be no freshness constraints in the solution. Thus we have:

▶ **Theorem 5.4.** *The nondeterministic algorithm* LetrecDagMatch *is sound and complete and requires polynomial time.*
In addition, the result is an at most exponential set of dag-compressed substitutions represented in polynomial space.
Thus the decision problem is NP-complete.

6 Hardness of Nominal Letrec Matching and Unification

▶ **Theorem 6.1.** *Nominal letrec matching (hence also unification) is NP-hard, where both expressions are letrec expressions, and the subexpressions are free of letrec.*

Proof. We encode the NP-hard problem of finding a Hamiltonian cycle in regular graphs. [23, 11]: Let $a_1, \ldots, a_n$ be the vertexes of the graph, and E be the set of edges. The first environment part is $env_1 = a_1.(node\ a_1); \ldots; a_n.(node\ a_n)$, and a second environment part env_2 consists of bindings $b.(f\ a\ a')$ for every edge $(a, a') \in E$ for fresh names b. Then let $s := ($`letrec` $env_1; env_2$ `in` $0)$ representing the graph.
Let the second expression encode the question whether there is a Hamiltonian cycle in a regular graph as follows. The first part of the environment is $env'_1 = a_1.(node\ X_1), \ldots, a_n.(node\ X_n)$. The second part is env'_2 consisting of $b_1.f\ X_1\ X_2; b_2.f\ X_2\ X_3; \ldots b_k.f\ X_n\ X_1$, and the third part consisting of a number of (dummy) entries of the form $b.f\ Z_2\ Z_3$, where b is always a fresh atom for every binding, and Z_2, Z_3 are fresh variables. The number of these dummy entries can be easily computed from the number of nodes and the degree of the graph, and it is less than the size of the graph.

Then the matching problem is solvable iff the graph has a Hamiltonian cycle. ◀

▶ **Theorem 6.2.** *The nominal letrec-unification problem is NP-complete.*

Proof. This follows from Theorems 4.3 and 6.1. ◀

We say that an expression (`letrec` env `in` r) *contains garbage*, iff the environment can be split into two environments $env = env_1; env_2$, such that env_1 is not trivial, and the atoms from $LA(env_1)$ do not occur in env_2 nor in r. Otherwise, the expression is *free of garbage*. Since extended α-equivalence of expressions is Graph-Isomorphism-complete [25], but the extended α-equivalence of garbage-free letrec-expressions is polynomial, it is useful to look for improvements of unification and matching for garbage-free expressions. As a remark: Graph-Isomorphism is known to be between $PTIME$ and NP; there are arguments that it is weaker than the class of NP-complete problems [27]. There is also a claim that it is quasi-polynomial [4], which means that it requires less than exponential time.

▶ **Theorem 6.3.** *Nominal letrec matching of an expression* $(s =$ `letrec` env `in` $X)$ *where* env *is ground, against a ground garbage-free expression* t *is Graph-Isomorphism-complete. I.e. Nominal Matching with one occurrence of a single variable is Graph-Isomorphism-hard.*

Proof. Let G_1, G_2 be two graphs. Let s be ($\texttt{letrec}$ env_1 $\texttt{in}$ f $a_1 \ldots$ a_n) the encoding of a graph G_1 where env_1 is the encoding as above and the nodes are encoded as $a_1 \ldots a_n$. Then the expression s is free of garbage. Let the environment env_2 be the encoding of G_2 in the expression $t = \texttt{letrec}$ env_2 $\texttt{in}$ X. Then t matches s iff the graphs are isomorphic. Hence we have GI-hardness. If there is an isomorphism of G_1 and G_2, then it is easy to see that this bijection leads to an equivalence of the environments, and we can instantiate X with $(f\ a_1 \ldots a_n)$. ◀

7 Nominal Letrec Matching with Environment Variables.

Extending the language by variables Env that may encode partial letrec-environments would lead to a larger coverage of unification problems in reasoning about the semantics of programming languages.

▶ **Example 7.1.** Consider as an example a rule that merges letrec environments:
(llet-e): ($\texttt{letrec}$ Env_1 $\texttt{in}$ ($\texttt{letrec}$ Env_2 $\texttt{in}$ X)) $\to$ ($\texttt{letrec}$ $Env_1; Env_2$ $\texttt{in}$ X)).
This can be applied to an expression ($\texttt{letrec}$ $a.0; b.1$ $\texttt{in}$ $\texttt{letrec}$ $c.(a,b,c)$ $\texttt{in}$ c) as follows: The left-hand side ($\texttt{letrec}$ Env_1 $\texttt{in}$ ($\texttt{letrec}$ Env_2 $\texttt{in}$ X)) of the reduction rule matches ($\texttt{letrec}$ $a.0; b.1$ $\texttt{in}$ ($\texttt{letrec}$ $c.(a,b,c)$ $\texttt{in}$ c)) with the match: $\{Env_1 \mapsto (a.0; b.1); Env_2 \mapsto c.(a,b,c); X \mapsto c\}$, producing the next expression as an instance of the right hand side ($\texttt{letrec}$ $Env_1; Env_2$ $\texttt{in}$ X), which is: ($\texttt{letrec}$ $a.0; b.1; c.(a,b,c)$ $\texttt{in}$ c). Note that in the application to extended lambda calculi, a bit more care (i.e. a further condition) is needed w.r.t. scoping in order to get valid reduction results in all cases.

We will now also have partial environments as syntactic objects.

The grammar for the extended language $LRLXE$ (**L**et**R**ec **L**anguage e**X**tended with **E**nvironments) is:

$$e \quad ::= \quad a \mid X \mid Env \mid \pi \cdot X \mid \lambda a.e \mid (f\ e_1\ \ldots e_{ar(c)}) \mid \texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e$$

We define a matching algorithm, where environment variables may occur in left hand sides. This algorithm needs a more expressive data structure in equations: a letrec with two environment-components, (i) a list of bindings that are already fixed in the correspondence to another environment, and (ii) an environment that is not yet fixed. We denote the fixed bindings as a list, which is the first component. In the notation we assume that the (non-fixed) letrec-environment part on the right hand side may be arbitrarily permuted before the rules are applied. The justification for this special data structure is the scoping in letrec expressions.

Note that suspensions do not occur in this algorithm.

▶ **Definition 7.2.** The matching algorithm LetrecEnvMatch for expressions where environment variables Env and expression variables X may occur only in the left hand sides of match equations is described below. It has the following rules:

$$\frac{\Gamma \dot\cup \{e \trianglelefteq e\}}{\Gamma} \qquad \frac{\Gamma \dot\cup \{(f\ s_1 \ldots s_n) \trianglelefteq (f\ s_1' \ldots s_n')\}}{\Gamma \dot\cup \{s_1 \trianglelefteq s_1', \ldots, s_n \trianglelefteq s_n'\}} \qquad \frac{\Gamma \dot\cup \{\lambda a.s \trianglelefteq \lambda a.t\}}{\Gamma \dot\cup \{s \trianglelefteq t\}}$$

$$\frac{\Gamma \dot\cup \{\lambda a.s \trianglelefteq \lambda b.t\}}{\Gamma \dot\cup \{s \trianglelefteq (a\ b) \cdot t\}} \quad \text{if } a\#t \text{, otherwise Fail.}$$

$$\frac{\Gamma\dot\cup\{(\mathtt{letrec}\ ls; a.s; env\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letrec}\ ls'; b.t; env'\ \mathtt{in}\ r')\}}{\underset{\forall(b.t)}{|}\quad \Gamma\dot\cup\{(\mathtt{letrec}\ ((a.s):ls); env\ \mathtt{in}\ r) \trianglelefteq (a\ b)(\mathtt{letrec}\ ((b.t):ls';\ env'\ \mathtt{in}\ r')\}}$$

if $a\#(\mathtt{letrec}\ ls'; b.t; env'\mathtt{in}\ r')$, otherwise Fail.

$$\frac{\Gamma\dot\cup\{(\mathtt{letrec}\ ls; Env; env\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letrec}\ ls'; env'_1; env'_2\mathtt{in}\ r')\}}{\underset{env'_1}{|}\quad \Gamma\dot\cup\{(\mathtt{letrec}\ (Env:ls); env\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letrec}\ ((env'_1):ls');\ env'_2\ \mathtt{in}\ r')\}}$$

$$\frac{\Gamma\dot\cup\{(\mathtt{letrec}\ ls; \emptyset\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letrec}\ ls'; \emptyset\ \mathtt{in}\ r')\}}{\Gamma\dot\cup\{ls \trianglelefteq ls'; r \trianglelefteq r'\}} \qquad \frac{\Gamma\dot\cup\{[e_1;\ldots;e_n] \trianglelefteq [e'_1;\ldots;e'_n]\}}{\Gamma\dot\cup\{e_1 \trianglelefteq e'_1;\ldots;e_n \trianglelefteq e'_n\}}$$

$$\frac{\Gamma\dot\cup\{X \trianglelefteq e_1, X \trianglelefteq e_2\}}{\Gamma\dot\cup\{X \trianglelefteq e_1, e_1 \doteq e_2\}} \qquad \frac{\Gamma\dot\cup\{Env \trianglelefteq env_1, Env \trianglelefteq env_2\}}{\Gamma\dot\cup\{Env \trianglelefteq env_1, env_1 \doteq env_2\}}$$

Performing $e_1 \doteq e_2$ and $env_1 \doteq env_2$ is done with high priority using the (nondeterministic) matching rules.
(Clash Failure:) if $s \doteq t$ is in Γ, where s, t are not variables nor suspensions, and $tops(s) \neq tops(t)$.

If the execution is successful, then the result will be a set of match equations with components $X \trianglelefteq e$, and $Env \trianglelefteq env$, which represents a matching substitution.

▶ **Theorem 7.3.** *The algorithm 7.2 (*LetrecEnvMatch*) is sound and complete. It runs in non-deterministic polynomial time. The corresponding decision problem is NP-complete. The determinized version returns an at most exponentially large, complete set of representations of matching substitutions, where the representations are of at most polynomial size.*

Proof. The reasoning for soundness, completeness and termination in polynomial time is a variation of previous arguments. The nonstandard part is fixing the correspondence of environment parts step-by-step and at the same time keeping the scoping. ◀

We leave the extension of nominal letrec unification by variables *Env* for future research. A plain AC-unification is not sufficient, since also suspensions $\pi{\cdot}Env$ occur in equations.

7.1 Nominal Unification and Equational Theories

The solution to the nominal letrec unification problem is related to nominal unification of terms in particular equational theories. The narrowing approach in [9] is general, but has a high complexity. We demonstrate that specialized algorithms can do better in particular for equational theories.

Let us transfer our methods to the following language that combines a higher-order language with the equational theory of commutativity. We only sketch the arguments, since these can be derived straightforwardly from our arguments and explanations for nominal letrec unification and matching.

The ground language is $e ::= a \mid \lambda a.e \mid (f\ e_1\ \ldots e_{ar(f)}) \mid (g_C\ e_1\ e_2)$, where f are uninterpreted functions. The relation $\sim_\alpha$ is generated by α-equivalence and all derivations of the commutativity axiom $(g_C\ x\ y) = (g_C\ y\ x)$. Similar as for letrec-unification, there are expressions that are nontrivial fixpoints for swappings and permutations, like $(a\ b){\cdot}(g_C\ a\ b) \sim_C (g_C\ a\ b)$.

The unification rules are as in LetrecUnify, where the letrec-decomposition is missing, and the following rule is added for treating commutativity:

$$\frac{\Gamma \mathbin{\dot\cup} \{(g_C\ s_1\ s_2) \doteq (g_C\ t_1\ t_2)\}, \nabla}{\Gamma \mathbin{\dot\cup} \{s_1 \doteq t_1, s_2 \doteq t_2\}, \nabla \ \mid\ \Gamma \mathbin{\dot\cup} \{s_1 \doteq t_2, s_2 \doteq t_1\}, \nabla}$$

The corresponding non-deterministic nominal commutative unification algorithm will result in either Fail, or a set of equations Γ that only consists of a (nonredundant) set of fixpoint equations of the form $X \doteq \pi{\cdot}X$, which indicates unifiability. The algorithm terminates in polynomial time, and the final results consist of three components: a substitution represented by components, a set of fixpoint equations and a set of atomic freshness constraints. NP-hardness holds already for nominal commutative matching by encoding the MONOTONE ONE-IN-THREE-3-SAT-problem [24], where a clause $x_1 \vee x_2 \vee x_3$ is encoded by the expression $(g_C (g_C\ x_1\ x_2)\ (g_C\ x_3\ 0))$.

Thus we have:

▶ **Theorem 7.4.** *The algorithm* LETRECCOMMUNIFY *decides nominal commutative unifiability in nondeterministic polynomial time. The nominal commutative unification problem is NP-complete. The determinized version of the algorithm* LETRECCOMMUNIFY *results in an at most exponential set of polynomial-sized solutions.*

8 Conclusion

We constructed a nominal letrec unification algorithm, several nominal letrec matching algorithms also for extensions, and a nominal commutative unification algorithm, which all run in nondeterministic polynomial time. Future research is to investigate nominal unification algorithms for letrec-expressions together with environment variables, and also to investigate the related problem of nominal matching also associative commutative function symbols.

References

1 Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL'95*, pages 233–246, San Francisco, California, 1995. ACM Press.

2 Zena M. Ariola and Jan Willem Klop. Cyclic Lambda Graph Rewriting. In *Proc. IEEE LICS*, pages 416–425. IEEE Press, 1994.

3 Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

4 Làszlò Babai. Graph isomorphism in quasipolynomial time. Available from http://arxiv.org/abs/1512.03547v2, 2016.

5 Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.

6 James Cheney. Relating higher-order pattern unification and nominal unification. In *Proc. 19th International Workshop on Unification, UNIF'05*, pages 104–119, 2005.

7 James Cheney. Toward a general theory of names: Binding and scope. In *MERLIN 2005*, pages 33–40. ACM, 2005.

8 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

9 Maribel Fernández, Daniele Nantes-Sobrinho, and Mauricio Ayala-Rincoón. Nominal narrowing. In Delia Kesner and Brigitte Pientka, editors, *FSCD 2016*, LIPIcs. Schloss Dagstuhl, 2016. Accepted for publication.

10 Merrick L. Furst, John E. Hopcroft, and Eugene M. Luks. Polynomial-time algorithms for permutation groups. In *21st FoCS*, pages 36–41. IEEE Computer Society, 1980.

11 M. R. Garey, David S. Johnson, and Robert Endre Tarjan. The planar hamiltonian circuit problem is NP-complete. *SIAM J. Comput.*, 5(4):704–714, 1976.

12 Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

13 Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

14 Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1-2):125–150, 2000.

15 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010.

16 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.

17 Eugene M. Luks. Permutation groups and polynomial-time computation. In Larry Finkelstein and William M. Kantor, editors, *Groups And Computation, Proceedings of a DIMACS Workshop*, volume 11 of *DIMACS*, pages 139–176. DIMACS/AMS, 1991.

18 Simon Marlow, editor. *Haskell 2010 – Language Report.* 2010.

19 Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

20 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

21 Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

22 Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.

23 Christophe Picouleau. Complexity of the Hamiltonian cycle in regular graph problem. *Theor. Comput. Sci.*, 131(2):463–473, 1994.

24 Thomas J. Schaefer. The complexity of satisfiability problems. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *Proc. 10th Annual ACM STOC*, pages 216–226. ACM, 1978.

25 Manfred Schmidt-Schauß, Conrad Rau, and David Sabel. Algorithms for Extended Alpha-Equivalence and Complexity. In Femke van Raamsdonk, editor, *24th International Conference RTA 2013)*, volume 21 of *LIPIcs*, pages 255–270. Schloss Dagstuhl, 2013.

26 Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

27 Uwe Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.

28 Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.

29 Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *17th CSL, 12th Annual Conf. EACSL, and 8th Kurt Gödel Colloquium, KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.

30 Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.