Towards Fast Nominal Anti-unification of Letrec-Expressions

Manfred Schmidt-Schauß[1] and Daniele Nantes-Sobrinho[2(✉)]

[1] Goethe-University Frankfurt, Frankfurt, Germany
schauss@em.uni-frankfurt.de
[2] Imperial College London, London, UK
dnantess@ic.ac.uk

Abstract. This paper describes anti-unification algorithms for computing least general generalizations of two expressions in a functional programming language with recursive let. First, by exploring a semantic approach to the problem, we argue for an improvement of the technique used in previous papers which avoids infinite chains of properly descending generalizations. Second, we present a (non-deterministic) nominal general anti-unification algorithm applicable to general expressions, which is complete, terminating and requires polynomial time. Third, we propose a specialized anti-unification algorithm applicable to two or more garbage-free ground expressions that produces a single least general generalization in polynomial time, and which can also exploit further semantically correct equivalences. Our results have potential applications in finding clones in functional programs.

Keywords: Anti-Unification · Nominal Techniques · Generalization · Functional Programming · Recursive Let

1 Introduction

Anti-unification problems (a.k.a. *generalization* problems) consist in finding a least general generalization (lgg) of two or more given expressions. This problem has interesting applications in computer science and software engineering, such as, symbolic mathematical computing [21], proof generalization [10], clone detection [8], among others; an overview is [6]. Early proposals to apply generalization for analyzing and improving programs by syntactic manipulations was given by Plotkin [12] and Reynolds [13].

We are interested in the anti-unification problem for languages with binders, such as the lambda-calculus, the pi-calculus, or the more general nominal language [11]. For instance, $\lambda x.Z$ is a generalization of the lambda-expressions $\lambda a.app(a,a)$, $\lambda a.\lambda b.a$, and $\lambda c.c$. In fact, from $\lambda x.Z$ one can retrieve any of the

Research reported in this paper is partially funded by the EPSRC Fellowship 'VeTSpec: Verified Trustworthy Software Specification' (EP/R034567/1).

B. Pientka and C. Tinelli (Eds.): CADE 2023, LNAI 14132, pp. 456–473, 2023.
https://doi.org/10.1007/978-3-031-38499-8_26

three expressions in the set by considering the appropriate instance of Z (where capturing is permitted), modulo renaming of bound variables: $Z \mapsto app(x, x)$, $Z \mapsto \lambda b.x$ and $Z \mapsto x$, respectively.

In the context of languages with recursive let (letrec), techniques for solving anti-unification problems would allow, for instance, to identify the program scheme $\texttt{letr}\ b.(\lambda x.N); a.(\lambda x.M)\ \texttt{in}\ b(y)$ as a generalization of the program [1]

$$\begin{aligned}&\texttt{letr}\ even.(\lambda x.\mathsf{if\text{-}else}\ (x = 0)\ (\texttt{true})\ (odd(x - 1)));\\ &\quad odd.(\lambda x.\mathsf{if\text{-}else}(x = 0)(\texttt{false})(even(x - 1)))\\ &\quad \texttt{in}\ (even\ y)\end{aligned}$$

or even identify both fragments of programs as possible clones [8].

In general, and as illustrated above, reasoning and automated deduction in higher order languages often require – as a very basic operation – to identify expressions up to α-equivalence. This means expressions are identified if they are syntactically equal up to a renaming of bound variables (which represent the binding structure). In addition, one has to have in mind that the letrec construct also satisfies laws like commutativity and associativity of its environment (e.g. we could permute the environment $b.(\lambda x.N); a.(\lambda x.M)$ as $a.(\lambda x.M); b.(\lambda x.N)$ above), which will be working in combination with binding primitives (i.e., also rename the bindings within the environment obtaining, e.g., $c.(\lambda x.M'); d.(\lambda x.N')$), and they also may occur nested.

Checking expressions for α-equivalence is an operation that is often performed on large and complex expressions. Ad-hoc algorithms for checking α-equivalence of such expressions are worst-case exponential due to searching for all possible permutations and renamings. An approach to handle α-equivalence in deduction systems is to use nominal techniques [5,11], where the focus is to ease formula specification and deduction rather than speeding up α-equivalence checking. In general, checking α-equivalence with the language extended with letrec using nominal techniques is a GI-hard problem [18]. Here, we follow the nominal approach to handle binding of names and their renaming.

In [17] we have proposed a semantic approach to anti-unification based on nominal techniques which uses atom-variables, and significantly improves an existing approach [4] to anti-unification for languages with binders, since it provides a finitary set of least general generalizations. In this work we propose a simplification of this semantic approach to a nominal language extended by the letrec construct, which we call $\texttt{NLL}_X$.

Our Results. We provide a nominal anti-unification algorithm (AntiUnifLetr) for $\texttt{NLL}_X$ which preserves the good properties of our semantic approach: it is terminating, sound, computes an exponential number of generalizations (Theorem 1) and weakly complete (Theorem 2). Completeness is achieved after further specialization of the computed generalization (Theorem 3).

The observation that *garbage* might be present in letrec expressions (for example, useless bindings in environments), and that they can be avoided by a semantically correct garbage collection algorithm, allows to apply the results and

methods in [18], which shows that α-equivalence and further algorithms could be considerably improved for garbage-free expressions. This leads to the design of AntiUnifNoGarbage, an anti-unification algorithm for *ground* garbage-free expressions, that is terminating, runs in polynomial time and produces one least general generalization, i.e. it is unitary (Theorem 4).

2 Preliminaries

We consider a countable infinite set of atoms $\mathbb{A}$ 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). We also consider a set $\mathcal{F}$ of function symbols with arity $ar(\cdot)$, and a countably infinite set of expression-variables *Var* ranged over by X, Y. We will use mappings on atoms from $\mathbb{A}$: a *swapping* $(a\ b)$ is a bijective function that maps 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 consists of a composition of swappings: in fact, every finite permutation π can be represented by a composition of at most $(|dom(\pi)| - 1)$ swappings, where $dom(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq a\}$. The identity permutation is denoted Id. Composition $\pi_1 \circ \pi_2$ and the inverse π^{-1} can be immediately computed, where the complexity is polynomial in the size of $dom(\pi)$.

Ground Expressions. The syntax of expressions $\bar{e}$ of the (ground) language `NLL` with recursive let is:

$$\bar{e} ::= a \mid \lambda a.\bar{e} \mid (f\ \bar{e}_1\ \ldots\ \bar{e}_{ar(f)}) \mid (\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$$

Ground expressions are either atoms, abstractions of an atom in an expression, function application, or a letrec expression. We assume that binding atoms $a_1, \ldots, a_n$ in a letrec-expression $(\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$ are pairwise distinct. Sequences of bindings $a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n$ may be abbreviated as *env* (environment). The *scope* of atom a in $\lambda a.\bar{e}$ is standard: a has scope $\bar{e}$. The `letr`-construct has a special scoping rule: in $(\texttt{letr}\ a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$, every atom a_i that is free in some $\bar{e}_j$ or $\bar{e}$ is bound by the environment $a_1.\bar{e}_1; \ldots; a_n.\bar{e}_n$. This defines in `NLL` the notion of free atoms $FA(\bar{e})$, bound atoms $BA(\bar{e})$ in expression $\bar{e}$, and all atoms $AT(\bar{e})$ that occur in $\bar{e}$. For an environment $env = \{a_1.\bar{e}_1, \ldots, a_n.\bar{e}_n\}$, we define the set of letrec-atoms as $LA(env) = \{a_1, \ldots, a_n\}$. We say a *is fresh for* $\bar{e}$ iff $a \notin FA(\bar{e})$, denoted as $a\#\bar{e}$.

Remark 1. The base language `NLL` is a lambda calculus extended with function constant and a recursive let constructor `letr`, and can also be interpreted as an untyped fragment of Haskell [7]. The function application operator in functional languages (implicit in some languages) can be encoded by a binary function `app`, and the case-construct in its plain form can be encoded as an application.

Example 1. The letrec-expression $(\texttt{letr}\ a.cons\ \bar{e}_1\ b; b.cons\ \bar{e}_2\ a\ \texttt{in}\ a)$ represents an infinite list $(cons\ \bar{e}_1\ (cons\ \bar{e}_2\ (cons\ \bar{e}_1\ (cons\ \bar{e}_2\ \ldots))))$, where $\bar{e}_1, \bar{e}_2$ are expressions and *cons* is the usual list constructor taken as a function symbol.

Syntactic α-equivalence on NLL is defined, following [16], as an extension of usual α-equivalence, where in addition the expressions $(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e})$ and $(\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')$ are α-equivalent iff the expressions can be made equal by correctly renaming them, possibly reordering the environment.

Definition 1. *The α-equivalence $\sim_\alpha$ on $\bar{e} \in$ NLL is defined as follows:*

- $a \sim_\alpha a$ *for atoms* a.
- *if* $\bar{e}_i \sim_\alpha \bar{e}'_i$ *for all* i, *then* $(f\ \bar{e}_1 \ldots \bar{e}_n) \sim_\alpha (f\ \bar{e}'_1 \ldots \bar{e}'_n)$ *for* n*-ary* $f \in \mathcal{F}$.
- *If* $\bar{e} \sim_\alpha \bar{e}'$, *then* $\lambda a.\bar{e} \sim_\alpha \lambda a.\bar{e}'$.
- *If* $a\#\bar{e}'$ *and* $\bar{e} \sim_\alpha (a\ b)\cdot\bar{e}'$, *then* $\lambda a.\bar{e} \sim_\alpha \lambda b.\bar{e}'$.
- $(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) \sim_\alpha (\texttt{letr}\ a_{\rho(1)}.\bar{e}_{\rho(1)};\ldots;a_{\rho(n)}.\bar{e}_{\rho(n)}\ \texttt{in}\ \bar{e})$ *for any permutation* ρ *of* $\{1,\ldots,n\}$.
- *The following holds for a permutation* π *on atoms* $\{a_1,\ldots,a_n\}\cup\{a'_1,\ldots,a'_n\}$:

$$\frac{\forall i.\ \pi(a'_i) = a_i \quad \pi\cdot\bar{e}'_i \sim_\alpha e_i \quad \pi\cdot\bar{e}' \sim_\alpha \bar{e} \quad a_i\#(\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')}{(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) \sim_\alpha (\texttt{letr}\ a'_1.\bar{e}'_1;\ldots;a'_n.\bar{e}'_n\ \texttt{in}\ \bar{e}')}$$

where, for $i = 1,\ldots,n$: a_i*'s are pairwise distinct, and* a'_i*'s are pairwise distinct.*

Permutations operate on NLL-expressions by recursing on their structure. For example, $\pi\cdot(\texttt{letr}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}) = (\texttt{letr}\ \pi\cdot a_1.\pi\cdot\bar{e}_1;\ldots;\pi\cdot a_n.\pi\cdot\bar{e}_n\ \texttt{in}\ \pi\cdot\bar{e})$.

General Expressions. The syntax of the nominal higher-order language $\texttt{NLL}_X$ with letrec and variables is:

$$\begin{array}{lll} e,s,t & ::= & a \mid \pi\cdot X \mid \lambda a.e \mid (f\ e_1\ \ldots\ e_{ar(f)}) \mid (\texttt{letr}\ a_1.e_1;\ldots;a_n.e_n\ \texttt{in}\ e) \\ \pi & ::= & \emptyset \mid (a\ b)\cdot\pi \end{array}$$

General expressions extend NLL with *suspensions*, i.e., expressions of the form $\pi \cdot X$, which denotes a variable X (also called a generalization variable) in which a permutation is suspended: π is waiting for some instantiation of X before its action. The basic properties and functions of NLL such as $FA(e)$, $BA(e)$, scope, fresh, etc., extend to $\texttt{NLL}_X$ as expected. In particular, $AT(e)$ is extended to suspensions as $AT(\pi \cdot X) = \{a \mid a \in dom(\pi)\}$. The suspension $Id{\cdot}X$ is written simply as X. We define $Head(s)$ either as the top function symbol in $\{a, f, \lambda, \texttt{letr}\}$ or $Head(\pi \cdot X)$ as X. More generally, for a non-variable expression e, the expression $\pi{\cdot}e$ means an operation, which is performed by shifting π into the expression, 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 remains in suspensions. For instance, $(a\ c) \cdot (\texttt{letr}\ a.(\lambda b.X)\ \texttt{in}\ f(a))$ denotes a renaming of a to c and vice-versa, which is equal to $(\texttt{letr}\ c.(\lambda b.(a\ c) \cdot X)\ \texttt{in}\ f(c))$.

An $\texttt{NLL}_X$*-freshness constraint* is an expression of the form $a\#e$, expressing that a is not free in (or is fresh for) e, where e is an $\texttt{NLL}_X$-expression. A conjunction (or set) of freshness constraints is called *freshness context* which is written using the notation ∇, Δ. Every $\texttt{NLL}_X$-freshness context can be transformed into

$$\frac{\{a\#b\}\cup\nabla}{\nabla} \text{ if } a \neq b \qquad \frac{\{a\#(\pi \cdot X)\}\cup\nabla}{\{\pi^{-1}(a)\#X\} \cup \nabla} \qquad \frac{\{a\#(f\ s_1 \dots s_n)\}\cup\nabla}{\{a\#s_1, \dots, a\#s_n\} \cup \nabla}$$

$$\frac{\{a\#(\lambda a.s)\}\cup\nabla}{\nabla} \qquad \frac{\{a\#a\}\cup\nabla}{\bot} \qquad \frac{\{a\#(\lambda b.s)\}\cup\nabla}{\{a\#s\} \cup \nabla} \text{ if } a \neq b$$

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

$$\frac{\{a\#(\texttt{letr}\ a_1.s_1; \dots, a_n.s_n\ \texttt{in}\ r)\}\cup\nabla}{\{a\#s_1, \dots a\#s_n, a\#r\} \cup \nabla} \text{ if } a \notin \{a_1, \dots, a_n\}$$

Fig. 1. Simplification of freshness constraints in $\texttt{NLL}_X$

a simpler one (flattened form) using the rules in Fig. 1 exhaustively until consisting only of constraints of the form $a\#X$ or $\bot$ (fail), which are called *atomic*. An $\texttt{NLL}_X$-freshness context ∇ is *consistent* if its flattened form does not contain $\bot$. The definition of α-equivalence extends to $\texttt{NLL}_X$ as expected. In the following, $[s]_\alpha$ denotes the equivalence class of the expression s induced by the equivalence relation $\sim_\alpha$.

Lemma 1. *Simplification using rules of Fig. 1 constitutes a polynomial decision algorithm for satisfiability of ∇: If $\bot$ is in the result, then unsatisfiable; otherwise, satisfiable.*

An $\texttt{NLL}_X$-substitution ρ is a finite mapping from generalization variables to $\texttt{NLL}_X$-expressions. Substitutions act on expressions homomorphically and this action extends to freshness constraints and contexts as follows: $(a\#X)\rho$ iff $a\#X\rho$ and $\nabla\rho = \{a\#e\rho \mid a\#e \in \nabla\}$. We will denote the domain of substitutions by $dom(\cdot)$. A substitution is *ground* if it maps (generalization) variables to NLL-expressions. For a ground substitution ρ: $\nabla\rho$ is called *valid* iff $\nabla\rho$ is consistent.

Permutations and Cycles. A *cycle* τ in $\mathbb{A}$ is a permutation represented by a sequence of different atoms $a_1, a_2, \dots, a_n$, such that $\tau(a_i) = a_{i+1}$ for $i = 1, \dots, n-1$ and $\tau(a_n) = a_1$. As standard, such cycle will be denoted as $\tau = (a_1\ a_2\ \dots a_n)$. Every permutation π has a representation $\tau_1\tau_2 \dots \tau_n$ (which abbreviates $\tau_1 \circ \tau_2 \circ \dots \circ \tau_n$) where τ_i are disjoint (primitive) cycles.

The disjoint cycles can be permuted. For instance, the permutation $(a\ b)(b\ d)(c\ e)$ has the cycle presentation $(a\ b\ d)(c\ e)$ which is the same as $(c\ e)(a\ b\ d)$.

2.1 Data-Structures of Anti-unification Algorithms

Anti-unification algorithms will produce as a result expressions that are restricted by a freshness context. These are called *expressions-in-context* and denoted as (∇, s), where ∇ is a freshness context and s is an $\texttt{NLL}_X$-expression.

The semantics of expressions-in-context follow the idea that syntactically used names of atoms in expressions are fixed, and atoms occurring in ∇, but not in s are viewed as existentially quantified: these are treated as arbitrary names of atoms.

Definition 2. *An* expression-in-context *is a pair* (∇, e)*, where* e *is an expression and* ∇ *is a (consistent) freshness context. The* semantics *of* (∇, e) *is the set of ground instances of* e *that satisfy* ∇*, i.e.,*

$$[\![(\nabla, e)]\!] = \{[r]_\alpha \mid \exists \hat{\rho} : \forall a \in AT(e).\ a\hat{\rho} = a \text{ and } [r]_\alpha = [e\hat{\rho}]_\alpha \text{ and } \nabla\hat{\rho} \text{ valid}\}$$

where $\hat{\rho}$ *is a mapping from* $Var \cup \mathbb{A}$ *to ground expressions such that* $\hat{\rho}|_{\mathbb{A}}$ *is a bijection on atoms.*

The existential quantification on valid instances of expressions gives additional power to the semantics of expressions-in-context: by considering a as existentially quantified, we obtain that $[\![(\{a\#X\}, X)]\!]$ is the same as $[\![(\emptyset, X)]\!]$.

Example 2. Consider the expression-in-context $(\{a\#X\}, f(X))$. We will argue that $[\![(\{a\#X\}, f(X))]\!] = [\![(\emptyset, f(X))]\!]$. First, notice that a does not occur syntactically in $f(X)$ and therefore we can take $\hat{\rho}$ mapping a to an arbitrary atom that does not break validity of ∇. In fact:

- It is obvious that $[\![(\{a\#X\}, f(X))]\!] \subseteq [\![(\emptyset, f(X))]\!]$, since the left one has more restriction on its elements than the right one.
- $[\![(\emptyset, f(X))]\!] \subseteq [\![(\{a\#X\}, f(X))]\!]$: Let $\hat{\rho}$ be a bijection on atoms that is the identity on the atoms occurring in $f(X)$ (there is none). Then, we select $a\hat{\rho} \notin [f(X)]_\alpha$ which trivially implies that $a\hat{\rho}\#X\hat{\rho}$ holds.

Our semantics for $(\{a\#X\}, X)$ differs from the one in Baumgartner et al. [3] where $[\![(\{a\#X\}, X)]\!]_B$ is the set of all ground instances of X, where a is not permitted to occur free. This will induce the negative effect of properly infinite descending chains[1] of expressions-in-context such as $\ldots \prec_B (\{a\#X, b\#X\}, f(X)) \prec_B (\{a\#X\}, f(X)) \prec_B (\emptyset, f(X))$, which is eliminated in our approach since in all these expressions-in-context have the same semantics.

Next we define an order relation on expressions-in-context which establishes when one expression-in-context is more general or more specific than another.

Definition 3 (Ordering, Generalization).

- *An expression-in-context* (Δ, r) *is* more specific *(or less general) than an expression-in-context* (∇, s)*, denoted* $(\nabla, s) \preceq (\Delta, r)$*, if* $[\![(\Delta, r)]\!] \subseteq [\![(\nabla, s)]\!]$*. The strict part of* $\preceq$ *is denoted* $\prec$*. This defines equivalence of two expressions-in-context via their semantics:* $(\nabla, s) \approx (\nabla', t)$ *iff* $[\![(\nabla, s)]\!] = [\![(\nabla', t)]\!]$.
- *An expression-in-context* (Δ, r) *is a* generalization *of* (∇, s) *and* (∇', t)*, if* $(\Delta, r) \preceq (\nabla, s)$ *and* $(\Delta, r) \preceq (\nabla', t)$.

[1] $[\![\cdot]\!]_B$ and $\prec_B$ denote the semantics and order relation in [4], resp.

– *A generalization* (Δ', r') *of* (∇, s) *and* (∇', t) *is the* most specific *(the least general) one, if for all generalizations* (Δ, r) *of* (∇, s) *and* (∇', t)*, we have* $(\Delta, r) \preceq (\Delta', r')$.

For instance, the expression-in-context $(\emptyset, \lambda e.app(e, X))$ is a generalization of $(\emptyset, \lambda a.app(a, c))$ and $(\emptyset, \lambda b.app(b, Z))$, for a new atom e. It is easy to verify that $(\emptyset, \lambda e.app(e, X)) \preceq (\emptyset, \lambda a.app(a, c))$ and $(\emptyset, \lambda e.app(e, X)) \preceq (\emptyset, \lambda b.app(b, Z))$.

3 The Anti-unification Problem for $\mathtt{NLL}_X$

We are interested in *the anti-unification problem for* $\mathtt{NLL}_X$:
Given two expressions-in-context (∇, s) and (∇, t),
Find a *least general generalization*, i.e., another expression-in-context (Δ, r) that satisfies $(\Delta, r) \preceq (\nabla, s)$ and $(\Delta, r) \preceq (\nabla, t)$.

The challenge in treating letrec-expressions in anti-unification algorithms is, on the one hand, its unusual scoping and; on the other hand, the multiple possibilities to formulate the same problem in several syntactically different ways.

Remark 2 [Permutations in the generalization of suspensions]. Generalization of suspensions, say $(\emptyset, \pi_1 \cdot Z)$ and $(\emptyset, \pi_2 \cdot Z)$, need some preparations based on properties of permutations: first, we decompose π_1 and π_2 into their cycle presentation, say $\pi_1 = \mu_1 \dots \mu_n$ and $\pi_2 = \mu'_1 \dots \mu'_m$; second, we work on generalizing $(\emptyset, \mu_1 \dots \mu_n \cdot Z)$ and $(\emptyset, \mu'_1 \dots \mu'_m \cdot Z)$ as follows: let π_3 be a permutation obtained from the set of common cycles of π_1 and π_2, say $\pi_1 = \pi_3 \pi'_1$ and $\pi_2 = \pi_3 \pi'_2$. Then, $\pi_3 \cdot X$ is a generalization for $(\emptyset, \pi_1 \cdot Z)$ and $(\emptyset, \pi_2 \cdot Z)$. In the following we will denote the common cycles of permutations π_1 and π_2 as $\pi_1 \cap \pi_2$. This will be addressed in details with the specific rule for suspensions in Fig. 2.

3.1 The Algorithm AntiUnifLetr and Its Rules

We first define the nominal generalization algorithm AntiUnifLetr that (nondeterministically) computes a single generalization of the input expressions, where the generalization can also be nonlinear in the generalization variables due to merging. We will argue that the algorithm is sound and weakly complete, and one run can be performed in polynomial time.

The data structure of the algorithm AntiUnifLetr is (Γ, M, ∇, L) where:

– Γ is a set of generalization triples of the form $X : s \triangleq t$, where X is a fresh (generalization-) variable, and s, t are $\mathtt{NLL}_X$-expressions;
– M is a set of solved generalization triples;
– ∇ is a set of freshness constraints, without freshness constraints for the fresh generalization variable for the input generalization triple;
– L is a substitution represented as a set of bindings; the empty set is $[]$. The result of applying the substitution L on the generalization variable X is denoted as $X \circ L$.

We call such a tuple a *state*. The rules of the algorithm AntiUnifLetr, given in Fig. 2, operate on states and $\uplus$ denotes disjoint union. Given two NLL expressions s and t, and a freshness context Δ (possibly empty), to compute generalizations for (Δ, s) and (Δ, t), we start with $(\{X : s \triangleq t\}; \emptyset; \Delta; [])$, the *initial state* (sometimes abbreviated to $(\Delta, \{X : s \triangleq t\})$), where X is a fresh generalization variable, and we apply the rules from Fig. 2 and Fig. 4 until no more rule applications are possible and we reach the *final state* which has the form $(\emptyset, M, \nabla, L)$, where M must be completely merged. We will denote the computation from initial to a final state: $(\Gamma; \emptyset; \Delta; []) \Longrightarrow^* (\emptyset; M; \nabla, L)$.

The output is an expression-in-context obtained from the generated substitution L and the final freshness constraint ∇, i.e. the output is $(\nabla, X \circ L)$, also called the *result computed* by the AntiUnifLetr algorithm. We say it is *complete* if every least general generalization (lgg) is found and it is *weakly complete* if every lgg is found up to some set of freshness constraints.

(Dec): Decomposition

$$\frac{\{X{:}f(s_1,\ldots,s_n) \triangleq f(t_1,\ldots,t_n)\} \uplus \Gamma, M, \nabla, L \quad X_i \text{ are fresh variables} \quad n = 0 \text{ is permitted}}{\Gamma \cup \{X_1{:}s_1 \triangleq t_1, \ldots, X_n{:}s_n \triangleq t_n\}, M, \nabla, L \cup \{X \mapsto f(X_1,\ldots,X_n)\}}$$

(Absaa): Abstraction

$$\frac{\{X{:}\lambda a.s \triangleq \lambda a.t\} \uplus \Gamma, M, \nabla, L \qquad Y \text{is a fresh variable}}{\Gamma \cup \{Y{:}s \triangleq t\}, M, \nabla, L \cup \{X \mapsto \lambda a.Y\}}$$

(Absab): Abstraction

$$\frac{\{X{:}\lambda a.s \triangleq \lambda b.t\} \uplus \Gamma, M, \nabla, L \quad Y \text{ is a fresh variable} \quad c \text{ is a fresh atom}}{\Gamma \cup \{Y{:}(c\ a){\cdot}s \triangleq (c\ b){\cdot}t\}, M, \nabla \cup \{c\#\lambda a.s, c\#\lambda b.t\}\}, L \cup \{X \mapsto \lambda c.Y\}}$$

(SusYY): SuspensionYY

$$\frac{\pi = \pi_1 \cap \pi_2, \pi_1 = \pi{\cdot}\pi_1', \pi_2 = \pi{\cdot}\pi_2' \quad \{X{:}\pi_1{\cdot}Y \triangleq \pi_2{\cdot}Y\} \uplus \Gamma, M, \nabla, L}{\{Z{:}\pi_1'{\cdot}Y \triangleq \pi_2'{\cdot}Y\} \cup \Gamma, M, \nabla, L \cup \{X \mapsto \pi{\cdot}Z\}}$$

(Mer): Merging

$$\frac{\Gamma, \{X{:}s_1 \triangleq t_1, Y{:}s_2 \triangleq t_2\} \uplus M, \nabla, L \quad \textsc{Eqvm}(\{(s_1,t_1) \preceq (s_2,t_2)\}) = \pi}{\Gamma, M \cup \{X{:}s_1 \triangleq t_1\}, \nabla, L \cup \{Y \mapsto \pi{\cdot}X\}}$$

(SolveYY)

$$\frac{\{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\} \uplus \Gamma, M, \nabla, L \quad \pi_1 \neq \pi_2, \pi_1 \cap \pi_2 = \emptyset}{\Gamma, M \cup \{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\}, \nabla, L}$$

(Solve)

$$\frac{\{X{:}s \triangleq t\} \uplus \Gamma, M, \nabla, L \quad \mathit{Head}(s) \neq \mathit{Head}(t) \text{ or } s,t \text{ letrec-expressions with a different number of bindings}}{\Gamma, M \cup \{X{:}s \triangleq t\}, \nabla, L}$$

Fig. 2. Rules of the algorithm AntiUnifLetr

Rules in Fig. 2 are similar to the ones in [3] without the parameter for the set of atoms occurring in the initial state and throughout the computation, and deal with abstractions, function application, and suspensions. The subalgorithm Eqvm, defined by the rules in Fig. 3, computes a matching permutation, say π, of

$$\frac{\Psi \cup \{f(s_1,\ldots,s_n) \preceq f(s'_1,\ldots,s'_n)\}}{\Psi \cup \{s_1 \preceq s'_1,\ldots,s_n \preceq s'_n\}} \qquad \frac{\Psi \cup \{\lambda a.s \preceq \lambda a.t\}}{\Psi \cup \{s \preceq t\}}$$

$$\frac{\Psi \cup \{\lambda a.s \preceq \lambda b.t\} \quad b \# \lambda a.s}{\Psi \cup \{(a\ b)\cdot s \preceq t\}} \qquad \frac{\Psi \cup \{\lambda a.s \preceq \lambda b.t\} \quad a \# \lambda b.t}{\Psi \cup \{(s \preceq (a\ b)\cdot t\}}$$

1. Exhaustively apply the rules above. If after the application Ψ contains pairs not of the form $a \preceq b$, then Fail.
2. Let $\Psi = \{a_1 \preceq b_1,\ldots,a_n \preceq b_n\}$. If the mapping $\{g : a_i \to b_i \mid i = 1,\ldots,n,$ where $a_i \preceq b_i \in \Psi\}$ is not injective, then Fail.
3. Return the bijective mapping π generated by $a_i \to b_i, i = 1,\ldots,n$.

Fig. 3. The permutation matching (sub-)algorithm EQVM

(Letraa): LETREC WITH ORDERED ATOMS

$$\frac{\{X{:}\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ a_1.t_1;\ldots;a_n.t_n\ \texttt{in}\ t\} \uplus \Gamma, M, \nabla, L}{\Gamma \cup \{X_1{:}s_1 \triangleq t_1,\ldots,X_n{:}s_n \triangleq t_n, Y{:}s \triangleq t\}, M, \nabla, L \cup \{X \mapsto \texttt{letr}\ a_1.X_1,\ldots,a_n.X_n\ \texttt{in}\ Y\}}$$

(Letperm): LETREC WITH PERMUTED BINDINGS IN ENVIRONMENTS

$$\frac{\{X{:}\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t\} \uplus \Gamma, M, \nabla, L \quad \rho \text{ is a permutation on } \{1,\ldots,n\}}{\{X : \texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ b_{\rho(1)}.t_{\rho(1)};\ldots;b_{\rho(n)}.t_{\rho(n)}\ \texttt{in}\ t\} \cup \Gamma, M, \nabla, L}$$

(Letrab): LETREC WITH ATOMS IN ENV SWAPPED WITH NEW NAMES

$$\frac{\begin{array}{c}\{X{:}\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ b_1.t_1,\ldots,b_n.t_n\ \texttt{in}\ t\} \uplus \Gamma, M, \nabla, L \\ \nabla' = \{c_i \# (\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s)\} \cup \{c_i \# (\texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t)\} \\ \pi_1 = (a_1\ c_1)\ldots(a_n\ c_n) \qquad \pi_2 = (b_1\ c_1)\ldots(b_n\ c_n) \quad c_i \text{ are fresh and different atoms}\end{array}}{\{X{:}\pi_1\cdot(\texttt{letr}\ a_1.s_1;\ldots;a_n.s_n\ \texttt{in}\ s) \triangleq \pi_2 \cdot (\texttt{letr}\ b_1.t_1;\ldots;b_n.t_n\ \texttt{in}\ t)\} \uplus \Gamma, M, \nabla \cup \nabla', L}$$

Fig. 4. Rules for `letrec` of the algorithm ANTIUNIFLETR

two expressions-in-context (say $s \preceq t$ in Ψ with context ∇), where EQVBIEX(Π) checks whether the set of swappings is injective and then adds a minimal set of mappings such that the result is a bijection, i.e. a permutation (on atoms). Rules in Fig. 4 are new and will be described in detail:

Rule (`Letraa`) acts as a decomposition rule with the `letr` construct and can only be applied if the bindings in the environment are the same, respecting the given order.

Rule (`Letrperm`) is branching and exhaustively tries to generalize the expressions by considering all permutations of the `letr` environment.

Rule (`Letrab`) deals with renaming of bound names; it consistently swaps the binding atoms of the `letr` environment with fresh names and propagates the obtained permutation throughout both expressions.

The latter rule exploits the following idea: if $\lambda a.s$ and $\lambda b.t$ are α-equivalent, then one can rename a and b with the same fresh name c and propagate the renaming within s and t and still obtain α-equivalent expressions.

Example 3. A generalization for the expressions-in-context $(\emptyset, \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b))$ and $(\emptyset, \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b))$ is computed as follows:

1. We cannot apply rule (`Letraa`) since the binding atoms in the environment are not corresponding to each other. We may rearrange the bindings using (`Letperm`). Then we apply rule `Letrab` for renaming: we choose d, e as fresh atoms and use the renaming $(a\ d)(b\ e)$ and $(c\ d)(b\ e)$, which leads to the check $\nabla' = \{d, e\#(\texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b))\} \cup \{d, e\#(\texttt{letr}\ c.c; b.a\ \texttt{in}\ f(a,b))\} = \emptyset$ which holds and evaluates to $\emptyset$, since the terms are ground. After an application (`Letraa`), which decomposes the letrec environments:

$$\dfrac{\dfrac{\dfrac{(\{X : \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b)\}, \emptyset, \emptyset, [])}{\{X{:}\texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ c.c; b.a\ \texttt{in}\ f(a,b)\}, \emptyset, \emptyset, [])}}{(\{X{:}\texttt{letr}\ d.d; e.c\ \texttt{in}\ f(d,e) \triangleq \texttt{letr}\ d.d; e.a;\ \texttt{in}\ f(a,e)\}, \emptyset, \emptyset, [])}}{(\{X_1{:}d \triangleq d, X_2{:}c \triangleq a, Y{:}f(d,e) \triangleq f(a,e)\}, \emptyset, \emptyset, \{X \mapsto \texttt{letr}\ d.X_1; e.X_2\ \texttt{in}\ Y\})}$$

2. After three applications of (`Dec`), one (`Solve`) and one (`Mer`) we obtain $(\emptyset, \{X_2 : c \triangleq a\}, \emptyset, \{X \mapsto \texttt{letr}\ d.d; e.X_2\ \texttt{in}\ f((c\ d) \cdot X_2, e)\})$. The output generalization is $(\emptyset, \texttt{letr}\ d.d; e.X_2\ \texttt{in}\ f((c\ d) \cdot X_2, e))$.

Another Solution: from $(X : \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b) \triangleq \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b))$ we could have immediately applied the rule (`Letrab`) using $\pi_1 = (a\ d)(b\ e)$ for the left and $\pi_2 = (b\ d)(c\ e)$ for the right expression. This finally leads to a generalization of the form $\texttt{letr}\ d.X_1, e.X_2\ \texttt{in}\ f(X_3, X_4)$ which is "weaker" (too general) than the one above.

Note that the environments of one of the expressions to be generalized contains *garbage*: the binding $c.c$ is not used in $f(a,b)$.

Theorem 1. *The algorithm* AntiUnifLetr *is terminating and sound. A single run requires polynomial time. The overall computation requires exponential time and may compute an exponential number of generalizations.*

Proof. Soundness and termination can be easily checked by inspection of the rules of Figs. 2, 4 and 3. The number of nondeterministic alternatives is exponential in the worst case, and it is induced by the rule (`Letperm`). A single run (one branch) can be performed in polynomial time.

Notice that except for rule (`Letrab`), all the rules in AntiUnifLetr algorithm preserve the context ∇. This differs from the approach taken in [3] which might add new freshness constraints with a rule similar to our rule (`SolveYY`), based on a set A of all atoms appearing throughout the computation of a generalization. We show in the next example that this choice of initially preserving the freshness context leads to a weak completeness result, but completeness is regained with a specialization algorithm that will be presented next.

Example 4 (Weak Completeness). The expressions-in-context $(\emptyset, f(c_1, a))$ and $(\emptyset, f(c_2, a))$ have the generalization $(\emptyset, f(X_1, a))$ computed by the rules of Fig. 2. However, this is not the lgg since $(\{a\#X_1\}, f(X_1, a))$ is a more specific generalization. In fact, $f(a, a) \in [\![(\emptyset, f(X_1, a)]\!]$, but $f(a, a) \notin [\![(\{a\#X_1\}, f(X_1, a)]\!]$.

Theorem 2 (Weak Completeness). *Given* $\mathtt{NLL}_X$ *expressions* e *and* e'*, and a freshness context* Δ*. If* (∇', r) *is a generalization of* (Δ, e) *and* (Δ, e')*, then there exists a* ∇'' *and a derivation* $(\{X : e \triangleq e'\}, \emptyset, \Delta, []) \Longrightarrow^* (\emptyset, M, \nabla, \sigma)$ *such that* $(\nabla \cup \nabla'', X\sigma)$ *is a generalization of* (Δ, e) *and* (Δ, e') *and* $(\nabla \cup \nabla'', X\sigma) \preceq (\nabla', r)$.

Proof. The proof is by induction on the structure of r.

Example 5 (Cont. Example 4). We remark another behaviour that can be seen from the execution of AntiUnifLetr: $(\{X{:}f(c_1, a) \triangleq f(c_2, a)\}, \emptyset, \emptyset, [])$ reduces to $(\emptyset, \{X_1{:}c_1 \triangleq c_2\}, \emptyset, \{X \mapsto f(X_1, a)\})$. Notice that (i) $f(a, a)$ is clearly not an element of $[\![(\emptyset, f(c_1, a))]\!]$ nor $[\![(\emptyset, f(c_2, a))]\!]$; (ii) the information that c_1 and c_2 were free names in the input problem was "forgotten" by the generalization $f(X_1, a)$, but it can be retrieved from the solved triple in the final state. (iii) $a\#c_1$ and $a\#c_2$ hold trivially.

3.2 From Weak Completeness to Completeness

Given a result (∇, s) of a run of the algorithm AntiUnifLetr, the result is in general only weakly complete, since the expressivity of the language may permit a better generalization. The true most specific generalization may have additional freshness constraints, as it was shown in Example 4. The problem of specializing the generalizer output by AntiUnifLetr is subtle: a different but related behaviour can be seen with the next example.

Example 6. Consider the expressions-in-context $(\emptyset, f(g(c_1, a), a))$ and $(\emptyset, f(c_2, a))$ as input for AntiUnifLetr. The output generalization is $(\emptyset, f(X_1, a))$, and this is the lgg. In fact, a run of the algorithm would terminate with the final state $(\emptyset, \{X_1{:}g(c_1, a) \triangleq c_2\}, \emptyset, \{X \mapsto f(X_1, a)\})$.

We can use the information in the solved part of the final state to build the substitutions $\sigma_1 = \{X_1 \mapsto g(c_1, a)\}$ and $\sigma_2 = \{X_1 \mapsto c_2\}$ that instantiate the generalization $f(X_1, a)$ back to the input terms. Notice that $a\#X_1\sigma_1$ is equal to $a\#g(c_1, a)$ and does not hold. Thus, we cannot add $\{a\#X_1\}$ as a constraint to the generalization, since $(\{a\#X_1\}, f(X_1, a))$ cannot be instantiated to $f(g(c_1, a), a)$.

Let $\gamma = (\emptyset; M; \nabla; L)$ be a final state. We define $AT_f(\gamma)$ as the set of unbound atoms that occur in M, ∇ or $codom(L)$. We say that a generalization variable X occurs in γ when it occurs in ∇, or as a subterm in M, or in $codom(L)$.

Algorithm 1 AntiUnifLetr- Phase 2

Input: (Δ, s) and (Δ, t)
$(\{X : s \triangleq t\}; \emptyset; \Delta; [])\stackrel{*}{\Longrightarrow}\ \gamma = (\emptyset; M; \nabla; L)$
Let $(\nabla, r) = (\nabla, X \circ L)$ be the resulting generalization.
Let X be a generalization variable occurring in r. ▷ Repeat for each X
if $a \in AT(r) \backslash BA(r)$ and $a \notin RelAtoms_\gamma(X)$ **then** ▷ Repeat for each $a \in AT(t)$
    $\nabla := \nabla \cup \{a \# X\}$
end if

Definition 4 (Relevant Atoms). *Let $\gamma = (\emptyset; M; \nabla; L)$ be a final state in a run of* AntiUnifLetr*. Let X be a generalization variable occurring in γ. The set of* relevant atoms for X*, denoted $RelAtoms_\gamma(X)$, is defined recursively:*

- *If there is no solved triple for X in M. Then, the relevant atoms are $RelAtoms_\gamma(X) = AT_f(\gamma) \backslash \{a \mid a \# X \in \nabla\}$, i.e., all atoms that are not bound and that occur syntactically in the state, but not the atoms that were excluded due to the freshness constraints in ∇.*
- *If there is a solved triple $X : s \triangleq t \in M$. Then, $RelAtoms_\gamma(X) = RelAtoms_\gamma(s) \cup RelAtoms_\gamma(t)$. The other cases are defined recursively in the structure of the expression:*
  - $RelAtoms_\gamma(a) = a$, $RelAtoms_\gamma(f\ s_1 \dots s_n) = \bigcup_i RelAtoms_\gamma(s_i)$;
  - $RelAtoms_\gamma(\pi \cdot s) = \pi \cdot RelAtoms_\gamma(s)$;
  - $RelAtoms_\gamma(\lambda a.s) = RelAtoms_\gamma(s) \backslash \{a\}$; *and*
  - $RelAtoms_\gamma(\texttt{letr}\ a_1.s_1; \dots; a_n.s_n\ \texttt{in}\ r) = RelAtoms_\gamma(s_1, \dots, s_n, r) \backslash \{a_1, \dots, a_n\}$.

For example, if we take $M = \{X{:}f(a,b) \triangleq g((a\ c) \cdot Y), Y{:}f(c,d) \triangleq g(e)\}$ and $\nabla = \{a \# Y\}$, then the set of relevant atoms for Y is $\{c, d, e\}$, and for X it is $\{a, b\} \cup (a\ c)\{c, d, e\} = \{a, b, d, e\}$, where it is noteworthy that atom c is missing.

We formulate a postprocessing algorithm (Algorithm 1) for AntiUnifLetr which is able to compute least general generalizations.

Theorem 3. *Adding (Algorithm 1) makes* AntiUnifLetr *complete.*

Note, however, that due to the non-determinism, it may be possible that one of the runs generates a generalization that is strictly less specific than the result in another run, see Example 3.

Example 7. This example shows the result of generalizing more complex expressions. Consider the generalization problem, and the sequence of generalization steps, where the last step abbreviates several steps.

$$\dfrac{\dfrac{\dfrac{(\{X_1 : \lambda a.f(a,a,c) \triangleq \lambda b.f(b,d,c)\}, \emptyset, [\,])}{(\{X_1 : \lambda e.f(e,e,c) \triangleq \lambda e.f(e,d,c)\}, \emptyset, [\,])}}{(\{X_2 : f(e,e,c) \triangleq f(e,d,c)\}, \emptyset, \{X_1 \mapsto \lambda e.X_2\})}}{(\emptyset, \{X_3 : e \triangleq d\}, \{X_1 \mapsto \lambda e.X_2, X_2 \mapsto f(e, X_3, c)\})}$$

Now the resulting lgg can be computed by adding only one freshness constraint: $(\{g\#X_3\}, \lambda e.f(e, X_3, c))$. This holds, since $d \in RelAtoms_\gamma(X_3)$, and hence does not occur in the freshness context. Notice that $c\#X_3$ is added as a freshness constraint since c occurs in the generalization expression, but $c \notin RelAtoms_\gamma(X_3)$.

4 Generalization Algorithm Under Semantic Equalities

We use semantic equivalences to specialize and extend our anti-unification algorithm to *ground expressions*. In particular, we exploit the fact that removal of *garbage* is semantically correct: it does not alter the meaning of the program. First, we develop a standardization algorithm for garbage-free expressions that helps in comparing the letrec-expressions and computing generalizations in polynomial time. Second, we propose a variation of our anti-unification algorithm called AntiUnifNoGarbage.

NLL-expressions may contain irrelevant bindings in the letrec environment: for instance, in ($\texttt{letr}\ a.\texttt{Nil}; b.b\ \texttt{in}\ f(a,a)$), the binding $b.b$ is useless for the expression, and will be considered as *garbage*. The garbage bindings do not contribute to the meaning of the functional expressions. It is shown in [18], that α-equivalence of garbage-free letrec-expressions can be checked in polynomial time, and that, in general, this problem is group-isomorphism-complete [2,20].

Definition 5. *Let $\bar{e}$ be an* NLL*-expression. We say that $\bar{e}$ contains* garbage *iff there is a subexpression* ($\texttt{letr}\ a_1.\bar{e}_1, \ldots, a_n.\bar{e}_n\ \texttt{in}\ \bar{e}'$) *in $\bar{e}$ such that the environment $a_1.\bar{e}_1, \ldots, a_n.\bar{e}_n$ can be split into two nonempty sub-environments $a_{i_1}.\bar{e}_{i_1}, \ldots, a_{i_k}.\bar{e}_{i_k}$ and $a_{j_1}.\bar{e}_{j_1}, \ldots, a_{j_{k'}}.\bar{e}_{j_{k'}}$, and the binding atoms $a_{i_h}, h = i_1, \ldots, i_k$ do not occur free in* $\texttt{letr}\ a_{j_1}.e_{j_1}, \ldots, a_{j_k}.\bar{e}_{j_k}\ \texttt{in}\ \bar{e}'$. *We say that $\bar{e}$ is* garbage-free (or garbage-collected) *iff it does not contain garbage.*

Making an expression garbage-free may require an iterated removal of garbage, using the **g**arbage **r**emoval rewriting rules below:

$$(\mathbf{gr}1)\ \texttt{letr}\ a_1.e_1; \ldots; a_n.e_n; b_1.e'_1; \ldots; b_m.e'_m\ \texttt{in}\ e'_{m+1} \longrightarrow$$
$$\texttt{letr}\ b_1.e'_1; \ldots; b_m; e'_m\ \texttt{in}\ e'_{m+1},\ \text{if}\ \bigcup FA(e'_i) \cap \{a_1, \ldots, a_n\} = \emptyset$$
$$(\mathbf{gr}2)\ \texttt{letr}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e \longrightarrow e,\ \text{if}\ FA(e) \cap \{a_1, \ldots, a_n\} = \emptyset$$

We illustrate our ideas for the generalization of garbage-free expressions. Note that the used equality of expressions makes a notable difference for the results as well as for the algorithmic steps.

Example 8. Let $\bar{s} = \texttt{let}\ c.a\ \texttt{in}\ f(g(c))$ and $\bar{t} = \texttt{let}\ d.b\ \texttt{in}\ f(h(d))$ two garbage-free ground expressions. A generalization of s and t w.r.t. $\sim_\alpha$ is $\bar{s}' = \texttt{let}\ c.X_1\ \texttt{in}\ f(X_2)$, which is also an lgg. If we would allow more equalities on the expressions, like $\sim_{gc}$ as a part of the equality or even an equality $\sim_{\alpha,gc,letcp}$ that allows also copying let-bindings, then $\bar{s}$ would be equivalent to $f(g(a))$ and $\bar{t}$ equivalent to $f(h(b)$, which have $f(X)$ as a generalization. The generalisation algorithm, however, would be much more complex.

$$\frac{\{X{:}\texttt{letr}\ a_{n-m+1}.s_{n-m+1};\ldots;a_n.s_n\ \texttt{in}\ s \triangleq\ \ \texttt{letr}\ a_1.t_1;\ldots;a_n.t_n\ \texttt{in}\ t\}\cup\Gamma, M, \nabla, L}{\{X{:}\texttt{letr}\ a_1.a_1;\ldots;a_{n-m}.a_{n-m};a_{n-m+1}.s_{n-m+1};\ldots;a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ a_1.t_1,\ldots,a_n.t_n\ \texttt{in}\ t\}\cup\Gamma, M, \nabla, L}\ (\texttt{Letrnm}){:}\ \text{Letrecnm}\quad(\text{for } m<n)$$

Fig. 5. Different lengths of `letrec`-environments in AntiUnifLetr

The next step is to standardize the sequence of bindings in garbage-collected expressions, which greatly supports further operations.

Standardization Algorithm. Consider $\texttt{let}\ a_1.\bar{e}_1;\ldots;a_n.\bar{e}_n\ \texttt{in}\ \bar{e}$ be a garbage-free NLL-expression. Then, rearrange the bindings as follows:

1. Let a_j be the atom from $\{a_1,\ldots,a_n\}$ that has the earliest occurrence as a free atom in the expression $\bar{e}$, in its printed string. Then select $a_j.\bar{e}_j$ as the leftmost binding in the fresh environment, i.e. $r_0=\bar{e}$; $r_1=\texttt{letr}\ a_j.\bar{e}_j\ \texttt{in}\ \bar{e}$.
2. Iterate this to compute r_k from $r_{k-1}=\texttt{letr}\ env_{k-1}\ \texttt{in}\ \bar{e}$ by selecting among the remaining binding atoms $a_{j'}\in\{a_1,\ldots,a_n\}\backslash\{a_j\}$ again the one which first occurs free in the printed string of r_{k-1}, and then add $a_{j'}.\bar{e}_{j'}$ as the leftmost binding in the `letr`-environment obtaining $r_k=\texttt{letr}\ a_{j'}.\bar{e}_{j'};env_{k-1}\ \texttt{in}\ \bar{e}$.

These steps are to be used iteratively: apply them to the smallest subexpression $\bar{e}'$ of $\bar{e}$, which is not yet correctly arranged. The result is a *gc-standardized* expression t_{gcst} of t.

Example 9. Consider the garbage-free expression $\texttt{let}\ a.app(b,\lambda c.c);b.\lambda d.d\ \texttt{in}\ a$, where *app* is a binary function symbol for denoting the usual application of the lambda calculus. The standardization algorithm returns the gc-standardized expression $\texttt{let}\ b.\lambda d.d;a.app(b,\lambda c.c)\ \texttt{in}\ a$.

Proposition 1. *For every garbage-free* NLL*-expression $\bar{e}$, the gc-standardized expression $\bar{e}'$ of $\bar{e}$ with $\bar{e}\sim_\alpha\bar{e}'$, has a sequence of bindings in all letrec environments that is unique and has a fixed ordering. The computation can be done in polynomial time.*

Proof. Garbage collection is polynomial: after every step the expression will be smaller, and a single step of detecting a set of redundant bindings is also polynomial. The rearrangement also can be done first for subexpressions of smaller size, and a single rearrangement of the top binding takes polynomial time.

4.1 Anti-unification of Garbage-Free Expressions

In this and the next subsection on generalization we will use a syntactically fixed ordering of bindings in a let environments, and denote this as `letf`.

AntiUnifLetr is adapted to the *ground* situation in several aspects: (i) There are no freshness constraints; (ii) expressions are first gc-standardized; (iii)

we permit that $n \geq 2$ expressions are to be generalized in one step; (iv) in a set of expressions to be generalized, we make all top-level letrec environments to be of the same (minimal) length by adding bindings $a.a$ with fresh atoms a; and (v) we fix the sequence of bindings in a `let` indicated by `letf`.

We remark that an iterated generalization of pairs (i.e., to generalize s_1, s_2 and s_3 one first generalizes s_1 and s_2, and from the result, say r, one repeat the generalization process with r and s_3) has the disadvantage that from the second step, after the first application of rule, there are generalization variables, and the semantic properties get lost, which means that, e.g., the standardization is no longer usable, and so the method does no longer work properly in the next generalization steps.

Therefore, for generalizing more than 2 expressions, the data structure adopted is: the generalized state is as $(\{X{:}s_1 \triangleq \ldots \triangleq s_n\}; M; \nabla; L)$, and we use generalization tuples of the form $\{X{:}s_1 \triangleq \ldots \triangleq s_n\}$ to denote that X is a variable generalizing expressions $s_1, \ldots, s_n$. Examples for the modified rules are

$$(\mathtt{Dec_m})\ \frac{\{X{:}f(s_{1,1},\ldots,s_{1,n}) \triangleq \ldots \triangleq f(s_{m,1},\ldots,s_{m,n})\} \cup \Gamma, M, L \quad X_i \text{ are fresh variables} \quad n = 0 \text{ is permitted}}{\Gamma \cup \{X_1{:}s_{1,1} \triangleq \ldots \triangleq s_{m,1}, \ldots, X_n{:}s_{1,n} \triangleq \ldots \triangleq s_{m,n}\}, M, L \cup \{X \mapsto f(X_1,\ldots,X_n)\}}$$

$$(\mathtt{Absaa_m})\ \frac{\{X{:}\lambda a.s_1 \triangleq \ldots \triangleq \lambda a.s_n\} \cup \Gamma, M, L}{\Gamma \cup \{Y{:}s_1 \triangleq \ldots \triangleq s_n\}, M, L \cup \{X \mapsto \lambda a.Y\}}$$

$$(\mathtt{Mer_m})\ \frac{\Gamma, \{X{:}s_1 \triangleq \ldots \triangleq s_n, Y{:}t_1 \triangleq \ldots \triangleq t_n\} \cup M, L \quad \textsc{Eqvm}(\{(s_1,\ldots,s_n) \preceq (t_1,\ldots,t_n)\}) = \pi}{\Gamma, M \cup \{X_1{:}s_1 \triangleq t_1\}, L \cup \{X \mapsto \pi{\cdot}Y\}}$$

Thus, we adapt the rules of AntiUnifLetr: it accepts $n \geq 2$ ground expressions; the permutation-rule (`Letrperm`) is inactive due to fixing the ordering of bindings; merging is supported, and the subalgorithms Eqvm and EqvBiEx are almost trivial and applied to larger tuples. Also the sequence of bindings in lets is fixed. All these adaptations can be done within the polynomial complexity.

These explanations suggest the algorithm AntiUnifNoGarbage, for $n \geq 2$ (ground) arguments, operating on a triple: (Γ, M, L). It is defined nondeterministically, but only one run will be done.

Example 10 (Fixed letr bindings). Generalizing the garbage-collected expressions $\mathtt{let}\ a'.a; b'.b; c'.c\ \mathtt{in}\ f(g(a',b',c'))$ and $\mathtt{let}\ a'.b; b'.c; c'.a\ \mathtt{in}\ f(h(a',b',c'))$ produces $\mathtt{let}\ a'.a; b'.b; c'.c\ \mathtt{in}\ f(X)$ since bindings can be rearranged, which requires exponential complexity for trying rearrangements. If we fix the sequence of bindings and generalize, then the algorithm requires only polynomial time in this step, then for $\mathtt{letf}\ a'.a; b'.b; c'.c\ \mathtt{in}\ f(g(a',b',c'))$ and $\mathtt{letf}\ a'.b; b'.c; c'.a\ \mathtt{in}\ f(h(a',b',c'))$, we obtain $\mathtt{letf}\ a'.X_1; b'.X_2; c'.X_3\ \mathtt{in}\ f(X)$.

Theorem 4. *Algorithm* AntiUnifNoGarbage *is sound, terminating and complete. It will compute a single least general generalization in polynomial time.*

Proof (Sketch). The main argument is that if no rule applies, then the result is already a generalization. Second, every applied rule keeps the semantics, i.e., does not lose information. The complexity has two components: one is the preparation of the input, which is polynomial. The second part is the test and computation of every rule, which is polynomial since there are no ∇-sets, and the execution of every rule requires polynomial time in the input size. Moreover, the size of the problem is decreased in every step.

4.2 Exploiting Semantic Equalities

Since we focus application of the algorithms in (functional) higher-order programming languages, it makes sense to take more semantic equations and properties into account to recognize semantic equality of syntactically different expressions, which improves the power of generalization algorithms.

Since there are various approaches and definitions to semantics, like variants of contextual equivalences or bisimulations [9,14,15,19] and we want to be consistent with most of them, we only investigate the equalities that are correct in a majority of the cases. By "cases" we mean different programming languages permitting `letr`, but with different operational and equational semantics.

The following semantically correct equalities, expressed as rewrite rules, in languages with letrec could also be used for further standardization of expressions, where we assume that there are no conflicts with variable names.

1. $x.f(s_1,\ldots,s_n) \to x.f(y_1,\ldots,y_n); y_1.s_1; \ldots; y_n.s_n$
2. $\texttt{let}\ (x = \texttt{letr}\ env\ \texttt{in}\ r); env'\ \texttt{in}\ s \to \texttt{let}\ x = r; env; env'\ \texttt{in}\ s$
3. $\texttt{let}\ env\ \texttt{in}\ (\texttt{let}\ env'\ \texttt{in}\ s) \to \texttt{let}\ env; env'\texttt{in}\ s.$
4. $f\ (\texttt{let}\ env\ \texttt{in}\ s_1)\ s_2 \to \texttt{let}\ env\ \texttt{in}\ (f\ s_1\ s_2).$

Note that these equalities if used to standardize expressions keep the polynomial complexity of generalizations of ground expressions.

5 Conclusion and Future Work

We formulated an anti-unification algorithm for expressions in a functional higher-order language with a let constructor that has mutually recursive bindings. We constructed a weakly complete anti-unification algorithm that in the general case is finitary, which is improved to being complete by a post-processing. In the worst case, the time for the computation as well as the number of generalizations are exponential.

In case the expressions are specialized to be ground and garbage-free, then the problem becomes unitary and the computation is polynomial. These properties make the method more friendly to applications. We also considered modifications of the generalization algorithm for functions in functional programming

languages with `letr` that has a wider coverage by abstracting from the syntactical details and by observing semantic equalities.

Further work is to generalize algorithms to other patterns and to experiment with the generalization method in practice.

References

1. Ariola, Z.M., Blom, S.: Skew confluence and the lambda calculus with letrec. Ann. Pure Appl. Log. **117**(1–3), 95–168 (2002). https://doi.org/10.1016/S0168-0072(01)00104-X
2. Babai, L.: Graph isomorphism in quasipolynomial time (2016). http://arxiv.org/abs/1512.03547v2
3. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Nominal anti-unification. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 2015. LIPIcs, Warsaw, Poland, 29 June–1 July 2015, vol. 36, pp. 57–73. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.57. http://www.dagstuhl.de/dagpub/978-3-939897-85-9
4. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Higher-order pattern anti-unification in linear time. J. Autom. Reason. **58**(2), 293–310 (2016). https://doi.org/10.1007/s10817-016-9383-3
5. Calvès, C., Fernández, M.: Matching and alpha-equivalence check for nominal terms. J. Comput. Syst. Sci. **76**(5), 283–301 (2010)
6. Cerna, D.M., Kutsia, T.: Anti-unification and generalization: a survey. https://doi.org/10.48550/arXiv.2302.00277
7. Haskell: Haskell, an advanced, purely functional programming language (2019). www.haskell.org
8. Li, H., Thompson, S.: Similar code detection and elimination for erlang programs. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 104–118. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11503-5_10
9. Moran, A., Sands, D., Carlsson, M.: Erratic fudgets: a semantic theory for an embedded coordination language. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 85–102. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48919-3_8
10. Pfenning, F.: Unification and anti-unification in the calculus of constructions. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS 1991), Amsterdam, The Netherlands, 15–18 July 1991, pp. 74–85. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151632
11. Pitts, A.: Nominal techniques. ACM SIGLOG News **3**(1), 57–72 (2016). https://doi.org/10.1145/2893582.2893594
12. Plotkin, G.D.: A note on inductive generalization. Mach. Intell. **5**(1), 153–163 (1970)
13. Reynolds, J.C.: Transformational systems and algebraic structure of atomic formulas. Mach. Intell. **5**, 135–151 (1970)
14. Sabel, D., Schmidt-Schauß, M.: A contextual semantics for concurrent haskell with futures. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th ACM PPDP 2011, pp. 101–112. ACM (2011)
15. Sangiorgi, D., Walker, D.: On barbed equivalences in π-calculus. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 292–304. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44685-0_20

16. Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M., Kutz, Y.D.K.: Nominal unification and matching of higher order expressions with recursive let. Fund. Inform. **185**(3), 247–283 (2022)
17. Schmidt-Schauß, M., Nantes-Sobrinho, D.: Nominal anti-unification with atom-variables. In: Felty, A.P. (ed.) 7th FSCD 2022. LIPIcs, Haifa, Israel, vol. 228, pp. 7:1–7:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.7
18. 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)
19. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker's strictness analysis. J. Funct. Program. **18**(04), 503–551 (2008). https://doi.org/10.1017/S0956796807006624
20. Schöning, U.: Graph isomorphism is in the low hierarchy. J. Comput. Syst. Sci. **37**(3), 312–323 (1988)
21. Watt, S.M.: Algebraic generalization. SIGSAM Bull. **39**(3), 93–94 (2005). https://doi.org/10.1145/1113439.1113452