John Wickerson

Nominal Prolog

Part II Computer Science Tripos

Churchill College

2 May 2008

Proforma

Name:	**John Wickerson**
College:	**Churchill College**
Project Title:	**Nominal Prolog**
Examination:	**Part II Computer Science Tripos**
Year:	**2008**
Word Count[1]:	**11952**
Project Originator:	Prof. A.M. Pitts
Supervisor:	Prof. A.M. Pitts

Original Aims of the Project

The Prolog language is to be extended to use nominal logic[11] rather than ordinary first-order logic. This will provide support for programming with names and name-bindings, such as those used to represent variables in the λ-calculus. The result, Nominal Prolog, will be similar to Cheney's αProlog[4], but will use a more efficient implementation of the Urban-Pitts-Gabbay algorithm for nominal unification[12], based on that of Calvès and Fernández[2]. [71 words]

Work Completed

A concrete syntax was devised and an interactive top-level shell was built to accept user input. A facility was produced to trace the execution of queries, the output being sequences of images charting the evolution of the goal. Nominal Prolog was shown to give the same results as αProlog on a range of test programs, albeit with a few minor discrepancies. In selected cases where αProlog took exponential time to return a result, Nominal Prolog took only polynomial time—a testimony to the efficient implementation of nominal unification that was developed and employed. [94 words]

Special Difficulties

None.

[1]The word count covers all material in chapters 1 through 5 save headers and footers.

Declaration

I, John Peter Wickerson of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is a name?

My name is John, but it need not be. I could change my name to Tom, and life would be largely the same, provided that I instruct all of my acquaintances henceforth to refer to me as Tom rather than John. Before changing my name, I ought to check that there isn't a Tom among my immediate family and closest friends, for were there already such a Tom then much confusion would ensue upon my becoming Tom too.

Names are given not just to humans but to all kinds of object. For instance, an instruction manual for some machine (illustrated in Figure 1.1) might begin with a diagram that labels one of the machine's buttons as 'A' and the other as 'B'. Consequential occurrences of the phrases 'A' and 'B' in the text are taken to refer to the first and second buttons, respectively. This labelling is arbitrary: an alternative instruction manual that described the first button as 'B' and the second as 'A' (and swapped 'A' and 'B' in the text accordingly) would be equally valid, as would one that named the buttons '1' and '2'. Yet we are not completely unrestricted on the choice of names: for instance, a manual that referred to both buttons as 'A' would be hopelessly ambiguous (Figure 1.2).

There is one further subtlety: that of *scoping*. Consider a manual that is divided into two sections, as shown in Figure 1.3. This manual uses the name 'C' to refer to two different objects. While this has previously led to ambiguity, there is no ambiguity in this case: occurrences of 'C' in the first section clearly refer to the button while those in the second section clearly refer to the handle. It is implicit that the buttons are *in scope* only in the first section and that the handle is in scope only in the second section. Since the sections are disjoint, the two different objects to which the name 'C' refers are never simultaneously in scope. In Figure 1.2, both buttons *are* in scope simultaneously so the use of the same name to

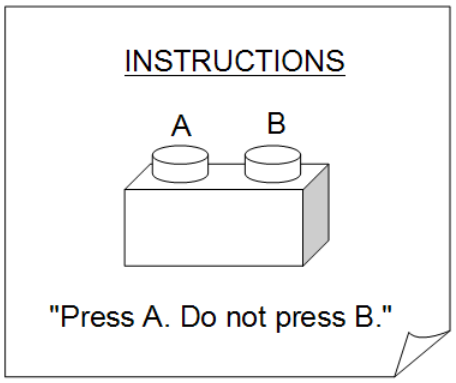


Figure 1.1: A simple instruction manual.

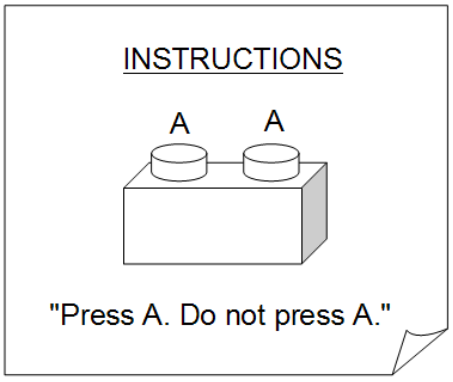


Figure 1.2: A silly instruction manual.

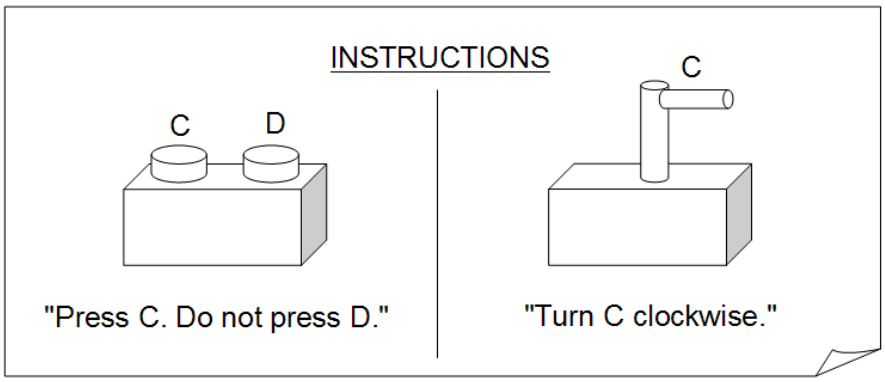


Figure 1.3: An instruction manual divided into two sections.

refer to both *does* lead to ambiguity.

These examples lead us to the two crucial properties that define the concept of a *name*:

The Uniqueness Condition. We must not use the same name to refer to different objects that are in scope simultaneously.

The Choice Axiom. Provided we do not violate Uniqueness then any name will suffice.

There are a few items of terminology that need to be introduced. A *name-binder* is where we fix which name is to be used. In Figure 1.1, the occurrences of 'A' and 'B' in the diagram are name-binders. Occurrences of 'A' and 'B' in the accompanying text are *bound* to those name-binders. Names that occur without a corresponding name-binder are said to be *free* or *unbound*.

Within the realm of computer science, one place with names galore is the λ-calculus. The names we give to bound variables such as the x and y in the term below are subject to the same Uniqueness and Choice constraints as in the definition above.

$$(\lambda x.x)(\lambda x.\lambda y.x)$$

We may, for instance, swap all occurrences of x and y. We may use the variables p and q instead of x and y. What we must not do is replace y with x, because this would change the meaning of the term. Clearly, whenever we write a program that deals with λ-calculus terms, we must ensure we implement the semantics of names correctly. This can be tedious; we would rather use a programming language that has an innate understanding of names already. Sadly, few programming languages do. Nominal Prolog is one of the few.

1.2 Background: Logic programming

Nominal Prolog is an extension of the logic programming language, Prolog. The reader is assumed to be familiar with both the syntax of Prolog and the way it uses unification to find solutions to queries, as presented in [9].

Why is it worthwhile to add support for programming with names to Prolog? Prolog is far from being a mainstream programming language, yet in selected situations it is the language of choice: Prolog and its derivatives have been used in such varied domains as the analysis of terrorist networks[1], the diagnosis of dengue fever[1], and the generation of end-user licence agreements for Microsoft

[1] `http://www.iconnect-online.org/Stories/Story.import5095`

products[2]. Prolog is particularly suitable for implementing relations that are defined by a set of inference rules. The typing relation for λ-calculus terms, for instance, is usually defined in this way. A Prolog implementation of this relation would see each inference rule in the definition correspond to a clause in the program, with the consequent and set of premises of each rule corresponding to the head and tail of a clause respectively. Such a tight correspondence between definition and implementation makes our program easy to write, read and debug. The only slack that remains is a result of Prolog's awkward handling of bindable variables in λ-calculus terms. The primary aim of Nominal Prolog is to reduce this slack.

[2] `http://www.business-integrity.com/press/news_2006_Microsoftrelease.htm`

Chapter 2

Preparation

In this chapter we describe the underlying theory of Nominal Prolog that needed to be understood before development could commence. The reader is referred to the original project proposal (included at the end of this document) for details of the back-up policy, choice of programming language, development model, possible extensions, evaluation strategy, success criteria and project timetable.

2.1 Why Prolog is bad with names

How might we represent names in ordinary Prolog? The terms we have available to do this are as follows.

Definition 2.1.1. *(Prolog terms.)* These are of the form given by the following grammar. Note that when we have a term of the form $\mathtt{f}\langle\rangle$ it is customary to omit the empty tuple and call the term a *constant*.

$$\begin{array}{rcll} t & ::= & \langle t_1, \ldots, t_n \rangle & \text{(tuple)} \\ & | & \mathtt{f}\ t & \text{(data)} \\ & | & X & \text{(unification variable)} \end{array}$$

We might try to represent names using unification variables, but this would be silly: we cannot restrict the scope of unification variables, nor can we stop Prolog unifying unification variables that are supposed to be representing distinct names. More sensible is the use of constants (e.g. strings) to represent names, but this approach is still far from perfect: it regards $\lambda x.x$ and $\lambda y.y$ as different terms, yet we would rather it regarded them as the same. (This was one strength of the otherwise silly approach suggested above.) Moreover, consider the implementation of a predicate that applies substitutions on λ-calculus terms. To evaluate $(\lambda y.xy)[y/x]$ we first substitute a fresh name for the bound variable y,

z say, giving $(\lambda z.xz)[y/x]$, which evaluates to $\lambda z.yz$. In order to make the choice of z sufficiently deterministic, we must keep a store of unused variable names. The additional complexity that results from maintaining this store and carrying it around as an auxiliary argument, together with the inefficiency of performing two substitutions in rapid succession, makes our implementation in Prolog far from satisfactory.

2.2 Why Nominal Prolog is good with names

Nominal Prolog restores this satisfactoriness. As a quick demonstration, here is an implementation of capture-avoiding substitution in Nominal Prolog that is concise, correct, efficient and intuitive.

```
subst(var(X), E, X, E).
subst(var(Y), E, X, var(Y)) :- X =\= Y.
subst(app(E1,E2), E, X, app(E3,E4)) :-
   subst(E1, E, X, E3), subst(E2, E, X, E4).
subst(lam(y\E1), E, X, lam(y\E2)) :-
   y # E, subst(E1, E, X, E2).
```

The problem with Prolog is that it knows only of constants and unification variables, yet neither are suitable for representing names. Nominal Prolog introduces a third category—*atoms*—which are suitable. An important property of atoms is that unlike unification variables they are mutually distinct; that is, we cannot unify two atoms that have different identifiers. Nominal Prolog also introduces an *atom abstraction* construction, rendered $a\backslash t$, which acts as a binder for all free occurrences of the atom a in the term t. The use of this construction tells Nominal Prolog that the choice of name for a is arbitrary and can be renamed as necessary. Roughly speaking, atoms are more mutable than constants, but less mutable than unification variables. Definition 2.2.1, which is based on that in [11], describes *nominal terms* more precisely.

Definition 2.2.1. *(Nominal terms.)* We extend ordinary Prolog terms (Definition 2.1.1) with atoms and atom abstractions. The presence of *permutations*, π, is a corollary of the algorithm that unifies nominal terms, which is the subject of Section 2.3.

$$
\begin{array}{rcll}
t & ::= & \langle t_1, \ldots, t_n \rangle & \text{(tuple)} \\
 & | & \texttt{f}\ t & \text{(data)} \\
 & | & a & \text{(atom)} \\
 & | & a\backslash t & \text{(atom abstraction)} \\
 & | & \pi \cdot X & \text{(suspended variable)}
\end{array}
$$

2.3 Nominal unification

Roughly speaking, Prolog execution proceeds by repeatedly unifying goals with clauses. In this section we consider how to extend the ordinary unification algorithm such that it is able to unify nominal terms. The resulting 'nominal unification' algorithm is that devised by Urban, Pitts and Gabbay in [12].

The goal of ordinary unification is to make a pair of terms equivalent. The goal of nominal unification is to make a pair of terms α-equivalent; that is, equivalent up to the Choice Axiom we defined earlier. Accordingly, we begin by defining a notion of α-equivalence for each type of nominal term in Definition 2.2.1.

2.3.1 Tuples, data, atoms

Two n-tuples are α-equivalent if each of the n corresponding pairs of elements are α-equivalent. Two data terms are α-equivalent if they have the same function symbol and their arguments are α-equivalent. We mentioned earlier that atoms are mutually distinct; consequently, two atoms are α-equivalent if they have the same identifier.

2.3.2 Atom abstractions

The first complicated case is the decision of whether two atom abstractions are α-equivalent, which divides into two subcases. If the binder is the same for both terms, then we need only consider whether the 'bodies' of the two terms are α-equivalent. Otherwise, the terms have different binders; that is, they are of the form $a\backslash t$ and $b\backslash t'$. Clearly these terms are not *equivalent*, but we may yet be able to prove them *α-equivalent*.

We would like to rename b to a throughout t', for should this make t equivalent to t' then our terms are indeed α-equivalent. This is a very dangerous operation if a already appears in t':

- If a appears *free* in t' then stop immediately: the terms are clearly distinct because a is free in the second term yet not free in the first.
- If a appears *bound* in t', then to prevent the binder for a capturing b atoms after they are renamed, we simultaneously rename a to b. This 'swapping' operation is written $(a\ b)$.

Swapping atoms in a ground term is simple enough, but we cannot apply a swap to a unification variable until we know what value it will take. Instead, we

$$\pi ::= \mathsf{id} \mid (a\ b)::\pi$$

$$\mathsf{id}(a) \stackrel{\text{def}}{=} a \qquad ((b\ c)::\pi)(a) \stackrel{\text{def}}{=} \begin{cases} b & \text{if } \pi(a) = c \\ c & \text{if } \pi(a) = b \\ \pi(a) & \text{otherwise} \end{cases}$$

$$\pi(\langle t_1, \ldots, t_n\rangle) \stackrel{\text{def}}{=} \langle \pi(t_1), \ldots, \pi(t_n)\rangle$$
$$\pi(\mathtt{f}\ t) \stackrel{\text{def}}{=} \mathtt{f}(\pi(t))$$
$$\pi(a\backslash t) \stackrel{\text{def}}{=} \pi(a)\backslash\pi(t)$$
$$\pi(\pi' \cdot X) \stackrel{\text{def}}{=} \pi@\pi' \cdot X$$

Figure 2.1: The syntax and semantics of permutations.

simply stack up a sequence of swaps outside the variable, ready for application as soon as the variable assumes a value. We call a sequence of zero or more swaps a *permutation* (see Figure 2.1), and a variable preceded by a permutation a *suspended variable*.

Now armed with a suitable permutation function, we can state that $a\backslash t$ and $b\backslash t'$ are α-equivalent if t is α-equivalent to $(a\ b)(t')$, and a does not appear free in t'. This latter condition is somewhat verbose, so we shall henceforth write it as "a is fresh for t'" or even just "$a \# t'$".

2.3.3 Suspended variables

For ordinary Prolog terms it is trivial to see that X and X are α-equivalent, and in the nominal world the fact that $\pi \cdot X$ is α-equivalent to $\pi \cdot X$ is equally obvious. What is somewhat less clear is whether we can show α-equivalence between $\pi \cdot X$ and $\pi' \cdot X$.

For a pedagogical example, consider the terms $(a\ b) \cdot Y$ and Y[1]. Were we later to substitute a for Y then the left-hand term would become b, the right-hand one a, and the terms would not be α-equivalent. The situation is the same were we to substitute b for Y. Nevertheless, provided Y is not later replaced with a or b or any term containing a free occurrence of a or b, then the terms are α-equivalent.

Returning to the general terms $\pi \cdot X$ and $\pi' \cdot X$, we find that a sufficient condition for α-equivalence is to require that we do not substitute for X any term that contains a free occurrence of any atom in the *disagreement set* of π and π'.

Definition 2.3.1. *(Disagreement set.)* The disagreement set of π and π' is defined as the set of atoms, a, for which $\pi(a) \neq \pi'(a)$. It is written $\mathrm{ds}(\pi, \pi')$.

The condition is satisfied exactly when we can show that a is fresh for X for all a in $\mathrm{ds}(\pi, \pi')$.

[1] Formally, the latter term should be written $\mathsf{id} \cdot Y$ but it is customary to omit the permutation when it is empty.

2.3.4 Freshness judgements, equational judgements

In both the atom abstraction case and the suspended variable case, the demonstration of α-equivalence depended upon being able to prove that some atom a is fresh for a term t. Doing this is trivial when t is a ground term; indeed, the only case worth mentioning here is that when t is a suspended variable. For instance, the problem of whether b is fresh for $(b\ c) \cdot Y$ is undecidable, simply because we can later substitute for Y any term we like. We can make such problems decidable, however, by augmenting them with extra information. The nature of this extra information is a finite set of *freshness constraints* (called a *freshness environment* and denoted ∇). Each freshness constraint is of the form $a \# X$; such a constraint expresses that we will never substitute for X any term containing a free occurrence of a. A freshness problem augmented with a freshness environment in this way is called a *freshness judgement*, is written $\nabla \vdash a \# t$, and expresses that under the assumptions in ∇ it is provable that a is fresh for t.

Since the demonstration of α-equivalence may depend on the demonstration of freshness, and freshness judgements require a freshness environment, then (by transitivity) judgements about α-equivalence (which we shall term *equational judgements*) also require a freshness environment. Accordingly, an equational judgement is written $\nabla \vdash t \approx t'$; it expresses that under the freshness constraints in ∇, it is provable that t and t' are α-equivalent.

We are now ready to present the formal inductive definitions of equational judgements and freshness judgements, which we do in Figures 2.2 and 2.3.

2.3.5 The unification process

The job of the unification algorithm is succinctly stated: given a list (denoted P) of *equational problems* and *freshness problems*, the algorithm must find a freshness environment, ∇, and a substitution of terms for variables, σ, such that for each equational problem $t \approx^? t'$ in P, the equational judgement $\nabla \vdash \sigma(t) \approx \sigma(t')$ holds, and for each freshness problem $a \#^? t$ in P, the freshness judgement $\nabla \vdash a \# \sigma(t)$ holds. The details of how the algorithm achieves this are given in Section 3.1.

A typical job given to the algorithm is to unify a pair of terms, t and t'; the input in this case is a list containing the single equational problem $t \approx^? t'$.

$$(\approx\text{-TUPLE})\ \frac{\nabla \vdash t_1 \approx t_1' \qquad \cdots \qquad \nabla \vdash t_n \approx t_n'}{\nabla \vdash \langle t_1, \ldots, t_n \rangle \approx \langle t_1', \ldots, t_n' \rangle} \qquad (\approx\text{-DATA})\ \frac{\nabla \vdash t \approx t'}{\nabla \vdash \mathtt{f}\ t \approx \mathtt{f}\ t'}$$

$$(\approx\text{-ATOM})\ \frac{}{\nabla \vdash a \approx a} \qquad (\approx\text{-ABST-1})\ \frac{\nabla \vdash t \approx t'}{\nabla \vdash a\backslash t \approx a\backslash t'}$$

$$(\approx\text{-ABST-2})\ \frac{a \neq a' \qquad \nabla \vdash t \approx (a\ a')(t') \qquad \nabla \vdash a \# t'}{\nabla \vdash a\backslash t \approx a'\backslash t'}$$

$$(\approx\text{-VAR})\ \frac{\{a \# X \mid a \in \mathrm{ds}(\pi, \pi')\} \subseteq \nabla}{\nabla \vdash \pi \cdot X \approx \pi' \cdot X}$$

Figure 2.2: Inductive definition of equational judgements.

$$(\#\text{-TUPLE})\ \frac{\nabla \vdash a \# t_1 \qquad \cdots \qquad \nabla \vdash a \# t_n}{\nabla \vdash a \# \langle t_1, \ldots, t_n \rangle} \qquad (\#\text{-DATA})\ \frac{\nabla \vdash a \# t}{\nabla \vdash a \# \mathtt{f}\ t}$$

$$(\#\text{-ATOM})\ \frac{a \neq a'}{\nabla \vdash a \# a'} \qquad (\#\text{-ABST-1})\ \frac{}{\nabla \vdash a \# a\backslash t}$$

$$(\#\text{-ABST-2})\ \frac{\nabla \vdash a \# t \qquad a \neq a'}{\nabla \vdash a \# a'\backslash t} \qquad (\#\text{-VAR})\ \frac{(\pi^{-1}(a) \# X) \in \nabla}{\nabla \vdash a \# \pi \cdot X}$$

Figure 2.3: Inductive definition of freshness judgements.

Chapter 3

Implementation

This chapter details the implementation of an interpreter for Nominal Prolog. Section 3.1 describes the implementation of nominal unification. Section 3.2 describes the solution-finding algorithm that employs nominal unification to try to find solutions to queries. Section 3.3 describes the interactive shell that allows a user to invoke the solution-finding algorithm.

A note on originality. The nominal unification algorithm is described in [12]. The implementation is original, but loosely based on that outlined in [2]. The solution-finding algorithm is original, but informed by the method suggested in [9].

We shall make use of the following definitions. A *program* is a set of clauses. A *clause* comprises a nominal term (the *head*) and a list of predicates (the *tail*). In Nominal Prolog, a *predicate* is either a freshness problem (which is true only when the atom a can be proved to be fresh for the term t) or a cut (which is always true) or a nominal term (which is true only if it appears as the head of some clause and all of the predicates comprising the tail of that clause are also true). A *goal* is a list of predicates.

3.1 Nominal unification

In Section 2.3.5 we established that the input to the nominal unifier is a list (P) of equational problems ($t \approx^? t'$) and freshness problems ($a \#^? t$), and that the output is a freshness environment (∇) and a substitution of terms for variables (σ) that are such that:

- for each $(t \approx^? t') \in P$, $\nabla \vdash \sigma(t) \approx \sigma(t')$, and
- for each $(a \#^? t) \in P$, $\nabla \vdash a \# \sigma(t)$.

Figure 3.1: Effecting the substitution $X := t$ by creating a pointer.

In this section we explain what happens in between. We begin with three reasons why this is a good implementation of the Urban-Pitts-Gabbay algorithm for nominal unification.

It represents terms as graphs rather than trees. In αProlog, a similar system to Nominal Prolog, a nominal term is represented as a tree. Nominal Prolog represents a nominal term as a directed acyclic graph instead. The main advantage of computing with a graph rather than a tree is that common sub-terms can be shared, thus improving the efficiency of computations in both space and time.

It deals with substitutions efficiently. Recall that part of the output of the nominal unifier is a substitution of terms for variables. A naïve implementation would effect the substitution $X := t$ by replacing each occurrence of the variable X with a copy of the term t. Since we are working with a graph, and because graphs allow us to share all common sub-terms, we know that although the variable X may appear numerous times in the textual representation of the term, there will never be more than one node representing it in the graph. Therefore, a substitution need only replace *one* occurrence of the variable X with a copy of the term t. We can do better yet. Rather than create a copy of t we need simply create a pointer from X to the t node that already exists. Figure 3.1 demonstrates. This operation is sound provided we check that t does not contain a pointer to X. This check is called the occurs-check. Note the use of a 'TRAN' node, which exists purely to redirect. It can be thought of as a 'transparent' or 'transitive' node.

It deals with permutations efficiently. In the standard formulation of nominal terms, a permutation may be suspended only over a variable. For our implementation, the definition is relaxed to allow permutations to be suspended over any nominal term. This allows us to be lazy, in that we no longer have to push permutations inside terms until it is absolutely necessary to do so. The naïve approach, which eagerly evaluates every permutation it comes across, is liable to waste effort pushing a permutation

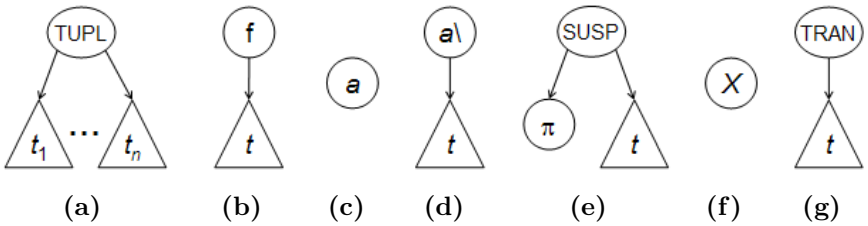


Figure 3.2: The seven kinds of graph node: (a) tuple, (b) data, (c) atom, (d) atom abstraction, (e) suspended permutation, (f) variable, (g) 'TRAN' node. This figure uses a triangle to denote an arbitrary term graph; the fact that the children of the tuple node appear disjoint should preclude neither the possibility that they are connected, nor that the graphs may even be identical.

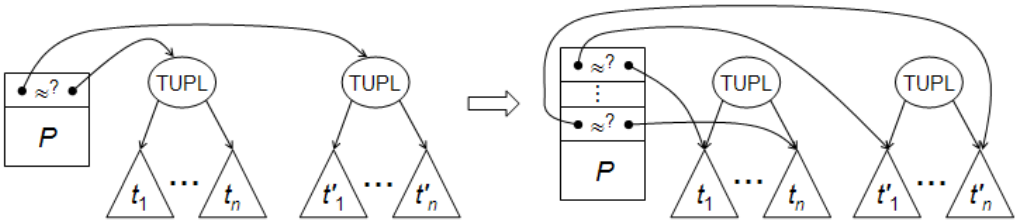
all the way through a large term, only to find that this term is not even needed.

This discussion leads us to the seven kinds of term graph that are given in Figure 3.2.

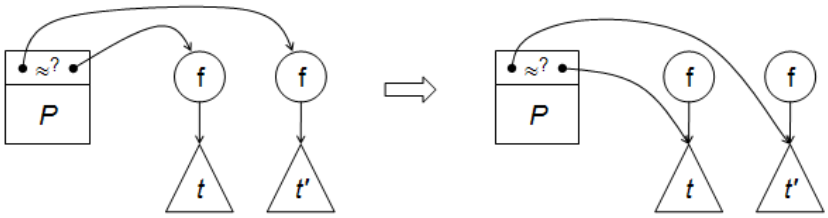
3.1.1 Overview of the algorithm

Central to the algorithm are the collection of transformations that are presented in Figures 3.3 and 3.4. The left-hand side of each transformation specifies the state in which the list of problems must be in order for that transformation to be applicable; for example, the second transformation in Figure 3.3 can be applied only if the first problem is an equational problem relating two data terms that have the same function symbol. The right-hand side describes the state after the transformation is applied; in the same example the transformation replaces the first problem in the list with a new problem that relates the bodies of the two data terms.

An overview of the procedure follows. The algorithm begins with an initial list of problems, P. The first stage of the process is to apply as many *equational transformations* as possible; that is, to construct a sequence $P \Longrightarrow \cdots \Longrightarrow P'$ such that P' matches the left-hand side of none of the transformations in Figure 3.3. Should P' still contain equational problems, we abort the procedure. Otherwise, we progress to the second stage, where we apply as many *freshness transformations* as possible; that is, we construct a sequence $P' \Longrightarrow \cdots \Longrightarrow P''$ such that P'' matches the left-hand side of none of the transformations in Figure 3.4. Should P'' be non-empty, then we abort the procedure; otherwise, we have succeeded.

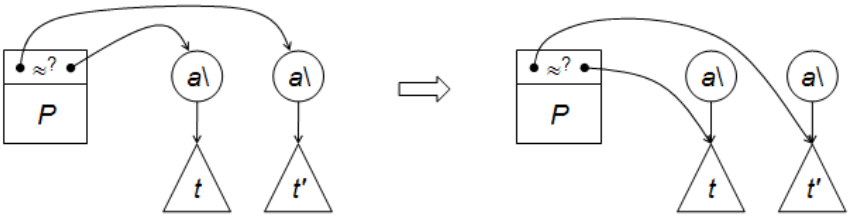


(1) Reducing tuples.



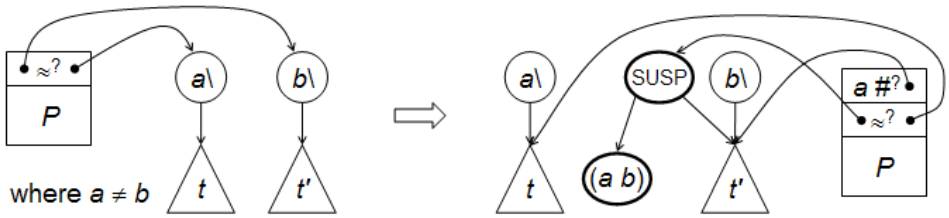
(2) Reducing data.

(3) Reducing atoms.

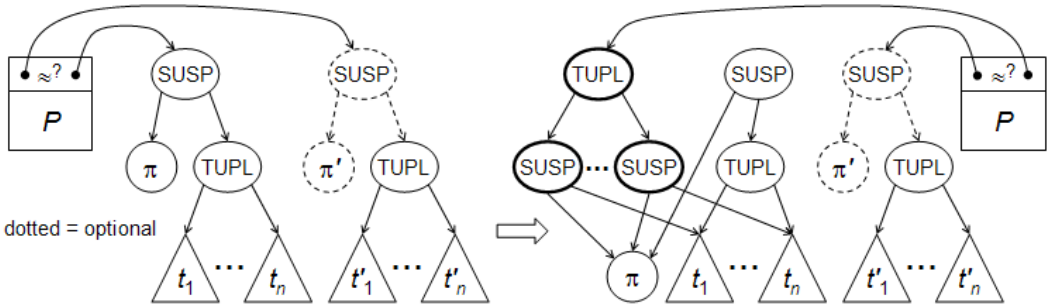


(4) Reducing atom abstractions that have identical binders.

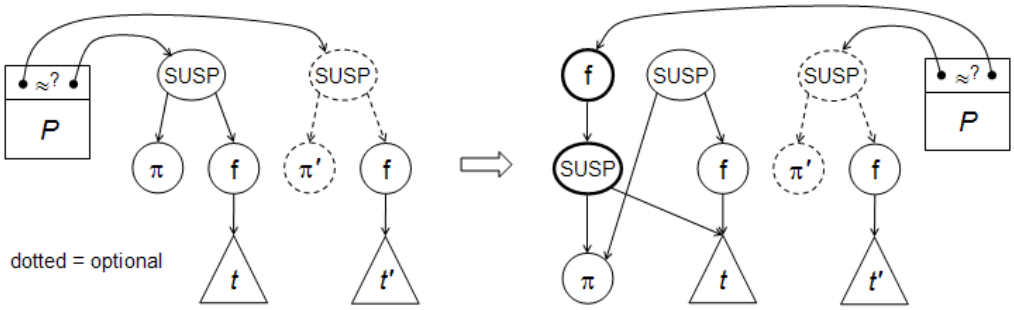
Figure 3.3: Equational transformations.



(5) Reducing atom abstractions that have different binders.

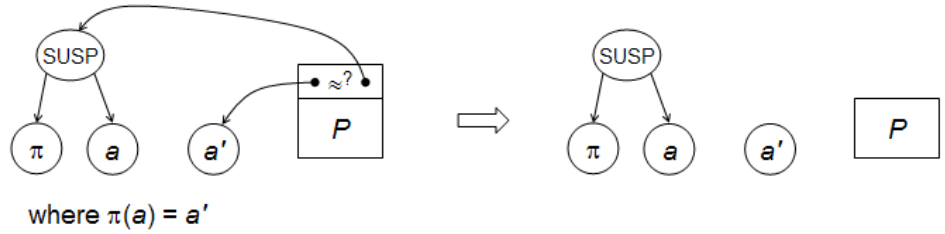


(6) Pushing suspension inside tuple.

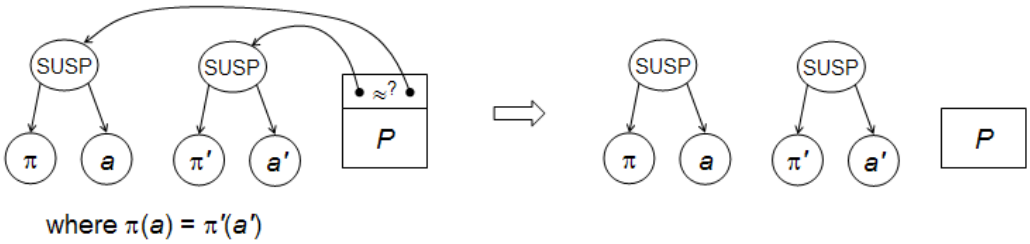


(7) Pushing suspension inside data.

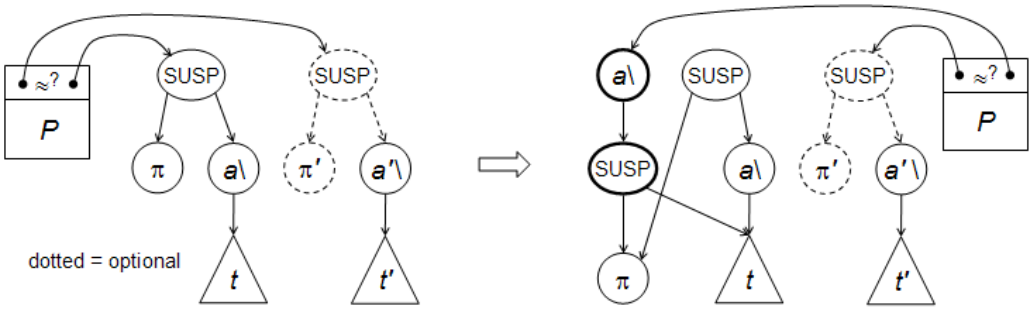
Figure 3.3: Equational transformations, *continued*.



(8) Pushing suspension inside one atom.

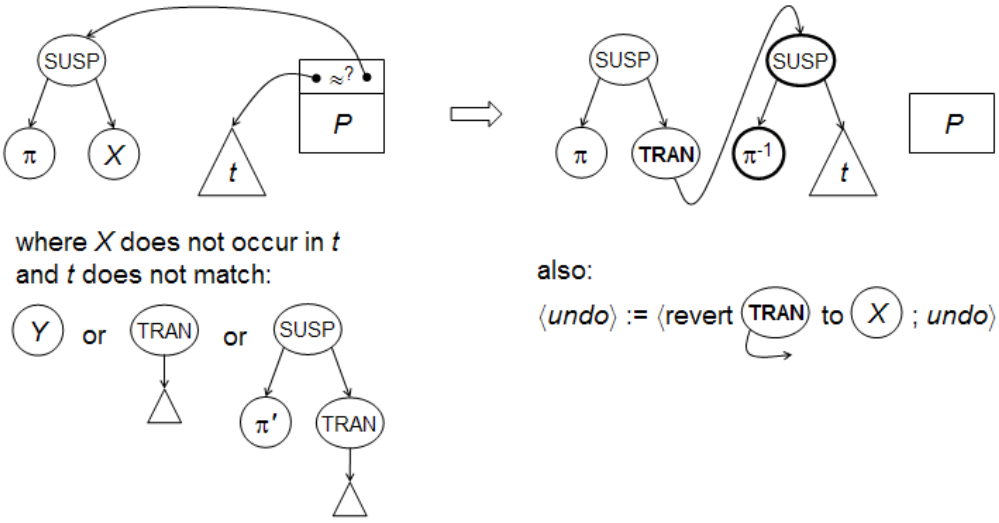


(9) Pushing suspension inside two atoms.

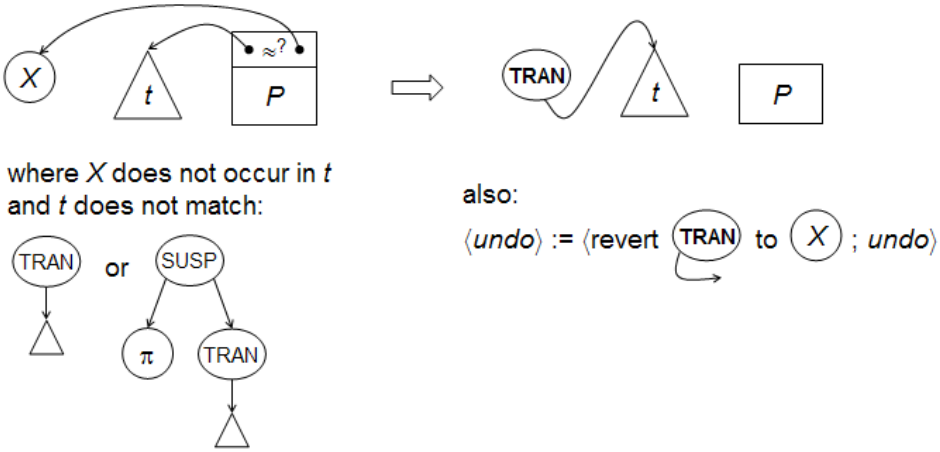


(10) Pushing suspension inside atom abstraction.

Figure 3.3: Equational transformations, *continued*.



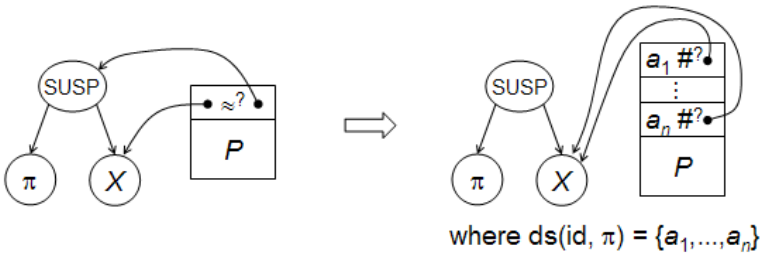
(11) Substituting suspended variable.



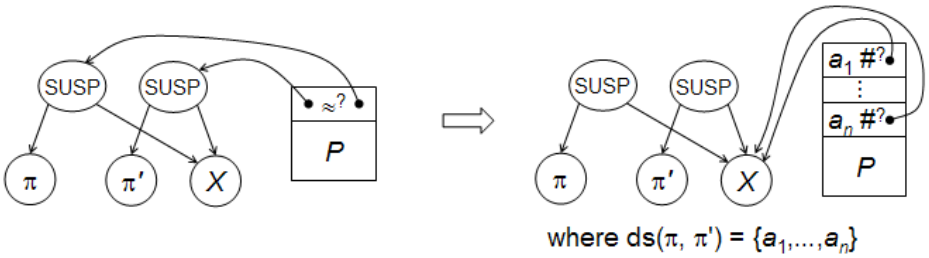
(12) Substituting un-suspended variable.

(13) Reducing variable.

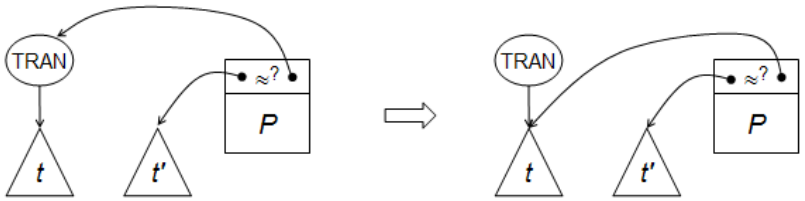
Figure 3.3: Equational transformations, *continued*.



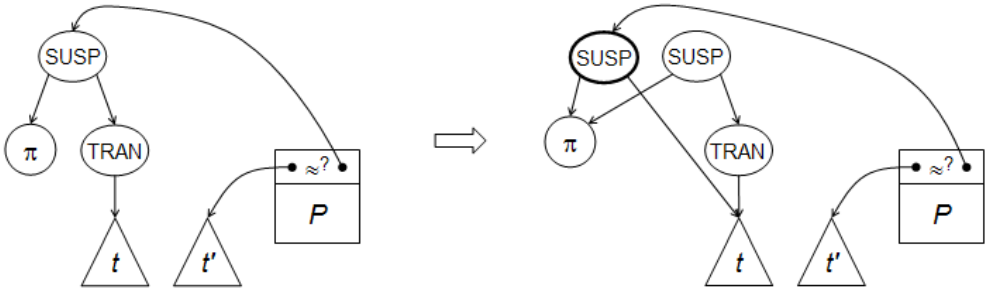
(14) Reducing variable, one suspension.



(15) Reducing variable, two suspensions.

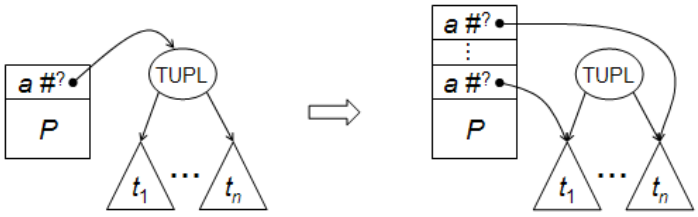


(16) Reducing TRAN node.

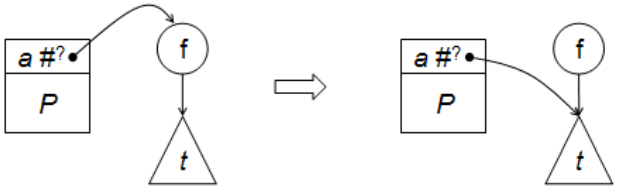


(17) Reducing TRAN node under suspension.

Figure 3.3: Equational transformations, *continued*.

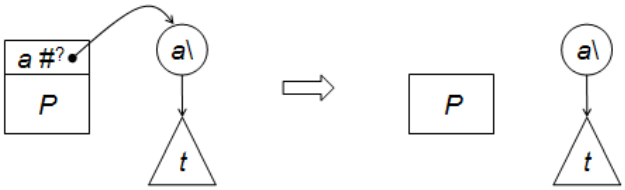


(1) Freshness of tuple.



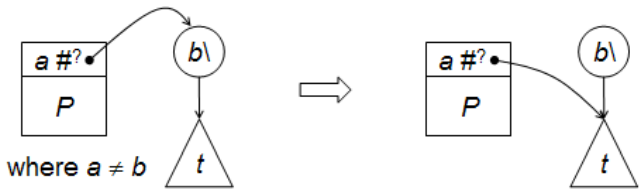
(2) Freshness of data.

(3) Freshness of atom.

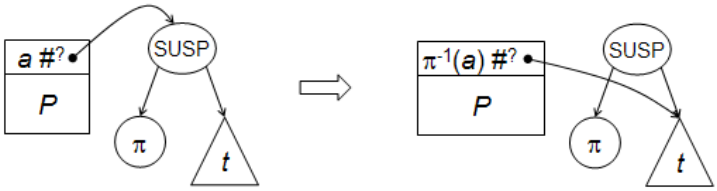


(4) Freshness of atom-abstraction that has common binder.

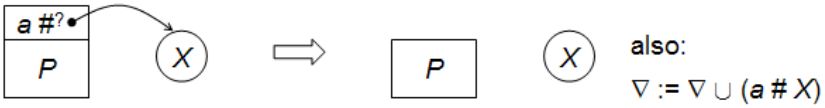
Figure 3.4: Freshness transformations.



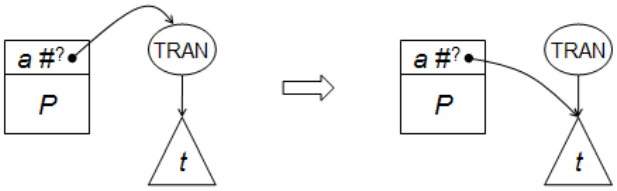
(5) Freshness of atom-abstraction that has different binder.



(6) Freshness of suspension.



(7) Freshness of variable.



(8) Freshness of TRAN node.

Figure 3.4: Freshness transformations, *continued*.

3.1.2 The equational transformations

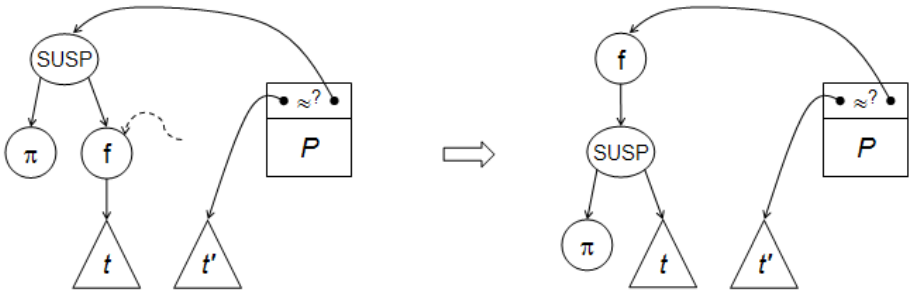
Transformations 1 through 5 are simple, and correspond exactly to the definition of $\approx$ in Figure 2.2. Transformation 5 is the first to introduce new nodes to the graph; such nodes are highlighted by a bold outline. It is crucial to note that no rules remove nodes from the graph. We can see that this is important when we realise that only part of the graph is depicted in the transformation diagrams. The rest of the graph must be unaffected by the transformation; ensuring this requires the preservation of any nodes that could be pointed to from nodes in the rest of the graph.

Transformations 6 through 10 all push a suspended permutation inside a term. They are based on the definition of the application of a permutation to a term (Figure 2.1). Some of the nodes and edges in these transformations are dotted; this means that the transformation applies both when all the dotted items are present (in which case they remain present after the transformation) and when all of the dotted items are absent (in which case they remain absent after the transformation). Transformation 6 is the first of several to have a (potentially) asymmetric left-hand side; for such transformations it is implicit that the operands of $\approx^?$ can be commuted.

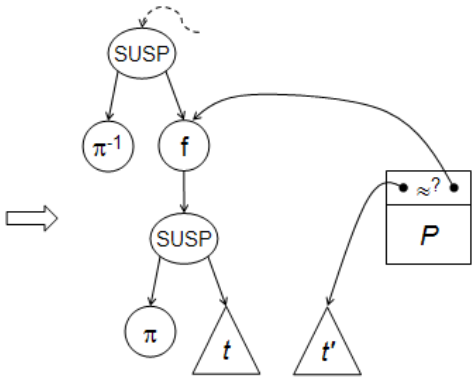
Remark 3.1.1. One might think that transformation 7 can equally well be formulated in the simpler way given in Figure 3.5a, and that the other suspension-pushing transformations can be reformulated similarly. It is certainly true that the left-hand and right-hand graphs are, as drawn, equivalent. Nonetheless, we must consider that there may be an edge from a part of the graph that is not depicted, pointing to the data term $\mathtt{f}\ t$. Such an edge is depicted in dashed form on the left-hand side of Figure 3.5a. The problem is that the right-hand side has no equivalent node to which this edge can point. The improved right-hand side given in Figure 3.5b provides such a node[1], thus restoring the transformation's soundness. Still, in making one problem less complicated, we may have made another problem more complicated (for instance, any problem involving $\mathtt{f}\ t$ now involves $\pi^{-1} \cdot \mathtt{f}(\pi \cdot t)$ instead). Fortunately, any sequence of transformations will still terminate, but proving this is much harder. Accordingly, in the interest of making the proof of termination easier (see Section 4.1.2), we shall stick with the original transformation given in Figure 3.3.

Transformations 11 and 12 are particularly important—they are how the algorithm performs the substitutions that are necessary to solve equational problems that include non-ground terms. The equational problem depicted on the left-hand side of transformation 11 is $\pi \cdot X \approx^? t$. This is solved by setting X to $\pi^{-1} \cdot t$. Transformation 12 deals with the simpler case where there is no permutation suspended in front of the variable. In both cases, the node that contained

[1] since $\pi^{-1} \cdot \mathtt{f}(\pi \cdot t)$ is equivalent to $\mathtt{f}\ t$



(a) A first attempt



(b) A refinement

Figure 3.5: An alternative version of equational transformation 7.

the variable X has been replaced by a 'TRAN' node. The diagrams emphasise the fact that the contents of the node has changed by setting the node's text in bold type. Indeed, these transformations are the only ones that actually modify nodes: the most the others do is to create new nodes. So that these modifications may later be reversed, we accumulate an $\langle undo \rangle$ function as the series of transformations progresses. Transformations 11 and 12 are the only ones that modify this function, and they both do so by prepending an instruction to revert the newly created 'TRAN' node to the original variable node. We introduce here the notation $\langle x \rangle$, which denotes a function that takes a unit argument and returns the value x—this is not to be confused with a 1-element tuple!

Remark 3.1.2. Transformations 11 and 12 impose seemingly unnecessary constraints on the type of node that t may be. For instance, the t in transformation 11 must not be a variable, yet the transformation would still be valid if it were. The constraints exist to give the transformations the nice property of determinacy: that is, for any list of problems there is never more than one transformation whose left-hand side matches. Determinacy is the reason for the harsh constraint imposed in transformation 7 (and the other suspension-pushing transformations): we allow $\pi \cdot \mathtt{f}\ t$ to unify only with a term of the form $\mathtt{f}\ t'$ or $\pi \cdot \mathtt{f}\ t'$ because it

reduces the number of pairs of terms that match this transformation. The alternative version of transformation 7 in Figure 3.5 neglects this constraint.

Transformations 13 through 15 correspond to the ($\approx$-VAR) rule in Figure 2.2. Recall that variables no longer need prefixing with a permutation. Accordingly, the three transformations correspond to neither, one and both variables being prefixed with a permutation, respectively. Transformations 13 and 14 can be derived from 15 by taking one or both of the permutations to be the identity permutation.

Finally, transformations 16 and 17 deal with the simple matter of reducing a 'TRAN' node. Since the 'TRAN' node is an invented concept, there is no formal framework upon which it can rest, as there was for the other transformations. Happily, the semantics of 'TRAN' nodes is very simple: the node should behave as if it were not there. Transformation 16 reduces a solitary 'TRAN' node, and transformation 17 reduces a 'TRAN' node while it is still inside a suspension.

3.1.3 The freshness transformations

Freshness transformations are fewer and simpler, and correspond exactly to the definition of # in Figure 2.3. While performing equational transformations we accumulate an $\langle undo \rangle$ function; while doing freshness transformations we accumulate a freshness environment, ∇, instead. This manifests itself only in transformation 7: to show that a is fresh for a variable X it is sufficient to add $a \# X$ to our freshness environment and rely upon the (#-VAR) rule of Figure 2.3.

3.1.4 Unit testing

The vast majority of unification problems either do not unify or have trivial solutions. Therefore, the testing of the nominal unifier is best done not by running it on thousands of randomly-generated problems, but on a few carefully-selected problems. A set of four unification problems that provide good testing coverage are presented in [12]; the results of running these are presented in Table 3.1.

3.2 The solution-finding algorithm

Figure 3.6 presents a pseudocode for the algorithm used by Nominal Prolog to find solutions to queries. This algorithm forms the core of Nominal Prolog, so it is appropriate to devote this section to a detailed explanation.

```
1  EXEC(pos, goal, ∇acc, ⟨bktrk⟩, ⟨abort⟩) =
2      match goal with
3      [] ⇒ Yes(∇acc, ⟨bktrk⟩, ⟨abort⟩)
4      pred :: rest_of_goal ⇒
5          match pred with
6          a #? t ⇒
7              try UNIFY(a #? t) with
8              Fail ⇒ bktrk
9              Succeed(∇, ⟨undo⟩) ⇒
10                 EXEC(0, rest_of_goal, ∇ ∪ ∇acc,
                       ⟨undo; bktrk⟩, ⟨undo; abort⟩)
11         ! ⇒
12             EXEC(0, rest_of_goal, ∇acc, ⟨No⟩, ⟨abort⟩)
13         t ⇒
14             if pos ≥ |program| then
15                 bktrk
16             else
17                 let (c, ⟨uncopy⟩) = COPY(program[pos]) in
18                 let ⟨next⟩ = ⟨EXEC(pos + 1, goal, ∇acc, ⟨bktrk⟩, ⟨abort⟩)⟩ in
19                 try UNIFY(c.head ≈? t) with
20                 Fail ⇒
21                     uncopy; next
22                 Succeed(∇, ⟨undo⟩) ⇒
23                     EXEC(0, c.tail @ rest_of_goal, ∇ ∪ ∇acc,
                           ⟨undo; uncopy; next⟩, ⟨undo; uncopy; abort⟩)
```

Figure 3.6: The solution-finding algorithm

Problem	Result
$\lambda a.\lambda b.X_1\, b \approx^? \lambda b.\lambda a.a\, X_1$	Passed: no solution.
$\lambda a.\lambda b.X_2\, b \approx^? \lambda b.\lambda a.a\, X_3$	Passed: $X_2 := b, X_3 := a$.
$\lambda a.\lambda b.b\, X_4 \approx^? \lambda b.\lambda a.a\, X_5$	Passed: $X_4 := (a\ b) \cdot X_5$
$\lambda a.\lambda b.b\, X_6 \approx^? \lambda a.\lambda a.a\, X_7$	Passed: $X_6 := (b\ a) \cdot X_7, b \# X_7$.

Table 3.1: The results of testing the nominal unifier. Here we use the syntax of the λ-calculus over that of nominal terms for better readability.

3.2.1 The return type

We shall begin the analysis by examining the return type of EXEC. There may be zero, one or many solutions for any given goal, and the return type of EXEC needs to reflect this. We might propose a list of solutions as a suitable datatype, but this would be inappropriate: the construction of such a list would necessitate the evaluation of all solutions, of which there may be an infinite number. The desired semantics is a result that expresses either "There are no solutions" or "There are one or more solutions, and here is the first one." Presented with a result of the latter variant, we need a way of asking for another solution. This is best achieved by including alongside the solution a function (which shall be known as $\langle bktrk \rangle$) that can be invoked with unit argument to return another solution (which itself might contain a $\langle bktrk' \rangle$ function that can lead to yet another solution, and so on).

The solution itself is a substitution of terms for variables, together with a freshness environment—at least, this is what is presented to the user. It is in fact unnecessary and inefficient to explicitly output a substitution. As was explained earlier, substitutions are best effected by simply creating redirection pointers. So that we can detect what substitutions have occurred, we create a pointer to each variable node in the graph before running the solution-finding algorithm. Afterwards, we follow each pointer: if a pointer still points to a variable node, then that variable must not have been substituted, but if the pointer now points to a 'TRAN' node, then the variable must have been substituted.

Once we have finished finding solutions to a query, we need to ensure that all the extra pointers that we created are erased, so that future queries have a 'blank slate' on which to work. Cunningly, we can embed this erasing process inside the $\langle bktrk \rangle$ function, such that it first destroys the pointers that were created by the previous solution before searching for the next solution. This means that the slate is only fully clean, and ready to accommodate a new query, after the final solution to the current query has been found. Since there may be an unlimited number of solutions (or the user may be impatient), we would like a way of 'aborting' the query before all the solutions are found. This is achieved by providing not

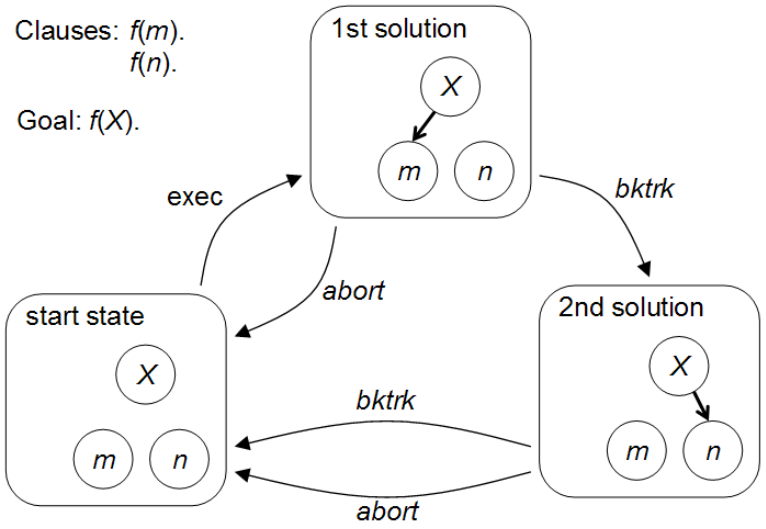


Figure 3.7: A demonstration of the $\langle bktrk \rangle$ and $\langle abort \rangle$ functions. From the start state, EXEC leads us to the first solution, $X := m$. From there we can either execute $\langle abort \rangle$ (which returns us to the start state) or we can execute $\langle bktrk \rangle$, which leads us to the second solution, $X := n$. There are no further solutions, so subsequent execution of either $\langle abort \rangle$ or $\langle bktrk \rangle$ will return us to the start state.

just the $\langle bktrk \rangle$ function, but an $\langle abort \rangle$ function too, which simply erases all the pointers and ignores any other solutions. The heavily simplified diagram in Figure 3.7 should clarify the usage of these functions.

We are thus led to the following return type of EXEC.

```
type execute_result = No
                    | Yes of freshness_environment
                            * unit → execute_result
                            * unit → unit
```

3.2.2 The arguments

Table 3.2 gives a brief description of each of EXEC's arguments. We give the type and briefly state the purpose of each argument. Being a recursive function, it is also informative to record what value should be given to each argument at the initial invocation.

Identifier	Type	Purpose	Initial value
pos	integer	the pos^{th} clause is the next one we should try to unify the *goal* with	0
goal	predicate list	a list of predicates that are yet to be solved	provided by user
∇_{acc}	freshness_ environment	accumulates the set of freshness constraints evoked by each successive unification	$\emptyset$
$\langle bktrk \rangle$	unit → execute_result	when executed, searches for an alternative solution	$\langle \text{No} \rangle$
$\langle abort \rangle$	unit → unit	accumulates a sequence of operations that, when executed, will restore our graph to its start state	$\langle \textbf{skip} \rangle$

Table 3.2: A description of each formal argument of EXEC.

3.2.3 The operation of the algorithm

Since a goal is a conjunction of predicates, then an empty *goal* is trivially true, so we have located a solution (line 3). Otherwise, *goal* must contain a first predicate, which we call *pred* (line 4). Predicates come in one of three variants: freshness problems, the cut predicate, and terms. We consider each case in turn.

Freshness problems (line 6)

A freshness problem, such as $a \#^? t$, is satisfied by demonstrating that the atom a is fresh for the term t. This is a problem of nominal unification. Should this unification fail (line 8), then we unable to prove our goal: a goal being a conjunction of predicates means that we require every predicate (including *pred*) to be provable. Nonetheless, it is possible that the original goal can yet be proved, by proving instead a different set of predicates not including *pred*. We explore this possibility by executing the $\langle bktrk \rangle$ function. Note, in line 8, that we have removed the angular braces from $\langle bktrk \rangle$—we are no longer referring to the function itself, but to the result of its application to a unit argument.

Should the unification succeed, we issue a recursive call of EXEC (line 10). Having proved *pred*, it remains only to prove *rest_of_goal*; this is our new goal. We add

to our accumulating freshness environment the set of constraints evoked by the unification. In order to backtrack or abort, we need first to undo the state changes that were effected by the successful unification. Thus we arrange for execution of the new ⟨*bktrk*⟩ and ⟨*abort*⟩ functions first to invoke ⟨*undo*⟩ (which reverses the changes to the graph made by the unification procedure) then to do the same work as that done by the previous one. Note that it was unnecessary to undo any state changes made by the unsuccessful unification on line 8, because by returning Fail, the UNIFY function guarantees that no state changes are exported.

The cut predicate (line 11)

The cut predicate, rendered as '!', always succeeds. Hence, in the recursive call to EXEC, we need only prove the predicates in *rest_of_goal*. No changes are necessary to the freshness environment, nor need we modify the ⟨*abort*⟩ function. Indeed, the cut predicate would be largely redundant, were it not for its effect on the ⟨*bktrk*⟩ function. Until this point, the ⟨*bktrk*⟩ function had contained all the information we needed to be able to find alternative solutions should we fail to solve a predicate. Upon solving a cut predicate, we reset the ⟨*bktrk*⟩ function to its default value, ⟨No⟩, effectively destroying all of this information. We are now in the unhappy situation whereby failure to prove *rest_of_goal* precipitates failure of the entire goal, despite the possibility that execution of the ⟨*bktrk*⟩ function (before it was erased) would have led to a solution. Still, in some cases, it is the programmer's intention that such a solution is not found, and this is where a cut is most handy.

Terms (line 13)

A term, t, is true if and only if there exists a clause whose head matches t and for which each predicate in its tail is also true. To find such a clause, we iterate over each of the program's clauses in turn, using *pos* as the iteration counter, checking each to see if it meets the above condition. At line 15, *pos* has exceeded the number of clauses in the program; this means that none of the clauses meet the condition for t to be given the value 'true', so we should abandon our attempt to prove this predicate and backtrack in the hope of finding an alternative solution.

Otherwise, we have established that *program*[*pos*] is a clause in the program[2]. Before we set about checking if it meets our condition, we take a copy of it, which we call c (line 17). This is necessitated by the destructive nature of the unification process—it permanently modifies the term graph by creating new pointers. It is commonplace for a single execution to use the same clause many times, and it is clearly unsatisfactory if, because of some previous unification

[2]Here we imagine *program* to be a zero-based array of clauses.

step, the clause is different every time we want to use it. Accordingly, we keep an 'original' of each clause, and only hand out copies to be unified. The COPY function, when given a clause, returns a deep copy of the clause that can be safely modified without affecting the original. It also returns an $\langle uncopy \rangle$ function that can later be invoked to remove the copy from the graph; this prevents memory from being filled up with redundant copies of clauses.

The term t is true if t matches the head of c and all of the predicates in the tail of c can be proved. The first hurdle—deciding whether t matches the head of c—is dealt with by our UNIFY function. Should this fail (line 20), it is clear that c is not the right clause to use, so we increment our iteration counter and re-invoke EXEC. If t does match the head of c, then all that remains is to prove each predicate in the tail of c. This is accomplished by yet another recursive call to EXEC (line 23). For this call, we reset *pos* to zero, ready for any future iteration over the set of clauses. Our goal now is to prove the tail of c; having done that we must finish proving the goal we were previously working on. Hence we set *goal* to be *c.tail* with *rest_of_goal* appended after it. We augment our freshness environment, ∇, with those freshness constraints that arose from the successful unification of *c.head* with t. The new $\langle abort \rangle$ function needs to undo the effects of the calls to COPY and UNIFY (and it should do so in the opposite order to that in which they were applied). Thus we arrange for execution of the new $\langle abort \rangle$ function first to invoke $\langle undo \rangle$, secondly to invoke $\langle uncopy \rangle$, and thirdly to do the same work as that done by the previous one. The $\langle bktrk \rangle$ function follows a similar structure in that it firstly reverses the effect of UNIFY and secondly reverses the effect of COPY. However, having done that, we realise that c happening to be a clause whose head matches t does not imply that there are no further clauses with this property. Accordingly, the third step of the new $\langle bktrk \rangle$ function should be to continue iterating over the rest of the clauses, starting at the $(pos + 1)^{\text{th}}$ clause. Only once these remaining clauses have been checked should the original $\langle bktrk \rangle$ function be invoked.

3.3 The interactive shell

An interactive shell was produced, to allow users to interact with Nominal Prolog. The accepted form of input, which is parsed by a lexer and parser built using *OCamlLex* and *OCamlYacc*, is given by the following grammar.

input ::= *directive* | *query* | *definition*
: The user provides *definitions* of clauses, then runs *queries* on them. *Directives* invoke miscellaneous commands.

directive ::= `#verbose on.`
| `#verbose off.`
: In 'verbose' mode, the interpreter prints the sequence of steps taken during execution of the query.

| `#trace on.`
| `#trace off.`
: In 'trace' mode, upon every iteration of the EXEC function, a pictorial representation of the current state of the graph is outputted to a `.ps` file. See Section 3.3.1.

| `#clear.`
: Deletes all clauses in the program.

| `#use` *filename*`.`
: Inputs the contents of the file with the given *filename*. The file should comprise a sequence of *input* commands, separated by line breaks.

| `#mkgraph` *filename*`.`
: Outputs a pictorial representation of the current state of the graph to *filename*`.ps`. See Section 3.3.1.

query ::= `?` *predlist*`.`
: A query is provided as a list of predicates preceded by a question mark.

definition ::= *term*`.` | *term* `:-` *predlist*`.`
: A clause is either a fact or a rule.

predlist ::= *pred* | *pred*`,` *predlist*

pred ::= *term* | `!` | *atom* `#` *term*
: A predicate is either a *term*, a cut or a freshness problem.

term ::= `(`*termlist*`)` | *func term* | *const* | `@`*atom*
| *atom* `\` *term* | `[`*susp*`]` *term* | *var*
: We see here the syntax for a tuple, data term, a constant (syntactic sugar for a data term whose body is the empty tuple), an atom (prepended with the `@` symbol in order to distinguish it from a constant), an atom abstraction, a suspended permutation and a variable. 'TRAN' nodes are only used internally so have no concrete syntactic form.

termlist ::= *term* | *term*`,` *termlist*

susp ::= *atom atom* | *atom atom*`,` *susp*

func, const, atom ::= lowercase alphanumeric*
That is, a lowercase letter followed by zero or more alphanumeric characters.

var ::= uppercase alphanumeric*
That is, an uppercase letter followed by zero or more alphanumeric characters.

Source files may contain comments written using the ML notation, `(* ...*)`.

Upon execution of a query, the interpreter responds with "`No.`" and returns to the main prompt if the query has no solutions. Should a solution exist, with perhaps $\sigma = [X := \texttt{f}\ Y]$ and $\nabla = \{a \# X\}$, the interpreter responds as follows:

```
Yes.
X := f(Y)
a # X
more?
```

At this point, the user must type either '`;`' to invoke the $\langle bktrk \rangle$ function that searches for further solutions, or '`.`' to invoke the $\langle abort \rangle$ function that returns the user to the main prompt.

3.3.1 Pictorial tracing

The pictorial traces of the execution are generated using the *dot* program from the *Graphviz* suite of graph-drawing software[3]. The input to *dot* is a description of the nodes and edges of the required graph; *dot* then lays these out in a way that minimises edge lengths and node overlaps, before outputting the graph as a `.ps` image. Nominal Prolog includes a module that is responsible for generating *dot* source from the current state of the term graph. The source code of this module is included as Appendix A.

Figure 3.8 illustrates an example trace. All three diagrams were produced *entirely autonomously* by Nominal Prolog. The trace was generated by the following program:

```
#trace on.
equal(X,X).
?equal(lam(a\@a), lam(b\Y)).
```

which, upon execution, gave the following output:

[3] `http://www.graphviz.org`

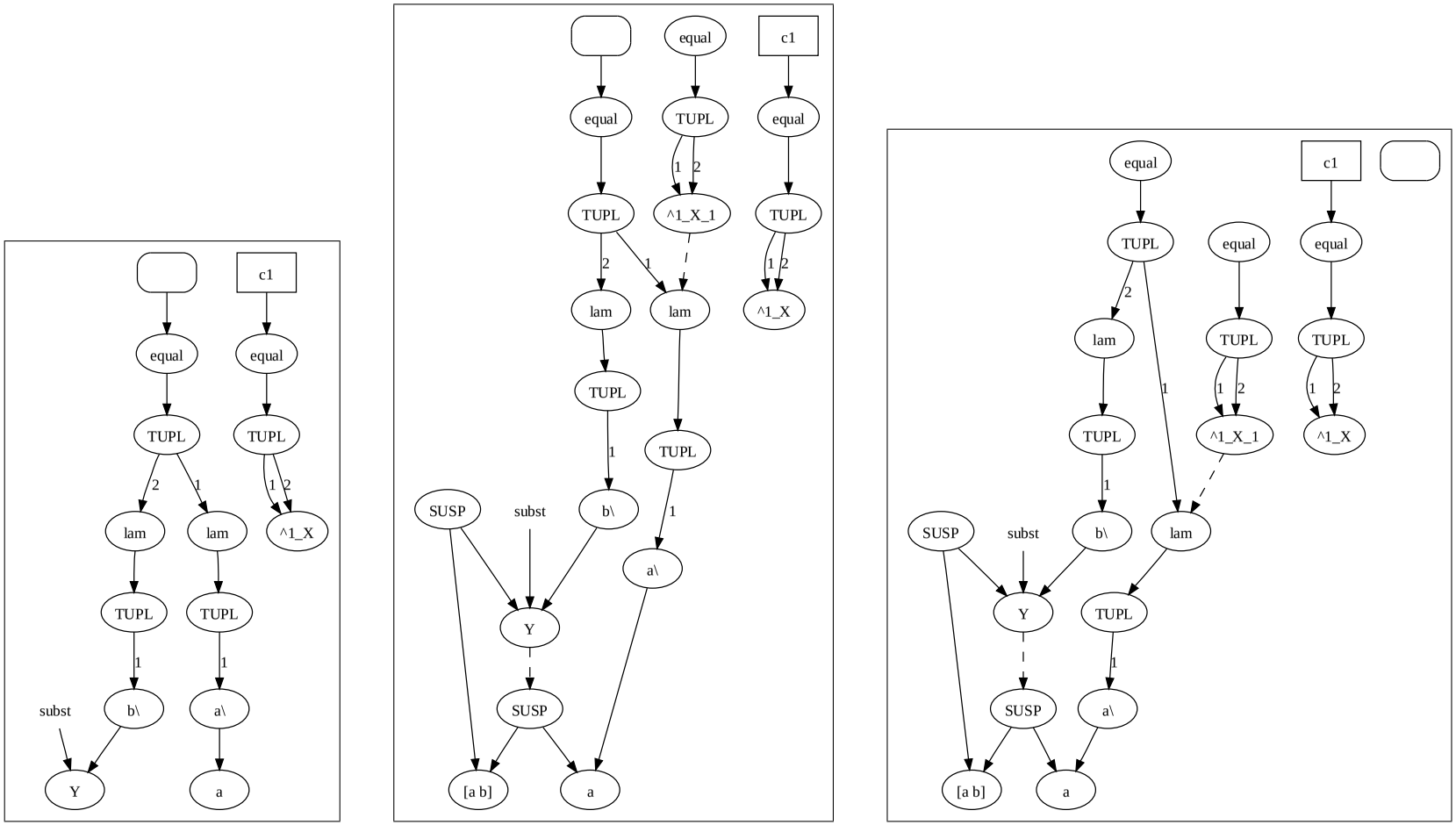


Figure 3.8: A sample execution trace. All three diagrams were produced *entirely autonomously* by Nominal Prolog. The single clause is identified as $c1$; a rounded rectangle points to the current goal; substitutions are drawn as dashed arrows; tuple elements are identified by numbered arrows; atoms and variables are made unique by prefixing them with a clause identifier and postfixing them with a number that is incremented each time the clause is copied; and the final solution can be extracted from the diagram by following the 'subst' pointer.

```
Yes.
Y:=@b
```

3.4 Testing

αProlog is distributed with a number of sample programs. An appropriate testing strategy, then, is to run each sample program on both αProlog and Nominal Prolog, and check that the results are equivalent. It is necessary first to manually rewrite each sample program into the syntax of Nominal Prolog. The thirteen sample programs, and the results of running each, are presented in the following table.

	Test program	Length of program	Result and comment
1.	A type-checker for the $\lambda\mu$-calculus	21 lines	**Passed**. Nominal Prolog sometimes generated more freshness constraints than αProlog, and sometimes fewer. This discrepancy is deemed not to be a cause for concern because those particular constraints were spurious anyway. The generation of such spurious freshness constraints seems to be an inherent limitation of this form of nominal logic programming, and is discussed further in section 4.3.1.
2.	Symbolic differentiation	12 lines	**Passed**. The conversion from αProlog syntax to Nominal Prolog syntax was hampered by Nominal Prolog's lack of support for numeric constants.
3.	Translation between de Bruijn and nominal formulations of λ-calculus terms	18 lines	**Passed**.
4.	Testing definitional equality	13 lines	**Passed**.
5.	An interpreter for the λ-calculus with explicit substitutions	22 lines	**Passed**.

	Test program	Length of program	Result and comment
6.	Rewriter and Skolemiser for first-order logic	91 lines	**Passed**.
7.	A type-checker for intuitionistic linear logic	43 lines	**Passed**. As in the first test, the results differed only in the number of spurious freshness constraints that were generated.
8.	An interpreter and type-checker for the λ-calculus	79 lines	**Failed**. Again there were some differences in the spurious freshness constraints. More worrying was the query for which αProlog outputted 'Yes' while Nominal Prolog outputted 'No'. It is thought that this is a result of the two systems' slightly different handling of atoms, which is discussed in Remark 3.4.1.
9.	A natural deduction calculus for pure nominal logic	53 lines	**Passed**.
10.	The Needham–Schroeder and Needham–Schroeder–Lowe cryptographic authentication protocols	66 lines	**Passed**. The conversion from αProlog syntax to Nominal Prolog syntax was hampered by Nominal Prolog's lack of support for 'definite clause grammar' notation.
11.	An interpreter for the π-calculus	61 lines	**Passed**. Once again, there were some differences in the spurious freshness constraints.
12.	Regular expressions and finite automata	72 lines	**Failed**. There were some discrepancies between several queries in this test. The reason is the same as that for test 8, and is discussed in Remark 3.4.1.

	Test program	Length of program	Result and comment
13.	Translation from the λ-calculus to an object calculus	54 lines	**Failed**. A small discrepancy arose owing to Nominal Prolog's syntax not quite being expressive enough to emulate the αProlog program. Consider the freshness problem $a \#^? t$: in Nominal Prolog the a must be an atom, but in αProlog it can be a variable that is substituted for an atom.

Remark 3.4.1. Nominal Prolog differs from αProlog in the way it makes atoms that appear in a clause unique. Given, for instance, the single clause $\mathtt{f}(a)$ and the query $?\mathtt{f}(a)$, Nominal Prolog would return 'No' while αProlog would return 'Yes'. Nominal Prolog represents the atom a in the clause internally as a_1 (the subscript indicating the clause number), because it takes the view that atoms should be local only to the clause in which they are defined. This makes it a distinct atom from that which appears in the query, so the query fails. αProlog represents both atoms as a, so the query succeeds. This discrepancy is considered to be largely a matter of taste.

Chapter 4

Evaluation

The evaluation is presented from three angles. We begin, in Section 4.1, with a 'mathematical' evaluation of the project, where we provide proofs of various properties of the implementation. Section 4.2 comprises a quantitative evaluation, and we conclude with qualitative comments in Section 4.3.

4.1 A 'mathematical' evaluation

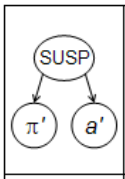
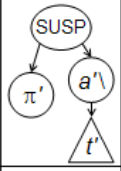
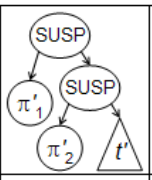
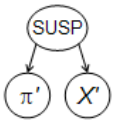
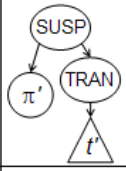
We begin by proving various properties about our implementation of nominal unification. The proofs of determinacy and completeness are original. The proof of termination is similar to that presented for a naïve implementation of nominal unification in [12], but has been significantly extended to make it suitable for our efficient implementation.

4.1.1 Determinacy and completeness of Nominal Unification

It is clear that the freshness transformations are deterministic (that is, there is never more than one suitable transformation for any list of freshness problems) and complete (that is, for any non-empty list of problems for which a solution exists, there is always at least one suitable transformation). The equational transformations also have these properties, though this is less trivial to see. In Table 4.1, we have listed each possible pairing of terms that can form an equational problem. Each row corresponds to a possible left-hand side, and each column corresponds to a possible right-hand side. Terms that have a suspended permutation have been expanded to the next level, which is why there are thirteen rows and columns rather than just seven. Each entry in the table is of one of the

lhs \ rhs	TUPL $t'_1 \cdots t'_n$	f′ t'	a′	a′\ t'	SUSP π′ TUPL $t'_1 \cdots t'_n$	SUSP π′ f′ t'
TRAN t	16	16	16	16	16	16
X	12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w
SUSP π TRAN t	17	17	17	17	17	17
SUSP π X	11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w
SUSP π_1 SUSP π_2 t	–	–	–	–	–	–
SUSP π a\ t	F	F	F	10	F	F
SUSP π a	F	F	8 $\pi(a) = a'$ F o/w	F	F	F
SUSP π f t	F	7 f = f' F o/w	F	F	F	7 f = f' F o/w
SUSP π TUPL $t_1 \cdots t_m$	6 $m = n$ F o/w	F	F	F	6 $m = n$ F o/w	
a\ t	F	F	F	4 $a = a'$ 5 o/w		
a	F	F	3 $a = a'$ F o/w			
f t	F	2 f = f' F o/w				
TUPL $t_1 \cdots t_m$	1 $m = n$ F o/w					

Table 4.1: A demonstration of both determinacy and completeness.

						
16	16	–	16	16	16	16
12 $X \notin$ rhs F o/w	12 $X \notin$ rhs F o/w	–	12 $X \neq X'$ 14 o/w	17	12 $X \neq X'$ 13 o/w	
17	17	–	17	17		
11 $X \notin$ rhs F o/w	11 $X \notin$ rhs F o/w	–	11 $X \neq X'$ 15 o/w			
–	–	–				
F	10					
9 $\pi(a) = \pi'(a')$ F o/w						

o/w = otherwise

Table 4.1: A demonstration of both determinacy and completeness, *continued*.

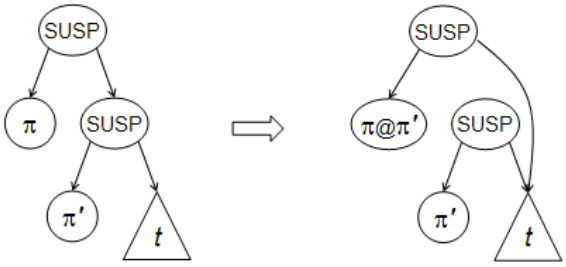


Figure 4.1: The normalisation of a suspension.

following forms:

- a number, which identifies the equational transformation to use in this situation;
- the letter 'F', which indicates that the left-hand side and right-hand side definitely do not unify, so the algorithm should fail; or
- a dash, –, which indicates that this situation can never arise (see Remark 4.1.1).

For determinacy it suffices to verify that at each place where the number of a particular transformation is given, no other transformation can be used instead. For completeness, it suffices to verify that at each place where 'F' is given, the left and right-hand sides definitely do not unify.

Remark 4.1.1. Table 4.1 assumes that the situation can never arise whereby one permutation is suspended outside another; that is, we never have terms of the form $\pi \cdot \pi' \cdot t$. This is because immediately such a term is created, it is normalised to the term $\pi@\pi' \cdot t$. The equational transformations that are liable to create such 'double suspensions' are 5, 6, 7, 10, 11 and 17, so after any of these transformations take place we perform the normalisation step illustrated in Figure 4.1.

4.1.2 A proof that UNIFY terminates

In this section we prove that there is no infinite sequence $P_1, P_2, ...$ such that for all $i \geq 1$, there exists some transformation $P_i \Longrightarrow P_{i+1}$. This allows us to deduce that unification will always terminate. We do not prove formally that the unification algorithm gives the *correct* result upon termination, although the justifications given in Sections 3.1.2 and 3.1.3 suggest fairly convincingly that the transformations are correct. The first part of this proof establishes that there are

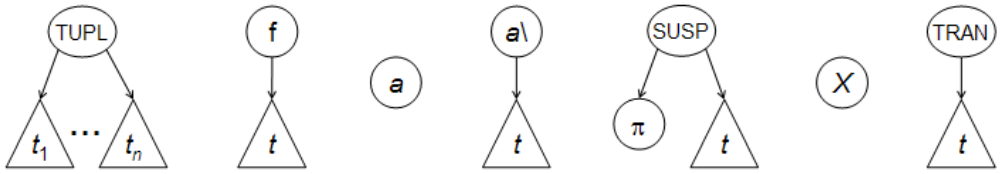


Figure 4.2: Size of a graph. The top row lists the seven types of graph node, and the bottom row defines the size of each. The size of the graph representing the term t is denoted $|t|$. Note that $|t| \geq 1$, and that even if some of the children of a tuple node are identical, each still contributes separately to the summation.

no infinite sequences of equational transformations; the second does the same for the freshness transformations.

For the first part, we define the size of a list of problems to be the pair of natural numbers, (n_1, n_2), where n_1 is the number of distinct variables in the list of problems and n_2 is defined by Equation 4.1, which uses the $|–|$ function defined in Figure 4.2. This definition of $|–|$ is rather surprising—why are there factors of two strewn seemingly at random? Still, after considering many alternatives, it is the only definition that was found to work.

$$n_2 \stackrel{\text{def}}{=} \sum_{(t \approx^? t') \in P} |t| + |t'|. \tag{4.1}$$

We show that every equational transformation strictly reduces (n_1, n_2), using a lexicographical ordering. Recall that for a lexicographical ordering $<_L$, we have:

$$(n_1, n_2) <_L (n_1', n_2') \Leftrightarrow n_1 < n_1' \vee (n_1 = n_1' \wedge n_2 < n_2').$$

Now, transformations 11 and 12 both remove a variable, thus decrementing n_1. All the other transformations leave n_1 unchanged, so we need to be sure that they reduce n_2. Table 4.2 demonstrates that this is indeed the case.

For the second part, we show (in Table 4.3) that every freshness transformation reduces the natural number n_3, defined by:

$$n_3 \stackrel{\text{def}}{=} \sum_{(a \#^? t) \in P} |t|.$$

Equational transformation	n_2 before	n_2 after
1	$1 + \sum_{i=1}^{n} \lvert t_i \rvert + 1 + \sum_{i=1}^{n} \lvert t'_i \rvert$	$\sum_{i=1}^{n} (\lvert t_i \rvert + \lvert t'_i \rvert)$
2	$1 + \lvert t \rvert + 1 + \lvert t' \rvert$	$\lvert t \rvert + \lvert t' \rvert$
3	2	0
4	$1 + 2\lvert t \rvert + 1 + 2\lvert t' \rvert$	$\lvert t \rvert + \lvert t' \rvert$
5	$1 + 2\lvert t \rvert + 1 + 2\lvert t' \rvert$	$\lvert t \rvert + 2\lvert t' \rvert$
6	$2(1 + \sum_{i=1}^{n} \lvert t_i \rvert) + \text{rhs}$	$1 + \sum_{i=1}^{n} 2\lvert t_i \rvert + \text{rhs}$
7	$2(1 + \lvert t \rvert) + \text{rhs}$	$1 + 2\lvert t \rvert + \text{rhs}$
8	3	0
9	4	0
10	$2(1 + 2\lvert t \rvert) + \text{rhs}$	$1 + 2(2\lvert t \rvert) + \text{rhs}$
13	2	0
14	3	0
15	4	0
16	$1 + \lvert t \rvert + \lvert t' \rvert$	$\lvert t \rvert + \lvert t' \rvert$
17	$2(1 + \lvert t \rvert) + \lvert t' \rvert$	$2\lvert t \rvert + \lvert t' \rvert$

Table 4.2: A demonstration that every equational transformation (except 11 and 12) reduces n_2. We write 'rhs' to signify the right-hand side of an equational problem if does not change. Note that the calculation of n_2 is incomplete: it considers only one equational problem at a time, despite n_2 being defined as the sum over *all* equational problems in the list of problems. Nonetheless, it is easy to verify that all the 'other' problems in the list remain unchanged by each transformation. This is because no transformations (save 11 and 12, which are not included in the table) modify existing nodes in the graph. Accordingly, we are simply omitting a constant from both columns in each row of the table, which clearly does not change the result.

Freshness transformation	n_3 before	n_3 after
1	$1 + \sum_{i=1}^{n} \lvert t_i \rvert$	$\sum_{i=1}^{n} \lvert t_i \rvert$
2	$1 + \lvert t \rvert$	$\lvert t \rvert$
3	1	0
4	$1 + 2\lvert t \rvert$	0
5	$1 + 2\lvert t \rvert$	$\lvert t \rvert$
6	$2\lvert t \rvert$	$\lvert t \rvert$
7	1	0
8	$1 + \lvert t \rvert$	$\lvert t \rvert$

Table 4.3: A demonstration that every freshness transformation reduces n_3.

4.2 A quantitative evaluation

We present the results of three tests that were designed to compare how quickly Nominal Prolog and αProlog execute programs of increasing complexity. In each case we define a variable n to represent the program complexity.

4.2.1 Test one

The family of programs used in this test were of the following form, for $n \geq 1$.

$$
\begin{array}{l}
\mathtt{f}_0(\mathtt{c}). \\
\mathtt{f}_1(X) :\!- \mathtt{f}_0(X), \mathtt{f}_0(X). \\
\mathtt{f}_2(X) :\!- \mathtt{f}_1(X), \mathtt{f}_1(X). \\
\quad \vdots \\
\mathtt{f}_n(X) :\!- \mathtt{f}_{n-1}(X), \mathtt{f}_{n-1}(X). \\
?\mathtt{f}_n(\mathtt{c}).
\end{array}
$$

The graph in Figure 4.3 compares how long Nominal Prolog and αProlog each take to execute the query $?\mathtt{f}_n(\mathtt{c})$. Execution times less than about 0.01 seconds were reported as zero, so cannot be plotted on the logarithmic scale. This is an artefact of using a system clock that advances in discrete time intervals.

Interestingly, at $n = 14$, αProlog raised an 'Out of stack space' exception. Nominal Prolog did not raise the same exception until it tried to execute the 18^{th} query. This would seem to mark a victory (at least in terms of memory efficiency) for the graph-based, term-sharing representation of terms of Nominal Prolog over the tree-based, non-sharing representation of αProlog.

This leaves us with few data points; fortunately, the points we have are very well-behaved. Clearly, αProlog executes convincingly quicker for every value of n: even when the data points are at their closest (at $n = 13$), αProlog is almost twice as quick. Nevertheless, the lines are unmistakably converging, and, given how well the data we have fits the drawn lines of best fit, it seems reasonable to suppose that for sufficiently large values of n, Nominal Prolog would execute faster. By extending the lines of best fit, we estimate the smallest 'sufficiently large' value of n to be 17.

This family of programs does not use atoms, atom abstractions or suspended permutations, so the reason for Nominal Prolog's (eventual) better performance most likely lies in its more efficient implementation of substitutions. There is a greater computational overhead when working with graphs (as Nominal Prolog does) rather than trees (as αProlog does), which is probably the main reason for αProlog's quicker execution when n is small.

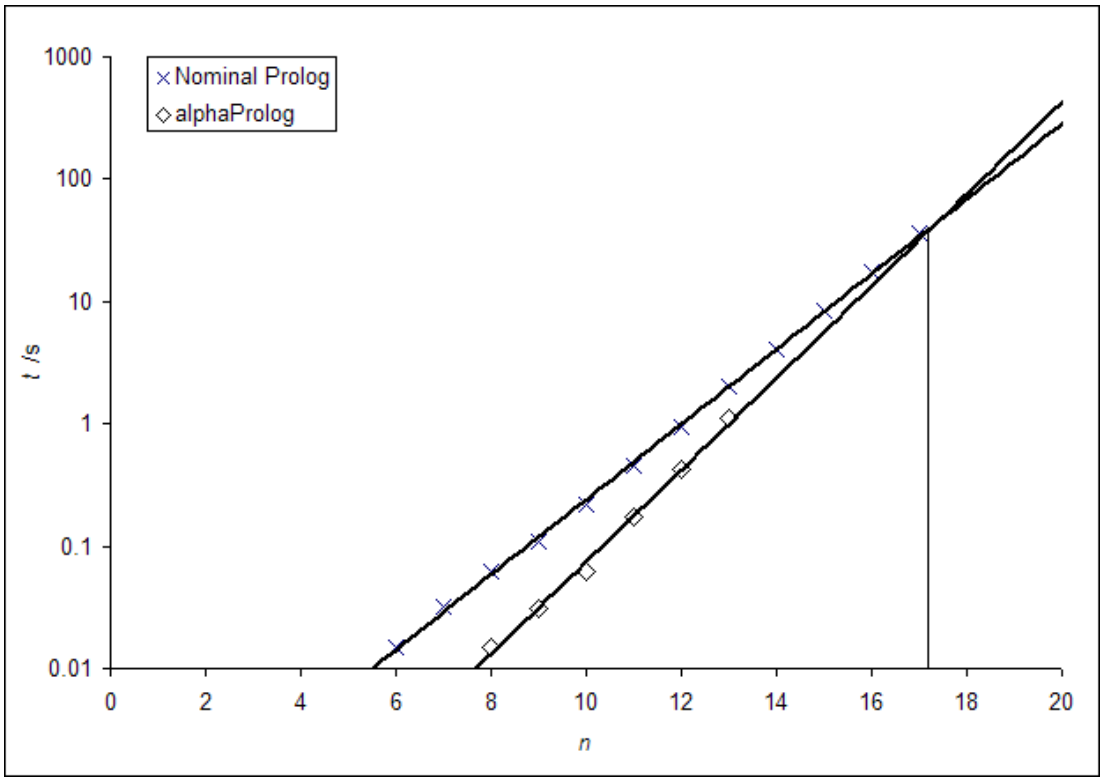


Figure 4.3: A graph of the time taken to execute queries of the form ?$\mathtt{f}_n(\mathtt{c})$ in terms of n. The lines of best fit are both exponential (despite appearing linear on the logarithmic scale) and were drawn by Microsoft Excel.

4.2.2 Test two: Naïve list reverse

Naïve list reverse is a classic benchmark for logic programming languages. The family of programs used in this test were of the following form, for $n \geq 1$.

$$
\begin{array}{l}
\texttt{reverse}\langle \texttt{nil}, \texttt{nil} \rangle. \\
\texttt{reverse}\langle \texttt{cons}\langle \mathit{Head}, \mathit{Tail} \rangle, R \rangle :- \\
\quad \texttt{reverse}\langle \mathit{Tail}, \mathit{RTail} \rangle, \\
\quad \texttt{append}\langle \mathit{RTail}, \texttt{cons}\langle \mathit{Head}, \texttt{nil} \rangle, R \rangle. \\
\texttt{append}\langle \texttt{nil}, L, L \rangle. \\
\texttt{append}\langle \texttt{cons}\langle \mathit{Head}, \mathit{Tail} \rangle, L, \texttt{cons}\langle \mathit{Head}, R \rangle \rangle :- \\
\quad \texttt{append}\langle \mathit{Tail}, L, R \rangle. \\
?\texttt{reverse}\langle \underbrace{\texttt{cons}\langle \texttt{c}, \texttt{cons}\langle \texttt{c}, \ldots \rangle \rangle}_{\text{a list of } n \text{ c's}}, \underbrace{\texttt{cons}\langle X, \texttt{cons}\langle X, \ldots \rangle \rangle}_{\text{a list of } n \ X\text{'s}} \rangle.
\end{array}
$$

The graph in Figure 4.4 compares how long Nominal Prolog and αProlog each take to execute the programs in the family defined above.

In this test, αProlog is the clear winner: it reverses the list faster than Nominal Prolog does. Moreover, the graphs show no signs of converging. It seems that the simpler, lower-overhead approach taken by αProlog still pays dividends in some cases.

For both graphs, the lowest degree of polynomial that fits well is a quartic, as plotted by Microsoft Excel in Figure 4.4. This suggests that both αProlog and Nominal Prolog perform the naïve list reverse algorithm in $O(n^4)$ time. Yet the naïve reverse algorithm is known have quadratic time complexity—from where have the two extra factors of n come? While *one* extra factor is justifiable by reference to the solution-finding algorithm presented in Figure 3.6 (we are making a copy of *rest_of_goal*, which is a list whose length is linear in n, for each recursive invocation of EXEC[1]), the reason for the presence of *two* extra factors remains unclear.

Of interest is the outlier at $n = 84$. One might assume this to be the result of a disruption caused by an ill-timed context switch, did it not appear in exactly the same position on three separate trials. The actual reason that Nominal Prolog takes unusally long to reverse a list of length 84 is to do with its implementation of directed acyclic graphs. A hash table stores mappings from terms to graph nodes; this lets us quickly find where a given term is located in the graph. The default size of this hash table is 100. The first time this proves inadequate is when we try to reverse a list of length 84—at this point, the hashtable is grown to twice its original size. It is this operation that causes the execution to take a little longer than expected.

[1] A better implementation might pass a reference to *rest_of_goal* rather than the value itself.

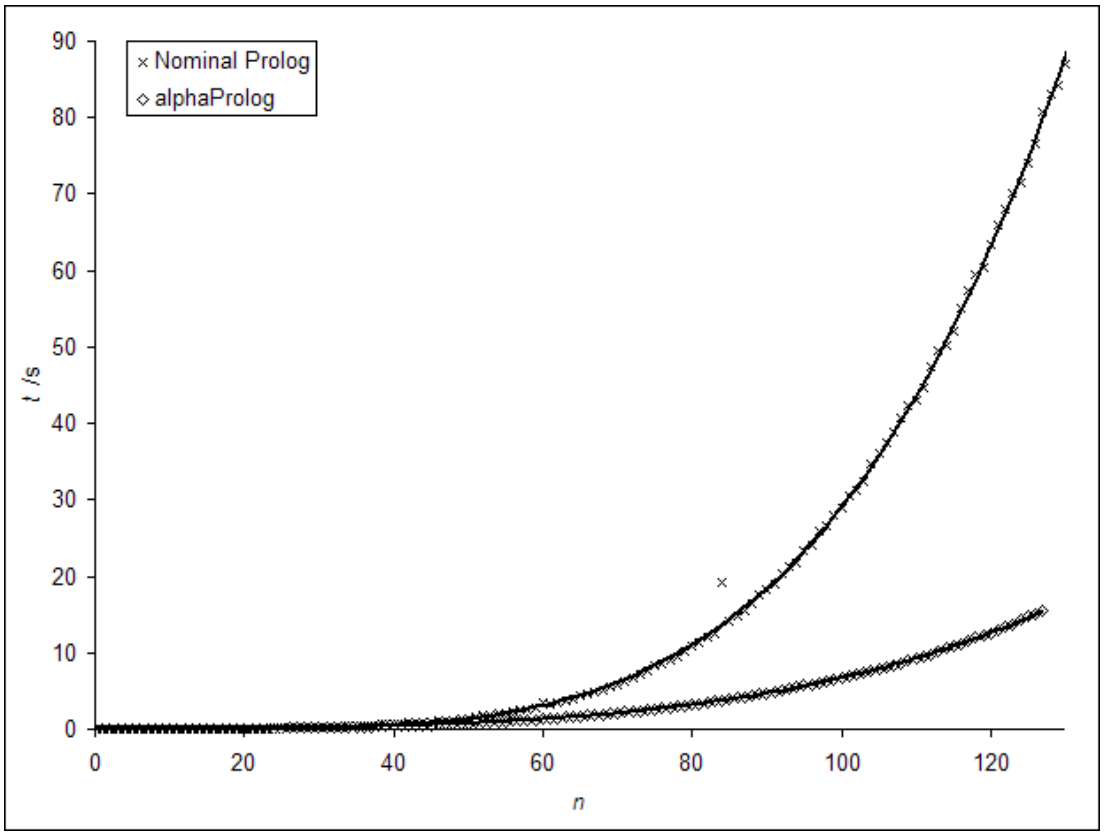


Figure 4.4: A graph of the time taken to reverse a list of length n. The lines of best fit are both quartics, and were drawn by Microsoft Excel.

4.2.3 Test three: Type inference

This final test analyses the performance of the 'nominal' features of Nominal Prolog, and makes use of the following program.

$\texttt{mem}\langle A, \texttt{cons}\langle A, T\rangle\rangle.$
$\texttt{mem}\langle A, \texttt{cons}\langle B, T\rangle\rangle :- \texttt{mem}\langle A, T\rangle.$
$\texttt{tc}\langle G, \texttt{var}\ X, A\rangle :- \texttt{mem}\langle\langle X, A\rangle, G\rangle.$
$\texttt{tc}\langle G, \texttt{app}\langle M, N\rangle, B\rangle :- \texttt{tc}\langle G, M, \texttt{arrow}\langle A, B\rangle\rangle,\ \ \texttt{tc}\langle G, N, A\rangle.$
$\texttt{tc}\langle G, \texttt{lam}(x\backslash M), \texttt{arrow}\langle A, B\rangle\rangle :- x\ \#^?\ G,\ \ \texttt{tc}\langle\texttt{cons}\langle\langle x, A\rangle, G\rangle, M, B\rangle.$

We use the simple type inferrer defined above to assign types to λ-calculus terms of the form tw^n, where tw is defined as $\lambda f.\lambda x.f(f\,x)$, and tw^3 is an abbreviation for $tw\ tw\ tw$. The type, for any n, is $(\alpha \to \alpha) \to \alpha \to \alpha$. The graph in Figure 4.5 compares how long Nominal Prolog and αProlog each take to infer this type, for increasing values of n.

The resounding victory in this test belongs to Nominal Prolog. The time taken by αProlog is exponential in n, while Nominal Prolog performs the same operation in polynomial time (probably cubic). There are a few reasons why this might be; the dominant factor is suspected to be Nominal Prolog's lazy evaluation of permutations, because this is the first test in which Nominal Prolog has really shone and is also the first test to make use of permutations.

4.3 A qualitative evaluation

The project generally proceeded very smoothly throughout development. The 'incremental and iterative' model of development described in the project proposal worked well. Despite being unfamiliar at the project outset, *OCaml* proved a suitable language in which to implement the interpreter. The main difficulty that was not foreseen in the planning phase was that of finding a way to pretty-print term graphs: a textual representation is satisfactory for trees but does not suffice for directed acyclic graphs. While the pictorial representation that was selected took a significant amount of time to develop, it proved invaluable during debugging. The backing-up process was unobtrusive, and back-ups were called upon a few times both to rectify mishaps and as a useful history of the evolution of the code, which was helpful in conjunction with the information in the project log when writing the progress report.

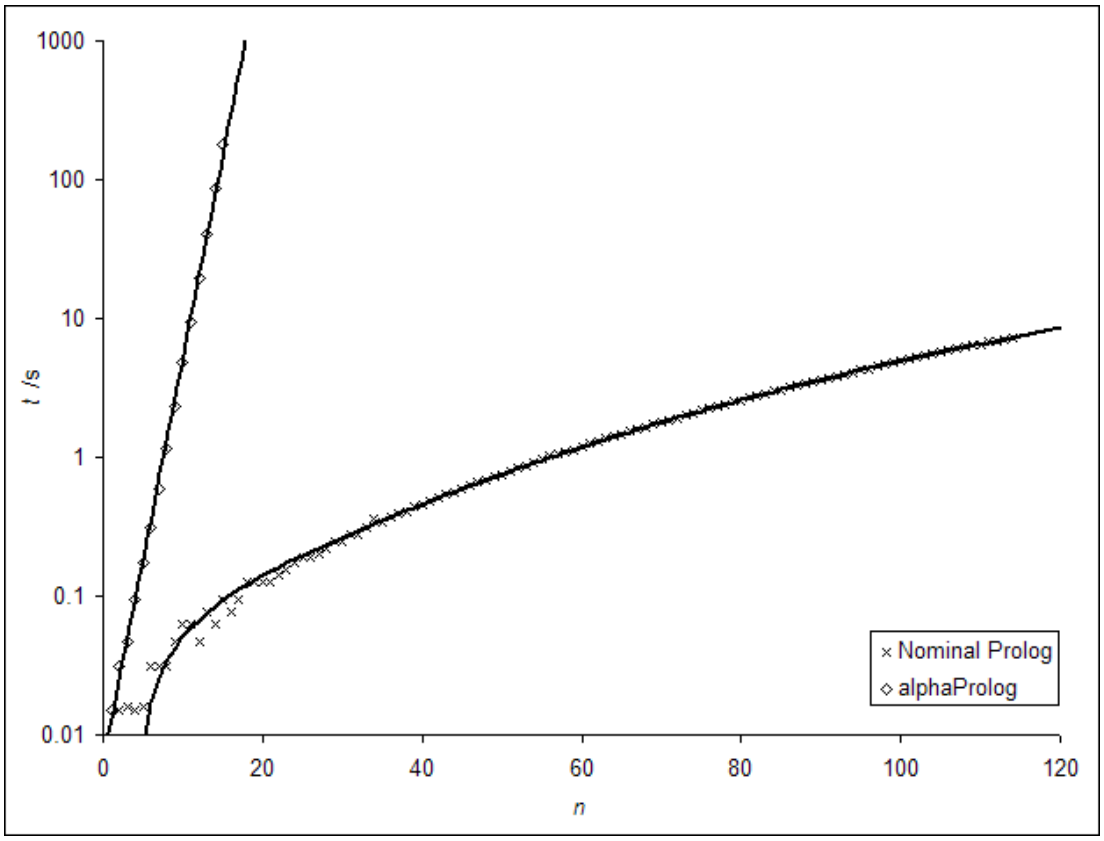


Figure 4.5: A graph of the time taken to infer the type of tw^n in terms of n. The line of best fit for αProlog is exponential while that for Nominal Prolog is a cubic.

4.3.1 Spurious freshness constraints

Both Nominal Prolog and αProlog are liable to output spurious freshness constraints. For instance, given the single clause $\texttt{f}(X) :- a \#^? X$ and the query $?\texttt{f}(X)$, both systems reply with 'Yes' plus the freshness constraint $a' \# X$, where a' is a freshened version of the atom a. This freshness constraint is spurious because a' is meaningless outside the clause in which it is used. Neither system reports the substitutions that are applied to internal unification variables, only those that affect the unification variables given in the query; surely then, we should only report the freshness constraints that are between atoms and unification variables given in the query. It transpires, however, that it is very hard to keep track of which atoms were given in the original query, and which have been generated during execution; to do so would require major changes to be made to the algorithm of nominal unification. For this reason, it seems that we must accept these spurious freshness constraints as a limitation of nominal logic programming.

4.3.2 Extensions

The project proposal (included at the end of this document) details two project extensions: the addition of a strong typing system and the caching of recently-unified terms. Neither extension was realised. It was decided that a typing system would add much complexity to the project while contributing little toward meeting the core aims. One unfortunate corollary of αProlog's strong type system is that its programs are cluttered with typing declarations, while Nominal Prolog's programs are refreshingly concise, and are hence better suited to a prototype language undergoing rapid development. Traditional Prolog does employ a minimal type system, which simply checks that a predicate is invoked with the right number of arguments. Nominal Prolog does not do this, but perhaps it should: such a system would have shortened a couple of debugging sessions that were caused by having the wrong number of arguments, and would not have required verbose typing declarations. The second extension, the caching of recently-unified terms, was soon discovered to be far more involved than first imagined, particularly because the process of unification incurs various side-effects. The idea merits further investigation, but was deemed to be beyond the scope of this project.

Chapter 5

Conclusions

5.1 Was the project a success?

According to the original project proposal, the project can be deemed a success upon delivering a working Nominal Prolog interpreter that...

1. ...accepts input as strings (i.e. concrete syntax) and pretty-prints results of queries.

   Status: **Achieved**. The interpreter accepts input directly from the command line or from a source file. The results of queries are clearly displayed. Moreover, the interpreter offers a pictorial tracing facility, whereby the evolution of the goal is outputted as a series of images.

2. ...uses the Urban-Pitts-Gabbay algorithm to perform nominal unification.

   Status: **Achieved**. The Urban-Pitts-Gabbay algorithm for nominal unification is at the heart of Nominal Prolog.

3. ...uses an efficient implementation of the Urban-Pitts-Gabbay algorithm (based on that of Calvès and Fernández).

   Status: **Achieved**. The most striking testimony to the efficiency of Nominal Prolog's implementation is the graph in Figure 4.5, which shows Nominal Prolog performing the same task in polynomial time that αProlog only managed in exponential time.

4. ...gives the same results as αProlog on a suite of test programs.

   Status: **Mostly achieved**. The remaining discrepancies were discussed in Section 3.4.

5.2 Alternative approaches

The project set out to extend an existing programming language with support for programming with names. We took the 'nominal' approach for representing names, but this is not the only way. We briefly consider a couple of alternative approaches.

de Bruijn indices

The de Bruijn representation of λ-calculus terms does away with names altogether[5]; it identifies an occurrence of a bound variable not by its name but the number of binders between that occurrence and its corresponding binder. For example, the term

$$\lambda x.\lambda y.\lambda z.xz(yz)$$

is written $\lambda(\lambda(\lambda(3\ 1(2\ 1))))$ using de Bruijn indices. One might think that the problem of unifying terms up to α-equivalence has already been solved by the use of de Bruijn indices, there being a bijection between de Bruijn terms and α-equivalence classes of λ-calculus terms. However, de Bruijn indices have a few drawbacks, the most obvious being that they are difficult for humans to work with—we much prefer to use names over numbers to refer to variables. Secondly, by giving a variable a name, we can refer to that variable again, outside the scope of the term in which it appears. Thirdly, the bijection breaks down when we allow terms to contain unification variables, X; the de Bruijn term $\lambda(\lambda(X))$ represents both $\lambda x.\lambda y.X$ and $\lambda x.\lambda x.X$ despite these terms not being α-equivalent[1].

Higher-order abstract syntax.

In HOAS[10], we can specify the syntax of the untyped λ-calculus as follows:

$$\begin{array}{rcl} t & ::= & \texttt{fn} : (t \to t) \to t \\ & | & \texttt{app} : t \times t \to t \end{array}$$

In the first-order representation of λ-calculus terms, the `fn` constructor is provided with both a binder and a body that may refer to that binder. In the higher-order approach, we simply provide the `fn` constructor with a function. As it happens, it remains convenient to express this function using names and name-bindings, but these are no longer part of the syntax. For instance, we can represent the term $\lambda x.\lambda y.x$ in HOAS as $\texttt{fn}(\lambda x.\texttt{fn}(\lambda y.\texttt{app}(x, y)))$. Much work has been done on

[1] Admittedly, there are ways to recover the bijection, such as the application of 'explicit substitutions' to the unification variables [7]. However, such approaches amass considerable complexity on top of an already unintuitive representation of terms.

'higher-order unification' that is based on such a representation of terms: see [6] for a survey. Sadly, unlike their nominal counterparts, higher-order unification problems may lack most general unifiers, or even be undecidable. A special case called 'higher-order pattern unification'[8] is decidable in linear time, and has been shown by Cheney[3] to be reducible to nominal unification.

5.3 Epilogue

The type inference task described in section 4.2.3 was later presented to both the OCaml and the Moscow ML toplevel interpreters. Both compilers processed the tw^6 case almost instantaneously, yet both ran out of memory before they could finish finding the type of tw^7. This is a rather surprising result, considering that Nominal Prolog accomplished the same task using its simplistic type inferrer in about fifty milliseconds. One wonders if they would both do better to implement their type inferrers using Nominal Prolog instead...

Bibliography

[1] Alex M. Andrew. The commercial use of PROLOG. *Kybernetes: The International Journal of Systems & Cybernetics*, 34(5):599–601, May 2005.

[2] Christophe Calvès and Maribel Fernández. Implementing nominal unification. *Electron. Notes Theor. Comput. Sci.*, 176(1):25–37, 2007.

[3] James Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*, pages 104–119, 2005.

[4] James Cheney and Christian Urban. Alpha-Prolog: a logic programming language with names, binding and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, Saint-Malo, France, September 2004. Springer-Verlag.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[6] Gilles Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, pages 1009–1062, 2001.

[7] Gilles Dowek, Therese Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 366, Washington, DC, USA, 1995. IEEE Computer Society.

[8] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[9] Lawrence C. Paulson. Logic and Proof. Lecture notes for the University of Cambridge Computer Science Tripos, 2007.

[10] F. Pfenning and C. Elliot. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, 1988.

[11] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[12] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

Appendix A

Sample source code

```
(** The code below comprises the module that is responsible for
the creation of pictorial representations of term dags. In
Section 3.3.1 we present some output from this module. *)

open Clause;;
open Term;;
open Unifier;;

(** The value of [ctr] is used to give unique ids to nodes in the
graphviz output. *)
let ctr = ref 0;;
let next_int() = (incr ctr; !ctr);;

(** An association list of (term reference [ref t], integer [i])
pairs, where "term[i]" is the unique id of the graphviz node
representing term [t]. *)
let keynums_term : (term ref * int) list ref
 = ref []
;;

(** An association list of (perm reference [ref p], integer [i])
pairs, where "perm[i]" is the unique id of the graphviz node
representing perm [p]. *)
let keynums_perm : (Perm.perm ref * int) list ref
 = ref []
;;

(** see below **)
let findnode (keynums : ('a ref * int) list ref)
```

```
  (k:'a ref) : string
 = string_of_int(
   try List.assq k (!keynums)
   with Not_found ->
      let i = next_int() in
      (keynums := ((k,i)::(!keynums));
      i));;

(** Finds the unique id of the given term reference in
[keynums_term], if it exists. Otherwise, makes, stores and returns
a new id. *)
let findnode_term (kt : term ref) : string
 = "term" ^ (findnode keynums_term kt)
;;

(** Finds the unique id of the given perm reference in
[keynums_perm], if it exists. Otherwise, makes, stores and returns
a new id. *)
let findnode_perm (kp: Perm.perm ref) : string
 = "perm" ^ (findnode keynums_perm kp)
;;

let gen_term (kt:term ref) : string
 = let termk = (findnode_term kt) in
   match (!kt) with
   | TUPL(kus) ->
      let f ku i = termk ^ " -> " ^ (findnode_term ku)
         ^ " [label = " ^ (string_of_int i) ^ "]; \n"
      in
      termk ^ " [label = TUPL]; \n"
      ^ (Mylist.implode2 0 "" f kus)
   | DATA(c,ku) ->
     termk ^ " [label = " ^ c ^ "]; \n"
      ^ termk ^ " -> " ^ (findnode_term ku) ^ "; \n"
   | ATOM(a) ->
     termk ^ " [label = \"" ^ a ^ "\"]; \n"
   | ABST(a,ku) ->
     termk ^ " [label = \"" ^ a ^ "\\\\\"]; \n"
      ^ termk ^ " -> " ^ (findnode_term ku) ^ ";\n"
   | SUSP(kp,ku) ->
     termk ^ " [label = SUSP]; \n"
      ^ termk ^ " -> " ^ (findnode_perm kp) ^ "; \n"
      ^ termk ^ " -> " ^ (findnode_term ku) ^ "; \n"
```

```
   | VAR(x) ->
     termk ^ " [label = \"" ^ x ^ "\"]; \n"
   | TRAN(x,ku) ->
     termk ^ " [label = \"" ^ x ^ "\"]; \n"
      ^ termk ^ " -> " ^ (findnode_term ku)
      ^ " [style = dashed]; \n"
;;

(** Generates the graphviz output for all the terms in the given
dag. *)
let gen_terms (): string
 = let f _ kt prev = (gen_term kt) ^ prev
   in Dag.fold_term f ""
;;

(** Generates the graphviz output for all the perms in the given
dag. *)
let gen_perms () : string
 = let p_str p = "\"" ^ (Perm.perm_toStr p) ^ "\"" in
   let f p k prev = (findnode_perm k)
      ^ " [label = " ^ p_str p ^ "]; \n" ^ prev
   in Dag.fold_perm f ""
;;

(** Generates the graphviz output for the given clause. *)
let gen_clause (c:clause) : string
 = let nodeid = "\"clause" ^ (string_of_int c.clause_id) ^ "\"" in
   let label =
      let f p i =
         "<t" ^ (string_of_int i) ^ ">"
         ^ (match p with
           | CUT -> " !"
           | TERM _ -> ""
           | FR_PRED(a,_) -> " " ^ a ^ "#")
      in
      let rest = Mylist.implode2 0 " | " f c.neglits in
      let first_bit = "<t0> c" ^ (string_of_int c.clause_id) in
      let mid_bit = if rest = "" then "" else " | " in
      first_bit ^ mid_bit ^ rest
   in
   let edges =
      let f p i =
         match p with
```

```
         | CUT -> ""
         | FR_PRED(_,kt)
         | TERM(kt) ->
            nodeid ^ ":t" ^ (string_of_int i)
            ^ " -> " ^ (findnode_term kt) ^ "; \n"
      in
      let rest = Mylist.implode2 0 "" f c.neglits in
      let first_bit = nodeid ^ ":t0"
            ^ " -> " ^ (findnode_term c.poslit) ^ "; \n" in
      first_bit ^ rest
   in
   nodeid ^ " [\n"
   ^ "label = \"" ^ label ^ "\" \n"
   ^ "shape = \"record\" \n"
   ^ "]; \n"
   ^ edges
;;

(** Generates the graphviz output for the given program. *)
let gen_prog (prog:program) : string
 = Hashtbl.fold (fun _ c e -> (gen_clause c) ^ e) prog ""
;;

(** Generates the graphviz output for the given goal. *)
let gen_goal (goal : goal) : string
 = let label =
      let f p i =
         "<t" ^ (string_of_int i) ^ ">"
         ^ (match p with
           | CUT -> " !"
           | TERM _ -> ""
           | FR_PRED(a,_) -> " " ^ a ^ "#")
      in
      Mylist.implode2 0 " | " f goal
   in
   let edges =
      let f p i =
         match p with
         | CUT -> ""
         | FR_PRED(_,kt)
         | TERM(kt) ->
            "goal:t" ^ (string_of_int i)
            ^ " -> " ^ (findnode_term kt) ^ "; \n"
```

```
      in
      Mylist.implode2 0 "" f goal
   in
   "\"goal\" [\n"
   ^ "label = \"" ^ label ^ "\" \n"
   ^ "shape = \"Mrecord\" \n"
   ^ "]; \n"
   ^ edges
;;

(** Generates the graphviz output for the given list of
problems. *)
let gen_probs (probs:prob list) : string
 = let rec gen_prob_nodes (probs:prob list) (len:int) : string
    = match probs with
      | [] -> ""
      | (EQ(kt1, kt2)) :: probs ->
         let problen = "prob" ^ (string_of_int len) in
         problen ^ " [label = \"=?\"]; \n"
         ^ (gen_prob_nodes probs (len-1))
      | (FR(a,kt)) :: probs ->
         let problen = "prob" ^ (string_of_int len) in
         problen ^ " [label = \"" ^ a ^"#?\"]; \n"
         ^ (gen_prob_nodes probs (len-1))
   in
   let rec gen_prob_edges (probs:prob list) (len:int) : string
    = match probs with
      | [] -> ""
      | (EQ(kt1, kt2)) :: probs ->
         let problen = "prob" ^ (string_of_int len) in
         problen ^ " -> " ^ (findnode_term kt1) ^ "; \n"
         ^ problen ^ " -> " ^ (findnode_term kt2) ^ "; \n"
         ^ (gen_prob_edges probs (len-1))
      | (FR(a,kt)) :: probs ->
         let problen = "prob" ^ (string_of_int len) in
         problen ^ " -> " ^ (findnode_term kt) ^ "; \n"
         ^ (gen_prob_edges probs (len-1))
   in
   let num_of_probs = List.length probs in
   "{\n"
   ^ "rank = source; \n"
   ^ (gen_prob_nodes probs num_of_probs)
   ^ "}\n"
```

```
   ^ (gen_prob_edges probs num_of_probs)
;;

let gen_subst (s:Subst.subst) : string
 = let f prev kt =
      let termk = findnode_term kt in
      "subst -> " ^ termk ^ "; \n" ^ prev
   in
   "subst[shape=plaintext];\n"
   ^ (List.fold_left f "" s)
;;

(** Generates the entire graphviz output. *)
let gen_all (s:Subst.subst) (probs:prob list)
  (prog:program) (goal:goal) : string
 = "digraph G { \n"
   ^ (gen_subst s)
   ^ (gen_probs probs)
   ^ (gen_prog prog)
   ^ (gen_goal goal)
   ^ (gen_terms ())
   ^ (gen_perms ())
   ^ "}"
;;

(** Resets the node identifiers. *)
let reset() =
   ctr := 0;
   keynums_term := [];
   keynums_perm := []
;;

(** [produce_graph d probs prog goal out_filename] makes a graph
of [d], [probs], [prog] and [goal], and stores it in
"graphs/[out_filename].ps". *)
let produce_graph (s:Subst.subst) (probs:prob list)
(prog:program) (goal:goal) (out_filename:string) : unit
 = reset();
   let dotcode = gen_all s probs prog goal in
   print_string("Putting dot code into " ^ out_filename ^
      ".tmp \n");
   let tmp_filename = "graphs\\" ^ out_filename ^ ".tmp" in
   let tmp_file = open_out tmp_filename in
```

```
    output_string tmp_file dotcode;
    close_out tmp_file;
    let filetype = "ps" in
    let mode = "ps2" in
    print_endline("Making " ^ out_filename ^ "." ^ filetype ^ ".");
    let _ = Sys.command("dot \"" ^ tmp_filename ^ "\" -T" ^ mode ^
        " > \"graphs\\" ^ out_filename ^ "." ^ filetype ^ "\"") in
    print_endline("Finished.")
;;
```

John Wickerson
Churchill College
JPW48

Part II Project Proposal

Nominal Prolog

19th October 2007

Project Originator: Prof. A. M. Pitts

Resources Required: See attached Project Resource Form

Project Supervisor: Prof. A. M. Pitts

Signature:

Director of Studies: Dr. J. K. Fawcett

Signature:

Overseers: Prof. A. Mycroft and Dr. F. M. Stajano

Signatures:

Introduction and Description of the Work

The main aim of the project is to design and implement an interpreter for *Nominal Prolog*, a Prolog-like language that, by performing unification up to α-equivalence, provides support for programming with names and name-binders.

The λ-calculus provides a motivating example; consider the following definition of capture-avoiding substitution:

$$\begin{array}{rcll} y[e/x] & = & \left\{ \begin{array}{ll} e & x = y \\ y & x \neq y \end{array} \right. & \\ (e_1\ e_2)[e/x] & = & e_1[e/x]\ e_2[e/x] & \\ (\lambda y.e_1)[e/x] & = & \lambda y.(e_1[e/x]) & y \notin FV(e) \cup \{x\} \end{array}$$

We need not consider the case where the condition on the third rule is not satisfied, because we can always rename the variable y to a new name, z say, such that the condition does hold. Although this function can be defined in Prolog, the renaming process is problematic because the choice of z is so open-ended. In solving the problem we are forced to write code that is either inefficient or obfuscated.

Meanwhile, in Nominal Prolog, the following definition of capture-avoiding substitution is correct, succinct and efficient.

```
subst(var(x), E, x, E).
subst(var(y), E, x, var(y)).
subst(app(E1,E2), E, x, app(E3,E4)) :-
   subst(E1, E, x, E3), subst(E2, E, x, E4).
subst(lam(y\E1), E, x, lam(y\E2)) :- y # E, subst(E1, E, x, E2).
```

The predicate is defined such that `subst(`e_1`, `e_2`, `x`, `R`)` is true if and only if $R = e_1[e_2/x]$. This example demonstrates two features of Nominal Prolog. Firstly, terms of the form $x\backslash t$ (where x is a variable and t is an arbitrary term) have the meaning of binding free occurrences of x in t. Secondly, the predicate $x\#t$ (read: "x is fresh for t") is true if and only if there are no free occurrences of x in t. These two features are made possible by the use of *nominal unification*, a method for solving both unification problems ("Can two terms be made α-equivalent?") and freshness problems ("Is x fresh for t?"). An algorithm for performing nominal unification has been devised by Urban, Pitts and Gabbay[6].

Nominal Prolog will be similar to an existing system called αProlog[2], designed by James Cheney. αProlog uses the Urban-Pitts-Gabbay algorithm to perform nominal unification, however the implementation of the algorithm lacks efficiency.[1] An efficient implementation

[1] Indeed, Cheney remarks in the source code: "`TODO: Use a faster version of unification such as using reference cells`".

has since been invented by Calvès and Fernández[1]. Nominal Prolog will base its implementation of the Urban-Pitts-Gabbay algorithm on that of Calvès and Fernández, and by doing so it has the potential to be more efficient (and hence more useful) than αProlog. The existence of a similar system will be useful during the evaluation phase: my interpreter can be compared against Cheney's interpreter for both efficiency and correctness.

Since the core of the interpreter is a resolution-based theorem prover, a functional programming language is the most suitable programming language to use for its implementation. I have selected OCaml[4], the language in which αProlog was written. OCaml is an ML variant that is both more widely used and better supported than Standard ML.

Resources Required

I will use the machine in my room for development, which has the following specification:

- 3.4GHz Intel Pentium 4 processor
- 1.0GB RAM
- 240GB hard disk space
- Microsoft Windows XP Home SP2

I will use the PWF for backup and for running overnight tests.

Back-up Policy

A CVS repository will be set up on my machine to store all files related to my project. Twice daily, a snapshot of the repository will be copied to both my PWF space and my external hard drive. A 'Scheduled Task' will be configured such that Windows does this automatically. Weekly backups will be made to CD using my machine's CD writer.

Starting Point

Although I have never programmed in OCaml, it should not prove too challenging to learn, as the differences it has with ML are almost all syntactic rather than semantic (at least for the language features that I will need).

In preparation for the project I have spent some time in the last few weeks studying research papers related to nominal unification. I am familiar with the process of resolution-based theorem-proving from the Part IB course *Logic and Proof*. I have built an interpreter for

a limited functional language as part of my work for the Part IB course *Semantics*, but I have not experimented with an interpreter for a logic programming langugage before.

I will be using the OCaml versions of *Lex* and *Yacc* to generate a lexer and a parser for my language. Part of the timetable has been set aside for gaining familiarity with these tools, as I have not used them before.

Substance and Structure of the Project

Nominal unification is part of a challenging and new area of Computer Science based on *nominal logic*[5], so a significant part of the initial phase of this project will comprise reading in and around the topic so that I am able to implement the algorithms involved.

I will observe an incremental and iterative model of development. By 'incremental', I mean that I will first identify and build the smallest subset of the proposed system that is able to stand alone, and then proceed to continually augment this small system with one new feature at a time. By 'iterative', I mean that when building a particular feature, I will first do so in a very naïve way, but over time I will revisit the feature and improve it.

With this model in mind, the first phase of development will comprise designing and building a naïve interpreter for a limited version of ordinary Prolog (that is, without nominal unification). The interpreter will operate on the abstract syntax trees of nominal terms, but it will ignore name-binding and alpha-conversion at this stage. Simple pretty-printing will be used to display output, and the implementation of the resolution-based proof search will be naïve. Later on, the use of 'success continuations'[3] in resolution-based proof search may give a more intuitive and efficient implementation. Work will be done to research this technique and investigate how it may be applied to this project.

The second phase will comprise implementing the Urban-Pitts-Gabbay algorithm for nominal unification. This will be done in an iterative way: first a naïve implementation of the algorithm will be produced, and then this will be refined to become an efficient implementation. The third phase will involve integrating the nominal unification algorithm into the Prolog interpreter built in the first phase. It is anticipated that this will be quite challenging for the following reason: the output of the ordinary unifier is a substitution, whereas the output of the nominal unifier is a substitution plus a set of freshness constraints. Considerable changes may have to be made to the interpreter to accommodate this.

The next phase of the project development will involve deciding upon a suitable concrete syntax for the language and generating a parser to accept this syntax. This stage has intentionally been left until this stage of the project so that the concrete syntax can be chosen to reflect the implementation of the underlying interpreter.

There are various ways in which the interpreter can be improved. One way to improve its efficiency is to keep a cache of recently unified terms, thus taking advantage of temporal

locality of reference (having unified a pair of terms, we are likely to ask to unify them again shortly afterwards). A strong typing system may be added (as used in αProlog) but this is currently deemed to be an optional extension.

How can we evaluate the project? The correctness of the interpreter can be justified by running test programs and comparing the output to that of αProlog. The efficiency of the interpreter can be assessed by running equivalent programs on both this interpreter and the αProlog interpreter, and plotting some measure of program complexity against execution time. It will be informative to compare the execution speed of the interpreter using the naïve implementation of the Urban-Pitts-Gabbay algorithm and the efficient implementation.

Success Criteria

The project can be deemed a success upon delivering a working Nominal Prolog interpreter that:

- accepts input as strings (i.e. concrete syntax) and pretty-prints results of queries,
- uses the Urban-Pitts-Gabbay algorithm to perform nominal unification,
- uses an efficient implementation of the Urban-Pitts-Gabbay algorithm (based on that of Calvès and Fernández), and
- gives the same results as αProlog on a suite of test programs.

Timetable and Milestones

Weeks 1 – 2 **Fri 19th Oct – Fri 2nd Nov**

Install OCaml. Set up CVS repository and practise the backing-up procedure. Install αProlog and experiment with some test programs. Begin study of relevant theory. Write a basic Prolog interpreter.

Deliverable:

- working prototype of basic Prolog interpreter (using ordinary unification)

Weeks 3 – 4 **Fri 2nd Nov – Fri 16th Nov**

Produce naïve version of Urban-Pitts-Gabbay algorithm. Produce efficient version.

Deliverable:

- two working nominal unifiers: one naïve version, one efficient version

Weeks 5 – 6 **Fri 16th Nov – Fri 30th Nov**

Incorporate Urban-Pitts-Gabbay algorithm into basic Prolog interpreter, giving first version of Nominal Prolog.

Deliverable:

- working first version of Nominal Prolog (with input as abstract syntax)

Weeks 7 – 12 (Christmas Vacation) **Fri 30th Nov – Tue 15th Jan**

Install the OCaml versions of *Yacc* and *Lex* and practise using them. Decide upon concrete syntax, and use *Yacc* and *Lex* to generate a lexer and a parser for Nominal Prolog. Research 'success continuations' and investigate how they can be added to the interpreter. Build harness for measuring execution speed of αProlog. Build harness for measuring execution speed of Nominal Prolog. Optional: add a strong typing system.

Deliverables:

- working version of Nominal Prolog (with input as concrete syntax)
- working test harness for αProlog
- working test harness for Nominal Prolog

Weeks 13 – 14 **Tue 15th Jan – Fri 25th Jan**

Use test harnesses to run overnight tests on PWF of αProlog and Nominal Prolog. Write progress report.

Complete progress report by Fri 25th Jan.

Deadline for submission of progress report: Fri 1st Feb.

Weeks 15 – 21 **Fri 25th Jan – Fri 14th Mar**

Write first draft of dissertation.

Submit to supervisor by Fri 7th Mar.

Write second draft of dissertation.

Submit to supervisor by Fri 14th Mar (end of Lent term).

Weeks 22 – 26 (Easter vacation) **Fri 14th Mar – Tue 22nd Apr**

Planned contingency. (Revise for examinations.)

Weeks 27 – 30 **Tue 22nd Apr – Fri 16th May**

Finalise dissertation. Get dissertation printed and bound.

Submit dissertation by Fri 2nd May.

Deadline for submission of dissertation: Fri 16th May.

References

[1] Christophe Calvès and Maribel Fernández. Implementing nominal unification. *Electron. Notes Theor. Comput. Sci.*, 176(1):25–37, 2007.

[2] James Cheney and Christian Urban. Alpha-prolog: A logic programming language with names, binding and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, Saint-Malo, France, September 2004. Springer-Verlag.

[3] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[4] Xavier Leroy et al. Objective caml 3.10.0. `http://caml.inria.fr/ocaml`, May 2007.

[5] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[6] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.