Nominal Unification of Higher Order Expressions with Recursive Let*

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

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

Abstract. A sound and complete algorithm for nominal unification of higher-order expressions with a recursive let is described, and shown to run in non-deterministic polynomial time. We also explore specializations like nominal letrec-matching for plain expressions and for DAGs and determine their complexity.

Keywords: Nominal unification, lambda calculus, higher-order expressions, recursive let, operational semantics

1 Introduction

Unification [7] 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. [18]), however, the problem is undecidable [17, 20].

Our motivation comes from syntactical reasoning on higher-order expressions, with equality being alpha-equivalence of expressions, and where a unification algorithm is demanded as a basic service. Nominal unification is the extension of first-order unification with abstractions. It unifies expressions w.r.t. alpha-equivalence, and employs permutations as a clean treatment of renamings. It is known that nominal unification is decidable [35, 36], where the complexity of the decision problem is polynomial time [9]. It can be seen also from a higher-order perspective [12, 22], as equivalent to Miller's higher-order pattern unification [26]. There are efficient algorithms [9, 21], formalizations of nominal unification [6], formalizations with extensions to commutation properties within expressions [4],

* This research has been partially founded by the MINECO/FEDER projects RASO (TIN2015-71799-C2-1-P) and LoCoS (TIN2015-66293-R) and the UdG project MPCUdG2016/055.

and generalizations of nominal unification to narrowing [5]. Equivariant (nominal) unification [14, 10, 11, 1] extends nominal unification by permutation-variables, but it can also be seen as a generalization of nominal unification by permitting abstract names for variables.

We are interested in unification w.r.t. an additional extension with cyclic let. To the best of our knowledge, there is no nominal unification algorithm for higher-order expressions permitting general binding structures like a cyclic let.

The motivation and intended application scenario is as follows: constructing syntactic reasoning algorithms for showing properties of program transformations on higher-order expressions in call-by-need functional languages (see for example [27, 31]) that have a letrec-construct (also called cyclic let) [3] as in Haskell [24], (see e.g. [13] for a discussion on reasoning with more general name binders, and [34] for a formalization of general binders in Isabelle). There may be applications also to coinductive extensions of logic programming [33] and strict functional languages [19]. Basically, overlaps of expressions have to be computed (a variant of critical pairs) and reduction steps (under some strategy) have to be performed. To this end, first an expressive higher-order language is required to represent the meta-notation of expressions. For example, the meta-notation $((\lambda x.e_1)\ e_2)$ for a beta-reduction is made operational by using unification variables X_1, X_2 for e_1, e_2. The scoping of X_1 and X_2 is different, which can be dealt with by nominal techniques. In fact, a more powerful unification algorithm is required for meta-terms employing recursive letrec-environments.

Our main algorithm LetrecUnify is derived from first-order unification and nominal unification: From first-order unification we borrowed the decomposition rules, and the sharing method from Martelli-Montanari-style unification algorithms [25]. The adaptations of decomposition for abstractions and the advantageous use of permutations of atoms is derived from nominal unification algorithms. Decomposing letrec-expression requires an extension by a permutation of the bindings in the environment, where, however, one has to take care of scoping. Since in contrast to the basic nominal unification, there are nontrivial fixpoints of permutations (see Example 2.2), novel techniques are required and lead to a surprisingly moderate complexity: a fixed-point shifting rule (FPS) and a redundancy removing rule (ElimFP) together 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 3.6) for obtaining efficiency.

Results: The nominal letrec unification algorithm LetrecUnify is complete and runs in nondeterministic polynomial time (Theorem 4.1, 4.3). The nominal letrec matching is NP-complete (Theorems 5.2, 6.1), as well as the nominal unification problem (Theorems 4.3, 6.1). Nominal letrec matching for dags is in NP and outputs substitutions only (Theorem 5.4), and a very restricted nominal letrec matching problem is graph-isomorphism hard (Theorem 6.3). Nominal matching including letrec-environment variables is in NP (Theorem 7.4).

2 The Ground Language of Expressions

The very first idea of nominal techniques [35] is to use concrete variable names in lambda-calculi (also in extensions), in order to avoid implicit α-renamings, and instead uses operations for explicitly applying α-renaming operations. Suppose $s = \lambda\mathtt{xx}.\mathtt{xx}$ and $t = \lambda\mathtt{yy}.\mathtt{yy}$ are concrete (syntactically different) lambda-expressions. The nominal technique provides explicit name-changes using permutations. These permutations are applied irrespective of binders. For example $(\mathtt{xx}\ \mathtt{yy})(\lambda\mathtt{xx}.\lambda\mathtt{xx}.\mathtt{a})$ results in $\lambda\mathtt{yy}.\lambda\mathtt{yy}.\mathtt{a}$. Syntactic reasoning on higher-order expressions, for example unification of higher-order expressions modulo α-equivalence will be done by nominal techniques on a language with concrete names, where the algorithms require certain extra constraints and operations. The gain is that all conditions and substitutions etc. can be computed and thus more reasoning tasks can be automated, whereas the implicit name conditions under implicit α-equivalence has a tendency to complicate (unification-) algorithms and to hide the required conditions on equality/disequality/occurrence/non-occurrence of names.

2.1 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 [35]. 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:

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

We also use tuples, which are written as $(e_1, \ldots, e_n)$, and which are treated as functional expressions in the language. We assume that binding atoms $a_1, \ldots, a_n$ in a letrec-expression $(\mathtt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \mathtt{in}\ e)$ are pairwise distinct. Sequences of bindings $a_1.e_1; \ldots; a_n.e_n$ are abbreviated as env.

The *scope* of atom a in $\lambda a.e$ is standard: a has scope e. The $\mathtt{letrec}$-construct has a special scoping rule: in $(\mathtt{letrec}\ a_1.s_1; \ldots; a_n.s_n\ \mathtt{in}\ r)$, every free atom a_i in some s_j or r is bound by the environment $a_1.s_1; \ldots; a_n.s_n$. This defines the notion of free atoms $FA(e)$, bound atoms $BA(e)$ in expression e, and all atoms $AT(e)$ in e. For an environment $env = \{a_1.e_1, \ldots, a_n.e_n\}$, we define the set of letrec-atoms as $LA(env) = \{a_1, \ldots, a_n\}$. We say *$a$ is fresh for e* iff $a \notin FA(e)$ (also denoted as $a\#e$). As an example, the expression $(\mathtt{letrec}\ f = cons\ s_1\ g; g = cons\ s_2\ f\ \mathtt{in}\ f)$ represents an infinite list $(cons\ s_1\ (cons\ s_2\ (cons\ s_1\ (cons\ s_2\ \ldots))))$, where s_1, s_2 are expressions. However, since our language *LRL* is only a fragment of core calculi [27, 31], the reader may find more programming examples there.

We will use mappings on atoms from $\mathbb{A}$. A *swapping* $(a\ b)$ is a function that maps an atom a to atom b, atom b to a, and is the identity on other atoms. We will also use finite permutations on atoms from $\mathbb{A}$, which are represented as a composition of swappings in the algorithms below. Let $dom(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq$

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

We will use the following definition of α-equivalence:

Definition 2.1. *The equivalence $\sim$ on expressions $e \in LRL$ is defined as follows:*

- $a \sim a$.
- *if $e_i \sim e_i'$ for all i, then $fe_1 \ldots e_n \sim fe_1' \ldots e_n'$ for an n-ary $f \in \mathcal{F}$.*
- *If $e \sim e'$, then $\lambda a.e \sim \lambda a.e'$.*
- *If $a\#e'$, $e \sim (a\ b) \cdot e'$, then $\lambda a.e \sim \lambda b.e'$.*
- $\texttt{letrec}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e_0 \sim \texttt{letrec}\ a_1'.e_1'; \ldots; a_n'.e_n'\ \texttt{in}\ e_0'$ *iff there is some permutation ρ on $\{1, \ldots, n\}$, such that $\lambda a_1. \ldots .\lambda a_n.(e_1, \ldots, e_n, e_0) \sim \lambda a_{\rho(1)}'. \ldots .\lambda a_{\rho(n)}'.(e_{\rho(1)}', \ldots, e_{\rho(n)}', e_0')$.* □

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

We need fixpoint sets of permutations π: We define $Fix(\pi) = \{e \mid \pi \cdot e \sim e\}$. In usual nominal unification, these sets can be characterized by using freshness constraints [35]. 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 2.2. The α-equivalence $(a\ b) \cdot (\texttt{letrec}\ c.a; d.b\ \texttt{in}\ True) \sim (\texttt{letrec}\ c.a; d.b\ \texttt{in}\ True)$ holds, which means that there are expressions t in LRL with $t \sim (a\ b) \cdot t$ and $FA(t) = \{a, b\}$. This is in contrast to usual nominal unification.

In the following we will use the results on complexity of operations in permutation groups, see [23], and [15]. We consider a set $\{a_1, \ldots, a_n\}$ of distinct objects (in our case the atoms), the symmetric group $\Sigma(\{a_1, \ldots, a_n\})$ (of size $n!$) of permutations of the objects, and consider its elements, subsets and subgroups. Subgroups are always represented by a set of generators (represented as permutations on $\{a_1, \ldots, a_n\}$). If H is a set of elements (or generators), then $\langle H \rangle$ denotes the generated subgroup. Some facts are:

- A permutation can be represented in space linear in n.
- Every subgroup of $\Sigma(\{a_1, \ldots, a_n\})$ can be represented by $\leq n^2$ generators.

However, elements in a subgroup may not be representable as a product of polynomially many generators.
The following questions can be answered in polynomial time:

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

However, intersection of groups and set-stabilizer (i.e. $\{\pi \in G \mid \pi(M) = M\}$) are not known to be computable in polynomial time, since those problems are as hard as graph-isomorphism (see [23]).

3 A Nominal Letrec Unification Algorithm

As an extension of *LRL*, there is also a countably infinite set of (unification) variables X, Y also denoted perhaps using indices.

The syntax of the language *LRLX* (**L**et**R**ec **L**anguage e**X**tended) is

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

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

The expression $\pi{\cdot}e$ for a non-variable e means an operation, which is performed by shifting π down, using the simplification $\pi_1{\cdot}(\pi_2{\cdot}X) \to (\pi_1 \circ \pi_2){\cdot}X$, apply it to atoms, where only expressions $\pi \cdot X$ remain, which are called *suspensions.*

A *freshness constraint* in our unification algorithm is of the form $a\#e$, where e is an *LRLX*-expression, and an *atomic* freshness constraint is of the form $a\#X$.

Definition 3.1 (Simplification of Freshness Constraints).

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

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

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

Definition 3.2. *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, permitting LRLX-expressions. A* (ground) solution *of (Γ, ∇) is a substitution ρ (mapping variables in $\mathit{Var}(\Gamma, \nabla)$ to ground expressions), such that $s_i\rho \sim t_i\rho$ for $i = 1, \ldots, n$ and for all $a\#e \in \nabla$: $a \notin \mathit{FA}(e\rho)$ holds.*
The decision problem is whether there is a solution for given (Γ, ∇).

Definition 3.3. *Let (Γ, ∇) be an LRLX-unification problem. We consider triples $(\sigma, \nabla', \mathcal{X})$, where σ is a substitution (compressed as a dag) mapping variables to LRLX-expressions, ∇' is a set of freshness constraints, and $\mathcal{X}$ is a set of fixpoint constraints of the form $X \in \mathit{Fix}(\pi)$, where $X \notin \mathit{dom}(\sigma)$. A triple $(\sigma, \nabla', \mathcal{X})$ is a* unifier *of (Γ, ∇), if (i) there exists a ground substitution ρ that solves $(\nabla'\sigma, \mathcal{X})$, i.e., for every $a\#e$ in ∇', $a\#e\sigma\rho$ is valid, and for every constraint $X \in \mathit{Fix}(\pi)$ in $\mathcal{X}$, $X\rho \in \mathit{Fix}(\pi)$; and (ii) for every ground substitution ρ that instantiates all variables in $Var(\Gamma, \nabla)$ which solves $(\nabla'\sigma, \mathcal{X})$, the ground substitution $\sigma\rho$ is a solution of (Γ, ∇). A set M of unifiers is* complete*, if every solution μ is covered by at least one unifier, i.e. there is some unifier $(\sigma, \nabla', \mathcal{X})$ in M, and a ground substitution ρ, such that $X\mu \sim X\sigma\rho$ for all $X \in \mathit{Var}(\Gamma, \nabla)$.* □

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

Since we want to avoid the exponential size explosion of the Robinson-style unification algorithms, keeping the good properties of Martelli Montanari-style unification algorithms [25], but not their notational overhead, we stick to a set of equations as data structure. As a preparation for the algorithm, all expressions in equations are exhaustively flattened as follows: $(f\ t_1 \ldots t_n) \to (f\ X_1 \ldots X_n)$ plus the equations $X_1 \doteq t_1, \ldots, X_n \doteq t_n$. Also $\lambda a.s$ is replaced by $\lambda a.X$ with equation $X \doteq s$, and $(\mathtt{letrec}\ a_1.s_1; \ldots, a_n.s_n\ \mathtt{in}\ r)$ is replaced by $(\mathtt{letrec}\ a_1.X_1; \ldots, a_n.X_n\ \mathtt{in}\ X)$ with the additional equations $X_1 \doteq s_1; \ldots; X_n \doteq s_n; X \doteq r$. The introduced variables are always fresh ones. We may denote the resulting set of equations of flattening an equation eq as $\mathit{flat}(eq)$. Thus, all expressions in equations are of depth at most 1, where we do not count the permutation applications in the suspensions.

A dependency ordering on $\mathit{Var}(\mathit{\Gamma})$ is required: If $X \doteq e$ is in $\mathit{\Gamma}$, and e is not a variable nor a suspension and $X \neq Y \in \mathit{Var}(e)$, then $X \succ_{vd} Y$. Let $>_{vd}$ be the transitive closure of $\succ_{vd}$. This ordering is only used, if no standard rules and no failure rules (see Def. 3.4) apply, hence if $>_{vd}$ is used in rule, there are no cycles.

3.1 Rules of the Algorithm LetrecUnify

LetrecUnify operates on a tuple $(\mathit{\Gamma}, \nabla, \theta)$, where $\mathit{\Gamma}$ is a set of flattened equations $e_1 \doteq e_2$, where we assume that $\doteq$ is symmetric, ∇ contains freshness constraints, θ represents the already computed substitution as a list of replacements of the form $X \mapsto e$. Initially θ is empty. The final state will be reached, i.e. the output, when $\mathit{\Gamma}$ only contains fixpoint equations of the form $X \doteq \pi{\cdot}X$ that are non-redundant, and the rule (Output) fires.

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

Standard (1,2,3,3') and decomposition rules (4,5,6,7):

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

$$(3)\frac{\mathit{\Gamma} \dot\cup \{X \doteq \pi{\cdot}Y\}, \nabla, \theta \qquad X \neq Y}{\mathit{\Gamma}[\pi{\cdot}Y/X], \nabla[\pi{\cdot}Y/X], \theta \cup \{X \mapsto \pi{\cdot}Y\}} \qquad (3')\frac{\mathit{\Gamma} \dot\cup \{X \doteq Y\}, \nabla, \theta \qquad X \neq Y}{\mathit{\Gamma}[Y/X], \nabla[Y/X], \theta \cup \{X \mapsto Y\}}$$

$$(4)\ \frac{\mathit{\Gamma} \dot\cup (f\ s_1 \ldots s_n) \doteq (f\ s'_1 \ldots s'_n)\}}{\mathit{\Gamma} \dot\cup \{s_1 \doteq s'_1, \ldots, s_n \doteq s'_n\}}$$

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

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

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

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

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

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

if X is maximal w.r.t. $>_{vd}$, $X \notin \mathit{Var}(\mathit{\Gamma})$, and e is neither a variable nor a suspension, and no failure rule (see below) is applicable.

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

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

We assume that the rule (ElimFP) will be applied whenever possible.

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 3.6). The rule (ElimFP) will limit the number of fixpoint equations by exploiting knowledge on operations on permutation groups.

The rule (Output) terminates an execution on $\mathit{\Gamma}_0$ by outputting a unifier $(\theta, \nabla', \mathcal{X})$. Note that in any case at least one solution is represented.

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

Definition 3.4. Failure Rules of LetrecUnify

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

Cycle Detection: *If there are equations $X_1 \doteq s_1,\ldots,X_n \doteq s_n$ where $\mathit{tops}(s_i) \in \mathcal{F}^x$, and X_{i+1} occurs in s_i for $i = 1,\ldots,n-1$ and X_1 occurs in s_n.*

Freshness Fail: *If there is a freshness constraint $a\#a$.*

Freshness Solution Fail: *If there is a freshness constraint $a\#X \in \nabla$, and $a \in \mathit{FA}((X)\theta)$.*

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

Example 3.5. We illustrate the letrec-rule by a ground example without flattening. Let the equation be: $\texttt{letrec}\ a.(a,b), b.(a,b)\ \texttt{in}\ b \doteq \texttt{letrec}\ b.(b,c), c.(b,c)\ \texttt{in}\ c$). Select the identity permutation ρ, which results in:

$$\lambda a.\lambda b.((a,b),(a,b),b) \doteq \lambda b.\lambda c.((b,c),(b,c),c). \quad \text{Then:}$$
$$\lambda b.((a,b),(a,b),b) \doteq (a\ b){\cdot}\lambda c.((b,c),(b,c),c) = \lambda c.((a,c),(a,c),c).$$

(The freshness constraint $a\#\ldots$ holds). Then the application of the λ-rule gives $((a,b),(a,b),b) \doteq (b\ c){\cdot}((a,c),(a,c),c)$ (the freshness constraint $b\#\ldots$ holds). The resulting equation is $((a,b),(a,b),b) \doteq ((a,b),(a,b),b)$, which is valid.

Example 3.6. This example shows that FPS (together with the standard and decomposition rules) may give rise to an exponential number of equations in the size of the original problem. Let there be variables $X_i, i = 0,\ldots,n$ and the equations $\Gamma = \{X_n \doteq \pi{\cdot}X_n,\ X_n \doteq (f\ X_{n-1}\ \rho_n{\cdot}X_{n-1}),\ldots,X_2 \doteq (f\ X_1\ \rho_2{\cdot}X_1)\}$ where $\pi,\rho_1,\ldots,\rho_n$ are permutations.

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

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

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

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

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

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

4 Soundness, Completeness, and Complexity of LetrecUnify

Theorem 4.1. *The decision variant of the algorithm* LetrecUnify *runs in nondeterministic polynomial time. Its collecting version returns a complete set of at most exponentially many unifiers, every one represented in polynomial space.*

Proof. Note that we assume that the input equations are flattened before applying the rules, which can be performed in polynomial time.

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

The termination measure $(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6)$, which is ordered lexicographically, is as follows: μ_1 is the number of letrec expressions in $\mathit{\Gamma}$, μ_2 is the number of letrec-, λ-symbols, function-symbols and atoms in $\mathit{\Gamma}$, μ_3 is the number of different variables in $\mathit{\Gamma}$, μ_4 is the number of occurrences of variables in $\mathit{\Gamma}$, μ_5 is the number of equations not of the form $X \doteq e$, and μ_6 is the number of equations.

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

Now it is easy to check that the rule applications strictly decrease μ: The rules (MMS) and (FPS) together with the subsequent decomposition strictly decrease (μ_1, μ_2). Since expressions in equations are flat, (MMS) does not increase the size: $X \doteq s_1, X \doteq s_2$ is first replaced by $X \doteq s_1, s_1 \doteq s_2$, and the latter is decomposed, which due to flattening results only in equations containing variables and suspensions. Thus μ_2 is reduced by the size of s_2. In the same way (FPS) strictly decreases (μ_1, μ_2). In addition μ_2 is at most S^2, since only the letrec-decomposition rule can add $\lambda a.$-constructs.

The number of fixpoint-equations for every variable X is at most $c_1 * S * \mathit{log}(S))$ for some (fixed) c_1, since the number of atoms is never increased, and since we assume that (ElimFP) is applied whenever possible. The size of the permutation group is at most $S!$, and so the length of proper subset-chains and hence the maximal number of generators of a subgroup is at most $\log(S!) = O(S{*}\mathit{log}(S))$. Note that the redundancy of generators can be tested in polynomial time depending on the number of atoms.

Now we prove a (global) upper bound on the number μ_3 of variables: An application of (7) may increase μ_3 at most by S. An application of (FPS) may increase this number at most by $c_1 * S \log(S) * S$, where the worst case occurs when e is a letrec-expression. Since (MMS) and (FPS) can be applied at most S times, the number of variables is smaller than $c_1 * S^3 \log(S)$.

The other rules strictly decrease (μ_1, μ_2), or they do not increase (μ_1, μ_2), and strictly decrease $(\mu_3, \mu_4, \mu_5, \mu_6)$ and can be performed in polynomial time. □

The problematic rule for complexity is (FPS), which does not increase μ_1 and μ_2, but may increase μ_3, μ_4 and μ_6 (see Example 3.6). This increase is defeated by the rule (ElimFP), which helps to keep the numbers μ_4 and μ_6 low.

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

Proof. Soundness of the algorithm holds, by easy arguments for every rule, similar as in [35], and since the letrec-rule follows the definition of $\sim$ in Def. 2.1. A further argument is that the failure rules are sufficient to detect final states without solutions.

Completeness requires more arguments. The decomposition and standard rules (with the exception of rule (7)), retain the set of solutions. The same for (MMS), (FPS), and (ElimFP). The nondeterministic Rule (7) provides all possibilities for potential ground solutions. Moreover, the failure rules are not applicable to states that are solvable.

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

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

Proof. This follows from Theorems 4.1 and 4.2.

5 Nominal Matching with Letrec: LetrecMatch

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

Example 5.1. Consider the (lbeta)-rule, which is the version of (beta) used in call-by-need calculi with sharing [2, 27, 31]. Note that only the sharing power of the recursive environment is used here.

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

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

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

The rules of the algorithm LetrecMatch are:

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

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

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

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

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

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

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

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

Proof. This follows by standard arguments.

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

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

A slightly more general situation for nominal 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 (plain) flattened expressions by encoding the dag-nodes as variables together with an 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 :

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

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 $\mathit{\Gamma}_0$. Since $\mathit{\Gamma}_0$ is a matching problem, the result will be free of fixpoint equations, and there will be no freshness constraints in the solution. Thus we have:

Theorem 5.4. *The collecting variant of* LetrecDagMatch *outputs an at most exponential set of dag-compressed substitutions that is complete and where every unifier is represented in polynomial space.*

6 Hardness of Nominal Letrec Matching and Unification

Theorem 6.1. *Nominal letrec matching (hence also unification) is NP-hard, for two letrec expressions, where subexpressions are free of letrec.*

Proof. We encode the NP-hard problem of finding a Hamiltonian cycle in a regular graph [28, 16], which are graphs where all nodes have the same degree $k \geq 3$. Let $a_1, \ldots, a_n$ be the vertexes of the graph, and E be the set of edges. The first environment part is $env_1 = a_1.(node\ a_1); \ldots; a_n.(node\ a_n)$, and a second environment part env_2 consists of bindings $b.(f\ a\ a')$ and $b'.(f\ a'\ a)$ for every edge $(a, a') \in E$ for fresh names b, b'. Then let $s := (\texttt{letrec}\ env_1; env_2\ \texttt{in}\ 0)$ representing the graph. Let the second expression encode the question whether there is a Hamiltonian cycle in a regular graph as follows. The first part of the environment is $env'_1 = a_1.(node\ X_1), \ldots, a_n.(node\ X_n)$. The second part is env'_2 consisting of $b_1.f\ X_1\ X_2; b_2.f\ X_2\ X_3; \ldots b_n.f\ X_n\ X_1$, and the third part consisting of a number of (dummy) entries of the form $b.f\ Z\ Z'$, where b is always a fresh atom for every binding, and Z, Z' are fresh variables for every entry. The number of these dummy entries is $k * n - n$. Then the matching problem is solvable iff the graph has a Hamiltonian cycle.

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

Proof. This follows from Theorems 4.3 and 6.1.

We say that an expression t *contains garbage*, iff there is a subexpression $(\texttt{letrec}\ env\ \texttt{in}\ r)$, and the environment env can be split into two environments $env = env_1; env_2$, such that env_1 is not trivial, and the atoms from $LA(env_1)$ do not occur in env_2 nor in r. Otherwise, the expression is *free of garbage.* Since α-equivalence of *LRL*-expressions is Graph-Isomorphism-complete [29], 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 [32]. There is also a claim that it is quasi-polynomial [8], which means that it requires less than exponential time.

Theorem 6.3. *Nominal letrec matching with one occurrence of a single variable and a garbage-free target expression is Graph-Isomorphism-hard.*

Proof. Let G_1, G_2 be two graphs. Let s be (`letrec` env_1 `in` $g\ b_1 \ldots, b_m$) the encoding of a arbitrary graph G_1 where env_1 is the encoding as in the proof of Theorem 6.1 and, nodes are encoded as $a_1 \ldots a_n$, and the edge-binders are b_i. Then the expression s is free of garbage. Let the environment env_2 be the encoding of G_2 in the expression $t =$ `letrec` env_2 `in` X. Then t matches s iff the graphs G_1, G_2 are isomorphic. Hence we have GI-hardness. If there is an isomorphism of G_1 and G_2, then it is easy to see that this bijection leads to an equivalence of the environments, and we can instantiate X with ($g\ b_1 \ldots, b_m$).

7 Nominal Letrec Matching with Environment Variables

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

Example 7.1. Consider as an example a rule (llet-e) of the operational semantics, that merges `letrec`-environments (see [31]):
(`letrec` E_1 `in` (`letrec` E_2 `in` X)) $\rightarrow$ (`letrec` $E_1; E_2$ `in` X).
It can be applied to an expression (`letrec` $a.0; b.1$ `in` `letrec` $c.(a, b, c)$ `in` c) as follows: The left-hand side (`letrec` E_1 `in` (`letrec` E_2 `in` X)) of the reduction rule matches (`letrec` $a.0; b.1$ `in` (`letrec` $c.(a, b, c)$ `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 (`letrec` $E_1; E_2$ `in` X), which is: (`letrec` $a.0; b.1; c.(a, b, c)$ `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, which means that all bound variables are different and that all free variables in the expression are different from all bound variables.
An alternative that is used for a similar unification task in [30] requires an additional construct of non-capture constraints: $NCC(env_1, env_2)$, which means that for every valid instantiation ρ: variables occurring free in $env_1\rho$ are not captured by the top letrec-binders in $env_2\rho$. In this paper we focus on matching, and leave the combination with reduction rules for further work.

Definition 7.2. *The grammar for the extended language LRLXE (**L**et**R**ec **L**anguage e**X**tended with **E**nvironments) is:*

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

We define a matching algorithm, where environment variables may occur in left hand sides. This algorithm needs a more expressive data structure in equations. The variant `letr*` of `letrec` is used with two environment-components, (i) a list of bindings that are already fixed in the correspondence to the bindings of the other environment, and (ii) an environment that is not yet fixed. We denote the fixed bindings as a list, which is the first component. The scoping is the same. In the notation we assume that the (non-fixed) letrec-environment part on the right hand side may be arbitrarily permuted before the rules are applied. The justification for this special data structure is the scoping in letrec expressions. The usual letrec is the extended letrec with an empty list as first component. Note that suspensions ($\pi{\cdot}E$, $\pi{\cdot}X$) are not generated nor a part of the result of this matching algorithm (but may be in the input).

Definition 7.3. *The matching algorithm* LetrecEnvMatch *for expressions where environment variables E and expression variables X may occur only in the left hand sides of match equations is described below. Initially, every* ($\mathtt{letrec}$ *env* $\mathtt{in}$ *e*) *is modified to* ($\mathtt{letr*}\ \emptyset; env\ \mathtt{in}\ e$)*. The rules are:*

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

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

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

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

$$\frac{\mathit{\Gamma}\dot\cup\{(\mathtt{letr*}\ ls; \pi{\cdot}E; env\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letr*}\ ls'; env'_1; env'_2\ \mathtt{in}\ r')\}}{\underset{env'_1}{|} \quad \mathit{\Gamma}\dot\cup\{(\mathtt{letr*}\ (E : ls); env\ \mathtt{in}\ r) \trianglelefteq (\mathtt{letr*}\ (\pi^{-1}{\cdot}(env'_1) : ls');\ env'_2\ \mathtt{in}\ r')\}}$$

$$\frac{\mathit{\Gamma}\dot\cup\left\{\begin{array}{l}(\mathtt{letr*}\ ls; \emptyset\ \mathtt{in}\ r)\\ \trianglelefteq (\mathtt{letr*}\ ls'; \emptyset\ \mathtt{in}\ r')\end{array}\right\}}{\mathit{\Gamma}\dot\cup\{ls \trianglelefteq ls'; r \trianglelefteq r'\}} \qquad \frac{\mathit{\Gamma}\dot\cup\{[e_1;\dots;e_n] \trianglelefteq [e'_1;\dots;e'_n]\}}{\mathit{\Gamma}\dot\cup\{e_1 \trianglelefteq e'_1;\dots;e_n \trianglelefteq e'_n\}}$$

$$\frac{\mathit{\Gamma}\dot\cup\{\pi{\cdot}X \trianglelefteq e\}}{\mathit{\Gamma}\dot\cup\{X \trianglelefteq \pi^{-1}e\}} \qquad \frac{\mathit{\Gamma}\dot\cup\{X \trianglelefteq e_1, X \trianglelefteq e_2\}}{\mathit{\Gamma}\dot\cup\{X \trianglelefteq e_1, e_1 \doteq e_2\}} \qquad \frac{\mathit{\Gamma}\dot\cup\{E \trianglelefteq env_1, E \trianglelefteq env_2\}}{\mathit{\Gamma}\dot\cup\{E \trianglelefteq env_1, env_1 \doteq env_2\}}$$

Testing $e_1 \doteq e_2$ and $env_1 \doteq env_2$ is done with high priority using the (non-deterministic) matching rules in Section 5, where for testing $env_1 \doteq env_2$ all permutations of the bindings are checked. Fail, if the equations does not hold.

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

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

Theorem 7.4. *The algorithm 7.3 (*LetrecEnvMatch*) is sound and complete. It runs in non-deterministic polynomial time. The corresponding decision problem is NP-complete. The collecting version of* LetrecEnvMatch *returns an at most exponentially large, complete set of representations of matching substitutions, where the representations are of at most polynomial size.*

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

8 Conclusion and Future Research

We constructed a nominal letrec unification algorithm, several nominal letrec matching algorithms for variants, which all run in nondeterministic polynomial time. Future research is to investigate extensions of unification with environment variables E, with abstract variables for (concrete) variables, (or alternatively extending equivariant nominal unification [14, 10, 11, 1] to letrec,) and to investigate nominal matching together with equational theories. Also applications of nominal techniques to reduction steps in operational semantics and transformations should be investigated.

References

1. Aoto, T., Kikuchi, K.: A rule-based procedure for equivariant nominal unification. In: informal proceedings HOR. p. 5 (2016)
2. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: A call-by-need lambda calculus. In: POPL'95. pp. 233–246. ACM Press, San Francisco, CA (1995)
3. Ariola, Z.M., Klop, J.W.: Cyclic Lambda Graph Rewriting. In: Proc. IEEE LICS. pp. 416–425. IEEE Press (1994)
4. Ayala-Rincón, M., de Carvalho-Segundo, W., Fernández, M., Nantes-Sobrinho, D.: A formalisation of nominal alpha-equivalence with A and AC function symbols. In: Proc. LSFA 2016. pp. 78–93 (2016)
5. Ayala-Rincón, M., Fernández, M., Nantes-Sobrinho, D.: Nominal narrowing. In: Pientka, B., Kesner, D. (eds.) Proc. first FSCD. pp. 11:1–11:17. LIPIcs (2016)
6. Ayala-Rincón, M., Fernández, M., Rocha-Oliveira., A.C.: Completeness in pvs of a nominal unification algorithm. ENTCS 323(3) (2016), to appear
7. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 445–532. Elsevier and MIT Press (2001)
8. Babai, L.: Graph isomorphism in quasipolynomial time. Available from http://arxiv.org/abs/1512.03547v2 (2016)
9. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theor. Comput. Sci. 403(2-3), 285–306 (2008)
10. Cheney, J.: Nominal Logic Programming. Ph.D. thesis, Cornell University, Ithaca, New York, U.S.A. (2004)
11. Cheney, J.: Equivariant unification. In: Giesl, J. (ed.) Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3467, pp. 74–89 (2005), http://dx.doi.org/10.1007/978-3-540-32033-3_7
12. Cheney, J.: Relating higher-order pattern unification and nominal unification. In: Proc. 19th International Workshop on Unification, UNIF'05. pp. 104–119 (2005)
13. Cheney, J.: Toward a general theory of names: Binding and scope. In: MERLIN 2005. pp. 33–40. ACM (2005)
14. Cheney, J.: Equivariant unification. J. Autom. Reasoning 45(3), 267–300 (2010), http://dx.doi.org/10.1007/s10817-009-9164-3

15. Furst, M.L., Hopcroft, J.E., Luks, E.M.: Polynomial-time algorithms for permutation groups. In: 21st FoCS. pp. 36–41. IEEE Computer Society (1980)
16. Garey, M.R., Johnson, D.S., Tarjan, R.E.: The planar Hamiltonian circuit problem is NP-complete. SIAM J. Comput. 5(4), 704–714 (1976)
17. Goldfarb, W.D.: The undecidability of the second-order unification problem. Theoretical Computer Science 13, 225–230 (1981)
18. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. 1(1), 27–57 (1975)
19. Jeannin, J.B., Kozen, D., Silva, A.: CoCaml: Programming with coinductive types. Tech. Rep. http://hdl.handle.net/1813/30798, Computing and Information Science, Cornell University (December 2012), fundamenta Informaticae, to appear
20. Levy, J., Veanes, M.: On the undecidability of second-order unification. Inf. Comput. 159(1-2), 125–150 (2000)
21. Levy, J., Villaret, M.: An efficient nominal unification algorithm. In: Lynch, C. (ed.) Proc. 21st RTA. LIPIcs, vol. 6, pp. 209–226. Schloss Dagstuhl (2010)
22. Levy, J., Villaret, M.: Nominal unification from a higher-order perspective. ACM Trans. Comput. Log. 13(2), 10 (2012)
23. Luks, E.M.: Permutation groups and polynomial-time computation. In: Finkelstein, L., Kantor, W.M. (eds.) Groups And Computation, Proceedings of a DIMACS Workshop. DIMACS, vol. 11, pp. 139–176. DIMACS/AMS (1991)
24. Marlow, S. (ed.): Haskell 2010 – Language Report (2010)
25. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems 4(2), 258–282 (1982)
26. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. 1(4), 497–536 (1991)
27. Moran, A.K.D., Sands, D., Carlsson, M.: Erratic fudgets: A semantic theory for an embedded coordination language. In: Coordination '99. LNCS, vol. 1594, pp. 85–102. Springer-Verlag (1999)
28. Picouleau, C.: Complexity of the Hamiltonian cycle in regular graph problem. Theor. Comput. Sci. 131(2), 463–473 (1994)
29. Schmidt-Schauß, M., Rau, C., Sabel, D.: Algorithms for Extended Alpha-Equivalence and Complexity. In: van Raamsdonk, F. (ed.) 24th RTA 2013). LIPIcs, vol. 21, pp. 255–270. Schloss Dagstuhl (2013)
30. Schmidt-Schauß, M., Sabel, D.: Unification of program expressions with recursive bindings. In: Cheney, J., Vidal, G. (eds.) 18th PPDP. pp. 160–173. ACM (2016), http://doi.acm.org/10.1145/2967973.2968603
31. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker's strictness analysis. J. Funct. Programming 18(04), 503–551 (2008)
32. Schöning, U.: Graph isomorphism is in the low hierarchy. J. Comput. Syst. Sci. 37(3), 312–323 (1988)
33. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczynski, M. (eds.) 22nd ICLP. pp. 330–345. LNCS (2006)
34. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in nominal Isabelle. Log. Methods Comput. Sci. 8(2) (2012)
35. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unification. In: 17th CSL, 12th EACSL, and 8th KGC. LNCS, vol. 2803, pp. 513–527. Springer (2003)
36. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theor. Comput. Sci. 323(1–3), 473–497 (2004)