Nominal Anti-Unification of Letrec-Expressions

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

[1] Dept. Computer Science and Mathematics,
Goethe-university Frankfurt, Germany
schauss@ki.informatik.uni-frankfurt.de
[2] Department of Computing, Imperial College London
Department of Mathematics, University of Brasília, Brazil
dnantes@unb.br

Abstract

This is a work-in-progress where we present a rule-based algorithm for nominal anti-unification of expressions in a higher-order language with lambda abstraction, functions symbols and recursive let. The algorithm is sound and weakly complete, but has high complexity.

1 Introduction

Our goal is to develop generalization algorithms for a `letrec` (programming) language with potential applications for example for Haskell programs (see for example [2] for more references, syntax, unification and more about letrec-languages.) We combine this with the techniques of [3] and apply it to letrec-languages. The semantics is a specialization of that for atom-variables, such that the infinite decreasing chains in anti-unification are avoided. The result is an anti-unification algorithm which outputs a weakly complete set of generalizations, which means complete up to the freshness context. Optimizations are left for future research.

2 Preliminaries

We define the nominal `letrec` language $\mathtt{NLL}_X$, which can be seen as a lambda calculus extended with a recursive let construct, and its sublanguage `NLL` consisting of ground expressions (without variables and permutations). For notation and unification in letrec-language see [2].

We consider a countable infinite set of atoms $\mathbb{A}$, as 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). We also consider a set $\mathcal{F}$ of function symbols with arity $ar(\cdot)$, and a countably infinite set of variables *Var* ranged over by X, Y. We will use mappings on atoms from $\mathbb{A}$. A *swapping* $(a\ b)$ is a bijective function (on `NLL`-expressions) that maps an atom a to atom b, atom b to a, and is the identity on other atoms. We will also use finite permutations π on atoms from $\mathbb{A}$, which could be represented as a composition of swappings in the algorithms below. Let $dom(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq a\}$. Then every finite permutation can be represented by a composition of at most $(|dom(\pi)| - 1)$ swappings. Composition $\pi_1 \circ \pi_2$ and inverse π^{-1} can be immediately computed, where the complexity is polynomial in the size of $dom(\pi)$.

The syntax of the expressions e of $\mathtt{NLL}_X$ is:

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

We assume that binding atoms $a_1, \ldots, a_n$ in a letrec-expression $(\mathtt{letr}\ a_1.e_1; \ldots; a_n.e_n\ \mathtt{in}\ e)$ are pairwise distinct. Sequences of bindings $a_1.e_1; \ldots; a_n.e_n$ may be abbreviated as *env* (as

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

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

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

Figure 1: Simplification of freshness constraints in $\texttt{NLL}_X$

a short for environment). The order of bindings can be swapped in environments. The expressions $(\texttt{letr}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$ and $(\texttt{letr}\ a_{\rho(1)}.e_{\rho(1)}; \ldots; a_{\rho(n)}.e_{\rho(n)}\ \texttt{in}\ e)$ are defined as equivalent for every permutation ρ of $\{1, \ldots, n\}$, i.e., in the following we view the environment $a_1.e_1; \ldots; a_n.e_n$ of a letrec-expression as a multiset. In algorithms this leads to an additional non-deterministic step of permuting environments.

As an example, the expression $(\texttt{letr}\ a.cons\ s_1\ b; b.cons\ s_2\ a\ \texttt{in}\ a)$ represents an infinite list $(cons\ s_1\ (cons\ s_2\ (cons\ s_1\ (cons\ s_2\ \ldots))))$, where s_1, s_2 are expressions. The functional application operator in functional languages (which is usually implicit) can be encoded by a binary function `app`, which also allows to deal with partial applications.

The *scope* of atom a in $\lambda a.e$ is standard: a has scope e. The `letr`-construct has a special scoping rule: in $(\texttt{letr}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e)$, every atom a_i that is free in some e_j or e is bound by the environment $a_1.e_1; \ldots; a_n.e_n$. This defines in NLL 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 $a\#e$). The α-equivalence is defined only on NLL (for more details see [2]).

Permutations π operate on expressions simply by recursing on the structure. For a letrec-expression this is $\pi \cdot (\texttt{letr}\ a_1.e_1; \ldots; a_n.e_n\ \texttt{in}\ e) = (\texttt{letr}\ \pi \cdot a_1.\pi \cdot e_1; \ldots; \pi \cdot a_n.\pi \cdot e_n\ \texttt{in}\ \pi \cdot e)$. More generally, for a non-variable expression e, the expression $\pi{\cdot}e$ means an operation, which is performed by shifting π down, using the additional simplification $\pi_1{\cdot}(\pi_2{\cdot}e) \to (\pi_1 \circ \pi_2){\cdot}e$, where after the shift, π only occurs in the subexpressions of the form $\pi \cdot X$, which are called *suspensions*. Usually, we do not distinguish X and $Id{\cdot}X$. A single *freshness constraint* in our anti-unification algorithm is of the form $a\#e$, where e is an $\texttt{NLL}_X$-expression, and an *atomic* freshness constraint is of the form $a\#X$. A conjunction (or set) of freshness constraints is sometimes called *freshness context*.

Lemma 2.1. *The rules in Fig. 1 for simplifying sets of freshness constraints in* $\texttt{NLL}_X$ *run in polynomial time and the result is either* $\bot$*, i.e. fail, or a set of freshness constraints where all single constraints are atomic. This constitutes a polynomial decision algorithm for satisfiability of* ∇*: If* $\bot$ *is in the result, then unsatisfiable, otherwise satisfiable.*

The intended semantics is that the results are independent of the (explicit) names of atoms. Thus the choice of a fresh atom in the algorithm is not a choice-point, but a deterministic step.

Definition 2.2. *A expression-in-context is a pair* (∇, t)*, where* t *is an expression and* ∇ *is a freshness context. The semantics is a set of valid instances as follows:*

$[\![(\nabla, t)]\!] := \{\varphi(t) \mid \varphi(\nabla)$ *holds where* $\varphi : Var \cup \mathbb{A} \to \texttt{NLL}$ *and* φ *acts as a bijection on atoms*$\}$.

An expression-in-context (Δ, r) *is* more general *than an expression-in-context* (∇, s)*, denoted* $(\Delta, r) \preceq (\nabla, s)$*, if* $[\![(\nabla, r)]\!] \subseteq [\![(\Delta, s)]\!]$*. This defines equivalence of two expressions-in-context by having the same 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* $(\Delta, r) \preceq (\nabla', t)$*. The strict part of* $\preceq$ *is denoted* $\prec$*.*

This semantics avoids the effect that freshness constraints of the form $a\#X$ where a is an irrelevant atom that leads to infinite sets of generalizations. In fact, this differs from Baumgartner and Kutsia [1], since the following $(\emptyset, X) \prec (\{a\#X\}, X) \prec (\{a\#X, b\#X\}, X) \prec \ldots$ does not represent an infinite descending chain of expressions-in-context. The semantics of these pairs remains the same: The inclusion $[\![(\{a\#X, b\#X\}, X)]\!] \subseteq [\![(\{a\#X\}, X)]\!]$ holds trivially. For the opposite inclusion, it is enough to observe that one can always find a ground ρ such that $\rho(X) \in [\![(\{a\#X\}, X)]\!]$ and $\rho(\{a\#X, b\#X\})$ holds, therefore, $\rho(X) \in [\![(\{a\#X, b\#X\}, X)]\!]$. The same reasoning can be used for other pairs more/less restricted in this "chain".

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 *generalization*, i.e., another expression-in-context (Δ, r) that satisfies $(\Delta, r) \preceq (\nabla, s)$ and $(\Delta, r) \preceq (\nabla, t)$, and is the *least general* one, that is, there is no other generalization (Δ', r') of (∇, s) and (∇, t) which satisfies $(\Delta, r) \prec (\Delta', r')$.

Example 3.1. *A generalization for the expressions-in-context* $(\emptyset, \mathtt{letr}\ a.a; b.c\ \mathtt{in}\ f(a,b))$ *and* $(\emptyset, \mathtt{letr}\ b.a; c.c\ \mathtt{in}\ f(a,b))$ *is* $(\emptyset, \mathtt{letr}\ d.X_1; e.X_2\ \mathtt{in}\ f(X_1, (c\ e)(e\ d) \cdot X_2))$*, where* d, e *are names that are fresh w.r.t. the environments of both expressions-in-context and* X_1, X_2 *are new variables. In fact, since environments can be permuted it follows that* $[\![(\emptyset, \mathtt{letr}\ b.a; c.c\ \mathtt{in}\ f(a,b))]\!] = [\![(\emptyset, \mathtt{letr}\ c.c; b.a\ \mathtt{in}\ f(a,b))]\!]$ *and another generalization can be found* $(\emptyset, \mathtt{letr}\ d.d; e.X_2\ \mathtt{in}\ f((c\ d) \cdot X_2, b))$*. These generalizations can be obtained via application of our anti-unification algorithm that will be presented next.*

3.1 The Algorithm AntiUnifLetr and its Rules

We first define the nominal generalization algorithm AntiUnifLetr that computes a single generalization of the input expressions, where the generalization can also be nonlinear in the generalization variables due to merging. It relies on the subalgorithm EQVM that computes a permutation similar to equivariance matching. We claim that the algorithm is sound and weakly complete, and can be performed in exponential time.

The data structure of the algorithm AntiUnifLetr is a tuple (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; L is a substitution represented as a list of bindings; the empty list is denoted as $[]$.

We call such a tuple a *state*. The rules of the AntiUnifLetr, given in Fig. 2, operate on states. Given two NLL expressions s and t, and a freshness context ∇ (possibly empty), to compute generalizations for (∇, s) and (∇, t), we start with $(\{X : s \triangleq t\}, \emptyset, \nabla, [])$, the *initial state* (sometimes we abbreviate it to $(\nabla, \{X : s \triangleq t\})$), where X is a fresh generalization variable, and we apply the rules from Fig. 2 and Fig. 4 as long as possible, until no more rule application is possible and we reach the *final state* which has the form $(\emptyset, M, \Delta, L)$. The output is a expression-in-context obtained from the generated substitution L and the final freshness constraint Δ, i.e. the output is $(\Delta, X \circ L)$, also called the *result computed* by the AntiUnifLetr algorithm. We say it is *weakly complete* if every generalization is covered up to the freshness constraints.

(`Dec`): Decomposition
$$\frac{\{X{:}f(s_1,\ldots,s_n) \triangleq f(t_1,\ldots,t_n)\}\cup\Gamma, M, \nabla, L}{\Gamma\cup\{X_1{:}s_1 \triangleq t_1,\ldots,X_n{:}s_n \triangleq t_n\}, M, \nabla, L\cup\{X \mapsto f(X_1,\ldots,X_n)\}}$$
where X_i are fresh variables

(`Absaa`): Abstraction
$$\frac{\{X{:}\lambda a.s \triangleq \lambda a.t\}\cup\Gamma, M, \nabla, L}{\Gamma\cup\{Y{:}s \triangleq t\}, M, \nabla, L\cup\{X \mapsto \lambda a.Y\}}$$

(`Absab`): Abstraction
$$\frac{\{X{:}\lambda a.s \triangleq \lambda b.t\}\cup\Gamma, M, \nabla, L \qquad \nabla' = \{c\#\lambda a.s, c\#\lambda b.t\}}{\Gamma\cup\{Y{:}(c\ a)\cdot s \triangleq (c\ b)\cdot t\}, M, \nabla\cup\nabla', L\cup\{X \mapsto \lambda c.Y\}}$$
where Y is a fresh variable, and c is a fresh atom

(`SusYY`): SuspensionYY
$$\frac{\{X{:}\pi_1\cdot Y \triangleq \pi_2\cdot Y\}\cup\Gamma, M, \nabla, L \quad \nabla \vDash \pi_1 = \pi_2}{\Gamma, M, \nabla, L\cup\{X \mapsto \pi_1\cdot Y\}}$$

(`Mer`): Merging
$$\frac{\Gamma, \{X{:}s_1 \triangleq t_1, Y{:}s_2 \triangleq t_2\}\cup M, \nabla, L \quad EQVM(\{(s_1,t_1) \preceq (s_2,t_2)\}, \nabla) = \pi}{\Gamma, M\cup\{X{:}s_1 \triangleq t_1\}, \nabla, L\cup\{Y \mapsto \pi\cdot X\}}$$

(`SolveYY`)
$$\frac{\{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\}\cup\Gamma, M, \nabla, L \quad \nabla \nvDash \pi_1 \neq \pi_2}{\Gamma, M\cup\{X{:}\pi_1 \cdot Y \triangleq \pi_2 \cdot Y\}, \nabla, L}$$

(`Solve`)
$$\frac{\{X{:}s \triangleq t\}\cup\Gamma, M, \nabla, L}{\Gamma, M\cup\{X{:}s \triangleq t\}, \nabla, L}$$
If $\mathit{Head}(s) \neq \mathit{Head}(t)$ or s,t are letrec-expressions with a different number of bindings.

Figure 2: Rules of the algorithm AntiUnifLetr

$$\frac{\Psi\cup\{e \preceq e\}, \Pi, \nabla}{\Psi, \Pi, \nabla} \qquad \frac{\Psi\cup\{(f\ s_1\ldots s_n) \preceq (f\ s'_1\ldots s'_n)\}, \Pi, \nabla}{\Psi\cup\{s_1 \preceq s'_1,\ldots,s_n \preceq s'_n\}, \Pi, \nabla}$$

$$\frac{\Psi\cup\{\pi_1\cdot X \preceq \pi_2\cdot X\}, \Pi, \nabla \quad \nabla \vDash \pi_1\cdot X = \pi_2\cdot X}{\Psi, \Pi, \nabla} \qquad \frac{\Psi\cup\{\lambda a.s \preceq \lambda a.t\}, \Pi, \nabla}{\Psi\cup\{s \preceq t\}, \Pi, \nabla}$$

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

$$\frac{\Psi\cup\{a \preceq b\}, \Pi, \nabla}{\Psi, \{a \mapsto b\}\cup\Pi, \nabla} \qquad \frac{\emptyset, \Pi, \nabla \qquad EQVBiEx(\Pi) = \pi}{\text{Return } \pi}$$

Figure 3: Rules of the permutation matching algorithm Eqvm

Rules in Fig. 2 are similar to the ones in [1] without the parameter for the set of atoms occurring in the initial state, and deal with abstractions, function application, and suspensions. The subalgorithm Eqvm, defined by the rules in Fig. 3, computes a matching permutation of two expressions-in-context (in Ψ with context ∇), where $EQVBiEx(\Pi)$ 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 deal with letrec.

(Letraa): LetrecEq

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

(Letperm): LetrecEq

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

where ρ a permutation on the set $\{1,\ldots,n\}$

(Letrab): Letrec

$$\frac{\{X{:}\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s \triangleq \texttt{letr}\ b_1.t_1,\ldots,b_n.t_n\ \texttt{in}\ t\}\cup\Gamma, M, \nabla, L}{\{X{:}\pi_1\cdot(\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s) \triangleq \pi_2\cdot(\texttt{letr}\ b_1.t_1,\ldots,b_n.t_n\ \texttt{in}\ t)\}\cup\Gamma, M, \nabla\cup\nabla', L}$$

where $\pi_1 = (a_1\ c_1)\ldots(a_n\ c_n)$ and $\pi_2 = (b_1\ c_1)\ldots(b_n\ c_n)$ and where c_i are fresh and different atoms and $\nabla' = \{c_i\#(\texttt{letr}\ a_1.s_1,\ldots,a_n.s_n\ \texttt{in}\ s)\}\cup\{c_i\#(\texttt{letr}\ b_1.t_1,\ldots,b_n.t_n\ \texttt{in}\ t)\}$ and $i = 1,\ldots n$.

Figure 4: Rules for `letrec` of the algorithm AntiUnifLetr

Theorem 3.2. *The algorithm* AntiUnifLetr *is terminating, sound and weakly complete.*

Example 3.3 (Cont. Example 3.1)**.** *The generalization presented for* $(\emptyset, \texttt{letr}\ a.a; b.c\ \texttt{in}\ f(a,b))$ *and* $(\emptyset, \texttt{letr}\ b.a; c.c\ \texttt{in}\ f(a,b))$ *was obtained via application of rules* (Letrab) *and* (Letraa) *to deal with* `letr`*, followed by application of other rules from Figure 2, including* Eqvm*. Another generalization can be obtained if we apply* (Letrperm) *before the application of* (Letrab)*:* $(\emptyset, \texttt{letr}\ d.d; e.X_2\ \texttt{in}\ f((c\ d)\cdot X_2, b))$*. The relation between different generalizations will be explored in future work.*

4 Conclusion and Further Work

We formulated an anti-unification algorithm for expressions in a higher-order language with letrec. The conjecture is that this algorithm is (non-deterministic) polynomial and weakly complete. We still need to prove that the obtained results are the least general generalizations and we conjecture that the solution type is finitary. Further work will be to develop a more practical algorithm for functional programming languages with `letr` that may use also semantical equivalences (like garbage-collection) in order to obtain a polynomial algorithm for a relevant set of generalizations.

References

[1] Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 57–73. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[2] Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, Mateu Villaret, and Yunus D. K. Kutz. Nominal unification and matching of higher order expressions with recursive let. *Fundamenta Informaticae*, 185(3):247–283, 2022. to be published.

[3] Manfred Schmidt-Schauß and Daniele Nantes-Sobrinho. Nominal anti-unification with atom-variables. In Amy Felty, editor, *FSCD 2022*, LIPIcs. Schloss Dagstuhl, 2022. Accepted for publication.