DOI 10.3233/FI-2016-0000
IOS Press

Nominal Unification and Matching of Higher Order Expressions with Recursive Let

Manfred Schmidt-Schauß[*] [C]
GU Frankfurt, Germany, schauss@ki.cs.uni-frankfurt.de

Temur Kutsia[†]
RISC, JKU Linz, Austria, kutsia@risc.jku.at

Jordi Levy[‡]
IIIA - CSIC, Spain, levy@iiia.scic.es

Mateu Villaret[§]
IMA, Universitat de Girona, Spain, villaret@ima.udg.edu

Yunus Kutz
GU Frankfurt, Germany, kutz@ki.cs.uni-frankfurt.de

Abstract. A sound and complete algorithm for nominal unification of higher-order expressions with a recursive let is described, and shown to run in nondeterministic polynomial time. We also explore specializations like nominal letrec-matching for expressions, for DAGs, and for garbage-free expressions and determine their complexity. Finally, we also provide a nominal unification algorithm for higher-order expressions with recursive let and atom-variables, where we show that it also runs in nondeterministic polynomial time.

Keywords: Nominal unification, lambda calculus, higher-order expressions, recursive let, atom variables

This paper is an extended version of the conference publication [1].

[*] Partially supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SCHM 986/11-1
[C] Corresponding author
[†] Partially supported by the Austrian Science Fund (FWF) project P 28789-N32.
[‡] Partially supported by the MINECO/FEDER projects RASO (TIN2015-71799-C2-1-P) and LoCoS (TIN2015-66293-R).
[§] Partially supported by UdG project MPCUdG2016/055.

1. Introduction

Unification [2] 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., [3]), however, the problem is undecidable [4, 5].

Our motivation comes from syntactical reasoning on higher-order expressions, with equality being α-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. α-equivalence, and employs permutations as a mathematically clean treatment of renamings. It is known that nominal unification is decidable [6, 7], where the complexity of the decision problem is polynomial time [8]. It can be seen also from a higher-order perspective [9], as equivalent to Miller's higher-order pattern unification [10]. There are efficient algorithms [8, 11], formalizations of nominal unification [12], formalizations with extensions to commutation properties within expressions [13], and generalizations of nominal unification to narrowing [14]. Equivariant (nominal) unification [15, 16, 17] 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. Higher-order unification could be applied, however, the algorithms are rather general and thus the obtained complexities of specializations is far too high. Thus we propose to extend and adapt usual nominal unification [6, 7] to languages with recursive 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 [18, 19]) that have a letrec-construct (also called cyclic let) [20] as in Haskell [21], (see e.g. [22] for a discussion on reasoning with more general name binders, and [23] for a formalization of general binders in Isabelle). Extended nominal matching algorithms are necessary for applying program transformations that could be represented as rewrite rules. Basic properties of program transformations like commuting properties of conflicting applications or overlaps can be analyzed in an automated way if there is a nominal unification algorithm of appropriate complexity. There may be applications also to coinductive extensions of logic programming [24] and strict functional languages [25]. 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-redex 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 borrow the decomposition rules, and the sharing method from Martelli-

Montanari-style unification algorithms [26]. 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 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) are required. 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 investigate 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 8.3).

As a further extension, we study the possibility to formulate input problems using atom variables as in [27, 28] in order to take advantage of the potential of less nondeterminism. The corresponding algorithm LETRECUNIFYAV requires permutation expressions and generalizes 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: Besides the already mentioned related work, we highlight further work. In nominal commutative unification [29], 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 [30], 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 [31] 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 [32] 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 the 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 [33, 34]. Later, in [35], 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 [33, 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 [35], 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.

There are investigations into variants of nominal techniques with a modified view of variables and their renamings and algorithms for the respective variants of nominal unification [36], however, it is unclear whether this can be extended to letrec.

Results: The nominal letrec unification algorithm is complete and runs in nondeterministic polynomial time (Theorem 5.2, 5.4). The nominal letrec matching is NP-complete (Theorems 6.3, 5.1), as well as the nominal letrec unification problem (Theorems 5.4, 5.1). Nominal letrec matching for DAGs is in NP and outputs substitutions only (Theorem 6.5), and a very restricted nominal letrec matching problem is already graph-isomorphism hard (Theorem 7.2). Nominal unification for garbage-free expressions can be done with simple fixpoint rules (Corollary 8.3). In the extension with atom variables, nominal unification can be done using further useful strategies with less nondeterminism and is NP-complete (Theorem 9.15). We construct an algorithm for nominal matching including letrec-environment variables, which runs in NP time (Theorem 10.4).

Structure of the paper. It starts with a motivating intuition on nominal unification (Sec. 2). After explaining the ground letrec-language LLR in Sec. 3, the unification algorithm LetrecUnify for LLR-expression is described in Sec. 4. Sec. 5 contains the arguments for soundness and completeness of LetrecUnify. Sec. 6 describes an improved algorithm for nominal matching on LLR: LetrecMatch. Further sections are on extensions. Sec. 7 shows Graph-Isomorphism-hardness of nominal letrec matching and unification on garbage-free expressions (Theorem 7.2). Sec. 8 shows that fixpoint-equations for garbage-free expressions can be translated into freshness constraints (Cor. 8.3). Sec. 9 considers nominal unification in an extension with atom variables, an nominal unification algorithm LetrecUnifyAV is defined and the differences to LetrecUnify are highlighted. It is shown that there is a simple strategy such that nominal unification runs in NP time (Theorem 9.15). The last section (Sec. 10) presents a nominal matching algorithm LetrecEnvMatch that is derived from the corresponding nominal unification algorithm LetrecUnifyAV. Sec. 11 concludes the paper.

2. Some Intuitions

In first order unification we have a language of applications of function symbols over a (possible empty) list of arguments $(f e_1 \dots e_n)$, 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 problems may be solved using the following two problem transformation rules:

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

(Instantiation) $\dfrac{\Gamma \uplus \{X \doteq e\}}{[X \mapsto e]\Gamma}$ If X 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 [26] described a set of improved rules that result into an $O(n\,\log n)$ time algorithm[1] where n is the size of the input equations. 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:

(Decomposition) $\dfrac{\Gamma \uplus \{(f\,X_1 \dots X_n) \doteq (f\,Y_1 \dots Y_n)\}}{\Gamma \cup \{X_1 \doteq Y_1, \dots, X_n \doteq Y_n\}}$

(Variable Instantiation) $\dfrac{\Gamma \uplus \{X \doteq Y\}}{[X \mapsto Y]\Gamma}$

(Elimination) $\dfrac{\Gamma \uplus \{X \doteq e\}}{\Gamma}$ If X neither occurs in e nor in Γ

(Merge) $\dfrac{\Gamma \uplus \{X \doteq (f\,X_1 \dots X_n), X \doteq (f\,Y_1 \dots Y_n)\}}{\Gamma \cup \{X \doteq (f\,X_1 \dots X_n), X_1 \doteq Y_1, \dots, X_n \doteq Y_n\}}$

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

$\dfrac{\Gamma \uplus \{X \doteq e_1, X \doteq e_2\}}{\Gamma \cup \{X \doteq e_1, e_1 \doteq e_2\}}$ If e_1 and e_2 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

[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.

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:

(Decomposition lambda 1) $$\frac{\Gamma\dot\cup\{\lambda a.s \doteq \lambda a.t\}}{\Gamma\cup\{s\doteq t\}}$$ (Decomposition lambda 2) $$\frac{\Gamma\dot\cup\{\lambda a.s \doteq \lambda b.t\}}{\Gamma\cup\{s\doteq (a\;b)\cdot t, a\#t\}}$$

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 is 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:

(Variable Instantiation) $$\frac{\Gamma\dot\cup\{\pi\cdot X\doteq\pi'\cdot Y\}\quad X\neq Y}{[X\mapsto(\pi^{-1}\circ\pi')\cdot Y]\Gamma}$$ (Fixpoint) $$\frac{\Gamma\dot\cup\{\pi\cdot X\doteq\pi'\cdot X\}}{\Gamma\cup\{a\#X\mid a\in dom(\pi^{-1}\circ\pi')\}}$$

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 $O(n^2\;\log n)$ algorithm, similar to the algorithms described in [8, 11]. 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 all 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. [37] prove that equivalence of letrec expressions is graph-isomorphism (GI) complete and Schmidt-Schauß and Sabel [35] 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 $(\texttt{letrec}\,v_1.a;v_2.a;v_3.a\ \texttt{in}\ \texttt{letrec}\,e_1.(c\,v_1\;v_2);e_2.(c\,v_2\;v_3)\ \texttt{in}\,a)$. Here,

v_i represent the nodes and $(c\, v_i\, v_j)$ the edges of the graph.

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 [6] 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:

$$\text{(FixPointShift)}\;\frac{\Gamma\dot{\cup}\{\pi_1{\cdot}X \doteq \pi_1'{\cdot}X, \ldots, \pi_n{\cdot}X \doteq \pi_n'{\cdot}X, \pi{\cdot}X \doteq e\}}{\Gamma \cup \{\pi_1\pi^{-1}{\cdot}e \doteq \pi_1'\pi^{-1}{\cdot}e, \ldots, \pi_n\pi^{-1}{\cdot}e \doteq \pi_n'\pi^{-1}{\cdot}e\}},\;\text{ if } X \text{ neither occurs in } e \text{ nor in } \Gamma.$$

The substitution is $X \to \pi^{-1}{\cdot}e$. This rule can generate an exponential number of equations (see Example 4.6). In order to avoid this effect, 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) nondeterministic choice.

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

Where the necessary freshness constraints are $\{a_i\#(\texttt{letrec}\; b_1.t_1; \ldots; b_n.t_n\; \texttt{in}\; r') \mid i = 1, \ldots, n\}$, ρ is a permutation on $\{1, \ldots, n\}$ and π is an (atom-)permutation that extends $\{b_{\rho(i)} \mapsto a_i \mid i = 1, \ldots, n)\}$ with $dom(\pi) \subseteq \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$.

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 [6] 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 bijective renamings. 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 $(\mathtt{x}\;\mathtt{y}){\cdot}(\lambda\mathtt{x}.\lambda\mathtt{x}.\mathtt{a})$ 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 emulated 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 non-bijective renamings have a tendency to complicate (unification-) algorithms and to hide the required conditions on equality/disequality/occurrence/non-occurrence of names. We will stick to a notation closer to lambda calculi than most other papers on nominal unification, however, note that in general the differences are only in notation and the constructs like application and abstraction can easily be translated into something equivalent in the other language without any loss.

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 [6]. 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 will 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 expressions $(\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$ and $(\texttt{letrec}\ a_{\rho(1)}.e_{\rho(1)}; \ldots; a_{\rho(n)}.e_{\rho(n)}\ \texttt{in}\ e)$ are defined as equivalent for every permutation ρ of $\{1, \ldots, n\}$.

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 atom a_i that is free in some s_j or r is bound by the environment $a_1.s_1; \ldots; a_n.s_n$. This defines in LRL 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. The functional application operator in functional languages (which is usally implicit) can be encoded by a binary function app, which also allows to deal with partial applications. Our language LRL is a fragment of core calculi [18, 19], since for example the case-construct is missing, but this could also be represented. For more programming examples the reader may consult the functional programming languages literature and web-sites.

We will use mappings on atoms from $\mathbb{A}$. A *swapping* $(a\ b)$ is a bijective function (on LRL-expressions) 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 could be 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, where the complexity is polynomial in the size of $dom(\pi)$. 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 (characterization) of α-equivalence:

Definition 3.1. The α-equivalence $\sim$ on expressions $e \in LRL$ is defined as follows:

- $a \sim a$.
- if $e_i \sim e'_i$ for all i, then $(f e_1 \ldots e_n) \sim (f e'_1 \ldots e'_n)$ for an n-ary $f \in \mathcal{F}$.
- 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 a permutation π on atoms such that
  - $dom(\pi) \subseteq \{a_1,\dots,a_n\} \cup \{b_1,\dots,b_n\}$, where $a_i \neq a_j$ and $b_i \neq b_j$ for all $i \neq j$,
  - $\pi(b_i) = a_i$ for all i,
  - $\{a_1,\dots,a_n\}\#(\texttt{letrec}\, b_1.t_1,\dots,b_n.t_n\ \texttt{in}\, r')$, and
  - $r \sim \pi(r')$ and $s_i \sim \pi(t_i)$ for $i = 1,\dots,n$ hold.

  Then $(\texttt{letrec}\, a_1.s_1,\dots,a_n.s_n\ \texttt{in}\, r) \sim (\texttt{letrec}\, b_1.t_1,\dots,b_n.t_n\ \texttt{in}\, r')$.

The last phrase includes that $(\texttt{letrec}\, a_1.s_1,\dots,a_n.s_n\ \texttt{in}\, r) \sim (\texttt{letrec}\, b_{\rho(1)}.t_{\rho(1)},\dots,b_{\rho(n)}.t_{\rho(n)}\ \texttt{in}\, r')$. for every permutation ρ on $\{1,\dots,n\}$, by the definition of syntactic equality.

Note that $\sim$ is identical to α-equivalence, i.e. the relation generated by renamings of binding constructs and permutation of bindings in a letrec. We omit a proof, since it detracts the attention from the main contents. Such a proof is not hard to construct by using that α-equivalence holds, if and only if the graph constructed by replacing bindings by pointing edges, where the outgoing edges from an environment are unordered and the one from a function application are ordered.

A nice and important property that is often implicitly used is: $e_1 \sim e_2$ is equivalent to $\pi{\cdot}e_1 \sim \pi{\cdot}e_2$ for any (atom-)permutation π.

In usual nominal unification, the solutions of fixpoint equations $X \doteq \pi{\cdot}X$, i.e. the sets $\{e \mid \pi\cdot e \sim e\}$ can be characterized by using finitely many freshness constraints [6]. 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 $(a\ b)\cdot(\texttt{letrec}\ c.a; d.b\ \texttt{in}\ \mathit{True}) \sim (\texttt{letrec}\ c.a; d.b\ \texttt{in}\ \mathit{True})$ holds, which means that there are expressions t in LRL with $t \sim (a\ b)\cdot t$ and $\mathit{FA}(t) = \{a,b\}$. This is in contrast to usual nominal unification.

3.2. Permutation Groups

Below we will use the results on complexity of operations in finite permutation groups, see [38, 39]. We summarize some facts on the so-called symmetric group and its properties. We consider a set $\{o_1,\dots,o_n\}$ of distinct objects o_i (in our case atoms), and the symmetric group $\Sigma(\{o_1,\dots,o_n\})$ (of size $n!$) of permutations of these objects. We will also look at its elements, subsets and subgroups. Subgroups of $\Sigma(\{o_1,\dots,o_n\})$ can always be represented by a set of generators (represented as permutations on $\{o_1,\dots,o_n\}$). If H is a set of elements (or generators), then $\langle H\rangle$ denotes the generated subgroup of $\Sigma(\{o_1,\dots,o_n\})$. Some facts are:

- A permutation can be represented in space linear in n.
- Every subgroup of $\Sigma(\{o_1,\dots,o_n\})$ can be represented by $\leq n^2$ generators.

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

The following questions can be answered in polynomial time:

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

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

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

Figure 1. Simplification of freshness constraints in $LRLX$

- 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 [38]).

4. A Nominal Letrec Unification Algorithm

As an extension of LRL, there is a countably infinite set of (unification) variables Var ranged over by X, Y where we also use indices. The syntax of the language $LRLX$ (**L**et**R**ec **L**anguage e**X**tended) is

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

$Var(e)$ 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 additional simplification $\pi_1{\cdot}(\pi_2{\cdot}e) \to (\pi_1 \circ \pi_2){\cdot}e$, where after the shift, π only occurs in the subexpressions of the form $\pi \cdot X$, which are called *suspensions*. Usually, we do not distinguish X and $Id{\cdot}X$, notationally. 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 $a\#X$.

Lemma 4.1. The rules in Fig. 1 for simplifying sets of freshness constraints in $LRLX$ run in polynomial time and the result is either $\bot$, i.e. fail, or a set of freshness constraints where all single constraints are atomic. This constitutes a polynomial decision algorithm for satisfiability of ∇: If $\bot$ is in the result, then unsatisfiable, otherwise satisfiable.

We can assume in the following algorithms that sets of freshness constraint are immediately simplified. In the following we will use $Var(\Gamma,\nabla)$, and $Var(\Gamma, e)$ and similar notation for the set of (unification-)variables occurring in the syntactic objects mentioned in the brackets.

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 $\{a_1\#X_1,\ldots,a_m\#X_m\}$. 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 $a_j\#(X_j\rho)$, for $j = 1, \ldots, m$.
The decision problem is whether there is a ground solution for a given (Γ, ∇) or not.

Definition 4.3. Let (Γ, ∇) be an $LRLX$-unification problem. We consider triples $(\sigma, \nabla', \mathit{FIX})$ as representing general unifiers, where σ is a substitution (compressed as a DAG (directed acyclic graph)) mapping variables to $LRLX$-expressions, ∇' is a set of freshness constraints, and FIX is a set of fixpoint equations of the form $\pi'{\cdot}X \doteq \pi{\cdot}X$, where $X \notin \mathit{dom}(\sigma)$.
A triple $(\sigma, \nabla', \mathit{FIX})$ is a *unifier* of (Γ, ∇), if

(i) there exists a ground substitution ρ that solves $(\nabla'\sigma, \mathit{FIX})$, i.e., for every $a\#X$ in ∇', $a\#X\sigma\rho$ is valid, and for every fixpoint equation $\pi'{\cdot}X \doteq \pi{\cdot}X \in \mathit{FIX}$, it holds $\pi'{\cdot}(X\rho) \sim \pi{\cdot}(X\rho)$; and

(ii) for every ground substitution ρ that instantiates all variables in $Var(\Gamma, \nabla)$ and which solves $(\nabla'\sigma, \mathit{FIX})$, 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 $(\sigma, \nabla', \mathit{FIX})$ in M, and a ground substitution ρ, such that $X\mu \sim X\sigma\rho$ for all $X \in Var(\Gamma, \nabla)$.

We will employ nondeterministic rule-based algorithms computing unifiers: There are clearly indicated disjunctive (don't know nondeterministic) rules, and all other rules are don't care nondeterministic. This distinction is related to the completeness of the solution algorithms: don't care means that the rule has several possibilities where it is sufficient for completeness to take only one. On the other hand, don't know means that for achieving completeness, every possibility of the rule has to be explored. The *collecting variant* of the algorithm runs and collects all solutions from all alternatives of the disjunctive rule(s). The *decision variant* guesses and verifies one possibility and tries to detect the existence of 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 [26], we stick to a set of equations as data structure. As a preparation for the algorithm, all expressions in equations are exhaustively flattened as follows: $(f\ t_1 \ldots t_n) \to (f\ X_1 \ldots X_n)$ plus the equations $X_1 \doteq t_1, \ldots, X_n \doteq t_n$. Also $\lambda a.s$ is replaced by $\lambda a.X$ with equation $X \doteq s$, and $(\texttt{letrec}\ a_1.s_1; \ldots, a_n.s_n\ \texttt{in}\ r)$ is replaced by $(\texttt{letrec}\ a_1.X_1; \ldots, a_n.X_n\ \texttt{in}\ X)$ with the additional equations $X_1 \doteq s_1; \ldots; X_n \doteq s_n; X \doteq r$. The introduced variables X_i, X are fresh ones. Thus, all expressions in equations are of depth at most 1, not counting the permutation applications in the suspensions.

In the notation of the rules, we use $[e/X]$ as substitution that replaces X by e, whereas $\{X \to t\}$ is used for constructing a syntactically represented substitution. In the written rules, we may omit ∇ or θ if they are not changed. We will use a notation "|" in the consequence part of a rule, usually 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 (6). The other rules can be applied in any order, where it is not necessary to explore alternatives.

$$(1)\ \frac{\Gamma\dot\cup\{e \doteq e\}, \nabla, \theta}{\Gamma, \nabla, \theta} \qquad (2)\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}X \doteq \pi_2{\cdot}Y\}, \nabla, \theta \qquad Y \neq X}{\Gamma[\pi_1^{-1}\pi_2{\cdot}Y/X], \nabla[\pi_1^{-1}\pi_2{\cdot}Y/X], \theta \cup \{X \mapsto \pi_1^{-1}\pi_2{\cdot}Y\}}$$

$$(3)\ \frac{\Gamma\dot\cup\{(f\ s_1 \ldots s_n) \doteq (f\ s_1' \ldots s_n')\}, \nabla, \theta}{\Gamma \cup \{s_1 \doteq s_1', \ldots, s_n \doteq s_n'\}, \nabla, \theta}$$

$$(4)\ \frac{\Gamma\dot\cup\{\lambda a.s \doteq \lambda a.t\}, \nabla, \theta}{\Gamma \cup \{s \doteq t\}, \nabla, \theta} \qquad (5)\ \frac{\Gamma\dot\cup\{\lambda a.s \doteq \lambda b.t\}, \nabla, \theta \qquad a \neq b}{\Gamma \cup \{s \doteq (a\ b){\cdot}t\}, \nabla \cup \{a\#t\}, \theta}$$

$$(6)\ \frac{\Gamma\dot\cup\{\texttt{letrec}\, a_1.s_1; \ldots; a_n.s_n\ \texttt{in}\, r \doteq \texttt{letrec}\, b_1.t_1; \ldots; b_n.t_n\ \texttt{in}\, r'\}, \nabla, \theta}{\begin{array}{ll} \Big| & (\Gamma\ \cup \{s_1 \doteq \pi{\cdot}t_{\rho(1)}, \ldots, s_n \doteq \pi{\cdot}t_{\rho(n)}, r \doteq \pi{\cdot}r'\}, \\ \forall\rho & \quad \nabla \cup \{a_i\#(\texttt{letrec}\, b_1.t_1; \ldots; b_n.t_n\ \texttt{in}\, r') \mid i = 1, \ldots, n\}, \theta) \end{array}}$$

where ρ is a permutation on $\{1, \ldots, n\}$ and π is an (atom-)permutation that extends $\{b_{\rho(i)} \mapsto a_i \mid i = 1, \ldots, n)\}$ with $dom(\pi) \subseteq \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$

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

4.1. Rules of the Algorithm LETRECUNIFY

The top symbol of an expression is defined as $tops(f\ s_1 \ldots s_n) = f$, $tops(a) = a$, $tops(\lambda a.s) = \lambda$, and $tops(\texttt{letrec}\ env\ \texttt{in}\ s) = (\texttt{letrec}, n)$, where n is the number of bindings in env. It is undefined for variables X.

Definition 4.4. The rule-based algorithm LETRECUNIFY is defined in the following. Its rules are in Figs. 2, 3 and 4. LETRECUNIFY operates on a tuple (Γ, ∇, θ), where Γ is a set of flattened equations $e_1 \doteq e_2$, and where we assume that $\doteq$ is symmetric, ∇ contains freshness constraints, and θ represents the already computed substitution as a list of mappings of the form $X \mapsto e$. Initially θ is empty.

The final state will be reached, i.e. the output, when Γ only contains fixpoint equations of the form $\pi_1{\cdot}X \doteq \pi_2{\cdot}X$ that are non-redundant, and the rule (Output) fires. Note that the rule (FPS) represents the usual solution rule if the premise is only a single equation.

The rules (1)–(6), and (ElimFP) have highest priority; then (MMS) and (FPS). The rule (Output) (lowest priority) terminates an execution on Γ_0 by outputting a unifier (θ, ∇', FIX).

We assume that the algorithm LETRECUNIFY halts if a failure rule (see Fig.4) is applicable.

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 bound the number of generated fixpoint equations by exploiting knowledge on operations within permutation groups.

Note that the application of every rule can be done in polynomial time. In particular rule (FailFS), since the computation of $FA((X)\theta)$ can be done in polynomial time by iterating over the solution components.

$$\text{(MMS)}\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}X \doteq e_1, \pi_2{\cdot}X \doteq e_2\},\nabla}{\Gamma\cup\{\pi_1{\cdot}X \doteq e_1\}\cup\Gamma',\nabla\cup\nabla'},$$

if e_1, e_2 are not suspensions, where Γ' is the set of equations generated by decomposing $\pi_1^{-1}{\cdot}e_1 \doteq \pi_2^{-1}{\cdot}e_2$ using (3)–(6), and where ∇' is the corresponding resulting set of freshness constraints.

$$\text{(FPS)}\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}X \doteq \pi_1'{\cdot}X,\ldots,\pi_n{\cdot}X \doteq \pi_n'{\cdot}X, \pi{\cdot}X \doteq e\},\theta}{\Gamma\cup\{\pi_1\pi^{-1}{\cdot}e \doteq \pi_1'\pi^{-1}{\cdot}e,\ldots,\pi_n\pi^{-1}{\cdot}e \doteq \pi_n'\pi^{-1}{\cdot}e\},\theta\cup\{X\mapsto\pi^{-1}{\cdot}e\}},$$

If neither $X \in \mathit{Var}(\Gamma, e)$, nor e is a suspension, nor (Cycle) (see Fig.4) is applicable.

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

If $\pi^{-1}\pi' \in \langle\pi_1^{-1}\pi_1',\ldots,\pi_n^{-1}\pi_n'\rangle$.

$$\text{(Output)}\ \frac{\Gamma,\nabla,\theta}{(\theta,\nabla,\Gamma)}\ \text{if } \Gamma \text{ only consists of fixpoint-equations.}$$

Figure 3. Main Rules of LETRECUNIFY

$$\text{(Clash)}\ \frac{\Gamma\dot\cup\{s \doteq t\},\nabla,\theta \quad \mathit{tops}(s) \neq \mathit{tops}(t) \text{ and } s \text{ and } t \text{ are not suspensions}}{\bot}$$

$$\text{(Cycle)}\ \frac{\begin{array}{c}\text{If } \pi_1{\cdot}X_1 \doteq s_1,\ldots,\pi_n{\cdot}X_n \doteq s_n \text{ in } \Gamma \text{ where } s_i \text{ are not suspensions}\\ \text{and } X_{i+1} \text{ occurs in } s_i \text{ for } i = 1,\ldots,n-1 \text{ and } X_1 \text{ occurs in } s_n.\end{array}}{\bot}$$

$$\text{(FailF)}\ \frac{a\#a \in \nabla}{\bot} \qquad \text{(FailFS)}\ \frac{a\#X \in \nabla \quad \text{and } a \text{ occurs free in } (X\theta)}{\bot}$$

Figure 4. Failure Rules of LETRECUNIFY

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 (position-)permutation ρ, which results in: $\pi = \{b \mapsto a; c \mapsto b; a \mapsto c\}$, where the third binding $a \mapsto c$ is irrelevant, but unique in this case.

Decomposition of the equations $(a,b) \doteq \pi{\cdot}(b,c), (a,b) \doteq \pi{\cdot}(b,c), b \doteq \pi{\cdot}c\}$ is possible without fail and yields only trivial equations.

The freshness constraint is $\{a,b\}\#(\texttt{letrec}\ b.(b,c), c.(b,c)\ \texttt{in}\ c)$, which holds.

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 $X_i, i = 1,\ldots,n$ 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):
$$\left\{\begin{array}{rcl} f\ X_{n-1}\ \rho_n{\cdot}X_{n-1} & \doteq & \pi{\cdot}(f\ X_{n-1}\ \rho_n{\cdot}X_{n-1}), \\ X_{n-1} & \doteq & (f\ X_{n-2}\ \rho_{n-1}{\cdot}X_{n-2}), \ldots \end{array}\right\}$$

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

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

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

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

5.1. NP-Hardness of nominal letrec unification and matching

First we show that a restricted problem class of nominal letrec unification is already NP-hard. If the equations for unification are of the form $s_1 \doteq t_1, \ldots, s_n \doteq t_n$, and the expressions t_i do not contain variables X_i, then this is a nominal letrec-matching problem (see also Section 6).

Theorem 5.1. Nominal letrec matching (hence also unification) in LRL 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 3-regular graph [40, 41], which are graphs where all nodes have the same degree $k = 3$. Let G be a graph, $a_1, \ldots, a_n$ be the vertexes of the graph G, and E be the set of edges of G. 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 $t := (\mathtt{letrec}\ env_1; env_2\ \mathtt{in}\, 0)$ which is intended to represent 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 $b_1.(f\ X_1\ X_2); b_2.(f\ X_2\ X_3); \ldots; b_n.(f\ X$ 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 can be computed as $3 * n - n$ due to the assumption that the degree of G is 3. 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 are shortcuts in the cycle. □

5.2. Properties of the Nominal Unification Algorithm LetrecUnify

We will use $size(\Gamma)$ for estimating the runtime of LetrecUnify, which is the sum of the sizes of the equated expressions, and where the size of an expression is its size as a term tree, without size of names. We do not count the size of permutations.

Theorem 5.2. 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, where Γ_0 is assumed to be flattened, and S_{all} be $size(\Gamma_0, \nabla_0)$. We use $S = size(\Gamma_0)$ to argue on the number of steps of LetrecUnify. 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 on the set of all atoms in the input is at most $S!$, and so the length of proper subset-chains and hence the maximal number of (necessary) generators of a subgroup is at most $\log(S!) \leq S * \log(S)$. 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 $(\#\mathrm{Var}, \#\mathrm{Lr}\lambda\mathrm{FA}, \#\mathrm{Eqs})$ is used:

1. $\#\mathrm{Var}$ is the number of different variables in Γ,

2. $\#\mathrm{Lr}\lambda\mathrm{FA}$ is the number of letrec-, λ, function-symbols and atoms in Γ, but not in permutations,

3. $\#\mathrm{Eqs}$ is the number of equations in Γ.

Since shifting permutations down and simplification of freshness constraints both terminate in polynomial time, 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:

The entries $+m$ represents an increase of at most m in the relevant measure component. This form of table permits an easy check that the complexity of a single run is polynomial. Note that we omit the failure rules in th table, since these stop immediately.

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

The table shows that every rule application strictly decreases the lexicographic measure as a combination of the three basic measures. 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 (overall) number of rule applications and hence the complexity: The rules (2) 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 $S\log(S)$. Since 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 $4S^3\log(S)$. Thus, the number of rule applications is $O(S^3\log(S))$.
The complexity of applications of single rules is polynomial, in particular (FPS), see Section 3.2. The complexity of the constraint simplification (Lemma 4.1) is also polynomial. We also have to argue on the failure rules. These detect all fail cases, and the size of the state part ∇ remains polynomial. The checks within the failure rules can be done in polynomial time in S_{all}, where the argument for polynomiality of the check in (FailFS) is an algorithm that iteratively applies parts of θ and checks. □

Theorem 5.3. The algorithm LETRECUNIFY is sound and complete.

Proof:
Soundness of the algorithm holds, by easy arguments for every rule, similar as in [6], 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 states without solutions.

Completeness requires more arguments. The decomposition and standard rules, with the exception of rule (6), 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 (6) provides all possibilities for potential ground solutions. Moreover, the failure rules are not applicable to states that are solvable.

A final output of LETRECUNIFY for a solvable input 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 5.4. The nominal letrec-unification problem is NP-complete.

Proof:
This follows from Theorems 5.2 and 5.3, and Theorem 5.1. □

6. Nominal Matching with Letrec: LETRECMATCH

Reductions using reductions rules of the form $l \to r$ in higher order calculi with letrec, in particular on a meta-notation, require a matching algorithm, matching the rules' left hand side to an expression or subexpression that is to be reduced. For the application it is sufficient if the instance of the right hand side $r\sigma$ is ground and the variable convention holds for $r\sigma$.

Example 6.1. Consider the (lbeta)-rule, which is the version of (beta) used in call-by-need calculi with sharing [42, 18, 19]. Note that only the sharing power of the recursive environment is used here.

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

An (lbeta) step, for example, on $(\lambda x.x)\ (\lambda y.y)$ is performed by representing the target in LRL and the beta-rule in the language LRLX, where e_1, e_2 are represented as variables X_1, X_2, 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 sequences permits α-equivalence as intermediate steps.

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 $s_1 \trianglelefteq t_1, \ldots, s_n \trianglelefteq t_n$ of match equations of expressions, where for all i: s_i may contain variables, and t_i is ground.

Definition 6.2. The rules of the nondeterministic algorithm LETRECMATCH are in Fig. 5.

The test $e_1 \sim e_2$ may for example 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. Standard arguments show:

Theorem 6.3. 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.

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

$$\frac{\Gamma\dot{\cup}\{\lambda a.s \trianglelefteq \lambda b.t \quad a\#t\}}{\Gamma \cup \{s \trianglelefteq (a\ b){\cdot}t\}} \qquad \frac{\Gamma\dot{\cup}\{\pi{\cdot}X \trianglelefteq e\}}{\Gamma \cup \{X \trianglelefteq \pi^{-1}{\cdot}e\}} \qquad \frac{\Gamma\dot{\cup}\{X \trianglelefteq e_1, X \trianglelefteq e_2 \quad e_1 \sim e_2\}}{\Gamma \cup \{X \trianglelefteq e_1\}}$$

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

where ρ is a permutation on $\{1, \ldots, n\}$, and π is an (atom-) permutation that extends $\{b_{\rho(i)} \mapsto a_i \mid i = 1, \ldots, n)\}$ with $dom(\pi) \subseteq \{a_1, \ldots, a_n, b_1, \ldots, b_n\}$, and $\{a_i\#(\texttt{letrec}\, b_1.t_1; \ldots; b_n.t_n\ \texttt{in}\, r') \mid i = 1, \ldots n\}$

$$\frac{s \trianglelefteq t \in \Gamma,\ \text{and } s \text{ is not a suspension, but } tops(s) \neq tops(t)}{\bot} \qquad \frac{\Gamma\dot{\cup}\{\lambda a.s \trianglelefteq \lambda b.t \quad \text{not } (a\#t)\}}{\bot}$$

$$\frac{\Gamma\dot{\cup}\{X \trianglelefteq e_1, X \trianglelefteq e_2 \quad e_1 \not\sim e_2\}}{\bot}$$

Figure 5. Rules of the matching algorithm LETRECMATCH

Theorem 6.4. Nominal letrec matching is NP-complete.

Proof:
The problem is in NP, which follows from Theorem 6.3. It is also NP-hard, which follows from Theorem 5.1. □

A slightly more general situation for nominal letrec matching occurs, when the matching equations Γ_0 are compressed using a DAG. We construct a practically more efficient algorithm LETRECDAGMATCH from LETRECUNIFY as follows. First we generate Γ_1 from Γ_0, which only contains flattened expressions by encoding the DAG-nodes as variables together with a unification equation. An expression is said Γ_0-ground, if it does not reference variables from Γ_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 :

$$\frac{\Gamma\dot{\cup}(\lambda a.s \doteq \lambda b.t\}, \nabla}{\Gamma \cup \{s \doteq (a\ b){\cdot}t\}, \nabla \cup \{a\#t\}} \quad \lambda b.t \text{ is } \Gamma_0\text{-ground}$$

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

Theorem 6.5. 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. Graph-Isomorphism-Hardness of Nominal Letrec Matching and Unification without Garbage

First we clarify the notion of garbage, which is a notion from a programming language point of view.

Definition 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 (the garbage) is not trivial, and the atoms from $LA(env_1)$ occur neither free in env_2 nor in r. Otherwise, the expression is *free of garbage* (or *garbage-free*).

Since α-equivalence of LRL-expressions is Graph-Isomorphism-complete [37], 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 [44], which means that it requires less than exponential time.

We will show that even very restricted nominal letrec matching problems are Graph-Isomorphism complete, which makes it very unlikely that there is a polynomial algorithm.

Theorem 7.2. 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 5.1, 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 [45], 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. On Fixpoints and Garbage

We will show in this section that $LRLX$-expressions without garbage only have trivial fixpointing permutations. Looking at 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, where $dom((a\ b)) \cap FA(\texttt{letrec}\ c.a; d.b\ \texttt{in}\ True) = \{a, b\} \neq \emptyset$. However, we see that the complete environment in this example is garbage (Def. 7.1).

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.

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

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

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.

A further example of garbage 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 $a.0$ is garbage. Another example is $e := (\texttt{letrec}\, a.d; b.1; c.d \;\texttt{in}\; (f\ b))$, which is an example with a free atom d, and the garbage consists of two bindings, $\{a.d; c.d\}$. It 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 permutations (bijective functions) mapping e' to (the α-equivalent) 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 8.1. 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: Let $s = (\texttt{letrec}\, a_1.e_1,\ldots,a_n.e_n \;\texttt{in}\; e) \sim (\texttt{letrec}\, b_1.f_1,\ldots,b_n.f_n \;\texttt{in}\; f) = t$. Note that due to syntactic equality all permutations of the environments are also to be considered. Then there is bijective mapping φ, with $\varphi(b_i) = a_{\rho(i)}, i = 1,\ldots,n$, where ρ is a permutation on $\{1,\ldots,n\}$, and such that $e_i \sim \varphi(f_{\rho(i)}), i = 1,\ldots,n$, $e \sim \varphi(f)$, and $(\{a_1,\ldots,a_n\} \setminus \{b_1,\ldots,b_n\})\#t$ holds. Let $\overline{\varphi}$ be the atom-permutation that extends φ, mapping $(\{a_1,\ldots,a_n\} \setminus \{b_1,\ldots,b_n\})$ to $(\{b_1,\ldots,b_n\} \setminus \{a_1,\ldots,a_n\})$.

The induction hypothesis implies a unique position correspondence of e and f, since $e \sim \overline{\varphi}(f)$. This implies that the bindings for $\{a_1,\ldots,a_n\} \cap FA(e)$ 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 8.2. 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 $e = f\ e_1 \ldots .e_n$, 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 $t = (\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$, 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 split 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 8.1 we obtain that there is a unique permutation ρ on $\{1, \ldots, n\}$, such that there is an injective mapping $\varphi : \pi(a_1) \mapsto a_{\rho(1)}, \ldots, \pi(a_n) \mapsto a_{\rho(n)}$, 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 $\{a_1, \ldots, a_n\} \setminus \{\pi(a_1), \ldots, \pi(a_n)\}$ bijectively to $\{\pi(a_1), \ldots, \pi(a_n)\} \setminus \{a_1, \ldots, a_n\}$. 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 8.3. 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 8.2. The other direction is easy. □

The proof also shows a slightly more general statement:

Corollary 8.4. Let e be an expression such that in all environments with at least two 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 semantics is defined such that only ground garbage free expressions are permitted, 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:

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

(ElimX) $\dfrac{\Gamma \dot\cup \{X \doteq e\}, \theta}{\Gamma, \theta \cup \{X \mapsto e\}}$, if $X \notin Var(\Gamma)$, and e is not a suspension of X.

Example 8.5. 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:

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

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 7.2, which shows that even nominal matching may have several incomparable solutions for garbage-free expressions.

9. 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 increase the expressive power of a term language with atoms alone. If in an application example it is known that in a pair (x_1, x_2) the expressions x_1, x_2 can only be atoms, but $x_1 = x_2$ as well as $x_1 \neq x_2$ is possible, then two different unification problems have to be formulated. If atom variables are possible, then the notation (A_1, A_2) covers both possibilities.

It is known that the nominal unification problem with atom-variables but without letrec is NP-complete [27]. An algorithm and corresponding rules and discussions can be found in [27]. An implication is NP-hardness of nominal unification with atom variables and letrec.

9.1. Extension with Atom-Variables

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

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

Let $Var(e)$ be the set of atom or expression variables occurring in e, and let $AtVar(e)$ be the set of atom variables occurring in e. Similarly for sequences of expressions or permutations.
The expression $\pi{\cdot}e$ for a non-variable expression e means an operation, which is performed by shifting π down in the expression, using the simplifications $\pi_1{\cdot}(\pi_2{\cdot}X) \rightarrow (\pi_1 \circ \pi_2){\cdot}X$, where only expressions $\pi{\cdot}X$ and $\pi{\cdot}V$ remain, where the latter are called *suspensions* and where $\pi{\cdot}V$ is abbreviated as W.

Remark 9.1. An **alert** for the reader: In this section the use of atom-variables induces generalizations and changes in the $LRLXA$-formulation of problems: binders may now be suspensions of atom-variables, and also "nested" permutation representations are permitted, which is due to atom variables, since in general, this permutation representation cannot be simplified.
Several simple facts and intuitions that are used for $LRLX$ no longer hold.

A *freshness constraint* in our unification algorithm is of the form $V\#e$ where e is an $LRLXA$-expression. The justification for the slightly more complex form as usual ($a\#X$) is that atom variables prevent a simplification to this form. The notation π^{-1} is defined as the reversed list of swappings of π. We also view $\pi \cdot V\#e$ as identical to the constraint $V\#\pi^{-1} \cdot e$.

Naively applying ground substitutions sometimes leads to syntactically invalid ground expressions, since instantiation may make binding atoms in letrecs equal, which is illegal.

Example 9.2. The equation

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

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,

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

is solvable.

Assumption 9.3. We circumvent the problem of illegal ground instances by assuming that for every letrec-expression in the input of unification and matching algorithms there are sufficiently many freshness constraint that prevent these illegal expressions.

Definition 9.4. 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 clarity), 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 $Var(\Gamma, \nabla)$ 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 9.5. The $LRLXA$-unification problem is in NP, and hence NP-complete.

Proof:
The argument is that every ground instantiation of an atom variable is an atom, which can be guessed and checked in polynomial time: guess the images of atom variables under a ground solution ρ in the set of atoms in the current state, or in an arbitrary set of fresh atom variables of cardinality at most the number of different atom variables in the input (i.e. guess the function γ). Then instantiate the problem using γ, thereby removing all atom-variables. The resulting problem can be decided (and solved) by an NP-algorithm as shown in this paper (Theorem 5.2). □

$$(1)\ \frac{\Gamma\dot\cup\{e \doteq e\}}{\Gamma} \qquad (2)\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}V_1 \doteq \pi_2{\cdot}V_2\}, \nabla, \theta}{\Gamma, \nabla \cup \{V_1 =_{\#} \pi_1^{-1}\pi_2{\cdot}V_2\}, \theta}$$

$$(3a)\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}X \doteq \pi_2{\cdot}Y\}, \nabla, \theta \qquad X \neq Y}{\Gamma[\pi_1^{-1}\pi_2{\cdot}Y/X], \nabla[\pi_1^{-1}\pi_2{\cdot}Y/X], \theta \cup \{X \mapsto \pi_1^{-1}\pi_2 Y\}}$$

$$(3b)\ \frac{\Gamma\dot\cup\{\pi_1{\cdot}X \doteq \pi_2{\cdot}V\}, \nabla, \theta}{\Gamma[\pi_1^{-1}\pi_2{\cdot}V/X], \nabla[\pi_1^{-1}\pi_2{\cdot}V/X], \theta \cup \{X \mapsto \pi_1^{-1}\pi_2 V\}}$$

$$(4)\ \frac{\Gamma\dot\cup(f\ (\pi_1{\cdot}X_1)\ldots(\pi_n{\cdot}X_n)) \doteq (f\ (\pi'_1{\cdot}X'_1)\ldots(\pi'_n{\cdot}X'_n))\}}{\Gamma \cup \{\pi_1{\cdot}X_1 \doteq \pi'_1{\cdot}X'_1, \ldots, \pi_n{\cdot}X_n \doteq \pi'_n{\cdot}X'_n\}}$$

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

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

$$(7)\ \frac{\Gamma\dot\cup\left\{\begin{array}{l}\texttt{letrec}\,W_1.\pi_1{\cdot}X_1;\ldots;W_n.\pi_n{\cdot}X_n\ \texttt{in}\ \pi{\cdot}Y \doteq\\ \texttt{letrec}\,W'_1.\pi'_1{\cdot}X'_1;\ldots;W'_n.\pi'_n{\cdot}X'_n\ \texttt{in}\ \pi'{\cdot}Y'\end{array}\right\}, \nabla}{\underset{\forall\rho}{\Big|}\quad\left(\begin{array}{l}\Gamma\cup\left\{\begin{array}{l}\mathrm{decompose}(n{+}1, \lambda W_1\ldots\lambda W_n.(\pi_1{\cdot}X_1,\ldots,\pi_n{\cdot}X_n,\pi{\cdot}Y)\\ \quad\doteq \lambda W'_{\rho(1)}.\ldots\lambda W'_{\rho(n)}.(\pi'_{\rho(1)}{\cdot}X'_{\rho(1)},\ldots,\pi'_{\rho(n)}{\cdot}X'_{\rho(n)},\pi'{\cdot}Y'))\end{array}\right\},\\ \nabla\cup\left\{\begin{array}{l}\mathrm{decompfresh}(n{+}1, \lambda W_1\ldots\lambda W_n.(\pi_1{\cdot}X_1,\ldots,\pi_n{\cdot}X_n,\pi{\cdot}Y)\\ \quad\doteq \lambda W'_{\rho(1)}.\ldots\lambda W'_{\rho(n)}.(\pi'_{\rho(1)}{\cdot}X'_{\rho(1)},\ldots,\pi'_{\rho(n)}{\cdot}X'_{\rho(n)},\pi'{\cdot}Y'))\end{array}\right\}\end{array}\right)}$$

where ρ is a permutation on $\{1,\ldots,n\}$ and $\mathrm{decompose}(n,.)$ is the equation part of n-fold application of rules (4), (5) or (6) and $\mathrm{decomposefresh}(n,.)$ is the freshness constraint part of the n-fold application of rules (4), (5) or (6); (in both cases after flattening).

Figure 6. Standard and decomposition rules with atom variables of LetrecUnifyAV.

Remark 9.6. 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 $V_1 =_{\#} \pi{\cdot}V_2$ as a more readable version of $V_1\#\lambda\pi{\cdot}V_2.V_1$.

9.2. 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 nondeterminism and such that it is an improvement over a brute-force guessing-algorithm that first guesses all atom instances of atom-variables and then uses Algorithm LetrecUnify (see Algorithm 9.13 for such an improvement). The

(MMS), (FPS), (ElimFP) and (Output) are almost the same as the ones in Fig 3.

(MMS) $$\frac{\Gamma \dot\cup \{\pi_1{\cdot}X \doteq e_1, \pi_2{\cdot}X \doteq e_2\}, \nabla}{\Gamma \cup \{\pi_1{\cdot}X \doteq e_1\} \cup \Gamma', \nabla \cup \nabla'},$$ if e_1, e_2 are not suspensions, where Γ' is the set of equations generated by decomposing $\pi_1^{-1}{\cdot}e_1 \doteq \pi_2^{-1}{\cdot}e_2$ using (1)–(7), and where ∇' is the corresponding resulting set of freshness constraints.

(FPS) $$\frac{\Gamma \dot\cup \{\pi_1{\cdot}X \doteq \pi_1'{\cdot}X, \ldots, \pi_n{\cdot}X \doteq \pi_n'{\cdot}X, \pi{\cdot}X \doteq e\}, \theta}{\Gamma \cup \{\pi_1\pi^{-1}{\cdot}e \doteq \pi_1'\pi^{-1}{\cdot}e, \ldots, \pi_n\pi^{-1}{\cdot}e \doteq \pi_n'\pi^{-1}{\cdot}e\}, \theta \cup \{X \mapsto \pi^{-1}{\cdot}e\}},$$
If $X \not\in \mathit{Var}(\Gamma, e)$, and e is not a suspension, and (Cycle) (see Fig.4) is not applicable.

(ElimFP) $$\frac{\Gamma \dot\cup \{\pi_1{\cdot}X \doteq \pi_1'{\cdot}X, \ldots, \pi_n{\cdot}X \doteq \pi_n'{\cdot}X, \pi{\cdot}X \doteq \pi'{\cdot}X\}, \theta}{\Gamma \cup \{\pi_1{\cdot}X \doteq \pi_1'{\cdot}X, \ldots, \pi_n{\cdot}X \doteq \pi_n'{\cdot}X\}, \theta},$$
If $\pi^{-1}\pi' \in \langle \pi_1^{-1}\pi_1, \ldots, \pi_n^{-1}\pi_n \rangle$,
and π_i, π_i', π, π' are ground, i.e. do not contain atom variables.

(Output) $$\frac{\Gamma, \nabla, \theta}{(\theta, \nabla, \Gamma)}$$ if Γ only consists of fixpoint-equations.

(ElimA) $$\frac{\Gamma, \nabla, \theta}{\Gamma[a/A], \nabla[a/A], \theta \cup \{A \mapsto a\}},$$ where we guess the following: some atom variable A occurring in Γ, ∇ and an atom a that occurs in Γ, ∇, θ, or is a fresh atom.

Figure 7. Main rules of LETRECUNIFYAV

simple idea is to only make these guesses if a certain space-bound of the whole state is exceeded and then use the guesses and further rules to shrink the size of the problem representation.

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 of the form $A \doteq e$, in particular $A \doteq \pi{\cdot}A'$, cannot be solved by substitutions ($A \mapsto \pi{\cdot}A'$) 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.

Atoms in the input are permitted. In the rules an extra mention of atoms is only in (2), (3), (ElimFP), (ElimA), (Clashab), (FailF), (FailFS) and in (ElimFP).

Definition 9.7. The algorithm LETRECUNIFYAV operates on a tuple (Γ, ∇, θ), where the rules are defined in Figs. 6 and 7, and failure rules are in Fig. 8. The following explanations are in order:

1. Γ is assumed to be a set of flattened equations $e_1 \doteq e_2$ (see the remarks after Definition 4.3).

2. We assume that $\doteq$ is symmetric,

3. ∇ contains freshness constraints, like $a\#e$, $A\#e$, which in certain cases may be written as equations of the form $A =_{\#} \pi{\cdot}A'$ (see Remark 9.6, for better readability and simplicity).

4. 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.

$$\text{(Clash)}\quad \frac{\Gamma\dot\cup\{s \doteq t\}, \nabla, \theta \quad tops(s) \neq tops(t) \text{ and } s \text{ and } t \text{ are not suspensions}}{\bot}$$

$$\text{(ClashA)}\quad \frac{\{s \doteq t\} \text{ is in } \Gamma, \text{ and} \quad \begin{array}{c} s \text{ is a suspension of an atom or atom variable} \\ \text{and } tops(t) \text{ is a function symbol, } \lambda \text{ or letrec} \end{array}}{\bot}$$

$$\text{(Clashab)}\quad \frac{\Gamma\dot\cup\{a \doteq b\}, \nabla, \theta \qquad a \neq b}{\bot}$$

$$\text{(Cycle)}\quad \frac{\begin{array}{c} \text{If } \pi_1{\cdot}X_1 \doteq s_1, \ldots, \pi_n{\cdot}X_n \doteq s_n \text{ in } \Gamma \text{ where } s_i \text{ are not suspensions} \\ \text{and } X_{i+1} \text{ occurs in } s_i \text{ for } i = 1, \ldots, n-1 \text{ and } X_1 \text{ occurs in } s_n. \end{array}}{\bot}$$

$$\text{(FailF)}\quad \frac{a\#a \in \nabla}{\bot} \qquad \text{(FailFS)}\quad \frac{a\#X \in \nabla \quad \text{and } a \text{ occurs free in } (X\theta)}{\bot}$$

Figure 8. Failure Rules of LETRECUNIFYAV

The final state will be reached, i.e. the output, when Γ only contains fixpoint equations of the form $\pi_1{\cdot}X \doteq \pi_2{\cdot}X$, 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 in the notation of a rule, 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 nondeterministic rules with disjunctive nondeterminism: 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.

We assume that permutations in the algorithm LETRECUNIFYAV are compressed using a grammar-mechanism, as a variation of grammar-compression in [46, 47]. However, we do not mention it in the rules of the algorithm, but we will use it in the complexity arguments.

The use of the iterated decomposition in rule (7) appears clumsy at a first look, however, it is an easy algorithmic representation of the method to define the permutations (with atom variables) in a recursive fashion, where the introduction of permutation variables is avoided.

Definition 9.8. 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 written as $\overline{P}_i$.
- Rules of the form $P_i \rightarrow w_1 \ldots w_n$, $n \geq 1$ where w_i is either a nonterminal or a terminal. At all times $\overline{P}_i \rightarrow \overline{w}_n \ldots \overline{w}_1$ holds, i.e., if a nonterminal is added its inverse is added accordingly. Usually, $n \leq 2$, but also another fixed bound for n is possible.

- Terminal elements are $\emptyset$, $(V_1\ V_2)$.

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

9.3. Arguments for Correctness and Completeness

Let S denote in the following the size of the initial unification problem.

Proposition 9.10. 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 9.11. Extending n times the grammar G 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 executions is polynomial in the size of the initial input. Also the operation of applying a compressed ground 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 [27]. The method is to nondeterministically instantiate atom-variables by atoms, and then checking the freshness constraints, which is in NP (see also Theorem 5.2).

We will show that the algorithm runs in polynomial time by applying (ElimA) following a strategy defined below. There are two rules, which can lead to a size increase of the unification problem if we ignore the size of the permutations: (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 $O(S)$. Disregarding the permutations of only 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 9.12. Let $p(x)$ be some easily computable 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 some variable X, then apply (ElimA) for all $A \in \mathit{AtVar}(\pi_1, \ldots, \pi_k)$. Then immediately apply (ElimFP) exhaustively.

Definition 9.13. The guided version LETRECUNIFYAVB(p) of LETRECUNIFYAV is obtained by replacing (ElimA) with ElimAB(p) where $p(x)$ is some (easily computable) function $\mathbb{R}^+ \to \mathbb{R}^+$, such that $\forall x \in \mathbb{R}^+ : q(x) \geq p(x) \geq x * \log(x)$ 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 9.14. Let Γ, ∇ be a solvable input. For every function $p(x)$ with $\forall x \in \mathbb{R}^+ : p(x) \geq x\log(x)$, the algorithm LETRECUNIFYAVB(p) does not get stuck, and for every intermediate state of the algorithm LETRECUNIFYAVB(p) it holds that the number of fixpoint equations per expression variable is bounded above by $p(S)$.

Proof:
The upper bound of the number of fixpoint equations is proved as follows: Let m be the number of atoms in the original unification problem. The rule (ElimA) (called by (ElimAB)) introduces at most $S - m$ new atoms, which implies at most S atoms at any time. If LETRECUNIFYAVB(p) exceeds its upper space 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 5.2).

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 $X \not\in \mathit{Var}(\Gamma, e)$ can be satisfied. □

Theorem 9.15. 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 5.2, and uses Lemma 9.14 that shows that 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 $(\#\text{Var}, \#\text{Lr}\lambda\text{FA}, \#\text{Eqs}, \#\text{EqNonX})$, which is ordered lexicographically, is as follows:

$\#\text{Var}$ is the number of different variables in Γ,

$\#\text{Lr}\lambda\text{FA}$ is the number of letrec-, λ, function-symbols and atoms in Γ, but not in permutations,

$\#\text{Eqs}$ is the number of equations in Γ, and

$\#\text{EqNonX}$ 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.

	$\#\text{Var}$	$\#\text{Lr}\lambda\text{FA}$	$\#\text{Eqs}$	$\#\text{EqNonX}$
(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 $\#\mathrm{Lr}\lambda\mathrm{FA}$ will never be greater than $2Sp(S)$.

The rule (MMS) strictly decreases $\#\mathrm{Lr}\lambda\mathrm{FA}$, hence $\#\mathrm{Eqs}$, i.e. the number of equations, is bounded by $4S^2p(S)$. The same bound holds for $\#\mathrm{EqNonX}$. 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 9.11 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. □

10. Nominal Letrec Matching with Environment Variables

We extend the language $LRLXA$ by variables E that may encode partial letrec-environments for a nominal matching algorithm, which leads to a larger coverage of nominal matching problems in reasoning about the (snall-step operational) semantics of programming languages.

Example 10.1. Consider as an example a rule (llet-e) of the operational semantics of a functional core language, which merges `letrec`-environments (see [19]): $(\texttt{letrec}\ E_1\ \texttt{in}\ (\texttt{letrec}\ E_2\ \texttt{in}\ X)) \quad \rightarrow \quad (\texttt{letrec}\ E_1; E_2\ \texttt{in}\ X)$. 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 if all free variables in the expression are different from all bound variables.

An alternative that is used for a similar unification task in [35] requires the 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 nominal matching for the extension with environment variables, and leave the investigation of reduction rules and sequences for further work.

Definition 10.2. The grammar for the extended language $LRLXAE$ (**L**et**R**ec **L**anguage e**X**tended with **A**tom variables and **E**nvironment) variables E is:

$$(1)\ \frac{\Gamma\dot{\cup}\{e \trianglelefteq e\}}{\Gamma} \qquad (2)\ \frac{\Gamma\dot{\cup}\{\pi_1{\cdot}A \trianglelefteq a\}, \nabla, \theta}{\Gamma[\pi_1^{-1}{\cdot}a/A], \nabla[\pi_1^{-1}{\cdot}a/A], \theta \cup \{A \mapsto \pi_1^{-1}{\cdot}a\}}$$

$$(3)\ \frac{\Gamma\dot{\cup}\{\pi_1{\cdot}X \trianglelefteq e\}, \nabla, \theta}{\Gamma[\pi_1^{-1}{\cdot}e/X], \nabla[\pi_1^{-1}{\cdot}e/X], \theta \cup \{X \mapsto \pi_1^{-1}{\cdot}e\}} \qquad (4)\ \frac{\Gamma\dot{\cup}\{(f\ e_1 \ldots e_n)) \trianglelefteq (f\ e_1' \ldots e_n')\}}{\Gamma \cup \{e_1 \trianglelefteq e_1', \ldots e_n \trianglelefteq e_n'\}}$$

$$(5)\ \frac{\Gamma\dot{\cup}\{(\lambda a.e_1 \trianglelefteq \lambda a.e_2\}}{\Gamma \cup \{e_1 \trianglelefteq e_2\}} \qquad (6)\ \frac{\Gamma\dot{\cup}\{(\lambda W.e_1 \trianglelefteq \lambda a.e_2, \nabla\}}{\Gamma \cup \{(W\ a){\cdot}e_1 \trianglelefteq e_2, \nabla \cup \{a\#\lambda W.e_1\}}$$

$$(7)\ \frac{\Gamma\dot{\cup}\left\{\begin{array}{l}\texttt{letrec}\, W_1.e_1; \ldots ; W_n.e_n\ \texttt{in}\ e \trianglelefteq \\ \texttt{letrec}\, a_1.e_1'; \ldots ; a_n.e_n'\ \texttt{in}\ e'\end{array}\right\}, \nabla \qquad \text{If the left hand side environment does not contain environment variables.}}{\mathop{\Bigg|}_{\forall\rho} \left(\begin{array}{l}\Gamma \cup \left\{\begin{array}{c}\mathrm{decompose}(n{+}1, \lambda W_1 \ldots \lambda W_n.(e_1, \ldots, e_n, e)) \\ \trianglelefteq \lambda a_{\rho(1)}.\ldots\lambda a_{\rho(n)}.(e_{\rho(1)}', \ldots, e_{\rho(n)}', e'))\end{array}\right\}, \\ \nabla \cup \left\{\begin{array}{c}\mathrm{decompfresh}(n{+}1, \lambda W_1 \ldots \lambda W_n.(e_1, \ldots, e_n, e)) \\ \doteq \lambda a_{\rho(1)}.\ldots\lambda a_{\rho(n)}.e_{\rho(1)}', \ldots, e_{\rho(n)}', e'))\end{array}\right\}\end{array}\right)}$$

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

$$(8)\ \frac{\Gamma\dot{\cup}\{(\texttt{letrec}\, W_1.e_1; \ldots ; E; \ldots ; W_n.e_n\ \texttt{in}\ e) \trianglelefteq \texttt{letrec}\, a_1.e_1'; \ldots ; a_n.e_m'\ \texttt{in}\ e'\}, \nabla, \theta}{\mathop{\Big|}_{\forall\sigma} ((\Gamma \cup \{(\texttt{letrec}\, W_1.e_1; \ldots ; E; \ldots ; W_n.e_n\ \texttt{in}\ e) \trianglelefteq \texttt{letrec}\, a_1.e_1'; \ldots ; a_n.e_m'\ \texttt{in}\ e'\})\sigma, \nabla\sigma, \theta \cup \sigma}$$

where $\sigma = \{E \mapsto A_1.X_1, \ldots A_k.X_k\}$ where A_i, X_i are fresh variables and $k \leq m - n$.

Figure 9. Standard and decomposition matching rules with environment variables of LETRECENVMATCH.

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

We define a nominal matching algorithm, where environment variables may occur (unrestricted) in left hand sides, but not in the right hand sides.

The matching algorithm with environment variables is described below. It can be obtained from the algorithm LETRECUNIFYAV by adding a rule that (nondeterministically) instantiates environment variables by environments of the form $W_1.X_1; \ldots ; W_k.X_k$. This can eliminate all environment variables. After this operation of eliminating all environment variables, it is possible to use simplified and optimized rules of LETRECUNIFYAV. Since the optimizations are not completely obvious, and require also slight modifications of the rules, we make them explicit.

Definition 10.3. The matching algorithm LETRECENVMATCH is described in Fig. 9. Permitted inputs are matching equations between expressions where environment variables E may only occur in the left hand sides of matching equations. The don't know-nondeterminism is indicated in the respective rules.
The result is a substitution and a freshness constraint including atom-constraints and a substitution.

We omit failure rules, since these obviously follow from the nominal matching algorithm. Guessing the number of instances into environment variables may lead to clashes due to a wrong number of bindings in environments. An implementation can be more clever by checking the possible number of bindings before guessing. The constraints can be solved in NP-time by guessing and instantiating the instances of atom-variables.
The rules terminate in polynomial time, since the right hand sides are made smaller, and completeness holds, since all cases are covered. Hence:

Theorem 10.4. The nominal matching algorithm LETRECENVMATCH is sound and complete and runs in NP time.

11. Conclusion and Future Research

We construct nominal unification algorithms for expressions with letrec, for the case where only atoms are permitted, and also for the case where in addition atom variables are permitted. We also describe several nominal letrec matching algorithms for variants, in particular also for expressions with environment variables. All algorithms run in (nondeterministic) polynomial time. Future research is to investigate extensions of nominal unification with environment variables E, perhaps as an extension of the matching algorithm.

Future work is also an investigation into the connection with equivariant nominal unification [15, 16, 17], 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 more deeply investigated.

References

[1] Schmidt-Schauß M, Kutsia T, Levy J, Villaret M. Nominal Unification of Higher Order Expressions with Recursive Let. In: Hermenegildo MV, López-García P (eds.), Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers, volume 10184 of *Lecture Notes in Computer Science*. Springer, 2016 pp. 328–344. doi: 10.1007/978-3-319-63139-4_19. URL https://doi.org/10.1007/978-3-319-63139-4_19.

[2] Baader F, Snyder W. Unification Theory. In: Robinson JA, Voronkov A (eds.), Handbook of Automated Reasoning, pp. 445–532. Elsevier and MIT Press, 2001.

[3] Huet GP. A Unification Algorithm for Typed lambda-Calculus. *Theor. Comput. Sci.*, 1975. **1**(1):27–57. doi:10.1016/0304-3975(75)90011-0.

[4] Goldfarb WD. The Undecidability of the Second-Order Unification Problem. *Theor. Comput. Sci.*, 1981. **13**:225–230. doi:10.1016/0304-3975(81)90040-2.

[5] Levy J, Veanes M. On the Undecidability of Second-Order Unification. *Inf. Comput.*, 2000. **159**(1-2):125–150. doi:10.1006/inco.2000.2877.

[6] Urban C, Pitts AM, Gabbay M. Nominal Unification. In: 17th CSL, 12th EACSL, and 8th KGC, volume 2803 of *LNCS*. Springer, 2003 pp. 513–527. doi:10.1007/978-3-540-45220-1_41.

[7] Urban C, Pitts AM, Gabbay MJ. Nominal unification. *Theor. Comput. Sci.*, 2004. **323**(1–3):473–497. doi:10.1016/j.tcs.2004.06.016.

[8] Calvès C, Fernández M. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 2008. **403**(2-3):285–306. doi:10.1016/j.tcs.2008.05.012.

[9] Levy J, Villaret M. Nominal Unification from a Higher-Order Perspective. *ACM Trans. Comput. Log.*, 2012. **13**(2):10. doi:10.1145/2159531.2159532.

[10] Miller D. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.*, 1991. **1**(4):497–536. doi:10.1093/logcom/1.4.497.

[11] Levy J, Villaret M. An Efficient Nominal Unification Algorithm. In: Lynch C (ed.), Proc. 21st RTA, volume 6 of *LIPIcs*. Schloss Dagstuhl, 2010 pp. 209–226. doi:10.4230/LIPIcs.RTA.2010.209.

[12] Ayala-Rincón M, Fernández M, Rocha-Oliveira AC. Completeness in PVS of a Nominal Unification Algorithm. *ENTCS*, 2016. **323**(3):57–74. doi:10.1016/j.entcs.2016.06.005.

[13] Ayala-Rincón M, de Carvalho Segundo W, Fernández M, Nantes-Sobrinho D. A Formalisation of Nominal α-equivalence with A and AC Function Symbols. *Electr. Notes Theor. Comput. Sci.*, 2017. **332**:21–38. doi:10.1016/j.entcs.2017.04.003.

[14] Ayala-Rincón M, Fernández M, Nantes-Sobrinho D. Nominal Narrowing. In: Pientka B, Kesner D (eds.), Proc. first FSCD, LIPIcs. 2016 pp. 11:1–11:17. doi:10.4230/LIPIcs.FSCD.2016.11.

[15] Cheney J. Equivariant Unification. *J. Autom. Reasoning*, 2010. **45**(3):267–300. doi:10.1007/s10817-009-9164-3.

[16] Cheney J. Nominal Logic Programming. Ph.D. thesis, Cornell University, Ithaca, New York, U.S.A., 2004.

[17] Aoto T, Kikuchi K. A Rule-Based Procedure for Equivariant Nominal Unification. In: Informal proceedings HOR. 2016 p. 5.

[18] Moran AKD, Sands D, Carlsson M. Erratic Fudgets: A semantic theory for an embedded coordination language. In: Coordination '99, volume 1594 of *LNCS*. Springer-Verlag, 1999 pp. 85–102. doi:10.1007/3-540-48919-3_8.

[19] Schmidt-Schauß M, Schütz M, Sabel D. Safety of Nöcker's Strictness Analysis. *J. Funct. Programming*, 2008. **18**(04):503–551. doi:10.1017/S0956796807006624.

[20] Ariola ZM, Klop JW. Cyclic Lambda Graph Rewriting. In: Proc. IEEE LICS. IEEE Press, 1994 pp. 416–425. doi:10.1109/LICS.1994.316066.

[21] Marlow S (ed.). Haskell 2010 – Language Report. 2010. URL `https://www.haskell.org`.

[22] Cheney J. Toward a General Theory of Names: Binding and Scope. In: MERLIN 2005. ACM, 2005 pp. 33–40. doi:10.1145/1088454.1088459.

[23] Urban C, Kaliszyk C. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Log. Methods Comput. Sci.*, 2012. **8**(2). doi:10.2168/LMCS-8(2:14)2012.

[24] Simon L, Mallya A, Bansal A, Gupta G. Coinductive Logic Programming. In: Etalle S, Truszczynski M (eds.), 22nd ICLP, LNCS. 2006 pp. 330–345. doi:10.1007/11799573_25.

[25] Jeannin J, Kozen D, Silva A. CoCaml: Functional Programming with Regular Coinductive Types. *Fundam. Inform.*, 2017. **150**(3-4):347–377. doi:10.3233/FI-2017-1473.

[26] Martelli A, Montanari U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 1982. **4**(2):258–282. doi:10.1145/357162.357169.

[27] Schmidt-Schauß M, Sabel D, Kutz YDK. Nominal unification with atom-variables. *J. Symb. Comput.*, 2019. **90**:42–64. doi:10.1016/j.jsc.2018.04.003.

[28] Schmidt-Schauß M, Sabel D. Nominal Unification with Atom and Context Variables. In: Kirchner [48], 2018 pp. 28:1–28:20. doi:10.4230/LIPIcs.FSCD.2018.28.

[29] Ayala-Rincón M, de Carvalho Segundo W, Fernández M, Nantes-Sobrinho D. Nominal C-Unification. In: Fioravanti F, Gallagher JP (eds.), 27th LOPSTR, Revised Selected Papers, volume 10855 of *LNCS*. Springer, 2017 pp. 235–251. doi:10.1007/978-3-319-94460-9_14.

[30] Ayala-Rincón M, Fernández M, Nantes-Sobrinho D. Fixed-Point Constraints for Nominal Equational Unification. In: Kirchner [48], 2018 pp. 7:1–7:16. doi:10.4230/LIPIcs.FSCD.2018.7.

[31] Schmidt-Schauss M, Kutsia T, Levy J, Villaret M. Nominal Unification of Higher Order Expressions with Recursive Let. RISC Report Series 16-03, RISC, Johannes Kepler University Linz, Austria, 2016.

[32] Baldan P, Bertolissi C, Cirstea H, Kirchner C. A rewriting calculus for cyclic higher-order term graphs. *Mathematical Structures in Computer Science*, 2007. **17**(3):363–406. doi:10.1017/S0960129507006093.

[33] Rau C, Schmidt-Schauß M. A Unification Algorithm to Compute Overlaps in a Call-by-Need Lambda-Calculus with Variable-Binding Chains. In: Proc. 25th UNIF. 2011 pp. 35–41.

[34] Rau C, Schmidt-Schauß M. Towards Correctness of Program Transformations Through Unification and Critical Pair Computation. In: Proc. 24th UNIF, volume 42 of *EPTCS*. 2010 pp. 39–54. doi:10.4204/EPTCS.42.4.

[35] Schmidt-Schauß M, Sabel D. Unification of program expressions with recursive bindings. In: Cheney J, Vidal G (eds.), 18th PPDP. ACM, 2016 pp. 160–173. doi:10.1145/2967973.2968603.

[36] Dowek G, Gabbay MJ, Mulligan DP. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Log. J. IGPL*, 2010. **18**(6):769–822. doi:10.1093/jigpal/jzq006. URL `https://doi.org/10.1093/jigpal/jzq006`.

[37] Schmidt-Schauß M, Rau C, Sabel D. Algorithms for Extended Alpha-Equivalence and Complexity. In: van Raamsdonk F (ed.), 24th RTA 2013, volume 21 of *LIPIcs*. Schloss Dagstuhl, 2013 pp. 255–270. doi: http://dx.doi.org/10.4230/LIPIcs.RTA.2013255.

[38] Luks EM. Permutation Groups and Polynomial-Time Computation. In: Finkelstein L, Kantor WM (eds.), Groups And Computation, volume 11 of *DIMACS*. DIMACS/AMS, 1991 pp. 139–176.

[39] Furst ML, Hopcroft JE, Luks EM. Polynomial-Time Algorithms for Permutation Groups. In: 21st FoCS. IEEE Computer Society, 1980 pp. 36–41. doi:10.1109/SFCS.1980.34.

[40] Picouleau C. Complexity of the Hamiltonian Cycle in Regular Graph Problem. *Theor. Comput. Sci.*, 1994. **131**(2):463–473. doi:10.1016/0304-3975(94)90185-6.

[41] Garey MR, Johnson DS, Tarjan RE. The Planar Hamiltonian Circuit Problem is NP-Complete. *SIAM J. Comput.*, 1976. **5**(4):704–714. doi:10.1137/0205049.

[42] Ariola ZM, Felleisen M, Maraist J, Odersky M, Wadler P. A call-by-need lambda calculus. In: POPL'95. ACM Press, San Francisco, CA, 1995 pp. 233–246. doi:10.1145/199448.199507.

[43] Schöning U. Graph Isomorphism is in the Low Hierarchy. *J. Comput. Syst. Sci.*, 1988. **37**(3):312–323. doi:10.1016/0022-0000(88)90010-4.

[44] Babai L. Graph Isomorphism in Quasipolynomial Time. http://arxiv.org/abs/1512.03547v2, 2016.

[45] Booth KS. Isomorphism Testing for Graphs, Semigroups, and Finite Automata Are Polynomially Equivalent Problems. *SIAM J. Comput.*, 1978. **7**(3):273–279. doi:10.1137/0207023.

[46] Lohrey M, Maneth S, Schmidt-Schauß M. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 2012. **78**(5):1651–1669. doi:10.1016/j.jcss.2012.03.003.

[47] Gascón A, Godoy G, Schmidt-Schauß M. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 2011. **12**(4):26:1–26:37. doi:10.1145/1970398.1970402.

[48] Kirchner H (ed.). 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.