GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik

Institut für Informatik

Masterarbeit

A Complete Unification Algorithm for Nominal Unification with Atom Variables

Yunus David Kerem Kutz

28.04.2017

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Institut für Informatik

Erklärung gemäß Master-Ordnung Informatik 2015 § 35 Abs. 16

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe. Ebenso bestätige ich, dass weder diese Arbeit noch Auszüge daraus für eine andere Prüfung oder Studienleistung verwendet wurden.

Ort, Datum

Unterschrift

Contents

Abstract

Unification describes the techniques of making sets of syntactically different equations equal by substituting variables with appropriate terms for the given context. In the nominal language with binders equality is defined up to alpha-equivalence and a unification replaces expression variables with nominal expressions. It is efficiently implementable[12] and hence, has found its applications in e.g. logic programming languages such as Alpha-Prolog.

However, in the base language bound names cannot be replaced, since names are always considered to be concrete rather than variables. Some research has been done on the unification problem with atom variables[11, 4, 3, 10].

In this thesis the unification problem with atom variables is further investigated. Moreover, as a novel result a proof for the existence of most general is provided. At last, an implementation of Schmidt-Schauß et.al.'s algorithm[11] in Haskell is outlined.

Chapter 1

Introduction

The nominal language serves as the ground language in nominal rewriting approaches [7] and in logic programming languages such as Alpha-Prolog[6, 5].

It consists of atoms/names (hence nominal language), binding constructs and functions, which are defined in the embedding. Names/Atoms differ from variables in that they are inherently interpreted to be unequal, i.e. $a \neq b$ for two different atoms a and b.

One of the advantages of this language lies in the ease with which alpha-equivalence is provable, since the concept of renaming is provided by adding name swappings to the language and the concept of a variable not occurring free in an expression by so called freshness constraints. Thus, two abstractions with different bound names are α-equivalent iff former's names is fresh in the latter expression, and the expression terms are α-equivalent modulo swapping. This can be written simply as:

$$\lambda a.e_1 \equiv_\alpha \lambda b.e_2 \iff a\#e_2 \wedge e \equiv_\alpha (a,b)e_2$$

The construct$a\#e_2$ is the freshness constraint and requires a not occurring free in e_2. The term $(a,b)e_2$ simply means, that a and b are swapped in e_2, which can be done instantaneously.

To define practical nominal rewrite rules, the nominal language needs to be extended by expression variables, for otherwise only rewriting on fully defined terms would be possible. In [7] a nominal rewrite rule is defined as a set of constraints ∇ and two expressions$l \rightarrow r$, with the idea being, that an expression can be rewritten to r, if it is unifiable with l and satisfies all the constraints in ∇ after unification.

As luck would have it, an efficient unification algorithm for the nominal language with expression variables exists[12], which also serves as the basis for the unification algorithm in Alpha-Prolog[5]. There are some issues with this approach however.

Consider for example the nominal rewriting system defined in [7] for β-reduction.

Example. As functions we use $\{App, sub, to\}$ each of arity 2. Furthermore we use infix notation for *to* and omit *App* for readability. As expression variables we write S_1, S_2, S_3 and as the only atom in these rules x.

1. $(\lambda x.S_1)\ S_2 \to sub\ S\ (x\ to\ S_2)$
2. $sub\ (S_1\ S_2)\ (x\ to\ S_3){\to}(sub\ S_1\ (x\ to\ S_3))\ (sub\ S_2\ (x\ to\ S_3))$
3. $sub\ x\ (x\ to\ S_1){\to}S_1$
4. $x\#S_1 \wedge sub\ S_1\ (x\ to\ S_2){\to}S_1$
5. $y\#S_2 \wedge sub\ (\lambda y.S_1)\ (x\ to\ S_2){\to}\lambda y.(sub\ S_1\ (x\ to\ S_2))$

The left-hand side of the third rule $sub\ x\ (x\ to\ S_1)$ and the expression $sub\ y\ (y\ to\ S_1)$ are not unifiable for two different names x and y. To get around this, a nominal rewriting system is defined as the equivariant hull, i.e. closure under names swappings, of nominal rewrite rules. This means however, that a different kind of unification is necessary, called equivariant unification, which is not efficiently decidable[4, 3].

There are also cases, where you might want to employ nominal rewrite rules with a modulo laxer than equivariance.

Example. Consider the first order theory with the axioms:

A1 $(\exists x.P(x) \wedge \exists y.Q(x, y)) \implies \varphi$

A2 $\exists x.P(x) \wedge Q(x, x)$

The second axiom is equivalent to $\exists x.P(x) \wedge \exists y.Q(x, x)$.

An embedding into the nominal language might transform the $\exists$ into λ, since both bind a variable and a rewrite system might look somewhat like:

1. $\lambda X.And\ (P\ X)\ (\lambda y.Q\ X\ Y) \to \varphi'$
2. $\bot \to \lambda X.And\ (P\ X)\ (\lambda y.Q\ X\ X)$

Where the second rule says that the right-hand side can be constructed out of nothing. The left-hand side of the first rule $\lambda X.And\ (P\ X)\ (\lambda Y.\ Q\ X\ Y)$ and the term $\lambda X.And\ (P\ X)\ (\lambda Y.Q\ X\ X)$ would need to be unifiable to proof φ', which requires X and Y not to be concrete atoms, but rather atom variables and the unification to be not (necessarily) to be equivariant.

Schmidt-Schauß et.al. developed a unification algorithm for this case[11], that computes a unifier for such a unification problem, i.e. a nominal unification problem with atom variables and expression variables, in polynomial time, and checks its satisfiability in non-deterministic polynomial time.

This thesis defines the ground nominal language NL_a and its extensions NL_{aS} which is well researched[7, 12, 9, 1], and NL_{AS} which is not.

It will give a detailed analysis of the latter one and as an original result prove the existence of most general unifiers. At last we will explain Schmidt-Schauß's unification algorithm[11] and showcase an implementation in Haskell.

Chapter 2

Mathematical Tools

In this chapter permutations and substitutions are quickly recaptured.

2.1 Permutations

In Algebra, permutations on a set S are defined as bijections from the set S to itself. For all finite sets S, the permutations form a non-abelian group called the symmetric group $Sym(S)$, which up to isomorphism depends only on the cardinality of the set.

Every infinite bijection which differs from the identity function on only a finite number of elements, can be represented as a bijection on only those elements. Thus, all renamings we deal with, have a finite representation as a permutation.

Definition 2.1. Let G be a group, X an arbitrary set and $\circ$ be a binary function from $G \times X$ to X. Then $\circ$ is a group action iff:

1. $(gh) \circ x = g(h \circ x)$
2. $id \circ x = x$

This brings as to our first proposition

Proposition 2.2. $(Sym(S), \circ)$ *forms a group, and the application on* S *defines a group action on* S.

We will usually omit the $\circ$ symbol for both group action and group operation.

Another useful fact for finite permutations is that every such permutation is equal to a composition of swappings(transpositions). A swapping refers to a bijection, which only differs from the identity on two elements and thus swaps them. We write the swapping of a and b as (a, b).

Consequently, the result of the application is:

- $(a, b)a = b$

- $(a,b)b = a$
- $(a,b)c = c$ for $c \notin \{a,b\}$

The representation as a composition of swappings is not unique, e.g. $(ab)(bc) = (ca)(ab)$. But since application forms a group action, the image of every element is independent of the representation.

We use this to identify an ordered list of swappings with its induced permutation and use the application of a swapping on an element as a full definition for the application of the list on S.

Another helpful result of the representation as a list of swappings is the ease with which the inverse can be computed. Since a swapping (a,b) is its own inverse, a list of swappings can be simply reversed to obtain the permutations inverse.

2.2 Substitutions

The exact definition of a substitution depends on any given context. However, some common features of substitutions are dealt with here.

Basic nomenclature and definitions

Given a language L with a set of variables V belonging to it, a substitution is defined as a partial function from V to L.

The domain of a substitution sigma, written $dom(\sigma)$, consists of all variables which are mapped to terms of the language not considering variables which are mapped to themselves. The codomain, written $codom(\sigma)$, consists of all terms to which variables in the domain are mapped to.

The application of a substitution σ on a term $t \in L$ is usually written as $t\sigma$ (postfix notation) or $\sigma(t)$. The result of the application is a term in which the variables of the substitution are replaced by the terms defined in the substitution.

Remark 2.3. The resulting term is not necessarily a term of the language, nor do all kinds of substitutions require all variables in the term, which occur in the domain to be replaced. Consider for example the substitution$\sigma = \{x \to f(a)\}$ on a First-Order language with a signature consisting of one function f of arity 1, and a constant a.
If we take for example the formula $e := x = f(a) \wedge \exists x.x = a$, we see x appearing once bound and once free. A First-Order substitution is commonly defined to only replace the free occurrences of a variable. Thus, we obtain $e\sigma = f(a) = f(a) \wedge \exists x.x = a$. Otherwise the result would be $e\sigma = f(a) = f(a) \wedge \exists f(a).f(a) = a$, which is not a First Order formula and which may or may not have a useful semantics (again, depending on the context).

Definition 2.4. The composition of two substitutions $\sigma = \{x_1 \to t_1, \ldots, x_1 \to t_1$, $\tau = \{y1 \to r1, \ldots, yk \to rk\}$ is defined as:

- $(\sigma \circ \tau)(x) = \tau(x)$ iff $x \notin dom(\sigma)$
- $(\sigma \circ \tau)(x)$=$(\sigma(x))\tau = \tau(\sigma(x))$ else

Intuitively this simply means, that σ is applied first and τ to the resulting term.

At last, a variable free term is referred to as ground and hence a substitution which maps variables only to ground terms is called a ground substitution.

Compression via composition

Given a set representation of substitutions, i.e. $\sigma = \{x_1 \to t_1, \ldots, x_n \to t_n\}$, the composition of substitutions can easily result in an exponentially larger data structure than the original substitutions.

Take for example the n substitutions $\sigma_1 = \{x_0 \to App\ x_1\ x_1\}$, ..., $\sigma_n = \{x_{n-1} \to App\ x_n\ x_n\}$. The composition $\sigma_1 \circ \cdots \circ \sigma_n$ has size$O(n)$ if it is not calculated. The calculated result is: $\{x_0 \to App(\ldots(App(Appx_nx_n)(Appx_nx_n))\ldots)\}$ which has size $O(2^n)$.

Since the application on a term in composition form can still be done efficiently, it is often more practical to not explicitly calculate the composition.

Chapter 3

Of Atoms and Variables

In this chapter, we define the ground nominal language NL_a and the extension NL_{aS}, which adds expression variables to the ground terms.

We also introduce the nominal unification problem in NL_{aS} – a concept which will reappear in the next chapter with all its necessary sub concepts like freshness constraints, α-equivalence, solutions and unifiers.

3.1 The Nominal Language - NL_a

The nominal language NL_a serves as the ground language for nominal unification problems. It consists of atoms, abstractions and functions. The definition via a grammar is:

Definition 3.1. Let $\mathcal{F}$ be set of functions, with each function $f \in \mathcal{F}$ being of fixed arity $ar(f)$. Let $\mathcal{A}t$ be a countable infinite set of atoms. The syntax of NL_a is then defined as:

$$e := a \mid f\ e_1 \dots e_{ar(f)} \mid \lambda a.e$$

As a convention we use letters a, b, c to denote atoms which are known to be distinct and x, y, z for atoms, which may be equal. This may seem a bit confusing at first but will proof useful, since you can distinguish between unequal and possibly equal variables.

For induction proofs, we define the depth of expressions in NL_a.

Definition 3.2. The depth on NL_a is defined as:

$$\begin{aligned}
&\mathrm{depth}(a) = 1\\
&\mathrm{depth}(\lambda a.e) = \mathrm{depth}(e) + 1\\
&\mathrm{depth}(f\ e_1 \dots\ e_k) = \max\{\mathrm{depth}(e_i) : i \in \{1, \dots, n\}\} + 1
\end{aligned}$$

This way any proof by induction over the depth needs to consider only atoms in the base case and function applications and λ-abstractions in the induction step, where the induction hypothesis holds for the subexpressions.

Alpha equivalence is traditionally defined by renaming bound variables, or names in our case.

Definition 3.3. An α-conversion $e \to e'$ is a capture avoiding renaming of a single bound atom in a single (sub)abstraction of e. Two NL_a expressions e_1, e_2 are α-equivalent, written $e_1 \equiv_\alpha e_2$, iff they are reducible to each other by α-conversions.

Remark 3.4. We defined α-conversion on single bound names and subexpressions to make the definition precise, though other definitions are possible. Later, we will give an alternative inductive characterization using permutation application. Note that two atoms are α-equivalent if they are the same atom, i.e. $a \equiv_\alpha a$ but $a \not\equiv_\alpha b$ and two function applications are α-equivalent if all sub expressions are α-equivalent, i.e. $f\ e_1 \dots e_{ar(f)} \equiv_\alpha f\ e_1 \dots e_{ar(f)} \iff \forall i : e_i \equiv_\alpha e_i$.

We will use an alternative definition equivalent to this one, which uses renamings on all atoms, instead of just bound ones. This has the advantage of not having to check, if an atom occurs free or bound in any given context and formulate context free rewrite rules.

Every finite renaming on the set of atoms can be represented as a finite list of swappings (2.1), i.e. permutations (a, b) with $(a, b)a = b$, $(a, b)b = a$ and $(a, b)c = c$ for atoms a, b, c. Thus, we use such representations throughout this paper and make no distinction between the representation and the permutation. As mentioned in (2.1), the above description of the application result of swappings is sufficient to characterize permutation application on atoms.

The application of permutation NL_a is defined inductively.

Definition 3.5. The application of a permutation $\pi' = \pi(a, b)$ or $\pi' = \emptyset = id$ on NL_a is defined as:

$$\begin{aligned}
\pi(a,b)a &:= \pi b & \emptyset \cdot e &:= e \\
\pi(a,b)b &:= \pi a & \pi' \cdot (\lambda a.e) &:= \lambda \pi \cdot a.\pi' \cdot e \\
\pi(a,b)c &:= \pi c & \pi' \cdot (f\ e_1 \dots e_{ar(f)}) &:= f\ (\pi \cdot e_1) \dots (\pi \cdot e_{ar(f)})
\end{aligned}$$

We also formalize the idea, that composition of permutation and application on NL_a are interchangeable.

Lemma 3.6. *The permutation application on* NL_a *defines a group action.*

Proof. The application on $\mathcal{A}t$ defines a group action, since it is defined as in 2.1. The rest follows from induction. $\square$

Remark 3.7. For any concrete NL_a expression e the application πe can immediately be evaluated. This will not hold, once we use variables for expressions or atoms.

To define α-equivalence with the help of this permutation application, we need a second construct, called a freshness constraint. A freshness constraint has the form $a\#e$ and requires a to not occur as a free variable in e.

Definition 3.8. A valid freshness constraint is defined inductively by the following rules:

$$\frac{}{a\#b} \qquad \frac{}{a\#\lambda a.e} \qquad \frac{a\#e}{a\#\lambda b.e} \qquad \frac{\bigwedge_{i=1}^{ar(f)} a\#e_i}{a\#f\ e_1\dots e_{ar(f)}}$$

Note that both directions are valid, i.e. $a\#e \iff a\#\lambda b.e$.

Lemma 3.9. *A freshness constraint $a\#e$ holds iff a does not occur freely in e*

Proof. For the base case the hypothesis holds, since all $a\#b$ is valid and $a\#a$ is not. For function application we have:

$$\begin{aligned} a \in free(f\ e_1 \dots e_{ar(f)}) &\iff \bigvee_{i=1}^{ar(f)} a \in free(e_i) \\ &\iff \neg(\bigwedge_{i=1}^{ar(f)} a\#e_i) \qquad \iff \neg(a\#f\ e_1 \dots e_{ar(f)}) \end{aligned}$$

For abstractions we have $a \notin free(\lambda a.e)$ and $a\#e$ for different bound names and

$$a \notin free(\lambda b.e) \iff a \notin free(e) \iff a\#e \iff a\#\lambda b.e$$

□

As an immediate consequence we gain that constraints are invariant under renaming, i.e. $a\#e \iff \pi a\#\pi e$. Hence any constraint of the form $a\#\pi e$ can be rewritten as $\pi^{-1}a\#e$ and vice versa.

Now we have all the tools we need to define α-equivalence via permutations and constraints.

Definition 3.10. The syntactic α-equivalence on NL_a is inductively defined by:

$$\frac{}{a \sim a} \qquad \frac{e_1 \sim e_2}{\lambda a.e_1 \sim \lambda a.e_2} \qquad \frac{a\#e_2 \wedge\ e_1 \sim (a,b)e_2}{\lambda a.e_1 \sim \lambda b.e_2} \qquad \frac{\bigwedge_{i=1}^{ar(f)} : e_i \sim e_i'}{f\ e_1 \dots e_k \sim f\ e_1' \dots, e_{ar(f)}'}$$

Lemma 3.11. *Two expressions $e_1, e_2 \in \text{NL}_a$ are syntactic α-equivalent iff they are α-equivalent.*

Proof. The base case where $e_1 = a$ holds since atoms have no bound atoms and therefore:

$$e_2 \equiv_\alpha a \iff e_2 = a \iff e_2 \sim e_1$$

For the induction step we assume the statement to hold for expressions of depth $\text{depth}(e_1) \leq n$.

Case 1. $e_1 = f\ e^1_1 \dots e^1_{ar(f)}$

Then

$$\begin{aligned} e_2 \equiv_\alpha e_1 &\iff e_2 = f\ e^2_1 \dots e^2_{ar(f)} \ \wedge\ \forall i : e^1_i \equiv_\alpha e^2_i \\ &\iff e_2 = f\ e^2_1 \dots e^2_{ar(f)} \ \wedge\ \forall i : e^1_i \sim e^2_i \iff e_1 \sim e_2 \end{aligned}$$

Case 2. $e_1 = \lambda a.e'_1$

Then e_2 has the form $\lambda x.e'_2$. We distinguish between two subcases: $x = a$ and $x = b$.

Case i. $e_2 = \lambda a.e'_2$

Then no α-conversion is done on the top expression of e_2 to check α-equivalence. Hence:

$$e_2 \equiv_\alpha e_1 \iff e'_1 \equiv_\alpha e'_2 \iff e'_1 \sim e'_2 \iff e_1 \sim e_2$$

Case ii. $e_2 = \lambda b.e'_2$

If the constraint $a\#e'_2$ does not hold, then a occurs free in e_2 but not in e_1 and they are neither α-equivalent nor syntactically α-equivalent. Thus, the hypothesis holds.

Now assume $a\#e'_2$. Then the swapping (a, b) only substitutes b with a if applied on e'_2 and hence also on e_2, i.e. $(a, b)e_2 = e_2[b := a]$. Furthermore, since e_2 is an abstraction with b as its bound variable, this substitution is an α-conversion. Hence:

$$\begin{aligned} e_2 \equiv_\alpha e_1 &\iff e_2[b := a] \equiv_\alpha e_1 &&\iff (a, b)(\lambda b.e'_2) \equiv_\alpha \lambda a.e'_1 \\ &\iff (a, b)e'_2 \equiv_\alpha e'_1 &&\iff e_1 \sim (a, b)e'_2 \\ & &&\iff e_1 \sim e_2 \end{aligned}$$

□

Since the two definitions are equivalent, $\sim$ is an equivalence and a congruence relation. This type of syntactic α-equivalence, provides an intuitive alternative to de Bruijn Indices at the cost of often poorer performance. A more detailed analysis of explicit implementations is provided by Berghofer, S., & Urban, C. in [1].

3.2 The first extension - NL$_{aS}$

As a first addition to the basic nominal language, expression variables are added[7, 12]. We will use capital letters S_i to denote such variables. Expression variables can be substituted to arbitrary expressions with ground substitutions. To add permutation application to the language, one has to add not only expression variables, but also the application of permutations on such variables. Such constructs πS are called suspensions. This yields the formal definition of NL$_{aS}$

Definition 3.12. Let $\mathcal{F}$ be set of functions, with each function $f \in \mathcal{F}$ being of fixed arity $ar(f)$. Let $\mathcal{A}t$ be a countable infinite set of atoms and $ExVar$ be a countable infinite set of expression variables. The syntax of NL_{aS} is then defined as:

$$e := a \mid f\ e_1 \dots e_{ar(f)} \mid \lambda a.e \mid S \mid \pi S$$

As a first notation on NL_{aS} we introduce $ExVar(e)$.

Definition 3.13. For any $e \in \mathrm{NL}_{aS}$ the set of expression variables in e is defined as $ExVar(e)$.

A unification problem in NL_{aS} is given as a set of "equations", i.e. $e_1 \doteq e_2$, which are supposed to be α-equivalent, and a set of constraints on NL_{aS}, which are required to hold. To lift these previously defined concepts into the new language, one needs substitutions, which map every expression variable to an expression in NL_a.

Definition 3.14. A substitution σ on NL_{aS} is a finite mapping from $ExVar$ to NL_{aS}. A ground substitution is a substitution with $codom(\sigma) \subset \mathrm{NL}_a$. The application is defined as:

$$e\sigma = \sigma(e) := e'$$

where e' is constructed by replacing all S with $\sigma(S)$ in e.

To formally define a unification problem, we still need concepts of constraints and equations. In NL_{aS} however, these are abstract construct, where no immediate, complete semantic can be defined on. This is due to the fact, that equations of the type $S \doteq e$ or constraints of the type $a\#S$ are true or false only when S is substituted. In this section, we will abstain from defining any semantic on it and instead only define solutions of sets of constraints and equations. Solutions are a certain type of ground substitution.

Definition 3.15. A constraint in NL_{aS} is a construct of the type $a\#e$ for any expression $e \in \mathrm{NL}_{aS}$.

Definition 3.16. An equation in NL_{aS} is a construct of the type $e_1 \doteq e_2$ for any expressions $e_1, e_2 \in \mathrm{NL}_{aS}$.

Definition 3.17. A unification problem in NL_{aS} is a tuple $P = (\Gamma, \nabla)$ where Γ consists of equations and ∇ of constraints in NL_{aS}.

Definition 3.18. A solution of a unification problem $P = (\Gamma, \nabla)$ is a ground substitution γ with:

- $\forall e_1 \doteq e_2 \in \Gamma :\ e_1\gamma \sim e_2\gamma$
- $\forall a\#e \in \nabla :\ a\#e\gamma$ holds

A unification problem is solvable if it has a solution. A constraint set ∇ is solvable if $(\emptyset, \nabla)$ is solvable.

A unification problem will usually have more than one solution.

Example 3.19. Consider for example the unification problem

$$P = (\{S_2 \doteq \lambda a.S_3, S_3 \doteq S_4\}, \{a\#S_1\})$$

The ground substitution $\gamma = \{S_1 \to S_2\} \circ \{S_2 \to \lambda a.a, \to a, S_4 \to a\}$ is a solution. Another one would be $\gamma = \{S_1 \to S_3\} \circ \{S_2 \to \lambda a.b, S_3 \to b, S_4 \to b\}$. In fact, there are an infinite number of solutions for this problem.

We introduce unifiers as a way to generalize the multitudes of solutions.

Definition 3.20. A unifier for a unification problem $P = (\Gamma, \nabla)$ in NL_{aS} is a tuple $\mu = (\sigma, \nabla')$ where:

- ∇' is solvable.
- For all ground substitutions γ holds:

$$\sigma \circ \gamma \text{ is a solution of } \nabla' \implies \sigma \circ \gamma \text{ is a solution of } P.$$

Taking a look at the previous example one notices to immediate benefits. First, one is no longer restricted to ground substitutions, i.e. $\{S_1 \to S_2\}$ is a valid substitution on its own. Second, we are free to use constraints to capture solutions.

This can be used to capture the previous solutions with the unifier $(\{S_3 \to S_4, S_2 \to \lambda a.S_3\}, \{a\#S_1\})$. This is in fact a representation of all solutions for the unification problem, called a most general unifier.

Definition 3.21. A unifier $\mu = (\sigma, \nabla')$ for a unification problem P is most general if for all solutions γ of P there is a ground substitution ρ with:

$$\forall S \in \mathit{ExVar}(P) : \gamma(S) \sim \sigma\rho(S)$$

It has been proven by Urban, C., Pitts, A. M., & Gabbay, M. J. in [12] that the unification problem in NL_{aS} is efficiently decidable and most general unifiers are computable efficiently as well.

Theorem 3.22. *(Urban et al. (2003); Calves and Fernández (2008); Levy and Villaret (2008, 2010)). The unification problem in* NL_{aS} *is solvable in quadratic time. Furthermore, for a solvable nominal unification problem* P*, a most general unifier can be computed in polynomial time.*

Chapter 4

The Nominal Language with Atom Variables - NL$_{AS}$

We introduce a third extension to NL$_a$, which adds not only expression variables but also atom variables. As in NL$_{aS}$ we define its semantics with respect to NL$_a$. Unlike NL$_{aS}$ however, we do not deal with concrete atoms anymore, which changes the renaming aspect quite a bit.

Take for example the permutation application $(a, b)c$ in NL$_{aS}$ from [12]. It can immediately be evaluated to c. In contrast, the permutation application $(A, B)C$ in NL$_{AS}$ cannot be evaluated without further knowledge about the equality/inequality of A, B, C, since$(A, B)C = A$ would be implied by $B = C$ and $(A, B)C = C$ by $A \neq C, B \neq C$. Thus, similar to the expression suspensions in NL$_{aS}$, we need atom suspensions of the form πA, if we want to properly work with permutation application.

In this chapter, we will lift the concepts required to define a unification problem in NL$_{aS}$ to the new settings with atom variables. The same names and symbols will often be used, since specification would be cumbersome and the concepts in NL$_{aS}$ can be seen as a special case of the ones in NL$_{AS}$.

Next, we will prove, that the complexity of the unification problem in NL$_{aS}$ is NP-complete in general and in P in some specific instances.

At last the existence of most general unifiers will be proven, which until now was an open problem.

4.1 Definitions

The language NL$_{AS}$ can be defined by the following grammar.

Definition 4.1. Let $\mathcal{F}$ be set of functions, with each function $f \in \mathcal{F}$ being of fixed arity $ar(f)$. Let $AtVar$ be a countable infinite set of atoms and $ExVar$

be a countable infinite set of expression variables. The syntax of NL$_{aS}$ is then defined as:

$$e := A \mid \pi A \mid S \mid \pi S \mid f\ e_1 \dots e_{ar(f)} \mid \lambda A.e \mid \lambda \pi A.e$$

where πA is called an atom suspension, πS an expression suspension and all other expressions are compound terms.

Definition 4.2. Let e be an expression in NL$_{AS}$. Then $AtVar(e)$ refers to all atom variables, which occur in e, $ExVar(e)$ to all expression variables and $Var(e) = AtVar \cup ExVar$.

As in NL$_a$ we use letters A, B, C when we talk about concrete atom variables and X, Y, Z when we talk about unknown atom variables, i.e. when we do not want to distinguish between $X = A$ and $X \neq A$. Furthermore, we define tools such as the depth to make proves by induction easier.

Definition 4.3. The depth on NL$_{AS}$ is defined as:

$$\begin{aligned}
\text{depth}(S) = \text{depth}(\pi S) &= 1\\
\text{depth}(A) = \text{depth}(\pi A) &= 1\\
\text{depth}(\lambda A.e) = \text{depth}(\lambda \pi A.e) &= \text{depth}(e) + 1\\
\text{depth}(f\ e_1 \dots\ e_k) &= \max\{\text{depth}(e_i) : i \in \{1, \dots, n\}\} + 1
\end{aligned}$$

Definition 4.4. The *tops* symbol of any expression in NL$_{AS}$ is defined as:

$$\begin{aligned}
tops(S) = tops(\pi S) &= S\\
tops(A) = tops(\pi A) &= A\\
tops(\lambda A.e) = tops(\lambda \pi A.e) &= \lambda\\
tops(f\ e_1 \dots\ e_k) &= f
\end{aligned}$$

Thus, the *tops* symbol refers to the type of the expression, after permutations would have been reduced.

One may have noticed by now, that the new language is not strictly speaking an extension of NL$_a$ since it does not allow concrete atoms. In the next section, we will see a formal explanation of why this does not change our view of NL$_{AS}$, but the basic idea is this: If you enforce all atom variables to be unequal, i.e. $\forall A \in AtVar, B \in AtVar \setminus \{A\} : A\#B$, we can embed any problem in NL$_{aS}$ into NL$_{AS}$ and vice versa.

The next concept to be lifted into NL$_{AS}$ is the concept of permutations. First, note that we do not deal with concrete permutations any longer, since any swapping (A, B) on atom variables may refer to the identity function if $A \doteq B$ is required by the permutation problem. Thus, any list of swappings represents a set of permutations. However, we do not distinguish terminologically between real permutations and permutation variables in the rest of this thesis, since the latter can be seen as a special case of the former.

Example 4.5. Consider the permutation $(A, B)(B, C)$. It could refer to the identity function if $C = A$, a circle (a, b, c) if all three variables are instantiated differently, or swappings (a, b), (b, c) if $B = C, A \neq B$ or $A = B, B \neq C$.

For now, permutations can be simply seen as lists.

Definition 4.6. A permutation in NL_{AS} is an ordered list of swappings (A, B) with $(A, B) = (B, A)$. Its inverse π^{-1} is defined as the reverse list.

The permutation application on NL_{AS} is defined by adding the permutation to the expression at the appropriate positions. For permutation on permutation application, we define $\pi_1 \cdot (\pi_2 \cdot e) := (\pi_1\pi_2) \cdot e$ where $\pi_1\pi_2$ refers to the concatination. Later, we will introduce rules, to simplify such permutations.

Definition 4.7. The application of a permutation π on NL_{AS} is defined as:

$$\begin{array}{rclrcl}
\pi \cdot S & := & \pi S & \pi \cdot (\pi' S) & := & (\pi\pi')S \\
\pi \cdot A & := & \pi A & \pi \cdot (\pi' A) & := & (\pi\pi')A \\
\pi \cdot (\lambda e_b.e) & := & \lambda\pi \cdot e_b.\pi \cdot e & \pi' \cdot (f\ e_1 \dots e_{ar(f)}) & := & f\ (\pi \cdot e_1) \dots (\pi \cdot e_{ar(f)})
\end{array}$$

with e_b as an expression either of type A or $\pi' A$.

Note that expressions e_b will be treated rather differently from other expressions, since such expressions can only evaluate to atoms. We take the time to define:

Definition 4.8. Expressions of the type A or πA are called basic expressions and written BEX. Note that $\mathrm{BEX} \subset \mathrm{NL}_{AS}$.

We define substitutions and ground substitutions next.

Definition 4.9. A substitution σ on NL_{AS} is a finite mapping from $AtVar$ to BEX and from $ExVar$ to NL_{AS}.

The application is defined as:

$$e\sigma = \sigma(e) := e'$$

where e' is constructed by replacing all S with $\sigma(S)$ and all A with $\sigma(A)$ in e.

Remark 4.10. A substitution as defined in 4.9 does not necessarily map expressions from NL_{AS} to NL_{AS}, since atom variables in swappings can be replaced with suspensions. If we allowed nested permutations, e.g. $(A, (B, (E, F)G)D)$, we could get around this problem. Analysising such a language is not straightforward however.

Definition 4.11. A ground substitution γ on NL_{AS} is a finite mapping from $AtVar$ to $\mathcal{A}t$ and from $ExVar$ to NL_a.

The application is defined as:

$$e\sigma = \sigma(e) := e'$$

where e' is constructed by replacing all S with $\sigma(S)$ and all A with $\sigma(A)$ in e.

In order to properly define a unification problem, only constraints and equations remain to be defined.

4.2 Constraints

As in NL_{aS} the idea behind constraints is to constrain possible ground substitutions. In this case however, we will do a more detailed analysis of how this is achieved.

Definition 4.12. A constraint in NL_{AS} is a construct of the type $A\#e$ for any expression $e \in \mathrm{NL}_{AS}$.

A constraint of the form $A\#B$ is called standardized constraint or standard constraint. If ∇ is a constraint set with $A\#B \in \nabla$ or $B\#A \in \nabla$ we write $\nabla \vdash A\#B$.

Definition 4.13. A ground substitution γ satisfies ∇, written as $\gamma \vDash \nabla$, iff $\gamma(\nabla) = \{A\gamma\#e\gamma : A\#e \in \nabla\}$ is a set of valid constraints.

This brings us to our first relation on constraint sets, which we will take as set of constraints being equivalent.

Definition 4.14. Given two constraint sets ∇_1, ∇_2, the constraint equivalence $\xleftrightarrow{\#}$ is defined as:

$$\nabla_1 \xleftrightarrow{\#} \nabla_2 := \text{ for all ground substitutions } \gamma\text{: } \gamma \vDash \nabla_1 \iff \gamma \vDash \nabla_2$$

For example $\{A\#f \;\; A \;\; B\} \xleftrightarrow{\#} \{A\#B, A\#C\}$, with the right-hand constraints being of smaller depth. We capture a few more facts about this relation.

Lemma 4.15. *The relation $\xleftrightarrow{\#}$ is an equivalence relation.*

Proof. Symmetry and reflexivity follow directly from the same properties of $\iff$. Transitivity follows, from

$$\begin{aligned} &(\forall\gamma : \gamma \vDash \nabla_1 \iff \gamma \vDash \nabla_2) \wedge (\forall\gamma : \gamma \vDash \nabla_2 \iff \gamma \vDash \nabla_3) \\ \equiv &\forall\gamma : (\gamma \vDash \nabla_1 \iff \gamma \vDash \nabla_2) \wedge (\gamma \vDash \nabla_2 \iff \gamma \vDash \nabla_3) \end{aligned}$$

and then the transitivity of $\iff$. □

Since semantics of expressions are supposed to be defined with respect to ground substitutions, we use the following definition to give an alpha like equivalence relation on expressions, similar to $\xleftrightarrow{\#}$, which will also be used to reduce expressions in a unification context.

Definition 4.16. For a constraint set ∇ the relation $\overset{\nabla}{\sim}$ (nabla equivalence) is defined as:

$$e_1 \overset{\nabla}{\sim} e_2 := \text{ for all ground substitutions } \gamma\text{: } \gamma \vDash \nabla \implies e_1\gamma \sim e_2\gamma$$

For $\nabla = \emptyset$ the relation is written as just $\sim$.

Note that in the above definition $e_1\gamma$, $e_2\gamma$ refer to ground expressions and thus, $e_1\gamma \sim e_2\gamma$ to α-equivalence.

The next lemma gives the justification for reducing sub expressions and formalizes the idea, that any reduction, possible in a constraint context, is also possible in a larger context.

Lemma 4.17. *The relation $\overset{\nabla}{\sim}$ is an equivalence relation and a congruence, i.e. for all contexts $C[\ \ ]$ and expression $e_1 \overset{\nabla}{\sim} e_2$ the relation $C[e_1] \overset{\nabla}{\sim} C[e_2]$ holds. Furthermore for $\nabla \subset \nabla'$ the implication $e_1 \overset{\nabla}{\sim} e_2 \implies e_1 \overset{\nabla'}{\sim} e_2$ holds.*

Proof. The relation is an equivalence relation because α-equivalence is an equivalence relation. Contexts in NL_{AS} are of the form:

$$[\ \ ],\ \pi[\ \ ],\ f\ e_1 \ldots [\ \ ] \ldots e_{ar(f)},\ \lambda[\ \ ].e,\ \lambda e_b.[\ \ ]$$

. The statement can be proven by induction over the depths of e_1, e_2 and the type of context. For the last statement, take in mind that $\rho \vDash \nabla' \implies \rho \vDash \nabla$. Hence:

$$(\forall \rho : \rho \vDash \nabla \implies e_1\gamma \sim e_2\gamma) \implies (\forall \rho : \rho \vDash \nabla' \implies e_1\gamma \sim e_2\gamma)$$

□

We want to give some examples for $\overset{\nabla}{\sim}$, which induce size decreasing rewrite rules.

Proposition 4.18. *On NL_{AS} the following $\overset{\nabla}{\sim}$ hold:*

- $(A, B)A \ \sim\ B$
- $(A, B)C \ \overset{\nabla}{\sim}\ C$ if $\nabla \vdash B\#C, A\#C$
- $\pi(A, A)\pi' \cdot e \ \sim\ \pi\pi' e$
- $\pi(A, B)(A, B)\pi' \cdot e \ \sim\ \pi\pi' e$

More rules are possible. For example, if $\nabla A\#C, B\#C, A\#D, B\#D$ the permutation $(A, B)(C, D)$ can be rewritten to $(C, D)(B, C)$. However, this rule does not reduce the size of an expression, nor is it terminating.

Such rules can also reduce constraints $A\#e$, by reducing the expression e. This, combined with other rules, brings us to our next proposition.

Proposition 4.19. *The following constraint equivalences holds:*

- $\{A\#e\} \cup \nabla \xleftrightarrow{\#} \{A\#e'\} \cup \nabla$ *if* $e \overset{\nabla}{\sim} e'$.
- $\{A\#f\ e_1 \ldots e_{ar(f)}\} \cup \nabla \xleftrightarrow{\#} \{A\#e_i : i \in \{1, \ldots, ar(f)\}\} \cup \nabla$
- $\{A\#\lambda A.e\} \cup \nabla \xleftrightarrow{\#} \nabla$

- $\{A\#e\} \cup \nabla \overset{\#}{\longleftrightarrow} \nabla$ *if* e *does not contain any atom or expression variables.*

- $\{A\#\lambda B.e\} \cup \nabla \overset{\#}{\longleftrightarrow} \{A\#e\} \cup \nabla$ *if* $\nabla \vdash A\#B$.

- $\{A\#(A,B)\pi \cdot e\} \cup \nabla \overset{\#}{\longleftrightarrow} \{B\#\pi \cdot e\} \cup \nabla$

- $\{A\#(B,C)\pi \cdot e\} \cup \nabla \overset{\#}{\longleftrightarrow} \{A\#\pi \cdot e\} \cup \nabla$ *if* $\nabla \vdash A\#B, A\#C$

Again, rules were chosen which strictly decrease the depth or size of constraints (or the set) and which induce terminating rewrite rules.

4.3 Unification Problem

To define a unification problem, we still need equations as in NL_{aS}. We define:

Definition 4.20. An equation in NL_{AS} is a construct of the type $e_1 \doteq e_2$ for any expressions $e_1, e_2 \in \text{NL}_{AS}$.

The unification problem consists again of equations Γ and constraints ∇, where we want $e_1 \overset{\nabla}{\sim} e_2$ for all equations in Γ

Definition 4.21. A unification problem in NL_{AS} is a tuple $P = (\Gamma, \nabla)$ where Γ consists of equations and ∇ of constraints in NL_{AS}.

Solutions and unifiers are extended naturally from the concepts in NL_{aS}.

Definition 4.22. A solution of a unification problem $P = (\Gamma, \nabla)$ is a ground substitution γ with:

- $\forall e_1 \doteq e_2 \in \Gamma : \ e_1\gamma \sim e_2\gamma$

- $\rho \vDash \nabla$

A unification problem is solvable if it has a solution. A constraint set ∇ is solvable if $(\emptyset, \nabla)$ is solvable.

Definition 4.23. A unifier for a unification problem $P = (\Gamma, \nabla)$ in NL_{AS} is a tuple $\mu = (\sigma, \nabla')$ where:

- ∇' is solvable.

- For all ground substitutions γ holds:

$$(\sigma \circ \gamma) \vDash \nabla' \implies \sigma \circ \gamma \text{ is a solution of } P.$$

The next concept to be lifted, is that of a general unifier. Since the existence of such unifiers remained until now an open question, we introduce the concept of a complete set of unifiers first.

Definition 4.24. A complete set for a unification problem $P = (\Gamma, \nabla)$ is a set of unifiers $M = \{\mu_1, \ldots, \mu_n\}$, $\mu_i = (\sigma_i, \nabla_i)$ with:

For all solutions γ of P there is a ground substitution ρ with:

$$\begin{aligned} \gamma(A) &\sim (\sigma \circ \rho)(A) \qquad \forall A \in P \\ \gamma(A) &\sim (\sigma \circ \rho)(S) \qquad \forall S \in P \end{aligned}$$

If $M = \{\mu\}$ is a complete set, μ is a most general unifier for P.

Example 4.25. Consider the unification problem $(\{(A, B)C \doteq D\}, \emptyset)$. A complete set is given by

$$\{\{C \to B, A \to D\}, \emptyset), \{C \to A, B \to D\}, \{B\#C\}), \{C \to D\}, \{B\#C, A\#C\})\}$$

This is in fact the smallest complete set with standardized constraints and only atom to atom substitutions. It illustrates why we allowed atom to suspension substitutions. Consider the unification problem

$$(\{(A_1, B_1)C_1 \doteq D_1, \ldots, (A_n, B_n)C_n \doteq D_n\}, \emptyset)$$

of size $4n$. Since all equations are independent of one another, it stands to reason, that a set of 3^n unifiers of this type are necessary for a complete set. But since we allow more general substitutions, we can compute the most general unifier

$$(\{D_1 \to (A_1, B_1)C_1, \ldots, D_n \to (A_n, B_n)C_n\}, \emptyset)$$

We capture the fact, that any unifier of a unification problem P needs to only substitute the variables in P with the following statements.

Definition 4.26. A substitution σ has the capsulation property for a unification problem P if $\sigma(V) = V$ for all $V \in \mathit{Var}$, which are not in P.

A unifier (σ, ∇') has the capsulation property if σ has it.

Proposition 4.27. *For every set of unifiers $M = (\mu_1, \ldots, \mu_n\}$ there is a set $M' = \{\mu'_1, \ldots, \mu'_n\}$ such that μ'_i agrees with μ_i on P and μ'_i has the capsulation property.*

Unless mentioned otherwise, we assume unifiers to always have the capsulation property.

At last we lift the previous defined concepts of equivalency to unification problems.

Definition 4.28. For two unification problems P_1, P_2 the problem equivalency $\longleftrightarrow$ is defined as:

$$P_1 \longleftrightarrow P_2 := \text{ for all ground substitutions } \gamma\text{: } \gamma \vDash P_1 \iff \gamma \vDash P_2$$

Proposition 4.29. *The following statments for single equations in unification problems hold:*

- $(\{e_1 \doteq e_2\} \cup \Gamma, \nabla) \longleftrightarrow \{(e_2 \doteq e_1\} \cup \Gamma, \nabla)$
- $(\{e_1 \doteq e_2\} \cup \Gamma, \nabla) \longleftrightarrow \{(\pi \cdot e_1 \doteq \pi \cdot e_2\} \cup \Gamma, \nabla)$
- $(\{e_1 \doteq e_2\} \cup \Gamma, \nabla) \longleftrightarrow \{(e'_1 \doteq e'_2\} \cup \Gamma, \nabla)$ *if* $e'_i \overset{\nabla}{\sim} e_i$
- $(\{f\ e_1 \dots e_{ar(f)} \doteq f\ e'_1 \dots e'_{ar(f)}\} \cup \Gamma, \nabla) \longleftrightarrow (\{e_i \doteq e'_i : \forall i \in \{1, \dots, n\}\} \cup \Gamma, \nabla)$
- *If* $(\{e_1 \doteq e_2\} \cup \Gamma, \nabla)$ *is solvable and* $tops(e_i) \neq S$ *then* $tops(e_1) = tops(e_2)$

4.4 Complexity of NL$_{AS}$

In this section we analyse the complexity of the unification problem in NL$_{AS}$. We will start by showcasing two special instances of this problem, which are in P. The first one is the case, mentioned in the introduction of the sectiobn 4.2, where ∇ contains all possible standardized constraints on $AtVar(\Gamma)$. We will show that any problem of this case, can be embedded into NL$_{aS}$. The second one is the case of ∇ being empty. Since both examples are borrowed from [11], no full proves will be provided here.

Next, it will be shown, that the problem is NP-complete in general. To that end, an encoding of the SAT problem as a unification problem in NL$_{AS}$ will be given to prove NP-hardness. Afterwards, a brute-force non-deterministic guessing algorithm will be provided to prove NP-completeness.

Lemma 4.30. *Let* $P = (\Gamma, \nabla)$ *be a unification problem with* $\{A\#B : A, B \in AtVar(P), A \neq B\} \subset \nabla$. *It is solvable in polynomial time and a most general unifier can be computed in polynomial time.*

Proof. We provide a short sketch. Since all atom variables are supposed to be instantiated differently, every solution γ of P is of the form $\gamma(A_i) = a_i$ for all atom variables $A_i \in P$. Using this and that solvability does not depend on any particular choice of names, we obtain an equivalent problem in NL$_{aS}$. The statement follows directly. □

Lemma 4.31. *Let* $P = (\Gamma, \emptyset)$ *be a unification problem for an arbitrary* Γ. *It can then be solved in polynomial time.*

Proof. Again, a short sketch. Suppose Γ is solvable. Since the problem does not contain any constraints, there is a solution γ, which instantiates all atom variables with the same atom. Afterwards the whole problem collapses to a first-order unification problem. □

Now, we get to the SAT-encoding.

Theorem 4.32. *For every clause set* C *there is a polynomial encoding to a unification problem* P*, such that* P *is solvable iff* C *is satisfiable.*

Proof. We give such an encoding for a single clause, since the statement follows directly from induction. Suppose $C = \{L_1, \ldots, L_n\}$ where L_i is a literal and suppose the Boolean variables range over $V_1, \ldots, V_n$. To encode the literals, add for every literal either the equation $L_i \doteq V_j$ if $L_i = V_j$ or the constraint $L_i \# V_j$ if $L_i = \neg V_j$.

Two more atom variables are added: $True$ and $False$. First we add $True \# False$. Next we need to make sure, that every variable either evaluates to true or false, i.e. $\rho(True) = \rho(V_i)$ or $\rho(False) = \rho(V_i)$ for every solution ρ. To ensure this we add the constraints $V_i \# \lambda True.\lambda False.V_i$. Since $V_i \# V_i$ would imply insolvability, this is achieved. We do the same for every literal next, i.e. add constraints $L_i \# \lambda True.\lambda False.L_i$.

At last to encode the clause being true, we add $True \# \lambda L_1.\lambda L_2. \ldots .\lambda L_n.True$. Since this is a polynomial encoding, and the method can be extended to multiple clauses without conflict, the statement follows. □

Corollary 4.33. *The unification problem in* NL_{AS} *is* NP-*hard.*

At last, we can show NP-completeness.

Corollary 4.34. *The unification problem in* NL_{AS} *is* NP-*complete.*

Proof. We need to show that the unification problem is in NP. We shall be content with providing the algorithm and refer for the full proof to Schauß et.al.[11]. Let $P = (\Gamma, \nabla)$ be an arbitrary unification problem. Then guess on all pairs A_i, A_j whether they are supposed to be equal or not. If the former is true apply $\sigma(A_i) = A_j$ on P. For the latter add $A_i \# A_j$ to ∇. The resulting problem (Γ', ∇') satisfies the conditions of 4.30 and thus, can be solved in polynomial time. □

4.5 On Most General Unifiers

For some time, the assumption stood, that most general unifiers do not exist for NL_{AS}. One motivating example for this conjecture was the unification problem $(\{(A, B)C \doteq C\}, \nabla)$[12]. A complete set of unifiers derived by guessing the equality/inequality of A, B and C is:

$$\{(\{A \to C, B \to C\}, \emptyset), (\emptyset, A\#C, B\#C)\}$$

There is no obvious way on how to combine both unifiers, which lead to the conjecture that most general unifiers do not exist. However, since we allow not only standardized constraints the alternative approach of rewriting equations as constraints yields the desired result. In this example a unifier would be given by $(\emptyset, C\#\lambda(A, B)C.C)$. This is due to the fact, that any solution γ has to instantiate C and $(A, B)C$ with the same atom to not get the contradictory constraint $C\#C$. We generalize this result.

Lemma 4.35. *Let $(\{\pi A \doteq X\}, \nabla)$ be a solvable unification problem. Then a most general unifier is given by:*

$$(\emptyset, \{X\#\lambda\pi A.X\} \cup \nabla)$$

Proof. Let ρ be a ground substitution. The constraint $\rho(x)\#\rho(\lambda\pi A.X)$ is satisfied iff $\rho(x) = \rho(\pi A)$ or $\rho(X)\#\rho(X)$, the latter of which is never true. Hence ρ is a solution of $(\{\pi A = X\}, \nabla)$ iff ρ satisfies $\{X\#\lambda\pi A.X\} \cup \nabla$. □

We proof the existence of most general unifiers for all solvable unification problems in three steps. First, we show, that if every unification problem with $|\Gamma| = 1$ has a most general unifier, then every unification problem has a most general unifier. This will be a simple consequence of the fact, that complete sets can be computed by going through equations iteratively. Afterwards, we will analyse for which unification problems with $|\Gamma| = 1$ we candirectly write down a unifier, and which can be transformed to an equivalent one of reduced depth. This will yield an algorithm for computing most general unifiers.

For the first part, we need the following lemma.

Lemma 4.36 (Continuation lemma)**.** *Let $P = (\Gamma_1 \cup \Gamma_2, \nabla)$ be a solvable unification problem. Let $M_1 = \{\mu_1, \dots, \mu_n\}$ be a complete set of unifiers for $P_1 = (\Gamma_1, \nabla)$ with $\mu_i = (\sigma_i, \nabla_i)$.*

Let $S_i = \{\mu^i_1, \dots, \mu^i_k\}$ be a complete set of unifiers for the problem $(\sigma_i(\Gamma_2), \nabla_i)$ with $\mu_i = (\sigma^i_j, \nabla^i_j)$ or $\emptyset$ if the problem is not solvable and let $C_i = \{(\sigma_i\sigma^i_j, \nabla^i_j) : (\sigma^i_j, \nabla^i_j) \in S_i\}$ be the continuation of S_i.

Then $\bigcup_{i=1}^n C_i$ is a complete set of unifiers for P.

Proof. First we need to show that $\bigcup_{i=1}^n C_i$ is a set of unifiers for P.

Let γ be a ground substitution with $\gamma \vDash \nabla^i_j$. Then $\sigma^i_j\gamma$ is a solution of $(\sigma_i(\Gamma_2), \nabla_i)$ and for all $e_1 \doteq e_2 \in \Gamma_2$:

$$\begin{aligned} &\sigma^i_j\gamma(e_1\sigma_i) \sim \sigma^i_j\gamma(e_2\sigma_i) \\ \iff &\sigma_i\sigma^i_j\gamma(e_1) \sim \sigma_i\sigma^i_j\gamma(e_2) \end{aligned}$$

Furthermore for all $e_1 \doteq e_2 \in \Gamma_1$ we have by definition $e_1\sigma_i \sim e_2\sigma_i$ and hence:

$$\begin{aligned} &\sigma_i\sigma^i_j\gamma(e_1) = \sigma^i_j\gamma(e_1\sigma_i) \\ \sim\ &\sigma^i_j\gamma(e_2\sigma_i) = \sigma_i\sigma^i_j\gamma(e_2) \end{aligned}$$

At last, since $\sigma^i_j\gamma \vDash \nabla_i$ and $\mu_i = (\sigma_i, \nabla_i)$ is a unifier for $P_1 = (\Gamma_1, \nabla)$ we get:

$$\sigma_i\sigma^i_j\gamma \vdash \nabla$$

So $(\sigma_i\sigma^i_j, \nabla^i_j)$ is a unifier for P.

Now we will show completeness.

Let ρ be a solution of P. Then ρ is also a solution of P_1 and there are $(\sigma_i, \nabla_i) \in M_1$ and a ground substitution γ_1 with:

$$\begin{aligned} \rho(A) &= \sigma_i\gamma_1(A) && \text{for } A \in AtVar(P_1) \\ \rho(S) &\sim \sigma_i\gamma_1(S) && \text{for } S \in ExVar(P_1) \end{aligned}$$

Let $\gamma(V) := \gamma_1(V)$ for $V \in Var(P_1)$ and $\gamma(V) := \rho(V)$ otherwise. Since $\sigma_i(V) = V$ if $V \notin Var(P_1)$ we get:

$$\begin{aligned} \rho(A) &= \sigma_i\gamma(A) && \text{for } A \in AtVarP) \\ \rho(S) &\sim \sigma_i\gamma(S) && \text{for } S \in ExVar(P) \end{aligned}$$

Since ρ is also a solution for (Γ_2, ∇) we get for all $e_1 \doteq e_2 \in \Gamma_2$:

$$\begin{aligned} \sigma_i\gamma(e_1) &\sim \rho(e_1) \\ &\sim \rho(e_2) \sim \sigma_i\gamma(e_2) \end{aligned}$$

Hence γ is a solution for $(\sigma_i(\Gamma_2), \nabla)$. Since S_i is complete there are $(\sigma^i_j, \nabla^i_j) \in S_i$ and a ground substitution γ_2 with:

$$\begin{aligned} \gamma(A) &= \sigma^i_j\gamma_2(A) && \text{for } A \in AtVar(\sigma_1(\Gamma_2)) \\ \gamma(S) &\sim \sigma^i_j\gamma_2(S) && \text{for } S \in ExVar(\sigma_1(\Gamma_2)) \end{aligned}$$

Now, let $\gamma'(V) := \gamma_2(V)$ for $V \in Var(\sigma_1(\Gamma_2))$ and $\gamma'(V) := \gamma(V)$ otherwise. Since $\sigma^i_j(V) = V$ if $V \notin Var(\sigma_1(\Gamma_2))$ we get:

$$\begin{aligned} \gamma(A) &= \sigma^i_j\gamma'(A) && \text{for } A \in AtVar \\ \gamma(S) &\sim \sigma^i_j\gamma'(S) && \text{for } S \in ExVar \end{aligned}$$

Combining our results, we derive:

$$\begin{aligned} \rho(A) &= \sigma_i\sigma^i_j\gamma'(A) && \text{for } A \in AtVar(P) \\ \rho(S) &\sim \sigma_i\sigma^i_j\gamma'(S) && \text{for } S \in ExVar(P) \end{aligned}$$

□

Corollary 4.37. *Suppose every problem of the form $(\{eq_1\}, \nabla)$ has a most general unifier. Then every unification problem has a most general unifier.*

Proof. By induction. The base case $|\Gamma| = 1$ holds, because it is the assumption. Now assume the induction hypothesis to hold for all $|\Gamma| = n - 1$ and let $P = (\{eq_1, \dots, eq_n\} \cup \{eq_{n+1}\}, \nabla)$ for some ∇. Due to the induction hypothesis $(\{eq_1, \dots, eq_n\}, \nabla)$ has a most general unifier $\mu_1 = (\sigma_1, \nabla_1)$ and the problem $(\{\sigma_2(eq_{n+1})\}, \nabla_1)$ has a most general unifier $\mu_2 = (\sigma_2, \nabla_2)$. Thanks to the continuation lemma, we know that $(\sigma_1 \circ \sigma_2, \nabla_2)$ is a most general unifier for P. □

Now to the second part. We have already proven, that equations of the form $X \doteq \pi A$ have a most general unifier. The same is true for $S \doteq e$ or $S \doteq e$

Proposition 4.38. *The following problems have most general unifiers, provided that they are solvable.*

- $(\{S \doteq e\}, \nabla)$ *has the most general unifier* $(\{S \to e\}, \nabla)$.
- $(\{\pi S \doteq e\}, \nabla)$ *has the most general unifier* $(\{S \to \pi^{-1} \cdot e\}, \nabla)$.

The same results also apply for $e \doteq S$, $e \doteq \pi S$ and $\pi A \doteq X$. We capture this with the following result.

Corollary 4.39. *Let* $\Gamma = \{e_1 \doteq e_2\}$, *with* $\mathrm{depth}(e_1) = 1$ *or* $\mathrm{depth}(e_2) = 1$. *Then all solvable unification problems* (Γ, ∇) *have a most general unifier*

Proof. Follows directly from lemma 4.35 and proposition 4.38. □

Now only equations with $tops(e_i) = f$ and $tops(e_i) = \lambda$ remain. For functions proposition 4.29 provides a method to transform the unification problem into another one with equations of lower depth. Abstractions however, need a little more work.

Lemma 4.40. *The following problem equivalency holds:*

$$(\{\lambda A.e_1 \doteq \lambda B.e_2\}, \nabla) \longleftrightarrow (\{e_1 \doteq (A, B) \cdot e_2\}, \nabla \cup (A\#\lambda B.e_2))$$

Proof. Let ρ be a ground substitution. We consider the cases $\rho(A) = \rho(B)$ and $\rho(A) \neq \rho(B)$ and show, that in both cases ρ is a solution of P iff it is a solution of P'.

First $\rho(A) = \rho(B)$. The constraint $A\#\lambda B.e_2$ is already satisfied and thus both constraint sets are satisfied simultaneously. We only need to show, that the equations are solved at the same time. Since $\rho(A) = \rho(B)$ the swapping $(A, B)\rho = id$ and hence

$$\rho(\lambda A.e_1) \sim \rho(\lambda B.e_2) \iff \rho(e_1) \sim \rho(e_2) \iff \rho(e_1) \sim \rho((A, B)e_2)$$

Now let $\rho(A) \neq \rho(B)$. By definition $\rho(\lambda A.e_1) \sim \rho(\lambda B.e_2)$ iff $\rho(e_1) \sim \rho((A, B) \cdot e_2)$ and $\rho(A)\#\rho(e_2)$. Since $\rho(A) \neq \rho(B)$ the latter constraint is satisfied iff $\rho(A)\#\rho(\lambda B.e_2)$ and hence either both constraint sets are satisfied or none is.

Thus, ρ is a solution of P iff it is a solution of P'. □

The next question is how to deal with abstractions of the form $\lambda\pi A.e$. If nested permutations were allowed, we could use lemma 4.40 in these cases as well, but since we restricted ourselves to proper permutations, a flattening of atom suspensions is necessary. We formalize this approach.

Definition 4.41. Let $\Gamma = \{e_1 \doteq e_2\} \cup \Gamma'$ be the equations of the unification Problem $P = (\Gamma, \nabla)$.

If $\pi A \in subEx(e_1)$ then a flattening can be conducted by replacing πA with a fresh flattening variable X.

To that end, let e_1^f be constructed by replacing πA with X in arbitrary subexpressions in e_1. A permutation flattened problem is then defined by:

$$\begin{aligned}\Gamma^f =& \{e_1^f \doteq e_2, X \doteq \pi A\} \cup \Gamma' \\ P^f =& (\Gamma^f, \nabla)\end{aligned}$$

Lemma 4.42. *Suppose M^f is a complete set of (standardized) unifiers for a permutation flattened problem $P^f = (\Gamma^f, \nabla)$, with construction names as in definition 4.41.*

Then M^f is a complete set of (standardized) unifiers for the original problem $P = (\Gamma, \nabla)$.

Proof. Let ρ be an arbitrary solution of P. First, we construct a solution for P^f by;

$$\begin{aligned}\rho^f(X) :=& \rho(\pi A) & \\ \rho^f(B) :=& \rho(B) & \text{for } B \neq X, B \in AtVar \\ \rho^f(S) :=& \rho(S) & \text{for } S \in ExVar\end{aligned}$$

Since $X \notin \nabla, \Gamma'$ the constructed substitution ρ^f operates like ρ on (Γ', ∇) and hence is a solution for it.

Furthermore, for the flattened part of the problem, i.e. $e_1 \doteq e_2, X \doteq \pi A$ we obtain:

$$\rho^f(X) = \rho^f(\pi A)$$
$$\rho^f(e_1^f) = \rho(e_1) \sim \rho(e_2) = \rho^f(e_2)$$

Because of $X \notin AtVar(e_2)$ and the construction method of e_1^f. So ρ^f is a solution of P^f.

Since M^f is complete, there is a ground substitution γ and a unifier $\mu = (\sigma, \nabla') \in M^f$ with:

$$\begin{aligned}\rho^f(B) =& \sigma\gamma(B) & \text{for } B \in AtVar(P^f) = AtVar(P) \cup \{X\} \\ \rho^f(S) \sim& \sigma\gamma(S) & \text{for } S \in ExVar(P)\end{aligned}$$

From the definition of ρ^f we derive

$$\begin{aligned}\rho(B) =& \sigma\gamma(B) & \text{for } B \in AtVar(P) \\ \rho(S) \sim& \sigma\gamma(S) & \text{for } S \in ExVar(P)\end{aligned}$$

So M^f is a complete set for P. □

At last we have everything we need to prove the existence of most general unifiers for all solvable unification problems.

Theorem 4.43. *Suppose $P = (\Gamma, \nabla)$ is solvable. Then there is a most general unifier (σ, ∇') of P.*

Proof. We prove the statement for $\Gamma = \{e_1 \doteq e_2\}$ by providing a recursive algorithm, which produces a most general unifier for solvable unification problems with only one equation.

If $\text{depth}(e_1) = 1$ or $\text{depth}(e_2) = 1$ corollary 4.39 provides the most general unifier. The only cases left to study are $f\ e_1^1 \ldots, e_{ar(f)}^1 \doteq f\ e_1^2 \ldots, e_{ar(f)}^2$ and $\lambda e_b.e \doteq \lambda e_b'.e'$.

Case 1. $f\ e_1^1 \ldots, e_{ar(f)}^1 \doteq f\ e_1^2 \ldots, e_{ar(f)}^2$
Due to proposition 4.29 the unification problem is equivalent to:

$$(\{e_1^1 \doteq e_1^2, \ldots, e_{ar(f)}^1 \doteq e_{ar(f)}^2\}, \nabla)$$

We compute a most general unifier for $(\{e_1^1 \doteq e_1^2\}, \nabla)$, apply the resulting substitution to the remaining equations and repeat, as in lemma 4.36. The result is a most general unifier.

Case 2. $\lambda e_b.e \doteq \lambda e_b'.e'$.

Case i. $e_b = X \wedge e_b' = Y$
We can apply lemma 4.40 to get the equivalent problem $(\{e_1 \doteq (A, B) \cdot e_2\}, \nabla \cup (A \# \lambda B.e_2))$, where the equation is of reduced depth. Apply the algorithm on this problem.

Case ii. $e_b = \pi X \vee e_b' = \pi' Y$
In this case, the expressions e_b, e_b' are flattened as in definition 4.41. The resulting unification problem is:

$$(\{X^f \doteq e_b, Y^f \doteq e_b', \lambda X^f.e \doteq \lambda Y^f.e'\}, \nabla)$$

where X^f, Y^f are new variables. The first two equations can directly be written as constraints according to lemma 4.35. For the remaining problem $(\{\lambda X^f.e \doteq \lambda Y^f.e'\}, \nabla')$ case i can be applied.

The statement follows for arbitrary unification problems from corollary 4.37. $\square$

Chapter 5

Schmidt-Schauß's Algorithm

The algorithm developed by Schmidt-Schauß et.al.[11] computes a complete set of unifiers in non-deterministic time, and checks the validity of the unifiers in NP-time. Despite working in NP-time, the algorithm is promising to have good practical properties for unification problems with a low amount of atom variables.

The algorithm consists of two sub algorithms. The first one, AvNomUnify, works mainly on the equations, while the second one, AvSolNabla, tests the validity of the computed unifiers. Both algorithm use lazy equality guessing of atom variables to their end.

In this chapter, we will deal with AvNomUnify. A description of the algorithm will be given as well as an outline of an implementation. The entire source code with AvSolNabla will be attached to this thesis.

5.1 AvNomUnify

The algorithm expects a unification problem(Γ, ∇) and a threshold $theq$ as the input. The threshold is supposed to be $p(size(\Gamma, \nabla))$ for some sufficiently large polynomial p to prevent possible exponential growth of the problem. The existence of such a polynomial is proven in [11]. Moreover, $4N^2 \cdot (Maxarity + 2)$ is proven to be a possible choice for the polynomial, where N is the size of the input.

AvNomUnify works on the triple (Γ, θ, ∇), where initially Γ and ∇ is taken from the input and θ is empty. The goal of the algorithm is to clear out Γ by deducing equivalent substitutions or constraints. If the rules to achieve this do not suffice, equality/inequality guesses are done on pairs of atom variables.

As a first step, the rules defined in proposition 4.18, proposition 4.19 and proposition 4.29 are used to define a terminating set of sound terminating rewrite rules on expressions, constraints and equations. We define the rewriting systems:

Definition 5.1. For a constraint set ∇, the expression reduction $\xrightarrow{\nabla}$ is defined as:

- $(A, B)A \xrightarrow{\nabla} B$
- $(A, B)C \xrightarrow{\nabla} C$ if $\nabla \vdash B\#C, A\#C$
- $\pi(A, A)\pi' \cdot e \xrightarrow{\nabla} \pi\pi' e$
- $\pi(A, B)(A, B)\pi' \cdot e \xrightarrow{\nabla} \pi\pi' e$
- $e \xrightarrow{\nabla} e'$ if e' arises by reducing subexpressions with $\xrightarrow{\nabla}$.

Definition 5.2. For a constraint set $\{A\#e\} \cup \nabla$, the expression reduction $\xrightarrow{\#}$ is defined as:

- $\{A\#e\} \cup \nabla \xrightarrow{\#} \{A\#e'\} \cup \nabla$ if $e_i \xrightarrow{\nabla} e'_i$
- $\{A\#f\ e_1 \dots e_{ar(f)}\} \cup \nabla \xrightarrow{\#} \{A\#e_i : i \in \{1, \dots, ar(f)\}\} \cup \nabla$
- $\{A\#\lambda A.e\} \cup \nabla \xrightarrow{\#} \nabla$
- $\{A\#e\} \cup \nabla \xrightarrow{\#} \nabla$ if e does not contain any atom or expression variables.
- $\{A\#\lambda B.e\} \cup \nabla \xrightarrow{\#} \{A\#e\} \cup \nabla$ if $\nabla \vdash A\#B$.
- $\{A\#(A, B)\pi \cdot e\} \cup \nabla \xrightarrow{\#} \{B\#\pi \cdot e\} \cup \nabla$
- $\{A\#(B, C)\pi \cdot e\} \cup \nabla \xrightarrow{\#} \{A\#\pi \cdot e\} \cup \nabla$ if $\nabla \vdash A\#B, A\#C$

Definition 5.3. For a unification problem $(\{e_1 \doteq e_2\} \cup \Gamma)$, the equation reduction $\xrightarrow{\Gamma}$ is defined as:

- $(\{e_1 \doteq e_2\} \cup \Gamma, \nabla) \longleftrightarrow (\{e'_1 \doteq e'_2\} \cup \Gamma, \nabla)$ if $e_i \xrightarrow{\Gamma} e'_i$
- $(\{e_1 \doteq e_1\} \cup \Gamma, \nabla) \longleftrightarrow (\Gamma, \nabla)$

Now to the algorithm. The rules by which it is defined can be divided into five categories. Rewriting rules, equation transforming rules, substitution inducing rules, equality guessing and failure rules. We keep the names from [11], though our ordering differs. All of these rules can only be applied if the size of (Γ, ∇) after application is not larger than the threshold.

Rewriting

(Simp1) $\dfrac{(\Gamma, \theta, \nabla) \wedge\ \Gamma \xrightarrow{\Gamma} \Gamma'}{(\Gamma', \theta, \nabla)}$

(Simp2) $\dfrac{(\Gamma, \theta, \nabla) \wedge\ \nabla \xrightarrow{\#} \nabla'}{(\Gamma, \theta, \nabla')}$

Substitution Inducing

(SD3) $\dfrac{(\{S \doteq \pi S'\} \cup \Gamma, \theta, \nabla) \wedge\ \nabla \xrightarrow{\#} \nabla'}{(\Gamma[\pi S'/S], \theta \circ \{S \to \pi S'\}, \nabla)}$, π can be $\emptyset$.

(SD4a) $\dfrac{(\{A \doteq \pi B\} \cup \Gamma, \theta, \nabla) \wedge\ A \text{ does not appear in a permutation of } (\Gamma,\nabla)}{(\Gamma[\pi B/A], \theta \circ \{A \to \pi B\}, \nabla)}$

(SD4b) $\dfrac{(\{A \doteq \pi B\} \cup \Gamma, \theta, \nabla) \wedge\ B \text{ does not appear in a permutation of } (\Gamma,\nabla)}{(\Gamma[\pi^{-1} A/B], \theta \circ \{B \to \pi^{-1} A\}, \nabla)}$

(SD4c) $\dfrac{(\{A \doteq B\} \cup \Gamma, \theta, \nabla)}{(\Gamma[B/A], \theta \circ \{A \to B\}, \nabla)}$

(SD4d) $\dfrac{(\{A \doteq (A,B)e_b\} \cup \Gamma, \theta, \nabla)}{(\{B \doteq e_b\} \cup \Gamma, \theta, \nabla)}$

(SD4e) $\dfrac{(\{A \doteq (B,C)e_b\} \cup \Gamma, \theta, \nabla) \wedge\ \nabla \vdash A\#B, A\#C}{(\{A \doteq e_b\} \cup \Gamma, \theta, \nabla)}$

(SD8) $\dfrac{(\{S \doteq \pi S\} \cup \Gamma, \theta, \nabla)}{(\Gamma, \theta, \nabla \cup \{A\#\lambda\pi A.S : A \in AtVar(\pi)\})}$

(MMS) If all e_i are compound terms and S does not occur in Γ nor in any e_i(see CycleDetection):

$$\frac{(\{S \doteq e_1, \ldots, S \doteq e_n\} \cup \Gamma, \theta, \nabla)}{(\{e_1 \doteq e_2, \ldots, e_1 \doteq e_n\} \cup \Gamma, \theta \circ \{S \to e_1\}, \nabla)}$$

Equation Transforming

(SD5) $\dfrac{(\{f\ e_1 \ldots e_{ar(f)} \doteq f\ e'_1 \ldots e'_{ar(f)}\} \cup \Gamma, \theta, \nabla)}{(\{e_i \doteq e'_i : \forall i \in \{1, \ldots, n\}\} \cup \Gamma, \theta, \nabla)}$

(SD6) $\dfrac{(\{\lambda\pi A.e \doteq \lambda\pi A.e'\} \cup \Gamma, \theta, \nabla)}{(\{e \doteq e'\} \cup \Gamma, \theta, \nabla)}$, where π can be $\emptyset$

(SD7) $\dfrac{(\{\lambda A.e \doteq \lambda B.e'\} \cup \Gamma, \theta, \nabla) \wedge\ \nabla \vdash A\#B}{(\{e \doteq (A,B) \cdot e'\} \cup \Gamma, \theta, \nabla \cup \{A\#e'\})}$

Failure

(ClashFailure) If e_1, e_2 are neither variables nor suspensions:

$$\frac{(\{e_1 \doteq e_2\} \cup \Gamma, \theta, \nabla) \wedge \ \mathit{tops}(e_1) \neq \mathit{tops}(e_2)}{Fail}$$

(VarFail) $\dfrac{(\{A \doteq e_2\} \cup \Gamma, \theta, \nabla) \wedge \ e_2 \text{ is not a suspension}}{Fail}$

(CycleDetection) If all e_i are compound terms:

$$\frac{(\{S \doteq e_1, \ldots, S \doteq e_n\} \cup \Gamma, \theta, \nabla) \longrightarrow \wedge\ \mathit{tops}(e_1) \neq \mathit{tops}(e_2)}{Fail}$$

.

(FreshFail) $\dfrac{(\Gamma, \theta, \nabla) \wedge \ \nabla \vDash A\#A}{Fail}$

Equality Guessing

(GuessEQ) $\dfrac{(\Gamma, \theta, \nabla) \wedge \ \nabla \nvDash A\#B}{(\Gamma[B/A], \theta \circ \{A \to B\}, \nabla[B/A]) \quad | \quad (\Gamma, \theta, \nabla \cup \{B\#A\})}$

The last rule is only applied, if no other rule can be, since exploring alternatives can be rather expensive. Once $\Gamma = \emptyset$ is reached, the unifier (θ, ∇) is returned. It will then be checked for solvability by AvSolNabla. While Sd8 is arguably not substitution inducing, we nevertheless categorize it as such, since it reasons on an expression variable.

5.2 Implementation

In this section, we describe the implementation of AvNomUnify bottom up, i.e. from the expression implementation, over constraints and constraint sets, equation sets, to a unifier and the algorithm itself. For concrete implementations of each of these, check the source code.

Notes for the Implementation

We do some analysis on the rules, to what type of equations they apply to and in which situations these equation types cannot be reduced by any rule. and in which cases they can be applied

We start with the identification of the types of equation to which the rules can be applied to. As a convention, we move variables to the left side of an equation if possible, with a priority on expression variables. This can be achieved by the prioritized ruleset:

$$\pi S \doteq e \rightarrow S \doteq \pi^{-1} e \qquad e \doteq \pi S \rightarrow S \doteq \pi^{-1} e$$
$$\pi A \doteq e \rightarrow A \doteq \pi^{-1} e \qquad e \doteq \pi A \rightarrow \pi^{-1} A \doteq e$$

, where $\pi = \emptyset$ is possible. The below markings are again prioritized, meaning that a lower marking is only be applied if none of the higher ranked marking conditions apply.

Any equation where the left side is an expression variable is substitution inducing. If the right side is a suspension, the rule can be applied without any check except for the size check. Otherwise it is a compound equation and need special care as defined in MMS. These equations are marked as **expression equations** with subtypes **suspension equations** and **compound equations**.

Equations, where the left side is an atom variable are substitution inducing as well. Such equations are marked as **atom equations**.

On any not clashing set only two types of equations apply, which will be mentioned in one run. Equations between functions are called **function equations** and equations between abstractions are called **lambda equations**.

Every substitution inducing and equation transforming can only be applied to exactly one of these types. The failure rules operate on each type of equation as follows:

- ClashFailure can occur on lambda equations and function equations
- VarFail can occur only on atom equations.
- CycleDetection can only occur on compound equations.

The last failure rule does not occur on equations.

Description

First, the nominal language needs to be defined. Variables are defined as simple containers i.e.

```
data AtomVariable a = AtVar a
data ExpressionVariable a = ExVar a
```

The expressions are implemented as two different kinds. First BasicNlasExpression as expressions of the form πAor A, second ExtendedNlasExpressions, as all valid expressions with possible permutation applications. This double implementation is useful to not allow complicated construct in abstractions. Constraints are defined as a tuple of an AtomVariable and an ExtendedNlasExpression, i.e. Data Constraint a = Con (AtomVariable a) (ExtendedNlasExpression a) The set of constraints, is defined as a tuple of standardized constraints, $A\#B$, $A\#S$, and non-standard constraints. The following functions are defined on it:

```
satsUnEqual :: ConstraintSet a -> AtomVariable a -> AtomVariable a -> Bool
size :: ConstraintSet a -> Int
isStandardized :: ConstraintSet a -> Bool
nablaIsConsistent :: ConstraintSet a -> Bool
substituteExVar :: (...) -> Either Failure (ConstraintSet a)
substituteAtom :: (...) -> Either Failure (ConstraintSet a)
applySimplification :: (...) Either Failure (ConstraintSet a)
```

where (...) has been used, to fit the signatures to the printable area. Note that every substitution and simplification can lead to a Failure, where we distinguish between:

```
data Failure = FreshFail | SubFail | SolveFail | TooLarge | VarFail | Clash
               | CycleInCompoundEquations
```

Next, the equation sets, unification problems and unifiers need to be implemented. Since the three concepts intersect in the implementation, we will explain them in one run.

The equation set is defined as a quadruple of different equations. The unifier is a tuple of substitution and a constraint set, and the unification problem a tuple of an equation set and a unifier (not just a constraint set). This is done, since the algorithm effectively work on $(\Gamma, (\theta, \nabla))$ not just on (Γ, ∇).

Unlike the constraint set, no function to apply simplification rules is directly defined on the unification problem, but rather those rules are directly defined on the appropriate equation types. Again, every rule application is checked for validity, where TooLarge and SubFail, indicate non-critical failures while the other rules imply insolvability of the unification problem.

At last we can go on to give a description of AvNomUnify. It expects a unification problem as its input and applies implication rules until a fixed state has been reached, where failures of the type SubFail and TooLarge are always reverted and thus, can never be reached. Once such a state has been reached a check is done. If Γ is empty or ∇ inconsistent the algorithm terminates. Otherwise a guess is performed on (Γ, ∇) and the resulting problems are attempted to be sovled.

Chapter 6

Conclusion and Outlook

In this thesis, we analyzed the ground nominal language NL_a and gave an alternative syntactical definition for alpha equivalence with the help of name swappings and freshness constraints.

We gave a quick overview of classical nominal rewrite systems with concrete atoms and expression variables, NL_{aS}, which were analyzed by [7, 9, 12] and for which an efficient unification algorithm was provided in [12].

Afterwards we lifted the previously defined concept of constraints and unification problems into a new setting without concrete atoms NL_{AS} but with atom variables instead. We showed what kind of equivalences hold in this setting, if nothing is known, or if constraints are given. In fact, further meaning was given to the word constraint, since we proved, that a set of constraints does strictly constrain the set of possible ground substitutions a unification problem can be solved by.

Some analysis of the complexity of the unification problem in NL_{AS}. We demonstrated its NP-completeness and showed, that even without concrete atoms, a similar environment to NL_{aS} can be induced.

Next the existence of most general unifiers, a problem which remained open until now, was proven algorithmically, mainly by flattening abstractions and expression atom to atom suspension equations as constraints.

At last an implementation of Schmidt-Schauß et.al. 's algorithm was outlined, which performs lazy equality guessing on atom variables. Compared to the algorithm induced by the proof of the most general unifier, it has the benefit of not introducing new atom variables to the problem and of guessing atom variables at the stage, where unifiers are computed, rather than delaying such a guessing to the solvability check.

The following interesting problems remain open, however. First, what complexity result does the restriction of the number of atom variables yield? Second, which practices in rewrite systems yields good or bad algorithmic properties? And connected with this, which classes of constraint sets are efficiently checkable for satisfiability? At last, an implementation of an algorithm, which produces most general unifiers remains to be done as well.

Bibliography

[1] Berghofer, S., and Urban, C. A head-to-head comparison of de bruijn indices and names. *Electronic Notes in Theoretical Computer Science 174*, 5 (2007), 53 – 67. Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[2] Calvès, C., and Fernández, M. *The First-Order Nominal Link.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 234–248.

[3] Cheney, J. *The Complexity of Equivariant Unification.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 332–344.

[4] Cheney, J. Equivariant unification. *Journal of Automated Reasoning 45*, 3 (Oct 2010), 267–300.

[5] Cheney, J., and Urban, C. System description: Alpha-prolog, a fresh approach to logic programming modulo alpha-equivalence, 2003.

[6] Cheney, J., and Urban, C. Nominal logic programming. *ACM Trans. Program. Lang. Syst. 30*, 5 (Sept. 2008), 26:1–26:47.

[7] Fernández, M., and Gabbay, M. J. Nominal rewriting. *Information and Computation 205*, 6 (2007), 917 – 965.

[8] Levy, J., and Villaret, M. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Logic 13*, 2 (Apr. 2012), 10:1–10:31.

[9] Pitts, A. M. Nominal logic, a first order theory of names and binding. *Information and Computation 186*, 2 (2003), 165 – 193. Theoretical Aspects of Computer Software (TACS 2001).

[10] Schmidt-Schauss, M., and Sabel, D. Unification of program expressions with recursive bindings. In *PPDP '16: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, September 2016), J. Cheney and G. Vidal, Eds., ACM, pp. 160–173.

[11] Schmidt-Schauss, M., Sabel, D., and Kutz, Y. Nominal unification with atom-variables. *J. Symbolic Comput.* (2017). accepted for publication, to appear.

[12] Urban, C., Pitts, A. M., and Gabbay, M. J. Nominal unification. *Theoretical Computer Science 323*, 1 (2004), 473 – 497.