Nominal Unification of Higher Order Expressions with Recursive Let

Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, Mateu Villaret, and Yunus Kutz

GU Frankfurt, Germany
RISC, JKU Linz, Austria
IIIA - CSIC, Spain
IMA, Universitat de Girona, Spain
GU Frankfurt, Germany

Technical Report Frank-62

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

October 23, 2019

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. We also explore specializations like nominal letrec-matching for expressions, for DAGs, and for garbage-free expressions and determine their complexity. As extension a nominal unification algorithm for higher-order expressions with recursive let and atom-variables is constructed, where we show that it also runs in non-deterministic polynomial time.

This paper is an extended version of the conference publication [36].

1 Introduction

Unification [9] 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., [21]), however, the problem is undecidable [20, 24].

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. 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 [46, 47], where the complexity of the decision problem is polynomial time [13]. It can be seen also from a higher-order perspective [26], as equivalent to Miller's higher-order pattern unification [31]. There are efficient algorithms [13, 25], formalizations of nominal unification [8], formalizations with extensions to commutation properties within expressions [4], and generalizations of nominal unification to narrowing [6]. Equivariant (nominal) unification [16, 14, 1] extends nominal unification by permutation-variables, but it can also be seen as a generalization of nominal unification by permitting abstract names for variables.

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 general binding structures like a cyclic let.

The motivation and intended application scenario is as follows: constructing syntactic reasoning algorithms for showing properties of program transformations on higher-order expressions in call-by-need functional languages (see for example [32, 42]) that have a letrec-construct (also called cyclic let) [3] as in Haskell [29], (see e.g. [15] for a discussion on reasoning with more general name binders, and [45] for a formalization of general binders in Isabelle). There may be applications also to coinductive extensions of logic programming [44] and strict functional languages [22]. Basically, 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.

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 [30]. 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 Example 3.2), novel techniques are required and lead to a surprisingly moderate complexity: a fixed-point shifting rule (FPS) and a redundancy removing rule (ElimFP). These rules bound the number of fixpoint equations $X \doteq \pi{\cdot}X$ (where π is a permutation) using techniques and results from computations in permutation groups. The application of these techniques is indispensable (see Example 4.6) for obtaining efficiency.

Inspired by the applications in programming languages, we investigated the notion of garbage-free expressions. The restriction to garbage-free expressions permits several optimizations of the unification algorithms. The first is that testing α-equivalence is polynomial. Another advantage is that due to the unique correspondence of positions for two α-equal garbage-free expressions, we show that in this case, fixpoint equations can be replaced by freshness constraints (Corollary 9.4).

As a further extension, we studied the possibility to formulate input problems using atom variables as in [41, 40] in order to take advantage of the potential of less non-determinism. The corresponding algorithm LetrecUnifyAV requires nested permutations and generalized freshness constraints as further expressibility, and also other techniques such as explicit compression of permutations. The algorithm runs in NP time. We added a strategy to really exploit the extended expressivity and the omission of certain nondeterministic choices.

Related Work: We have already mentioned some related work about nominal unification and its extensions. In one of them, nominal commutative unification [5], one can observe that there are nontrivial fixpoints of permutations. This is similar to what we have in nominal unification with recursive let (when garbage-freeness is not required), which is not surprising, because, essentially, this phenomenon is related to the lack of the ordering: in one case among the arguments of a commutative function symbol, in the other case among the bindings of recursive let. Consequently, nominal C-unification reduces to fixpoint constraints. Those constraints may have infinitely many incomparable solutions expressed in terms of substitutions and freshness constraints (which is the standard way to represent nominal unifiers). In [7], the authors proposed to use fixpoint constraints as a primitive notion (instead of freshness constraints) to axiomatize α-equivalence and, hence, use them in the representation of unifiers, which helped to finitely represent solutions of nominal C-unification problems. The technical report [37] contains explanations how to obtain a nominal C-unification algorithm from a letrec unification algorithm and transfers the NP-completeness result for letrec unification to nominal commutative unification.

The ρ_g-calculus [11] integrates term rewriting and lambda calculus, where cyclic, shared terms are permitted. Such term-graphs are represented as recursion constraints, which resemble to recursive let environments. The evaluation mechanism of ρ_g-calculus is based on matching for such shared structures. Matching and recursion equations are incorporated in the object level and rules for their evaluation are presented.

Unification of higher-order expressions with recursive let (but without nominal features) has been studied in the context of proving correctness of program transformations in call-by-need λ-calculi [35, 34]. Later, in [39], the authors proposed a more elaborated approach to address semantic properties of program calculi, which involves unification of meta-expressions of higher-order lambda calculi with letrec environments. This unification problem extends those from [35, 34]: environments are treated as multisets, different kinds of variables are considered (for letrec environments, contexts, and binding chains), more than one environment variable is permitted, and non-linear unification problems are allowed. Equivalence there is syntactic, in contrast to our nominal approach where equality modulo α is considered. Unlike [39], our unification problems do not involve context and chain variables, but we do have environment variables in matching problems. Moreover, we permit atom variables in an extension of nominal letrec unification.

Results: The nominal letrec unification algorithm is complete and runs in nondeterministic polynomial time (Theorem 5.1, 5.3). The nominal letrec matching is NP-complete (Theorems 6.2, 7.1), as well as the nominal unification problem (Theorems 5.3, 7.1). Nominal letrec matching for dags is in NP and outputs substitutions only (Theorem 6.4), and a very restricted nominal letrec matching problem is graph-isomorphism hard (Theorem 7.3). Nominal matching including letrec-environment variables is in NP (Theorem 8.6). Nominal unification for garbage-free expressions can be done with simple fixpoint rules (Corollary 9.4). In the extension with atom variables, nominal unification can be done using practically useful strategies with less non-determinism and is NP-complete (Theorem 10.13).

2 Some Intuitions

In first order unification we have a language of applications of function symbols over a (possible empty) list of arguments $(fe_1 \dots e_n)$, where n is the arity of f, and variables X. Solutions of equations between terms are substitutions for variables that make both sides of equations syntactically equal. First order unification may be solved using the following two problem transformation rules:

$$\text{(Decomposition)}\ \frac{\Gamma \dot\cup \{(f\, e_1 \dots e_n) \doteq (f\, e'_1 \dots e'_n)\}}{\Gamma \cup \{e_1 \doteq e'_1 \dots e_n \doteq e'_n\}}$$

$$\text{(Instantiation)}\ \frac{\Gamma \dot\cup \{X \doteq e\}}{[X \mapsto e]\Gamma} \quad \text{If } X \text{ does not occur in } e.$$

The substitution solving the original set of equation may be easily recovered from the sequence of transformations. However, the algorithm resulting from these rules is exponential in the worst case.

Martelli and Montanari [30] described a set of improved rules that result into a $\mathcal{O}(n\ \log(n))$ time algorithm.[1] In a first phase the problem is flattened,[2] resulting into equations where every term is a variable or of the form $(f\ X_1 \dots X_n)$. The second phase is a transformation using the following rules:

$$\text{(Decomposition)}\ \frac{\Gamma \dot\cup \{(f\, X_1 \dots X_n) \doteq (f\, Y_1 \dots Y_n)\}}{\Gamma \cup \{X_1 \doteq Y_1, \dots, X_n \doteq Y_n\}} \qquad \text{(Variable Instantiation)}\ \frac{\Gamma \dot\cup \{X \doteq Y\}}{[X \mapsto Y]\Gamma}$$

$$\text{(Elimination)}\ \frac{\Gamma \dot\cup \{X \doteq e\}}{\Gamma} \quad \text{If } X \text{ does not occur in } e \text{ or } \Gamma$$

$$\text{(Merge)}\ \frac{\Gamma \dot\cup \{X \doteq (f\, X_1 \dots X_n), X \doteq (f\, Y_1 \dots Y_n)\}}{\Gamma \cup \{X \doteq (f\, X_1 \dots X_n), X_1 \doteq Y_1, \dots, X_n \doteq Y_n\}}$$

[1] The original Martelli and Montanari's algorithm is a bit different. In fact, they do not flatten equations. However, the essence of the algorithm is basically the same as the one described here.

[2] In the flattening process we replace every proper subterm $(fe_1 \dots e_n)$ by a fresh variable X, and add the equation $X \doteq (fe_1 \dots e_n)$. We repeat this operation (at most a linear number of times) until all proper subterms are variable occurrences.

Notice that in these rules the terms involved in the equations are not modified (they are not instantiated), except by the replacement of a variable by another in the Variable Instantiation rule. We can define a measure on problems as the number of distinct variables, plus the number of equations, plus the sum of the arities of the function symbols occurrences. All rules decrease this measure (for instance, the merge rule increases the number of equations by $n-1$, but removes a function symbol occurrence of arity n). Since this measure is linear in the size of the problem, this proves that the maximal number of rule applications is linear. The Merge rule is usually described as

$$\frac{\Gamma \uplus \{X \doteq e_1, X \doteq e_2\}}{\Gamma \cup \{X \doteq e_1, e_1 \doteq e_2\}} \quad \text{If } e_1 \text{ and } e_2 \text{ are not variables}$$

However, this rule does not decrease the proposed measure. We can force the algorithm to, if possible, immediately apply a decomposition of the equation $e_1 \doteq e_2$. Then, the application of both rules (resulting into the first proposed Merge rule) do decrease the measure.

2.1 Nominal Unification

Nominal unification is an extension of first-order unification where we have lambda-binders. Variables of the target language are called atoms, and the unification-variables are simply called variables. Bound atoms can be renamed. For instance, $\lambda a.(f\,a)$ is equivalent to $\lambda b.(f\,b)$. We also have permutations of atom names (represented as swappings) applied to expressions of the language. When these permutations are applied to a variable, this is called a *suspension*. The action of a permutation on a term is simplified until we get a term where permutations are innermost and only apply to variables. For instance, $(a\,b)\cdot\lambda a.(f\,X\,a\,(f\,b\,c))$, where $(a\,b)$ is a swapping between the atoms a and b, results into $\lambda b.(f\,(a\,b)\cdot X\,b\,(f\,a\,c))$. As we will see below, we also need a predicate to denote that an atom a cannot occur free in a term e, noted $a\#e$.

We can extend the previous first-order unification algorithm to the nominal language modulo α-equivalence. The decomposition of λ-expressions distinguishes two cases, when the binder name is the same and when they are distinct and we have to rename one of them:

$$\begin{array}{c}\text{(Decomposition} \\ \text{lambda 1)}\end{array} \frac{\Gamma \uplus \{\lambda a.s \doteq \lambda a.t\}}{\Gamma \cup \{s \doteq t\}} \qquad \begin{array}{c}\text{(Decomposition} \\ \text{lambda 2)}\end{array} \frac{\Gamma \uplus \{\lambda a.s \doteq \lambda b.t\}}{\Gamma \cup \{s \doteq (a\ b)\cdot t, a\#t\}}$$

As we see in the second rule, we introduce a *freshness constraint* that has to be checked or solved, so we need a set of transformations for this kind of equations. This set of freshness constraints are solved in a second phase of the algorithm.

As we have said, permutations applied to variables cannot be longer simplified and result into suspensions. Therefore, now, we deal with suspensions instead of variables, and we do not make any distinction between X and $Id{\cdot}X$. Variable instantiation distinguishes two cases:

$$\begin{array}{l}\text{(Variable} \\ \text{Instantiation)}\end{array} \frac{\Gamma \uplus \{\pi\cdot X \doteq \pi'\cdot Y\}}{[X \mapsto (\pi^{-1}\circ\pi')\cdot Y\ X \neq Y]\Gamma} \qquad \text{(Fixpoint)} \frac{\Gamma \uplus \{\pi\cdot X \doteq \pi'\cdot X\}}{\Gamma \cup \{a\#X \mid a \in dom(\pi^{-1}\circ\pi')\}}$$

Notice that equations between the same variable $X \doteq X$ that are trivially solvable in first-order unification, adopt now the form $\pi\cdot X \doteq \pi'\cdot X$. This kind of equations are called *fixpoint equations* and impose a restriction on the possible instantiations of X, when π and π' are not the identity. Namely, $\pi\cdot X \doteq \pi'\cdot X$ is equivalent to $\{a\#X \mid a \in dom(\pi^{-1}\circ\pi')\}$, where the domain $dom(\pi)$ is the set of atoms a such that $\pi(a) \neq a$.

From this set of rules we can derive an $\mathcal{O}(n^2 \log n)$ algorithm, similar to the algorithms described in [13, 25]. This algorithm has three phases. First, it flattens all equations. Second, it applies this set of problem transformation rules. Using the same measure as in the first-order case (considering lambda abstraction as a unary function symbol and not counting the number of freshness equations), we can prove that the length of problem transformation sequences is always linear. In a third phase, we deal with freshness equations. Notice that the number of distinct non-simplifiable freshness equations $a\#X$ is quadratically bounded.

2.2 Letrec Expressions

Letrec expressions have the form ($\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e$). Variables a_i are binders where the scope is in any of the expressions e_j and in e. We can rename these binders, obtaining an equivalent expression. For instance, $(\texttt{letrec}\ a.(f\,a)\ \texttt{in}\ (g\,a)) \sim (\texttt{letrec}\ b.(f\,b)\ \texttt{in}\ (g\,b))$. Moreover, we can also swap the order of definitions. For instance, $(\texttt{letrec}\ a.f; b.g\ \texttt{in}\ (h\,a\,b)) \sim (\texttt{letrec}\ b.g; a.f\ \texttt{in}\ (h\,a\,b))$. Schmidt-Schauß et al. [38] prove that equivalence of letrec expressions is graph-isomorphism (GI) complete and Schmidt-Schauß and Sabel [39] prove that unification is NP-complete. The GI-hardness can be elegantly proved by encoding any graph, like $G = (V, E) = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3)\})$, into a letrec expression, like $(\texttt{letrec}\ v_1.a; v_2.a; v_3.a\ \texttt{in}\ \texttt{letrec}\ e_1.(c\,v_1\,v_2); e_2.(c\,v_2\,v_3)\ \texttt{in}\ a)$.

Unfortunately, there are nontrivial fixpoints of permutations in the letrec-language. For example, $(\texttt{letrec}\ a_1.b_1,\ a_2.b_2, a_3.a_3\ \texttt{in}\ a_3)$ is a fixpoint of the equation $X \doteq (b_1\ b_2) \cdot X$, although b_1 and b_2 are not fresh in the expression. Therefore, the fixpoint rule of the nominal algorithm in [46] would not be complete in our setting: to ensure $X \doteq (b_1\ b_2) \cdot X$ we cannot require $b_1\#X$ and $b_2\#X$. See also Example 3.2. Hence, fixpoint equations can in general not be replaced by freshness constraints. For the general case we need a complex elimination rule, called fixed point shift:

$$\text{(FixPointShift)}\ \frac{\Gamma \dot\cup \{X \doteq \pi_1 \cdot X, \ldots, X \doteq \pi_n \cdot X, X \doteq e\}}{\Gamma \cup \{e \doteq \pi_1 \cdot e, \ldots, e \doteq \pi_n \cdot e\}} \quad \text{If } X \text{ does not occur in } e \text{ or } \Gamma.$$

This rule can generate an exponential number of equations (see Example 4.6). In order to avoid it, we will use a property on the number of generators of permutation groups (see end of Section 3).

For the decomposition of letrec expressions we also need to introduce a (don't know) non-deterministic choice. With this artifact, we can reduce equations between letrec expressions into equations between lambda expressions:

$$\text{(Decomposition Letrec)}\ \frac{\Gamma \dot\cup \{\texttt{letrec}\ a_1.s_1; \ldots; a_n.s_n \texttt{in}\ r \doteq \texttt{letrec}\ b_1.t_1; \ldots; b_n.t_n \texttt{in}\ r'\}}{|\forall_\rho\ \Gamma \cup \{\lambda a_1 \ldots \lambda a_n.(s_1, \ldots, s_n, r) \doteq \lambda b_{\rho(1)} \ldots \lambda b_{\rho(n)}.(t_{\rho(1)}, \ldots, t_{\rho(n)}, r')\}}$$

In order to decrease the measure, we should immediately apply decomposition of lambda expressions (similarly to the decomposition applied immediately after the Merge rule). In section 4, we will describe in full detail all the transformation rules of our algorithm.

3 The Ground Language of Expressions

The very first idea of nominal techniques [46] is to use concrete variable names in lambda-calculi (also in extensions), in order to avoid implicit α-renamings, and instead use operations for explicitly applying α-renaming. Suppose $s = \lambda\mathtt{x}.\mathtt{x}$ and $t = \lambda\mathtt{y}.\mathtt{y}$ are concrete (syntactically different) lambda-expressions. The nominal technique provides explicit name-changes using permutations. These permutations are applied irrespective of binders. For example $(\mathtt{x}\ \mathtt{y})(\lambda\mathtt{x}.\lambda\mathtt{x}.\mathtt{a})$ results in $\lambda\mathtt{y}.\lambda\mathtt{y}.\mathtt{a}$. Syntactic reasoning on higher-order expressions, for example unification of higher-order expressions modulo α-equivalence will be done by nominal techniques on a language with concrete names, where the algorithms require certain extra constraints and operations. The gain is that all conditions and substitutions etc. can be computed and thus more reasoning tasks can be automated, whereas the implicit name conditions under implicit α-equivalence have a tendency to complicate (unification-) algorithms and to hide the required conditions on equality/disequality/occurrence/non-occurrence of names.

3.1 Preliminaries

We define the language *LRL* (**L**et**R**ec **L**anguage) of (ground-)expressions, which is a lambda calculus extended with a recursive let construct. The notation is consistent with [46]. The (infinite) set of atoms $\mathbb{A}$ is a set of (concrete) symbols a, b which we usually denote in a meta-fashion; so we can use symbols

a, b also with indices (the variables in lambda-calculus). There is a set $\mathcal{F}$ of function symbols with arity $ar(\cdot)$. The syntax of the expressions e of *LRL* is:

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

We also use tuples, which are written as $(e_1, \ldots, e_n)$, and which are treated as functional expressions in the language. 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 abbreviated as *env*.

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 $a \notin FA(e)$ (also denoted as $a\#e$). As an example, the expression $(\texttt{letrec}\ a.cons\ s_1\ b; b.cons\ s_2\ a\ \texttt{in}\ a)$ represents an infinite list $(cons\ s_1\ (cons\ s_2\ (cons\ s_1\ (cons\ s_2\ \ldots))))$, where s_1, s_2 are expressions. However, since our language *LRL* is only a fragment of core calculi [32, 42], the reader may find more programming examples there.

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 π on atoms from $\mathbb{A}$, which are represented as a composition of swappings in the algorithms below. 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 $(|dom(\pi)| - 1)$ swappings. Composition $\pi_1 \circ \pi_2$ and inverse π^{-1} can be immediately computed. Permutations π operate on expressions simply by recursing on the structure. For a letrec-expression this is $\pi \cdot (\texttt{letrec}\ a_1.s_1; \ldots; a_n.s_n\ \texttt{in}\ e) = (\texttt{letrec}\ \pi \cdot a_1.\pi \cdot s_1; \ldots; \pi \cdot a_n.\pi \cdot s_n\ \texttt{in}\ \pi \cdot e)$. Note that permutations also change names of bound atoms.

We will use the following definition of α-equivalence:

Definition 3.1. *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 $(fe_1 \ldots e_n) \sim (fe'_1 \ldots e'_n)$ for an n-ary $f \in \mathcal{F}$.*
- *If $e \sim e'$, then $\lambda a.e \sim \lambda a.e'$.*
- *If $a\#e'$ and $e \sim (a\ b) \cdot e'$, then $\lambda a.e \sim \lambda b.e'$.*
- *If there is some permutation ρ on $\{1, \ldots, n\}$, such that* $\lambda a_1. \ldots .\lambda a_n.(e_1, \ldots, e_n, e_0) \sim \lambda a'_{\rho(1)}. \ldots .\lambda a'_{\rho(n)}.(e'_{\rho(1)}, \ldots, e'_{\rho(n)}, e'_0)$ *implies* $(\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e_0) \sim (\texttt{letrec}\ a'_1.e'_1; \ldots; a'_n.e'_n\ \texttt{in}\ e'_0)$.

Note that $\sim$ is identical to the equivalence relation generated by α-equivalence of binding constructs and permutation of bindings in a letrec. Note also that $e_1 \sim e_2$ is equivalent to $\pi{\cdot}e_1 \sim \pi{\cdot}e_2$, which will be implicitly used in the arguments below.

We need fixpoint sets of permutations π: We define $Fix(\pi) = \{e \mid \pi \cdot e \sim e\}$. In usual nominal unification, these sets can be characterized by using freshness constraints [46]. Clearly, all these sets and also all finite intersections are nonempty, since at least fresh atoms are elements and since $\mathbb{A}$ is infinite. However, in our setting, these sets are nontrivial:

Example 3.2. The α-equivalence $(a\ b) \cdot (\texttt{letrec}\ c.a; d.b\ \texttt{in}\ True) \sim (\texttt{letrec}\ c.a; d.b\ \texttt{in}\ True)$ holds, which means that there are expressions t in *LRL* with $t \sim (a\ b) \cdot t$ and $FA(t) = \{a, b\}$. This is in contrast to usual nominal unification.

Below we will use the results on complexity of operations in permutation groups, see [28, 17]. We consider a set $\{a_1, \ldots, a_n\}$ of distinct objects (in our case the atoms), the symmetric group $\Sigma(\{a_1, \ldots, a_n\})$ (of size $n!$) of permutations of the objects, and its elements, subsets and subgroups. Subgroups are always represented by a set of generators (represented as permutations on $\{a_1, \ldots, a_n\}$). If H is a set of elements (or generators), then $\langle H \rangle$ denotes the generated subgroup. Some facts are:

- A permutation can be represented in space linear in n.

– Every subgroup of $\Sigma(\{a_1,\ldots,a_n\})$ can be represented by $\leq n^2$ generators.

However, elements in a subgroup may not be representable as a product of polynomially many generators.

The following questions can be answered in polynomial time:

– 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, since those problems are as hard as graph-isomorphism (see [28]).

4 A Nominal Letrec Unification Algorithm

As an extension of *LRL*, there is a countably infinite set of (unification) variables X, Y also denoted perhaps using indices. The syntax of the language *LRLX* (**L**et**R**ec **L**anguage e**X**tended) is

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

Var is the set of variables and $Var(e)$ is the set of variables X occurring in e.

The expression $\pi{\cdot}e$ for a non-variable e means an operation, which is performed by shifting π down, using the simplification $\pi_1{\cdot}(\pi_2{\cdot}X) \to (\pi_1\circ\pi_2){\cdot}X$, apply it to atoms, where only expressions $\pi\cdot X$ remain, which are called *suspensions*. A *freshness constraint* in our unification algorithm is of the form $a\#e$, where e is an *LRLX*-expression, and an *atomic* freshness constraint is of the form $a\#X$.

Definition 4.1 (Simplification of Freshness Constraints).

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

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

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

Definition 4.2. *An LRLX-*unification problem *is a pair (Γ,∇), where Γ is a set of equations $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, and ∇ is a set of freshness constraints $\{a_1\#e_1,\ldots,a_m\#e_m\}$.*
A (ground) solution *of (Γ,∇) is a substitution ρ (mapping variables in $Var(\Gamma,\nabla)$ to ground expressions), such that $s_i\rho \sim t_i\rho$, for $i=1,\ldots,n$, and $a_j\#(e_j\rho)$, for $j=1,\ldots,m$.*
The decision problem is whether there is a solution for a given (Γ,∇).

Definition 4.3. *Let (Γ,∇) be an LRLX-unification problem. We consider triples $(\sigma,\nabla',\mathcal{X})$, where σ is a substitution (compressed as a dag) mapping variables to LRLX-expressions, ∇' is a set of freshness constraints, and $\mathcal{X}$ is a set of fixpoint constraints of the form $X\in Fix(\pi)$, where $X\notin dom(\sigma)$.*
A triple $(\sigma,\nabla',\mathcal{X})$ is a unifier *of (Γ,∇), if*

(i) there exists a ground substitution ρ that solves $(\nabla'\sigma,\mathcal{X})$, i.e., for every $a\#e$ in ∇', $a\#e\sigma\rho$ is valid, and for every constraint $X\in Fix(\pi)$ in $\mathcal{X}$, $X\rho\in Fix(\pi)$; and
(ii) for every ground substitution ρ that instantiates all variables in $Var(\Gamma,\nabla)$ which solves $(\nabla'\sigma,\mathcal{X})$, the ground substitution $\sigma\rho$ is a solution of (Γ,∇).

A set M of unifiers is complete*, if every solution μ is covered by at least one unifier, i.e. there is some unifier $(\sigma,\nabla',\mathcal{X})$ in M, and a ground substitution ρ, such that $X\mu\sim X\sigma\rho$ for all $X\in Var(\Gamma,\nabla)$.*

$$(1)\ \frac{\varGamma\dot\cup\{e \doteq e\}}{\varGamma} \qquad (2)\ \frac{\varGamma\dot\cup\{\pi\cdot X \doteq s\}\quad s \notin \mathit{Var} \text{ and } \pi \neq \emptyset}{\varGamma \cup \{X \doteq \pi^{-1}\cdot s\}}$$

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

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

$$(7)\ \frac{\varGamma\dot\cup\{\texttt{letrec}\ a_1.s_1;\dots;a_n.s_n\ \texttt{in}\ r \doteq \texttt{letrec}\ b_1.t_1;\dots;b_n.t_n\ \texttt{in}\ r'\}, \nabla}{\Big|_{\forall\rho} \left(\begin{array}{l} \varGamma \cup \left\{\begin{array}{c}\text{decompose}(n+1, \lambda a_1 \dots \lambda a_n.(s_1,\dots,s_n,r)\\ \doteq \lambda b_{\rho(1)}.\dots.\lambda b_{\rho(n)}.(t_{\rho(1)},\dots,t_{\rho(n)},r'))\end{array}\right\}, \\ \nabla \cup \left\{\begin{array}{c}\text{decompfresh}(n+1, \lambda a_1 \dots \lambda a_n.(s_1,\dots,s_n,r)\\ \doteq \lambda b_{\rho(1)}.\dots.\lambda b_{\rho(n)}.(t_{\rho(1)},\dots,t_{\rho(n)},r'))\end{array}\right\}\end{array}\right)}$$

where ρ is a permutation on $\{1,\dots,n\}$ and decompose$(n,.)$ is the equation part of n-fold application of rules (5) or (6) and decomposefresh$(n,.)$ is the freshness constraint part of the n-fold application of rules (5) or (6).

Fig. 1. Standard (1,2,3) and decomposition rules (4,5,6,7)

(MMS) $\dfrac{\varGamma\dot\cup\{X \doteq e_1, X \doteq e_2\}, \nabla}{\varGamma \cup \{X \doteq e_1\} \cup \varGamma', \nabla \cup \nabla'}$, if e_1, e_2 are neither variables nor suspensions. where $\varGamma'$ is the set of equations generated by decomposing $e_1 \doteq e_2$ using (4)–(7), and where ∇' is the set of freshness constraints generated during decomposing $e_1 \doteq e_2$ using (4)–(7).

(FPS) $\dfrac{\varGamma\dot\cup\{X \doteq \pi_1{\cdot}X, \dots, X \doteq \pi_n{\cdot}X, X \doteq e\}, \theta}{\varGamma \cup \{e \doteq \pi_1{\cdot}e, \dots, e \doteq \pi_n{\cdot}e\}, \theta \cup \{X \mapsto e\}}$, If $X \notin \mathit{Var}(\varGamma, e)$, and e is neither a variable nor a suspension, and no failure rule (see below) is applicable.

(ElimFP) $\dfrac{\varGamma\dot\cup\{X \doteq \pi_1{\cdot}X, \dots, X \doteq \pi_n{\cdot}X, X \doteq \pi{\cdot}X\}, \theta}{\varGamma \cup \{X \doteq \pi_1{\cdot}X, \dots, X \doteq \pi_n{\cdot}X\}, \theta}$, if $\pi \in \langle\pi_1, \dots, \pi_n\rangle$.

(Output) $\dfrac{\varGamma, \nabla, \theta}{\theta, \nabla, \{\text{“}X \in \mathit{Fix}(\pi)\text{”} \mid X \doteq \pi \cdot X \in \varGamma\}}$ if $\varGamma$ only consists of fixpoint-equations.

Fig. 2. Main Rules of LetrecUnify

We will employ nondeterministic rule-based algorithms computing unifiers: There is a clearly indicated disjunctive (don't know non-deterministic) rule, all other rules are don't care non-deterministic. The *collecting variant* of the algorithm runs and collects all solutions from all alternatives of the disjunctive rule(s). The *decision variant* guesses one possibility and tries to compute a single unifier.

Since we want to avoid the exponential size explosion of the Robinson-style unification, keeping the good properties of Martelli Montanari-style algorithms [30], 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: $(f\ t_1 \dots t_n) \to (f\ X_1 \dots X_n)$ plus the equations $X_1 \doteq t_1, \dots, X_n \doteq t_n$. Also $\lambda a.s$ is replaced by $\lambda a.X$ with equation $X \doteq s$, and $(\texttt{letrec}\ a_1.s_1; \dots, a_n.s_n\ \texttt{in}\ r)$ is replaced by $(\texttt{letrec}\ a_1.X_1; \dots, a_n.X_n\ \texttt{in}\ X)$ with the additional equations $X_1 \doteq s_1; \dots; X_n \doteq s_n; X \doteq r$. The introduced variables are fresh ones. Thus, all expressions in equations are of depth at most 1, not counting the permutation applications in the suspensions.

4.1 Rules of the Algorithm LetrecUnify

The top symbol of an expression is defined as $\mathit{tops}(X) = X$, $\mathit{tops}(\pi{\cdot}X) = X$, $\mathit{tops}(f\ s_1 \dots s_n) = f$, $\mathit{tops}(a) = a$, $\mathit{tops}(\lambda a.s) = \lambda$, $\mathit{tops}(\texttt{letrec}\ \mathit{env}\ \texttt{in}\ s) = \texttt{letrec}$. Let $\mathcal{F}^x := \mathcal{F} \cup \mathbb{A} \cup \{\texttt{letrec}, \lambda\}$.

Definition 4.4. *The rule-based algorithm* LetrecUnify *is defined in the following. Its rules are in Figs. 1, 2 and 3.* LetrecUnify *operates on a tuple $(\varGamma, \nabla, \theta)$, where $\varGamma$ 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 substitution as a list of replacements of the form $X \mapsto e$. Initially θ is empty.*

Clash Failure: If $s \doteq t \in \Gamma$, $tops(s) \in \mathcal{F}^x$, $tops(t) \in \mathcal{F}^x$, but $tops(s) \neq tops(t)$.
Cycle Detection: If there are equations $X_1 \doteq s_1, \ldots, X_n \doteq s_n$ where $tops(s_i) \in \mathcal{F}^x$, 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 $a\#a$.
Freshness Solution Fail: If there is a freshness constraint $a\#X \in \nabla$, and not $a\#((X)\theta)$.

Fig. 3. Failure Rules of LetrecUnify

The final state will be reached, i.e. the output, when Γ only contains fixpoint equations of the form $X \doteq \pi{\cdot}X$ that are non-redundant, and the rule (Output) fires.

The rules (1)–(7), and (ElimFP) have highest priority; then (MMS) and (FPS). The rule (Output) terminates an execution on Γ_0 by outputting a unifier $(\theta, \nabla', \mathcal{X})$.

We assume that the algorithm LetrecUnify *stops if a failure rule is applicable.*

We can also apply LetrecUnify to arbitrary input equations by first flattening them, which can be performed in polynomial time.

Note that the two rules (MMS) and (FPS), without further precaution, may cause an exponential blow-up in the number of fixpoint-equations (see Example 4.6). The rule (ElimFP) will limit the number of fixpoint equations by exploiting knowledge on operations on permutation groups.

Note that in any case at least one solution is represented.

The computation of $FA((X)\theta)$ can be done in polynomial time by iterating over the solution components.

In the notation of the rules, we use $[e/X]$ as substitution that replaces X by e. In the rules, we may omit ∇ or θ if they are not changed. We will use a notation "|" in the consequence part of one rule, perhaps with a set of possibilities, to denote disjunctive (i.e. don't know) nondeterminism. The only nondeterministic rule that requires exploring all alternatives is rule (7). The other rules can be applied in any order, where it is not necessary to explore alternatives.

Example 4.5. We illustrate the letrec-rule by a ground example 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 identity permutation ρ, which results in: $\lambda a.\lambda b.((a,b),(a,b),b) \doteq \lambda b.\lambda c.((b,c),(b,c),c)$. Decomposition yields: $\lambda b.((a,b),(a,b),b) \doteq (a\ b){\cdot}\lambda c.((b,c),(b,c),c) = \lambda c.((a,c),(a,c),c)$. (The freshness constraint $a\#\ldots$ holds). Then decomposing using the λ-rule gives $((a,b),(a,b),b) \doteq (b\ c){\cdot}((a,c),(a,c),c)$ (the freshness constraint $b\#\ldots$ holds). The resulting equation is $((a,b),(a,b),b) \doteq ((a,b),(a,b),b)$, which is valid.

Example 4.6. This example shows that FPS (together with the standard and decomposition rules) may give rise to an exponential number of equations in the size of the original problem. Let there be variables $X_i, i = 0, \ldots, n$ and the equations $\Gamma = \{X_n \doteq \pi{\cdot}X_n, X_n \doteq (f\ X_{n-1}\ \rho_n{\cdot}X_{n-1}), \ldots, X_2 \doteq (f\ X_1\ \rho_2{\cdot}X_1)\}$ where $\pi, \rho_1, \ldots, \rho_n$ are permutations. We prove that this unification problem may give rise to 2^{n-1} equations, if the redundancy rule (ElimFP) is not there.

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

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

After (FPS): $$\left\{\begin{array}{r} (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}) \doteq \pi{\cdot}(f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \\ (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}) \doteq \rho_n^{-1}{\cdot}\pi{\cdot}\rho_n{\cdot}(f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \\ X_{n-2} \doteq (f\ X_{n-3}\ \rho_{n-2}{\cdot}X_{n-3}), \ldots \end{array}\right\}$$

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

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 $\rho_n, \rho_{n-1}, \ldots, \rho_2\}$, 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.

5 Soundness, Completeness, and Complexity of LetrecUnify

Theorem 5.1. *The decision variant of the algorithm* LetrecUnify *runs in nondeterministic polynomial time. Its collecting version returns a complete set of at most exponentially many unifiers, every one represented in polynomial space. The number of rule applications is $O(S^3 \log(S))$ where S is the size of the input.*

Proof. Let Γ_0, ∇_0 be the input, and let $S = \mathit{size}(\Gamma_0, \nabla_0)$. The execution of a single rule can be done in polynomial time depending on the size of the intermediate state, thus we have to show that the size of the intermediate states remains polynomial and that the number of rule applications is at most polynomial.

The number of fixpoint-equations for every variable X is at most $S * \log(S)$ since the number of atoms is never increased, and since we assume that (ElimFP) is applied whenever possible. The size of the permutation group is at most $S!$, and so the length of proper subset-chains and hence the maximal number of generators of a subgroup is at most $\log(S!) \leq S*\log(S)$. Note that the redundancy of generators can be tested in polynomial time depending on the number of atoms. Note also that applicability of (ElimFP) can be tested in polynomial time by checking the maximal possible subsets.

The lexicographically ordered termination measure $(\#\mathrm{Var}, \#\mathrm{Lr}\lambda\mathrm{FA}, \#\mathrm{Eqs}, \#\mathrm{EqNonX})$ is:
#Var is the number of different variables in Γ,
#LrλFA is the number of letrec-, λ, function-symbols and atoms in Γ, but not in permutations,
#Eqs is the number of equations in Γ, and
#EqNonX is the number of equations where none of the equated expressions is a variable.

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. The following table shows the effect of the rules: Let S be the size of the initial (Γ_0, ∇_0) where Γ is already flattened.

The entries $+W$ represents a size increase of at most W in the relevant measure component.

	#Var	#LrλFA	#Eqs	#EqNonX
(3)	$<$	$\leq$	$<$	$\leq$
(FPS)	$<$	$+2S\log(S)$	$<$	$+S\log(S)$
(MMS)	$=$	$<$	$+2S$	$=$
(4), (5), (6), (7)	$=$	$<$	$+S$	$\leq$
(ElimFP)	$=$	$=$	$<$	$\leq$
(1)	$\leq$	$\leq$	$<$	$\leq$
(2)	$=$	$=$	$=$	$<$

The table shows that the rule applications strictly decrease the measure. The entries can be verified by checking the rules, and using the argument that there are not more than $S\log(S)$ fixpoint equations for a single variable X. We use the table to argue on the number of rule applications and hence the complexity: The rules (3) and (FPS) strictly reduce the number of variables in Γ and can be applied at most S times. (FPS) increases the second measure at most by $2 * S\log(S)$, since the number of symbols may be increased as often as there are fixpoint-equations and there are at most $S\log(S)$. Because no other rule increases the measure #LrλFA will never be greater than $2S^2\log(S)$. The rule (MMS) strictly decreases #LrλFA. Hence#Eqs, i.e. the number of equations is bounded by $4S^3\log(S)$. The same bound holds for #EqNonX. Thus, the number of rule applications is $O(S^3\log(S))$.

Theorem 5.2. *The algorithm* LetrecUnify *is sound and complete.*

Proof. Soundness of the algorithm holds, by easy arguments for every rule, similar as in [46], and since the letrec-rule follows the definition of $\sim$ in Def. 3.1. 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 rule (7)), retain the set of solutions. The same for (MMS), (FPS), and (ElimFP). Note that the nondeterminism in (ElimFP) does not affect completeness. The nondeterministic rule (7) provides all possibilities for potential ground solutions. Moreover, the failure rules are not applicable to states that are solvable.

A final output of LetrecUnify 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 5.3. *The nominal letrec-unification problem is in NP.*

Proof. This follows from Theorems 5.1 and 5.2.

6 Nominal Matching with Letrec: LetrecMatch

Reductions in higher order calculi with letrec, in particular on a meta-notation, require a matching algorithm, matching its left hand side to an expression.

Example 6.1. Consider the (lbeta)-rule, which is the version of (beta) used in call-by-need calculi with sharing [2, 32, 42]. Note that only the sharing power of the recursive environment is used here.

$$(\mathit{lbeta}) \quad (\lambda x.e_1)\ e_2 \rightarrow\ \texttt{letrec}\ x.e_2\ \texttt{in}\ e_1.$$

An (lbeta) step, for example, on $(\lambda x.x)\ (\lambda y.y)$ is performed by switching to the language LRL and then matching $(\mathit{app}\ (\lambda c.X_1)\ X_2) \trianglelefteq (\mathit{app}\ (\lambda a.a)\ (\lambda b.b))$, where app is the explicit representation of the binary application operator. This results in $\sigma := \{X_1 \mapsto c; X_2 \mapsto (\lambda b.b)\}$, and the reduction result is the σ-instance of ($\texttt{letrec}\ c.X_2\ \texttt{in}\ X_1$), which is ($\texttt{letrec}\ c.(\lambda b.b)\ \texttt{in}\ c$). Note that this form of reduction implicitly uses α-equivalence.

We derive a nominal letrec matching algorithm as a specialization of LetrecUnify. We use non-symmetric equations written $s \trianglelefteq t$, where s is an $LRLX$-expression, and t does not contain variables. Note that neither freshness constraints nor suspensions are necessary (and hence no fixpoint equations) in the solution. We assume that the input is a set of equations of expressions.

The rules of the algorithm LetrecMatch are:

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

$$\frac{\Gamma \dot\cup \{\lambda a.s \trianglelefteq \lambda b.t\}}{\Gamma \cup \{s \trianglelefteq (a\ b){\cdot}t\}} \ \text{if } a\#t\text{, otherwise Fail.} \qquad \frac{\Gamma \dot\cup \{\pi{\cdot}X \trianglelefteq e\}}{\Gamma \cup \{X \trianglelefteq \pi^{-1}{\cdot}e\}}$$

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

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

$$\frac{\mathit{\Gamma}\dot{\cup}\{X \trianglelefteq e_1, X \trianglelefteq e_2\}}{\mathit{\Gamma}\cup\{X \trianglelefteq e_1\}}\ \text{if } e_1 \sim e_2\text{, otherwise Fail}$$

The test $e_1 \sim e_2$ will be performed as a subroutine call to this (nondeterministic) matching procedure in the collecting version, i.e. the test succeeds if there is a nondeterministic execution with success as result.

Clash Failure: if $s \doteq t \in \mathit{\Gamma}$, $tops(s) \in \mathcal{F}^x, tops(t) \in \mathcal{F}^x$, but $tops(s) \neq tops(t)$.

Standard arguments show:

Theorem 6.2. LetrecMatch *is sound and complete for nominal letrec matching. It decides nominal letrec matching in nondeterministic polynomial time. Its collecting version returns a finite complete set of an at most exponential number of matching substitutions, which are of at most polynomial size.*

Theorem 6.3. *Nominal letrec matching is NP-complete.*

Proof. The problem is in NP, which follows from Theorem 6.2. It is also NP-hard, which follows from the (independent) Theorem 7.1.

A slightly more general situation for nominal letrec matching occurs, when the matching equations $\mathit{\Gamma}_0$ are compressed using a dag. We construct a practically more efficient algorithm LetrecDagMatch from LetrecUnify as follows. First we generate $\mathit{\Gamma}_1$ from $\mathit{\Gamma}_0$, which only contains flattened expressions by encoding the dag-nodes as variables together with a unification equation. An expression is said $\mathit{\Gamma}_0$-ground, if it does not reference variables from $\mathit{\Gamma}_0$ (also via equations). 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 :

$$\frac{\mathit{\Gamma}\dot{\cup}(\lambda a.s \doteq \lambda b.t\}, \nabla}{\mathit{\Gamma}\cup\{s \doteq (a\ b)\cdot t\}, \nabla\cup\{a\#t\}}\qquad \lambda b.t \text{ is } \mathit{\Gamma}_0\text{-ground}$$

The extra conditions $a\#t$ and $\mathit{\Gamma}_0$-ground can be tested in polynomial time. The equations $\mathit{\Gamma}_1$ 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 $\mathit{\Gamma}_0$. Since $\mathit{\Gamma}_0$ 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 6.4. *The collecting variant of* LetrecDagMatch *outputs an at most exponential set of dag-compressed substitutions that is complete, where every unifier is represented in polynomial space.*

7 Hardness of Nominal Letrec Matching and Unification

Theorem 7.1. *Nominal letrec matching (hence also unification) is NP-hard, for two letrec expressions, where subexpressions are free of letrec.*

Proof. We encode the NP-hard problem of finding a Hamiltonian cycle in a regular graph [33, 18], which are graphs where all nodes have the same degree $k = 3$. 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 $b.(f\ a\ a')$ and $b'.(f\ a'\ a)$ for every edge

$(a, a') \in E$ for fresh names b, b'. Then let $t := (\texttt{letrec}\ env_1; env_2\ \texttt{in}\ 0)$ 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 $b_1.f\ X_1\ X_2; b_2.f\ X_2\ X_3; \ldots; b_n.f\ X_n\ X_1$, where all b_i are different atoms, and the third part env'_3 consists of a number of (dummy) entries of the form $b.f\ Z\ Z'$, where b is always a fresh atom for every binding, and Z, Z' are fresh variables for every entry. The number of these dummy entries is $k * n - n$. Let $s := (\texttt{letrec}\ env'_1; env'_2; env'_3\ \texttt{in}\ 0)$, representing the question of the Hamiltonian cycle existence. Then the matching problem $s \trianglelefteq t$ is solvable iff the graph has a Hamiltonian cycle. The degree is 3, hence it is not possible that there shortcuts in the cycle.

Theorem 7.2. *The nominal letrec-unification problem is NP-complete.*

Proof. This follows from Theorems 5.3 and 7.1.

We say that an expression t *contains garbage*, iff there is a subexpression $(\texttt{letrec}\ env\ \texttt{in}\ r)$, and the environment env can be split into two environments $env = env_1; env_2$, such that env_1 is not trivial, and the atoms from $LA(env_1)$ occur neither in env_2 nor in r. Otherwise, the expression is *free of garbage*. Since α-equivalence of LRL-expressions is Graph-Isomorphism-complete [38], but α-equivalence of garbage-free LRL-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 have complexity between $PTIME$ and NP; there are arguments that it is weaker than the class of NP-complete problems [43]. There is also a claim that it is quasi-polynomial [10], which means that it requires less than exponential time.

Theorem 7.3. *Nominal letrec matching with one occurrence of a single variable and a garbage-free target expression is Graph-Isomorphism-hard.*

Proof. Let G_1, G_2 be two regular graphs with degree ≥ 1. Let t be $(\texttt{letrec}\ env_1\ \texttt{in}\ g\ b_1 \ldots, b_m)$ the encoding of an arbitrary graph G_1 where env_1 is the encoding as in the proof of Theorem 7.1, nodes are encoded as $a_1 \ldots a_n$, and the edge-binders are b_i. Then t is free of garbage. Let the environment env_2 be the encoding of G_2 in $s = (\texttt{letrec}\ env_2\ \texttt{in}\ X)$. Then s matches t iff the graphs G_1, G_2 are isomorphic. Since the graph-isomorphism problem for regular graphs of degree ≥ 1 is GI-hard [12], 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 $(g\ b_1 \ldots, b_m)$.

8 Nominal Letrec Matching with Environment Variables

We extend the language LRLX by variables E that may encode partial letrec-environments, which leads to a larger coverage of unification problems in reasoning about the semantics of programming languages.

Example 8.1. Consider as an example a rule (llet-e) of the operational semantics, that merges letrec-environments (see [42]): $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ X)) \ \rightarrow\ (\texttt{letrec}\ E_1; E_2\ \texttt{in}\ X)$. It 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 $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ X))$ 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: $\{E_1 \mapsto \{a.0; b.1\}; E_2 \mapsto \{c.(a, b, c)\}; X \mapsto c\}$, producing the next expression as an instance of the right hand side $(\texttt{letrec}\ E_1; E_2\ \texttt{in}\ X)$, which is $(\texttt{letrec}\ a.0; b.1; c.(a, b, c)\ \texttt{in}\ c)$. Note that for application to extended lambda calculi, more care is needed w.r.t. scoping in order to get valid reduction results in all cases. The restriction that a single letrec environment binds different variables becomes more important. The reduction (llet-e) is correctly applicable, if the target expression satisfies the so-called distinct variable convention, i.e., if all bound variables are different and that all free variables in the expression are different from all bound variables.

An alternative that is used for a similar unification task in [39] requires an additional construct of non-capture constraints: $NCC(env_1, env_2)$, which means that for every valid instantiation ρ, variables occurring free in $env_1\rho$ are not captured by the top letrec-binders in $env_2\rho$. In this paper we focus on matching, and leave the combination with reduction rules for further work.

Definition 8.2. *The grammar for the extended language LRLXE (***L***et***R***ec* **L***anguage e***X***tended with* **E***nvironments) is:*

$$\begin{array}{lcl} env & ::= & E \mid \pi \cdot E \mid a.e \mid env; env \\ e & ::= & a \mid X \mid \pi \cdot X \mid \lambda a.e \mid (f\ e_1\ \ldots e_{ar(c)}) \mid (\texttt{letrec}\ env\ \texttt{in}\ e) \end{array}$$

We define a matching algorithm, where environment variables may occur in left hand sides. It needs a more expressive data structure in equations. The variant `letr*` of `letrec` is used with two environment-components: (i) a list of bindings that are already fixed in the correspondence to the bindings of the other environment, and (ii) an environment that is not yet fixed. We denote the fixed bindings as a list, which is the always the first component. The scoping is the same. 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 data structure is the scoping in letrec expressions. The initial `letrec`-expression is represented as an expression `letr`$*$ with an empty list as first component, and the environment as the second component. We assume that there are no environment-variable suspensions. This is appropriate, since the algorithm also does not generate these suspensions.

Definition 8.3. *The matching algorithm* LetrecEnvMatch *for expressions where environment variables E and expression variables X may occur only in the left hand sides of match equations is described below. We assume that there are no suspension of environment variables.*
Initially, every (`letrec` *env* `in` *e*) *is modified to* (`letr`$*$ $\emptyset$; *env* `in` *e*). *The don't know-non-determinism is indicated in the rules, and additionally there is a choice between the two alternatives of the environment-match rule.*
The results are either match-equations $x \trianglelefteq e$, $E \trianglelefteq \emptyset$, or $E \trianglelefteq a.e; E'$, where we indicate the latter as part of the solution by writing $E \mapsto a.e; E'$.
The rules are:

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

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

$$\frac{\Gamma \dot\cup \{(\texttt{letr}*\ ls; a.s; env\ \texttt{in}\ r) \trianglelefteq (\texttt{letr}*\ ls'; b.t; env'\ \texttt{in}\ r')\}}{\mid_{\forall (b.t)}\ \Gamma \cup \{(\texttt{letr}*\ ((a.s) : ls); env\ \texttt{in}\ r) \trianglelefteq (a\ b)(\texttt{letr}*\ ((b.t) : ls');\ env'\ \texttt{in}\ r')\}}$$

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

$$\frac{\Gamma \dot\cup \{(\texttt{letr}*\ ls; E; env\ \texttt{in}\ r) \trianglelefteq (\texttt{letr}*\ ls'; b.t; env'\ \texttt{in}\ r')\}}{\mid_{\forall (b.t)}\ \Gamma \cup \{(\texttt{letr}*\ ((a.X) : ls); E'; env\ \texttt{in}\ r) \trianglelefteq (a\ b){\cdot}(\texttt{letr}*\ ((b.t) : ls');\ env'\ \texttt{in}\ r')\} \cup \{E \mapsto a.X; E'\}}$$

If $a\#(\texttt{letr}*\ ls'; b.t; env'\ \texttt{in}\ r')$.
Where a is guessed as an atom occuring in the problem, or a fresh one, such that $a \notin LA(env)$.

$$\frac{\Gamma \dot\cup \{(\texttt{letr}*\ ls; E; env\ \texttt{in}\ r) \trianglelefteq (\texttt{letr}*\ ls'; env'\ \texttt{in}\ r')\}}{\Gamma \cup \{(\texttt{letr}*\ ls; env\ \texttt{in}\ r) \trianglelefteq (\texttt{letr}*\ ls'; env'\ \texttt{in}\ r')\} \cup \{E \trianglelefteq \emptyset\}}$$

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

$$\frac{\Gamma \dot\cup \{\pi{\cdot}X \trianglelefteq e\}}{\Gamma \cup \{X \trianglelefteq \pi^{-1}e\}} \qquad \frac{\Gamma \dot\cup \{X \trianglelefteq e_1, X \trianglelefteq e_2\}}{\Gamma \cup \{X \trianglelefteq e_1\}} \quad \textit{if } e_1 \sim e_2 \textit{ holds}$$

$$\frac{\Gamma \dot\cup \{E \trianglelefteq env_1, E \trianglelefteq env_2\}}{\Gamma \cup \{E \trianglelefteq env_1\}} \quad \textit{if } env_1 \sim env_2 \textit{ holds.}$$

Clash Failure: *If $s \trianglelefteq t \in \Gamma$, $tops(s) \in \mathcal{F}^x$, $tops(t) \in \mathcal{F}^x$, but $tops(s) \neq tops(t)$.*

After successful execution, the result will be a set of match equations with components $X \trianglelefteq e$, and $E \trianglelefteq env$, and $E \mapsto a.e; E'$, which represents a matching substitution, where the `letr`$*$-expressions are retranslated to `letrec`-expressions.

Example 8.4. An example illustrates the effects of iterating the guessing:
Consider $\{(\texttt{letr}*\ [];a.(1,c);b.2;c.3\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [];b.(1,a);c.2;a.3\ \texttt{in}\ 0)\}$.
It is arranged that the guesses can be made from left to right: After the first step:
$\{(\texttt{letr}*\ [a.(1,c)];b.2;c.3\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [a.(1,b)];c.2;b.3\ \texttt{in}\ 0)\}$.
After the next steps we obtain:
$\{(\texttt{letr}*\ [a.(1,c),b.2];c.3\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [a.(1,c),b.2];c.3\ \texttt{in}\ 0)\}$, and
$\{(\texttt{letr}*\ [a.(1,c),b.2;c.3];\emptyset\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [a.(1,c),b.2,c.3];\emptyset\ \texttt{in}\ 0)\}$.
Then $\{a.(1,c) \trianglelefteq a.(1,c), b.2 \trianglelefteq b.2, c.3 \trianglelefteq c.3, 0 \trianglelefteq 0\}$, and the match will succeed.

Example 8.5. Another example illustrates the effects of iterating the guessing using environment variables: Consider $\{(\texttt{letr}*\ [];E;b.2;c.(3,a)\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [];b.(1,a);c.2;a.(3,b)\ \texttt{in}\ 0)\}$. It is arranged that the guesses can be made from left to right. It is not possible to guess $E \mapsto b.X; E'$, since this would violate the binding constraints for letrec-expressions. However, we can guess E to be $a.X; E'$: $\{(\texttt{letr}*\ [a.X];E';b.2;c.(3,a)\ \texttt{in}\ 0) \trianglelefteq (\texttt{letr}*\ [a.(1,b)];c.2;b.(3,a)\ \texttt{in}\ 0), E \mapsto a.X;E'\}$. The next step is to guess that E' is empty. In particular a free atom in the left hand side can be guessed.

We claim that the algorithm is complete. The algorithm has to guess from an infinite set of fresh atoms, however, since the names of fresh atoms are irrelevant, it is sufficient to use any fresh atom. A clean mathematical treatment could be done using atom-variables using the techniques in section 10.

Theorem 8.6. *The algorithm 8.3 (*LetrecEnvMatch*) is sound and complete. It runs in non-deterministic polynomial time. The corresponding decision problem is NP-complete. The collecting version of* LetrecEnvMatch *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 keeping the scoping.

9 On Fixpoints and Garbage

We show in this section that *LRLX*-expressions without garbage have only trivial fixpointing permutations. The idea is that garbage is defined as a subset of bindings that can be removed without losing any definition of variables (atoms) in the `in`-part of the letrecs. Looking at Example 3.2, the α-equivalence $(a\ b) \cdot (\texttt{letrec}\ c.a;d.b\ \texttt{in}\ \mathit{True}) \sim (\texttt{letrec}\ c.a;d.b\ \texttt{in}\ \mathit{True})$ holds, where $dom((a\ b)) \cap FA(\texttt{letrec}\ c.a;d.b\ \texttt{in}\ \mathit{True}) = \{a,b\} \neq \emptyset$. However, we see that the complete environment in this example is garbage, from a programming language point of view.

As a helpful information, we write the α-equivalence-rule for letrec-expressions in the ground language *LRL* as an extension of the rule for lambda-abstractions.

$$\frac{r \sim \varphi'{\cdot}r',\ s_i \sim \varphi'{\cdot}t_{\rho(i)},\ i = 1,\ldots,n, \quad M\#(\texttt{letrec}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ r')}{\texttt{letrec}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ r \sim \texttt{letrec}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ r'}$$

where ρ is a permutation on $\{1,\ldots,n\}$, $M = \{a_1,\ldots,a_n\} \setminus \{b_1,\ldots,b_n\}$, and φ is a smallest atom-permutation-extension of the bijective function $\{b_i \mapsto a_{\rho(i)}, i = 1,\ldots,n\}$ such that $dom(\varphi) \subseteq (\{b_1,\ldots,b_n\} \cup \{a_1,\ldots,a_n\})$, and $\varphi(M) = (\{b_1,\ldots,b_n\} \setminus \{a_1,\ldots,a_n\})$.

Note that α-equivalence of s, t means structural equivalence of s, t as trees, and a justification always comes with a bijective relation between the positions of s, t where only the names of atoms at nodes may be different.

Definition 9.1. *An expression* ($\texttt{letrec}\ E\ \texttt{in}\ e$) *contains garbage, if E is not empty and there is a nonempty subset of bindings $E' \subseteq E$, such that $FA(\texttt{letrec}\ E'\ \texttt{in}\ e) = FA(\texttt{letrec}\ E\ \texttt{in}\ e)$.*

An example is ($\texttt{letrec}\ a.0; b.1\ \texttt{in}\ (f\ \ b)$), where a is unused, but b is used in the right hand side. In this case we say, $a.0$ is garbage. Another example is ($\texttt{letrec}\ a.d; b.1; c.d\ \texttt{in}\ (f\ \ b)$), which is an example with a free atom d, and two bindings, $a.d$ and $c.d$ which is α-equivalent to ($\texttt{letrec}\ a'.d; b.1; c'.d\ \texttt{in}\ (f\ \ b)) =: e$. Note that in this case, there are two different bijective functions (modulo αequivalence) on e: $\{a' \mapsto a; c' \mapsto c\}$ and $\{a' \mapsto c; c' \mapsto a\}$.

The next lemma shows that this situation is only possible if the expressions contain garbage.

Lemma 9.2. *If $s \sim t$, and s is free of garbage, then α-equivalence provides a unique correspondence of the positions of s and t.*

Proof. The proof is by induction on the structure and size of expressions. For the structure, the only nontrivial case is letrec: If $s = (\texttt{letrec}\ a_1.e_1, \dots, a_n.e_n\ \texttt{in}\ e) \sim (\texttt{letrec}\ b_1.f_1, \dots, b_n.f_n\ \texttt{in}\ f) = t$, then there is bijective mapping φ, with $\varphi(b_i) = a_{\rho(i)}, i = 1, \dots, n$, where ρ is a permutation on $\{1, \dots, n\}$, and such that $e_i \sim \varphi(f_{\rho(i)}), i = 1, \dots, n$, $e \sim \varphi(f)$, and $(\{a_1, \dots, a_n\} \setminus \{b_1, \dots, b_n\})\#t$ holds. Let $\overline{\varphi}$ be the atom-permutation that extends φ, mapping $(\{a_1, \dots, a_n\} \setminus \{b_1, \dots, b_n\})$ to $(\{b_1, \dots, b_n\} \setminus \{a_1, \dots, a_n\})$.

The induction hypothesis implies a unique position correspondence of e and f, since $e \sim \overline{\varphi}(f)$. This implies that the bindings for $\{a_1, \dots, a_n\} \cap FA(e)$ have a unique correspondence to the bindings in t. This is continued by exhaustively following free occurrences of atoms a_i in the right hand sides of the top bindings in s. Since there is no garbage in s, all bindings can be reached by this process, hence we have uniqueness of the correspondence of positions.

Proposition 9.3. *Let e be an expression that does not have garbage, and let π be a permutation. Then $\pi{\cdot}e \sim e$ implies $dom(\pi) \cap FA(e) = \emptyset$.*

Proof. The proof is by induction on the size of the expression.

- If e is an atom, then this is trivial.
- If $e = f\ e_1 \dots . e_n$, then no e_i contains garbage, and $\pi{\cdot}e_i \sim e_i$ implies $dom(\pi) \cap FA(e_i) = \emptyset$, hence also $dom(\pi) \cap FA(e) = \emptyset$.
- If $e = \lambda a.e'$, then there are two cases:
  1. $\pi(a) = a$. Then $\pi{\cdot}e' \sim e'$, and we can apply the induction hypothesis.
  2. $\pi(a) = b \neq a$. Then $(a\ b){\cdot}\pi$ fixes e', and $b\#e'$. The induction hypothesis implies $dom((a\ b){\cdot}\pi) \cap FA(e') = \emptyset$. We have $dom(\pi) \subseteq dom((a\ b){\cdot}\pi)) \cup \{a, b\}$, hence $dom(\pi) \cap FA(\lambda a.e') = \emptyset$.
- First a simple case with one binding in the environment: $t = (\texttt{letrec}\ a_1.e_1\ \texttt{in}\ e)$, $\pi{\cdot}t \sim t$. If $\pi(a_1) = a_1$, then $\pi{\cdot}(e, e_1) \sim (e, e_1)$, and the induction hypothesis implies $dom(\pi) \cap FA(e, e_1) = \emptyset$, which in turn implies $dom(\pi) \cap FA(t) = \emptyset$.
  If $\pi(a_1) = b \neq a_1$, then $b\#(e, e_1)$ and for $\pi' := (a_1\ b){\cdot}\pi$, it holds $\pi'{\cdot}(e, e_1) \sim (e, e_1)$, and so $dom((a_1\ b){\cdot}\pi) \cap FA(e, e_1) = \emptyset$. since $dom(\pi) \subseteq dom((a_1\ b){\cdot}\pi) \cup \{a_1, b\}$, we obtain $dom(\pi) \cap t = \emptyset$. In the case of one binding, it is irrelevant whether the binding is garbage or not.
- Let $t = (\texttt{letrec}\ a_1.e_1; \dots; a_n.e_n\ \texttt{in}\ e)$, and t is a fixpoint of π, i.e. $\pi(t) \sim t$. Note that no part of the environment is garbage. The permutation π can be splitted into $\pi = \pi_1{\cdot}\pi_2$, where $dom(\pi_1) \subseteq FA(t)$ and $dom(\pi_2) \cap FA(t) = \emptyset$. From $t \sim \pi{\cdot}t$ and Lemma 9.2 we obtain that there is a unique permutation ρ on $\{1, \dots, n\}$, such that there is an injective mapping $\varphi : \pi(a_1) \mapsto a_{\rho(1)}, \dots, \pi(a_n) \mapsto a_{\rho(n)}$, and $e \sim \varphi\pi(e)$, $e_{\rho(i)} \sim \varphi\pi(e_i)$. Then α-equivalence implies that $\varphi\pi$ can be extended to a atom-permutation $\overline{\varphi}\pi$ by mapping the atoms in $\{a_1, \dots, a_n\} \setminus \{\pi(a_1), \dots, \pi(a_n)\}$ bijectively to $\{\pi(a_1), \dots, \pi(a_n)\} \setminus \{a_1, \dots, a_n\}$. By the freshness constraints for α-equivalences of

letrec-expressions, $\overline{\varphi}\pi(e) = \varphi\pi(e)$ and $\overline{\varphi}\pi(e_i) = \varphi\pi(e_i)$ which in turn implies that $e \sim \overline{\varphi}\pi(e)$ and $e_i \sim \overline{\varphi}\pi(_ie)$, and we can apply the induction hypothesis.
This shows that $FA(e) \setminus \{a_1, \ldots, a_n\}$ are not moved by $\overline{\varphi}\pi$, and the same for all e_i, hence this also holds for t.

Corollary 9.4. *Let e be an expression that does not have garbage, and let π be a permutation. Then $\pi{\cdot}e \sim e$ is equivalent to $dom(\pi) \cap FA(e) = \emptyset$.*

Proof. This follows from Proposition 9.3. Note that the other direction is easy.

The proof also shows a slightly more general statement:

Corollary 9.5. *Let e be an expression such that in all environments with at least 2 bindings there are no garbage bindings, and let π be a permutation. Then $\pi{\cdot}e \sim e$ is equivalent to $dom(\pi) \cap FA(e) = \emptyset$.*

In case that the input does not represent garbage-parts, and the solutions are not intended to represent expressions with garbage, the set of rules in the case without atom-variables can be optimized as follows: (ElimFP) can be omitted and instead of (FPS) there are two rules:

$$\text{(FPS2)}\ \frac{\Gamma \mathbin{\dot\cup} \{X \doteq \pi{\cdot}X\}, \nabla}{\Gamma, \nabla \cup \{a\#X \mid a \in dom(\pi)\}},$$

$$\text{(ElimX)}\ \frac{\Gamma \mathbin{\dot\cup} \{X \doteq e\}, \theta}{\Gamma, \theta \cup \{X \mapsto e\}}, \quad \text{if } X \notin Var(\Gamma) \text{, and } e \text{ is not of the form } \pi{\cdot}X \text{ for any } \pi.$$

Example 9.6. It cannot be expected that the letrec-decomposition rule (7) can be turned into a deterministic rule, and to obtain a unitary nominal unification, under the restriction that input expressions are garbage-free, and also instantiations are garbage-free. Consider the equation:

$$(\texttt{letrec}\ a_1.e_1; a_2.e_2\ \texttt{in}\ ((a_1, a_2), X)) \doteq (\texttt{letrec}\ b_1.f_1; b_2.f_2\ \texttt{in}\ (X', (b_1, b_2))).$$

Then the in-expressions do not enforce a unique correspondence between the bindings of the left and right-hand bindings. An example also follows from the proof of Theorem 7.3, which shows that even nominal matching may have several incomparable solutions for garbage-free expressions.

10 Nominal Unification with Letrec and Atom-Variables

In this section we extend the unification algorithm to the language *LRLXA*, which is an extension of *LRLX* with atom variables. Atom-variables have a higher expressive power: For example if in an application example it is known that in a pair (x, y) there must be atoms, but $x = y$ as well as $x \neq y$ is possible, then two different unification problems have to be formulated. If atom variables are possible, then the fomulation (A_1, A_2) covers both possibilities.

It is known that the nominal unification problem (without letrec) but with atom-variables is NP-complete [41]. An algorithm and corresponding rules and discussions can be found in [41].

10.1 Extension with Atom-Variables

As an extension of *LRLX*, we define *LRLXA* as follows: Let A denote atom variables, V denote atom variables or atoms, W denote suspensions of atoms or atom variables, π a permutation, and e an expression. The syntax of the language *LRLXA* is

$$\begin{array}{lcl} V & ::= & a \mid A \\ W & ::= & \pi \cdot V \\ \pi & ::= & \emptyset \mid (W\ W) \mid \pi \circ \pi \\ e & ::= & \pi{\cdot}X \mid W \mid \lambda W.e \mid (f\ e_1\ \ldots e_{ar(f)}) \mid (\texttt{letrec}\ W_1.e_1; \ldots; W_n.e_n\ \texttt{in}\ e) \end{array}$$

Var is the set of variables and $\mathit{Var}(e)$ is the set of variables A, X occurring in e.

The expression $\pi{\cdot}e$ for a non-variable expression e means an operation, which is performed by shifting π down, removing the permutation $\emptyset$, using the simplifications $\pi_1{\cdot}(\pi_2{\cdot}X) \to (\pi_1 \circ \pi_2){\cdot}X$, and apply it to atoms, where only expressions $\pi{\cdot}X$ or $\pi{\cdot}V$ remain, which are called *suspensions* and where $\pi{\cdot}V$ is denoted as W. However, note that nested permutations are permitted, since in general, these cannot be simplified.

A *freshness constraint* in our unification algorithm is of the form $V\#e$ where e is an *LRLXA*-expression. The notation π^{-1} is defined as the reversed list of swappings of π. We also use $\pi \cdot V\#e$ as syntactic sugar for the constraint $V\#\pi^{-1} \circ e$.

Not all ground substitutions map *LRLXA*-expressions to valid *LRL* expressions. It is also not sufficient to only take 'valid' ground substitutions into account, as this leads to problems with the composability of unifications problems and renaming of bound variables within expressions, as the following example shows.

Example 10.1. The equation

$$(\mathit{app}\ (\texttt{letrec}\ A.a, B.a\ \texttt{in}\ B)\ A) \doteq (\mathit{app}\ (\texttt{letrec}\ A.a, B.a\ \texttt{in}\ B)\ B)$$

enforces that A, B are instantiated with the same atom, which contradicts the syntactic assumption on distinct atoms for the binding names in letrec-expressions. However,

$$(\mathit{app}\ (\texttt{letrec}\ A.a, C.a\ \texttt{in}\ C)\ A) \doteq (\mathit{app}\ (\texttt{letrec}\ A.a, D.a\ \texttt{in}\ D)\ B)$$

is solvable. To avoid this problem, the freshness constraints in unification problems need to ensure distinct binding variables in every `letrec`-expression in the input.

Definition 10.2. *An LRLXA-*unification problem *is a pair (Γ, ∇), where Γ is a set of equations $s \doteq t$, and ∇ is a set of freshness constraints $V\#e$. In addition, for every letrec-subexpression* $\texttt{letrec}\ W_1.e_1, \ldots, W_m.e_m\ \texttt{in}\ e$*, which occurs in Γ or ∇, the set ∇ must also contain the freshness constraint $W_i\#W_j$ for all $i, j = 1, \ldots, m$ with $i \neq j$.*

A (ground) solution *of (Γ, ∇) is a substitution ρ (mapping variables in $\mathit{Var}(\Gamma, \nabla)$ to ground expressions), such that $s\rho \sim t\rho$ for all equations $s \doteq t$ in Γ, and for all $V\#e \in \nabla$: $V\rho\#(e\rho)$ holds.*

The decision problem *is whether there is a solution for a given (Γ, ∇).*

Proposition 10.3. *The LRLXA-unification problem is in NP.*

Proof. The argument is that every ground instantiation of an atom variable is an atom, which can be guessed: guess the images of atom variables under a ground solution ρ in the set of atom variables in the current state, and in an arbitrary set of fresh atom variables of cardinality at most the number of different atom variables in the input; then instantiate using ρ, thereby removing all atom-variables. The resulting problem can be decided (and solved) by an NP-algorithm as shown in this paper (Theorem 5.1).

Remark 10.4. Note that the equation $A = \pi{\cdot}B$ for atom variables A, B can be encoded as the freshness constraint $A\#\lambda\pi{\cdot}B.A$. In the following we may use equations $A =_\# \pi{\cdot}B$ as a readable version of $A\#\lambda\pi{\cdot}B.A$.

10.2 Rules of the Algorithm LetrecUnifyAV

Now we describe the nominal unification algorithm LetrecUnifyAV for *LRLXA*. It will extend the algorithm LetrecUnify by a treatment of atom variables that extend the expressibility. It has flexible rules, such that a strategy can be added to control the non-determinism and such that it is an improvement over a brute-force guessing-algorithm (see the algorithm 10.11 for such an improvement).

$$(1)\ \frac{\mathit{\Gamma}\dot\cup\{e \doteq e\}}{\mathit{\Gamma}} \qquad (2)\ \frac{\mathit{\Gamma}\dot\cup\{\pi\cdot X \doteq e\}\quad e \notin \mathit{Var}}{\mathit{\Gamma}\cup\{X \doteq \pi^{-1}\cdot e\}}$$

$$(3)\ \frac{\mathit{\Gamma}\dot\cup\{X \doteq \pi{\cdot}Z\}, \nabla, \theta \qquad X \neq Z}{\mathit{\Gamma}[\pi{\cdot}Z/X], \nabla[\pi{\cdot}Z/X], \theta\cup\{X \mapsto \pi{\cdot}Z\}} \quad (3')\ \frac{\mathit{\Gamma}\dot\cup\{\pi{\cdot}A \doteq \pi'{\cdot}B\}, \nabla, \theta}{\mathit{\Gamma}, \nabla\cup\{A =_{\#} \pi^{-1}\pi'{\cdot}B\}, \theta}$$

$$(4)\ \frac{\mathit{\Gamma}\dot\cup(f\ (\pi_1{\cdot}X_1)\dots(\pi_n{\cdot}X_n)) \doteq (f\ (\pi_1'{\cdot}X_1')\dots(\pi_n'{\cdot}X_n'))\}}{\mathit{\Gamma}\cup\{\pi_1{\cdot}X_1 \doteq \pi_1'{\cdot}X_1', \dots, \pi_n{\cdot}X_n \doteq \pi_n'{\cdot}X_n'\}}$$

$$(5)\ \frac{\mathit{\Gamma}\dot\cup(\lambda W.\pi_1{\cdot}X_1 \doteq \lambda W.\pi_2{\cdot}X_2\}}{\mathit{\Gamma}\cup\{\pi_1{\cdot}X_1 \doteq \pi_2{\cdot}X_2\}}$$

$$(6)\ \frac{\mathit{\Gamma}\dot\cup(\lambda W_1.\pi_1{\cdot}X_1 \doteq \lambda W_2.\pi_2{\cdot}X_2\}, \nabla}{\mathit{\Gamma}\cup\{\pi_1{\cdot}X_1 \doteq (W_1\ W_2){\cdot}\pi_2{\cdot}X_2\}, \nabla\cup\{W_1\#(\lambda W_2.\pi_2{\cdot}X_2)\}}$$

$$(7)\ \frac{\mathit{\Gamma}\dot\cup\left\{\begin{array}{l}\texttt{letrec}\ W_1.\pi_1{\cdot}X_1;\dots;W_n.\pi_n{\cdot}X_n\ \texttt{in}\ \tau{\cdot}Y \doteq\\ \texttt{letrec}\ W_1'.\pi_1'{\cdot}X_1';\dots;W_n'.\pi_n'{\cdot}X_n'\ \texttt{in}\ \tau'{\cdot}Y'\end{array}\right\}, \nabla}{\Big|_{\forall\rho}\left(\begin{array}{l}\mathit{\Gamma}\cup\left\{\begin{array}{l}\text{decompose}(n{+}1, \lambda W_1\dots\lambda W_n.(\pi_1{\cdot}X_1,\dots,\pi_n{\cdot}X_n,\tau{\cdot}Y)\\ \quad\doteq \lambda W_{\rho(1)}'.\dots\lambda W_{\rho(n)}'.(\pi_{\rho(1)}'{\cdot}X_{\rho(1)}',\dots,\pi_{\rho(n)}'{\cdot}X_{\rho(n)}',\tau'{\cdot}Y'))\end{array}\right\},\\ \nabla\cup\left\{\begin{array}{l}\text{decompfresh}(n{+}1, \lambda W_1\dots\lambda W_n.(\pi_1{\cdot}X_1,\dots,\pi_n{\cdot}X_n,\tau{\cdot}Y)\\ \quad\doteq \lambda W_{\rho(1)}'.\dots\lambda W_{\rho(n)}'.(\pi_{\rho(1)}'{\cdot}X_{\rho(1)}',\dots,\pi_{\rho(n)}'{\cdot}X_{\rho(n)}',\tau'{\cdot}Y'))\end{array}\right\}\end{array}\right)}$$

where ρ is a permutation on $\{1, \dots, n\}$ and decompose$(n, .)$ is the equation part of n-fold application of rules (4), (5) or (6) and decomposefresh$(n, .)$ is the freshness constraint part of the n-fold application of rules (4), (5) or (6).

Fig. 4. Standard and decomposition rules with atom variables of LetrecUnifyAV.

The rules are the same as (MMS), (FPS), (ElimFP) and (Output) as in Fig 2. In addition:

$$\text{(ElimA)}\ \frac{\mathit{\Gamma}, \nabla, \theta}{\mathit{\Gamma}[a/A], \nabla[a/A], \theta\cup\{A \mapsto a\}},$$

where we guess the following: some atom variable A occurring in $\mathit{\Gamma}, \nabla$ and an atom a that occurs in $\mathit{\Gamma}, \nabla, \theta$, or is a fresh atom.

Fig. 5. Main rules of LetrecUnifyAV

Note that permutations with atom variables may lead to an exponential blow-up of their size, which is defeated by a compression mechanism. Note also that equations $A \doteq e$, in particular $A \doteq \pi{\cdot}A'$, cannot be resolved by substitution for two reasons: (i) the atom variable A may occur in the right hand side, and (ii) due to our compression mechanism (see below), the substitution may introduce cycles into the compression, which is forbidden.

Definition 10.5. *The algorithm* LetrecUnifyAV *operates on a tuple $(\mathit{\Gamma}, \nabla, \theta)$, where the rules are defined in Figs. 4 and 5, and the following explanations:*

1. *$\mathit{\Gamma}$ is a set of flattened equations $e_1 \doteq e_2$, where we assume that $\doteq$ is symmetric,*
2. *∇ contains freshness constraints, where some may be written as equations of the form $A =_{\#} \pi{\cdot}A'$ [41] , for simplicity.*
3. *θ represents the already computed substitution as a list of replacements of the form $X \mapsto e$. We assume that the substitution is the iterated replacement. Initially θ is empty.*

The final state will be reached, i.e. the output, when $\mathit{\Gamma}$ only contains fixpoint equations of the form $X \doteq \pi{\cdot}X$, and the rule (Output) fires.

In the notation of the rules, we will use $[e/X]$ as substitution that replaces X by e. We may omit ∇ or θ, if they are not changed. We will also use a notation "$|$" in the consequence part of one rule, with a set of possibilities, to denote disjunctive (i.e. don't know) nondeterminism. There are two non-deterministic rules with disjunctive non-determinism: the letrec-decomposition rule (7) exploring all alternatives of the correspondence between bindings; the other one is (ElimA) that guesses the instantiation of an atom-variable. In case it is guessed to be different from all currently used atoms, we remember this fact (for simplicity) by selecting a fresh atom for instantiation. The other rules can be applied in any order, where it is not necessary to explore alternatives.

As a last step the fresh atoms have to be replaced by fresh atom-variables together with the following extra constraints in ∇: $A\#A'$ for different fresh atom variables A, A', and $A\#a$ for fresh atom variables A and atoms a in the problem.

We assume that permutations in the algorithm LetrecUnifyAV are compressed using a grammar-mechanism, as a variation of grammar-compression in [27, 19]. However, we do not mention it in the rules of the algorithm, but we will use it in the complexity arguments.

Definition 10.6. *The components of a* permutation grammar G*, used for compression, are:*

- *Nonterminals P_i.*
- *For every nonterminal P_i there is an associated inverse P_j, which can also be rewritten $\overline{P}_i$.*
- *Rules of the form $P_i \to w_1 \dots w_n$, where w_i is either a nonterminal or a terminal. At all times $\overline{P}_i \to \overline{w}_n \dots \overline{w}_1$ holds, i.e. if a nonterminal is added its inverse is added accordingly. Usually, $n \leq 2$, but also another fixed bound is possible.*
- *Terminal elements are $\emptyset$, or $(P{\cdot}V_1\ P'{\cdot}V_2)$.*

The grammar is deterministic: every nonterminal is on the left-hand side of exactly one rule. It is also non-recursive: the terminal index is such that P_i can only be in right-hand sides of the nonterminal P_j with $j < i$.

The function inv, mapping $P_i \to \overline{P}_i$ and $T \to T$ for terminals T computes the inverse in constant time. This is true by construction, because if $P \to w_1 \dots w_n$ then $inv(P) \to inv(w_n) \dots inv(w_1)$ and $inv(T) = T$ for terminals.

Every nonterminal P represents a permutation $val(P)$, which is computed from the grammar as follows:

1. *$val(P) = val(w_1)\ \dots\ val(w_n)$ (as a composition of permutations), if $P \to w_1 \dots w_n$.*
2. *$val(\emptyset) = Id$.*
3. *$val((P_1{\cdot}V_1\ P_2{\cdot}V_2)) = (val(P_1){\cdot}V_1\ \ val(P_2){\cdot}V_2)$.*

Lemma 10.7. *For nonterminals P of a permutation grammar G, the permutation $val(inv(P))$ is the inverse of $val(P)$.*

Let S be the size of the initial unification problem.

Proposition 10.8. *Let G be a permutation grammar, and let P be a nonterminal, such that $val(P)$ contains n atoms, and does not contain any atom variables. Then $val(P)$ can be transformed into a permutation of length at most n in polynomial time.*

Proof. For every P the size of the set $At(P)$ has an upper bound S and can be computed in time $O(S{\cdot}\log(S)$ For every such atom $a \in At(P)$ we compute its image $P{\cdot}a$ and save the result in a mapping from atoms to atoms. The computation of $P \cdot a$ can be done in $O(S|^2)$, yielding a total of $O(S^3)$ for the construction of this map, which has size $O(S)$. At last, the construction of the permutation list can be done in linear time, i.e. $O(S)$.

Now we consider the operations to extend the grammar during the unification algorithm.

Proposition 10.9. *Extending the grammar G n times can be performed in polynomial time in n, and the size of the initial grammar G.*

Proof. We check the extension operations:
Adding a nonterminal can be done in constant time. Adding an inverse of P is in constant time, since the inverses of the sub-permutations are already available. Adding a composition $P = P_1{\cdot}P_2$ and at the same time the inverse, can be done in constant time.

As a summary we obtain: Generating the permutation grammar on the fly during the execution of the unification rules can be done in polynomial time, since (as we will show below) the number of rule exeuctions is polynomial in the size of the initial input. Also the operation of applying a compressed atom-only permutation to an atom is polynomial.

Note that (MMS) and (FPS), without further precaution, may cause an exponential blow-up in the number of fixpoint equations (see Example 4.6). The rule (ElimFP) will limit the number of fixpoint equations for atom-only permutations by exploiting knowledge on operations on permutation groups. The rule (ElimA) can be used according to a dynamic strategy (see below): if the space requirement for the state is too high, then it can be applied until simplification rules make (Γ, ∇) smaller.

The rule (Output) terminates an execution on Γ_0 by outputting a unifier $(\theta, \nabla', \mathcal{X})$, where the solvability of ∇' needs to be checked using methods as in the algorithm proposed in [41]. The methods are to non-deterinistically instantiate atom-variables by atoms, and then checking the freshness constraints, which is in NP (see also Theorem 5.1).

We will show that the algorithm runs in polynomial space and time without brute forcing the (ElimA) rule by specifying a strategy. Let S be the size of the original unification problem. There are two rules, which can lead to a size increase of the unification problem – not taking the permutations into account – (MMS) and (FPS).

- (MMS) Given the equations $X \doteq e_1, X \doteq e_2$, the increase of the size of Γ after the application of the rule has an upper bound $O(S)$.
- (FPS) Given $X \doteq \pi_1 \cdot X, \ldots, X \doteq \pi_k \cdot X, X \doteq e$, the size increase has an upper bound $O(S)$. Unlike with permutations with atoms, it is not known whether there exists a polynomial upper bound of the number of independent permutations with atom variables - but it seems very unlikely.

Definition 10.10. *Let $p(x)$ be some function $\mathbb{R}^+ \to \mathbb{R}^+$. The rule ElimAB($p$) is defined as follows:*

- *ElimAB(p): If there are $k > p(S)$ fixpoint equations $X \doteq \pi_1 \cdot X, \ldots, X \doteq \pi_k \cdot X$ in Γ for the same variable X, then apply (ElimA) on all $A \in AtVar(\pi_1, \ldots, \pi_k)$. Then immediately apply (ElimFP) exhaustively.*

Definition 10.11. *The guided version* LetrecUnifyAVB(p) *of* LetrecUnifyAV *is obtained by replacing (ElimA) with ElimAB(p) where $p(x)$ is some (easily computable) function $\mathbb{R}^+ \to \mathbb{R}^+$, s.t. $\forall x \in \mathbb{R}^+ : q(x) \geq p(x) \geq x * \log(x)$ holds for some polynomial q. In addition the priority of the rules is as follows, where highest priority comes first: (1), ..., (6), (ElimFP), (MMS), (Output). Then ElimAB(p), (FPS), and the nondeterministic rule (7) with lowest priority.*

Lemma 10.12. *Let Γ, ∇ be a solvable input. For every function $p(x)$ with $\forall x \in \mathbb{R}^+ : p(x) \geq x \log(x)$,* LetrecUnifyAVB($p$) *does not get stuck, and the number of fixpoint equations per expression variable is limited by $p(S)$, where S is the size of the original input.*

Proof. The upper bound of the number of fixpoint equations is proven as follows: Let m be the number of atoms in the original unification problem. The rule (ElimA) introduces at most $S - m$ new atoms, which implies at most S atoms at any time. If LetrecUnifyAVB(p) reaches its upper bound and applies ElimAB(p) on the fixpoint equations $X \doteq \pi_1 \cdot X, \ldots, X \doteq \pi_k \cdot X$, the number of fixpoint equations of X can be reduced to at most $S \log(S) \leq p(S)$ (see the proof of Theorem 5.1).

Since the input is solvable, the choices can be made accordingly, guided by the solution, and then it is not possible that there is an occurs-check-fail for the variables. Hence if the upper line of the preconditions of (FPS) is a part of Γ, there will also be a maximal variable X, such that the condition $X \notin Var(\Gamma, e)$ can be satisfied.

Theorem 10.13. *Let Γ, ∇ be a solvable input. For every function $p(x)$ such that there is a polynomial $q(x)$ with $\forall x : q(x) \geq p(x) \geq x \log(x)$,* LetrecUnifyAVB($p$) *does not get stuck and runs in polynomial space and time.*

Proof. The proof is inspired by the proof of Theorem 5.1, and uses Lemma 10.12 according to which the number of fixpoint-equations for a single variable is at most $p(S)$.

Below we show some estimates on the size and the number of steps. The termination measure ($\#\mathrm{Var}, \#\mathrm{Lr}\lambda\mathrm{FA}, \#\mathrm{Eqs}, \#\mathrm{EqNonX}$), which is ordered lexicographically, is as follows:
#Var is the number of different variables in Γ,
#LrλFA is the number of letrec-, λ, function-symbols and atoms in Γ, but not in permutations,
#Eqs is the number of equations in Γ, and
#EqNonX is the number of equations where non of the equated expressions is a variable.

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.

The following table shows the effect of the rules: Let S be the size of the initial (Γ_0, ∇_0) where Γ is already flattened. Again, the entries $+W$ represent a size increase of at most W in the relevant measure component.

	#Var	#LrλFA	#Eqs	#EqNonX
(3)	$<$	$\leq$	$=$	$\leq$
(FPS)	$<$	$+2p(S)$	$<$	$+2p(S)$
(MMS)	$=$	$<$	$+2S$	$=$
(4), (5), (6), (7)	$=$	$<$	$+S$	$\leq$
ElimAB(p)	$=$	$=$	$<$	$\leq$
(1)	$\leq$	$\leq$	$<$	$\leq$
(2)	$=$	$=$	$=$	$<$

The table shows that the rule applications strictly decrease the measure. The entries can be verified by checking the rules, and using the argument that there are not more than $p(S)$ fixpoint equations for a single variable X. We use the table to argue on the number of rule applications and hence the complexity: The rules (3) and (FPS) strictly reduce the number of variables in Γ and can be applied at most S times. The rule (FPS) increases the second measure at most by $2p(S)$, since the number of symbols may be increased as often as there are fixpoint-equations, and there are at most $p(S)$. Thus the measure #LrλFA will never be greater than $2Sp(S)$.

The rule (MMS) strictly decreases #LrλFA, hence#Eqs, i.e. the number of equations is bounded by $4S^2p(S)$. The same bound holds for #EqNonX. Hence the number of rule applications is $O(S^2p(S))$. Of course, there may be a polynomial effort in executing a single rule, and by Proposition 10.9 the contribution of the grammar-operations is also only polynomial. Finally, since $p(x)$ is polynomially bounded by $q(x)$, the algorithm can be executed in polynomial time.

11 Conclusion and Future Research

We constructed nominal letrec unification algorithms, for the case where only atoms are permitted, and also for the case where atom variables are permitted. We also described several nominal letrec matching algorithms for variants, which all run in nondeterministic polynomial time. Future research is to investigate extensions of unification with environment variables E as an extension of the matching algorithm with environment variables, to investigate the connection with equivariant nominal unification [16, 14, 1], and to investigate nominal matching together with equational theories. Also applications of nominal techniques to reduction steps in operational semantics of calculi with letrec and transformations should be investigated.

References

1. Takahito Aoto and Kentaro Kikuchi. A rule-based procedure for equivariant nominal unification. In *Informal proceedings HOR*, page 5, 2016.

2. 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, CA, 1995. ACM Press.
3. Zena M. Ariola and Jan Willem Klop. Cyclic Lambda Graph Rewriting. In *Proc. IEEE LICS*, pages 416–425. IEEE Press, 1994.
4. Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. A formalisation of nominal α-equivalence with A and AC function symbols. *Electr. Notes Theor. Comput. Sci.*, 332:21–38, 2017.
5. Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal c-unification. In Fabio Fioravanti and John P. Gallagher, editors, *27th LOPSTR, Revised Selected Papers*, volume 10855 of *LNCS*, pages 235–251. Springer, 2017.
6. Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In Brigitte Pientka and Delia Kesner, editors, *Proc. first FSCD*, LIPIcs, pages 11:1–11:17, 2016.
7. Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Fixed-point constraints for nominal equational unification. In Kirchner [23], pages 7:1–7:16.
8. Mauricio Ayala-Rincón, Maribel Fernández, and Ana Cristina Rocha-Oliveira. Completeness in pvs of a nominal unification algorithm. *ENTCS*, 323(3):57–74, 2016.
9. 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.
10. Làszlò Babai. Graph isomorphism in quasipolynomial time. http://arxiv.org/abs/1512.03547v2, 2016.
11. Paolo Baldan, Clara Bertolissi, Horatiu Cirstea, and Claude Kirchner. A rewriting calculus for cyclic higher-order term graphs. *Mathematical Structures in Computer Science*, 17(3):363–406, 2007.
12. Kellogg S. Booth. Isomorphism testing for graphs, semigroups, and finite automata are polynomially equivalent problems. *SIAM J. Comput.*, 7(3):273–279, 1978.
13. Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.
14. James Cheney. *Nominal Logic Programming.* PhD thesis, Cornell University, Ithaca, New York, U.S.A., 2004.
15. James Cheney. Toward a general theory of names: Binding and scope. In *MERLIN 2005*, pages 33–40. ACM, 2005.
16. James Cheney. Equivariant unification. *J. Autom. Reasoning*, 45(3):267–300, 2010.
17. 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.
18. 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.
19. Adrià Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 12(4):26:1–26:37, 2011.
20. Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.
21. Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.
22. Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundam. Inform.*, 150(3-4):347–377, 2017.
23. Hélène Kirchner, editor. *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*. Schloss Dagstuhl, 2018.
24. Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1-2):125–150, 2000.
25. 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.
26. Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.
27. Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.
28. Eugene M. Luks. Permutation groups and polynomial-time computation. In Larry Finkelstein and William M. Kantor, editors, *Groups And Computation*, volume 11 of *DIMACS*, pages 139–176. DIMACS/AMS, 1991.
29. Simon Marlow, editor. *Haskell 2010 – Language Report.* 2010.
30. Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
31. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.
32. 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 *LNCS*, pages 85–102. Springer-Verlag, 1999.
33. Christophe Picouleau. Complexity of the Hamiltonian cycle in regular graph problem. *Theor. Comput. Sci.*, 131(2):463–473, 1994.
34. Conrad Rau and Manfred Schmidt-Schauß. Towards correctness of program transformations through unification and critical pair computation. In *Proc. 24th UNIF*, volume 42 of *EPTCS*, pages 39–54, December 2010.
35. Conrad Rau and Manfred Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *Proc. 25th UNIF*, pages 35–41, July 2011.

36. Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In Manuel V. Hermenegildo and Pedro López-García, editors, *26th LOPSTR, Revised Selected Papers*, volume 10184 of *LNCS*, pages 328–344. Springer, 2016.
37. Manfred Schmidt-Schauss, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal Unification of Higher Order Expressions with Recursive Let. RISC Report Series 16-03, RISC, Johannes Kepler University Linz, Austria, 2016.
38. Manfred Schmidt-Schauß, Conrad Rau, and David Sabel. Algorithms for Extended Alpha-Equivalence and Complexity. In Femke van Raamsdonk, editor, *24th RTA 2013*, volume 21 of *LIPIcs*, pages 255–270. Schloss Dagstuhl, 2013.
39. Manfred Schmidt-Schauß and David Sabel. Unification of program expressions with recursive bindings. In James Cheney and Germán Vidal, editors, *18th PPDP*, pages 160–173. ACM, 2016.
40. Manfred Schmidt-Schauß and David Sabel. Nominal unification with atom and context variables. In Kirchner [23], pages 28:1–28:20.
41. Manfred Schmidt-Schauß, David Sabel, and Yunus D. K. Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, 90:42–64, 2019.
42. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
43. Uwe Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.
44. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In Sandro Etalle and Miroslaw Truszczynski, editors, *22nd ICLP*, LNCS, pages 330–345, 2006.
45. Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in nominal Isabelle. *Log. Methods Comput. Sci.*, 8(2), 2012.
46. Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *17th CSL, 12th EACSL, and 8th KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.
47. Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.