Avoiding Equivariance in Alpha-Prolog

Christian Urban[1] and James Cheney[2]

[1] Ludwig-Maximilians-University Munich (urban@mathematik.uni-muenchen.de)
[2] Edinburgh University (jcheney@inf.ed.ac.uk)

Abstract. αProlog is a logic programming language which is well-suited for rapid prototyping of type systems and operational semantics of typed λ-calculi and many other languages involving bound names. In αProlog, the nominal unification algorithm of Urban, Pitts and Gabbay is used instead of first-order unification. However, although αProlog can be viewed as Horn-clause logic programming in Pitts' *nominal logic*, proof search using nominal unification is incomplete in nominal logic. Because of nominal logic's *equivariance principle*, complete proof search would require solving NP-hard *equivariant unification* problems. Nevertheless, the αProlog programs we studied run correctly without equivariant unification. In this paper, we give several examples of αProlog programs that do not require equivariant unification, develop a test for identifying such programs, and prove the correctness of this test via a proof-theoretic argument.

1 Introduction

Logic programming is particularly suited for implementing inference rules defining relations over terms. Many interesting examples of such inference rules, however, involve terms with binders and α-equivalence, for which Prolog, for example, provides little assistance. In [3] we presented αProlog, which is designed to simplify programming with binders. For instance, the operation of capture-avoiding substitution for λ-terms can be implemented in αProlog as follows:

```
id(X,X).
subst(var(X),X,T,T).
subst(var(X),Y,T,var(X))      :- not(id(X,Y)).
subst(app(M,N),X,T,app(M',N')):- subst(M,X,T,M'), subst(N,X,T,N').
subst(lam(a.M),X,T,lam(a.M')) :- a # T, a # X, subst(M,X,T,M').
```

where the terms `var(X)`, `app(M,N)` and `lam(a.M)` encode variables, applications and λ-abstractions. The predicate `subst(E,X,T,E')` defined by the clauses holds only if `E'` contains the result of the usual capture-avoiding substitution `E[X:=T]` in the λ-calculus.

Two features of αProlog are immediately visible to the user. First, the term language includes the term-constructor $-.-$ for forming abstractions, which are used to encode binding. Second, αProlog has a freshness-predicate, written as $-\#-$, built into the language; this predicate ensures that a name does not occur freely in a term (by a name we mean lower-case symbols, for instance `a` in the expression `lam(a.M)`). In this `subst`-program, the freshness-predicate is used to make sure that no variable capture occurs inside the term being substituted.

To illustrate how the `subst`-program calculates the result of the capture-avoiding substitution $(\lambda b.a\ b)[a := b]$, we consider the query:

$$\texttt{subst(lam(b.app(var(a),var(b)),a,var(b),R)} \tag{1}$$

To solve this query, αProlog unifies it with the head of the fourth `subst`-clause

$$\texttt{subst(lam(a}_1\texttt{.M}_1\texttt{),X}_1\texttt{,T}_1\texttt{,lam(a}_1\texttt{.M'}_1\texttt{)):- a}_1\texttt{\#T}_1\texttt{,a}_1\texttt{\#X}_1\texttt{,subst(M}_1\texttt{,X}_1\texttt{,T}_1\texttt{,M'}_1\texttt{).}$$

where, as in Prolog, the variables `M`, `X`, `T` and `M'` have been replaced with fresh variables (indicated by the subscript), and also the name `a` has been freshened (we shall return to the difference between variables and names later). The unifier that αProlog calculates is $\texttt{app(var(a),var(a}_1\texttt{))}$ for $\texttt{M}_1$, `a` for $\texttt{X}_1$, `var(b)` for $\texttt{T}_1$ and $\texttt{lam(a}_1\texttt{.M'}_1\texttt{)}$ for `R`. Next, αProlog checks that the freshness-predicates $\texttt{a}_1\texttt{\#}$ `var(b)` and $\texttt{a}_1\texttt{\#}$ `a` hold, and continues unifying the new query $\texttt{subst(app(var(a),var(a}_1\texttt{)),a,var(b),M'}_1\texttt{)}$ with the third `subst`-clause. Then it uses the first and second `subst`-clause and after they succeed, αProlog returns $\texttt{lam(a}_1\texttt{.app(var(b),var(a}_1\texttt{)))}$ as the answer for `R`.

Another example, which illustrates how easily inference rules can be implemented in αProlog, is the following program

```
mem(X,[X|T]).
mem(X,[Y|T]) :- mem(X,T).
type(Gamma,var(X),T)   :- mem((X,T),Gamma).
type(Gamma,app(M,N),T):- type(Gamma,M,arr(S,T)), type(Gamma,N,S).
type(Gamma,lam(x.M),arr(S,T)):-x#Gamma, type([(x,S)|Gamma],M,T).
```

implementing the usual inference rules for inferring the types of λ-terms.

$$\frac{x : T \in \Gamma}{\Gamma \triangleright x : T}\ var \qquad \frac{\Gamma \triangleright M : S \to T \quad \Gamma \triangleright N : S}{\Gamma \triangleright MN : T}\ app \qquad \frac{x : S, \Gamma \triangleright M : T \quad (x \notin FV(\Gamma))}{\Gamma \triangleright \lambda x.M : S \to T}\ lam$$

Note that, in contrast to for example λProlog, abstractions in αProlog bind a concrete name which is *not* restricted to the scope of the abstractions. Therefore it is possible in αProlog to use a name of a binder in the body of the clause, for instance to append `(x,S)` to the context `Gamma` in the third `type`-clause. The implicit side-condition in the rule lam requiring that Γ has no type-assignment for x is implemented in αProlog by the freshness-predicate `x#Gamma`.

We have implemented a large number of such λ-calculus examples, including type systems and operational semantics for System F, $\lambda\mu$ and linear λ-calculi. Our experience from these examples suggests that the combination of concrete names in abstractions and the freshness-predicate is very useful for programming with binders. One question, however, might arise: what are the advantages of αProlog relative to, for example, λProlog [7], which has both α-equivalence and capture-avoiding substitution built-in and the typing rules can be correctly implemented by the two clauses:

```
(type (app M N) T) :- type M (arr S T), type N S.
(type (lam M) (arr S T)) :- (pi x\(type x S => type (M x) T)).
```

(Notice that in this program the typing-context is implicitly given by the "surrounding" program-context. This program-context can be modified using the universal quantification (i.e. `pi x\...`) and implications in goal-formulae. Therefore there is no clause for the variable case.) We find the most important reason in favour of αProlog is that by having concrete names (namely `x` in the `type`-example) and freshness-predicates one can almost directly translate the three typing rules into three clauses and obtain a correct implementation. This should be seen in the context that, despite the elegance of λProlog, some recent textbooks use (standard) Prolog for implementing inference rules over λ-terms. For example one of them presents the following implementation of the typing rules:

```
mem(X,[X|T]).
mem(X,[Y|T]) :- mem(X,T).

type(Gamma,var(X),T)   :- mem((X,T),Gamma).
type(Gamma,app(M,N),T):- type(Gamma,M,arr(S,T)), type(Gamma,N,S).
type(Gamma,lam(X,M),arr(S,T)):-type([(X,S)|Gamma],M,T).
```

which calculates the *wrong* type for λ-terms such as $\lambda x.\lambda x.(x\ x)$. Although this problem can be fixed by judicious use of cut or side-effects, first-order terms of Prolog are unwieldy for implementing relations over syntax with binders correctly. On the other hand, λProlog does not allow concrete names as binders and therefore operations such as adding the type for `x` to the typing-context need to be encoded using universal quantification, implications in goals and beta-reduction.

The αProlog language is based on nominal terms and uses the nominal unification algorithm of Urban, Pitts, and Gabbay [9], which calculates (most general) unifiers modulo α-equivalence. For example, the query `?- id(a.a,b.X)` is solved in αProlog by the capturing substitution $[\mathtt{X} := \mathtt{b}]$ since `a.a` and `b.b` are α-equivalent. However, nominal unification is not enough to make the programs given earlier function as intended. For this αProlog generates fresh names during proof-search. As seen above, before a query is unified with the fourth `subst`-clause, αProlog generates a fresh name for `a`. This ensures that substitutions can always be "pushed" under a binder without risk of capture.

While in [3] we have described our implementation of αProlog, its behaviour can be justified in terms of nominal logic [8, 2]. For instance, the generation of a fresh name can be expressed in terms of the И-quantifier of nominal logic, and an αProlog clause `A :- B1,..,Bn` can viewed as the formula $\text{И}a_1..a_n.\forall X_1..X_n.B_1\wedge\cdots\wedge B_n\supset A$, where the X_i and a_i are the variables, respectively names, in the clause. The problem is that the generation of fresh names is more subtle in αProlog than the usual "freshening" of variables when backchaining a Prolog-clause. The reason is that distinct names are always considered to denote different values. Consider the clause $\forall$`X.p(X)` and the query `p(b)` written as a sequent as follows

$$\forall\mathtt{X.p(X)}\ \vdash\ \mathtt{p(b)} \tag{2}$$

When constructing a proof for (2), Prolog generates a fresh name for the variable `X`, say `X'`, and then unifies `p(X')` and `p(b)` giving the solution `[X':=b]`. A similar αProlog-clause that has a name in place of the variable behaves differently: if we have the sequent

$$\text{И}\mathtt{a.p(a)}\ \vdash\ \mathtt{p(b)} \tag{3}$$

with the clause `Иa.p(a)`, then "freshening" `a` to `a'` leads to the unification problem `p(a')`$\approx$?`p(b)`. Since nominal unification treats names as distinct constants, this problem is unsolvable. (Treating names as distinct constants is important, because treating them as substitutable entities would break the most-general unifier property of nominal unification, see [9].) On the other hand, (3) *is* provable in nominal logic. This is because after freshening `a` to `a'` one can in nominal logic apply the equivariance principle—expressed as an inference rule[3]

$$\frac{\pi{\cdot}B, \Gamma \Rightarrow C}{B, \Gamma \Rightarrow C}\ \pi$$

where π is a permutation of names, B, C stand for formulae and Γ for a multiset of formulae. This means if the *full* Horn-fragment of nominal logic were used as the basis of αProlog, then we need equivariant unification for complete proof search. Equivariant unification solves a problem not just by finding a substitution but also by finding a permutation; for example in (3) the identity substitution and the permutation `(a' b)`.

The second author has shown in [1] that equivariant unification and equivariant matching problems are NP-hard. For proof-search in αProlog this means that one needs to guess which permutation π leads to a proof. However, in experimenting with αProlog we found that such guessing is never needed in the programs we considered. In this paper we identify a class of nominal Horn-clause programs for which the π-rule can be eliminated from deductions (this is the place where equivariant unification problems arise), and thus nominal unification is complete for proof-search. In order to show this result, we introduce a *well-formedness* condition which guarantees that nominal unification-based proof search is complete. This condition roughly says that a clause is "insensitive" to the particular choice of names occurring in it.

Some programs do not satisfy this condition. For example, in the following program calculating a list of bound variables of a λ-term, the last clause is *not* well-formed.

```
bv(var(X),[]).
bv(app(E1,E2),L)   :- bv(E1,L1), bv(E2,L2), append(L1,L2,L).
bv(lam(x.E),[x|L]) :- bv(E,L).
```

In the last clause, the result accumulated in the second argument depends on which name is chosen for the binder `x`. In contrast, the names chosen in the `subst` and `type` example do not matter (up to α-equivalence) and therefore will satisfy our well-formedness condition.

The existence of a trivial syntactic criterion for deciding when a clause is "insensitive" to the choice of a name seems unlikely. Consider, for example, allowing names to only occur bound or in binding position—then the `type`-program would be ruled out since `x` occurs free in the body of the clause. Restricting free names to occur only in the body of a clause would permit the clause `r(X):-id(X,a)` which *is* sensitive to the choice of `a` since `id` "propagates" the choice for the name `a` back to the head of the clause (`id` is defined in the `subst`-example). Our well-formedness condition is therefore more subtle; it is a test whether a certain matching problem derived from the clause is solvable. Despite being technically relatively complex, well-formedness can be automatically verified.

[3] The corresponding right-rule has been shown to be admissible in nominal logic in [5].

The paper is organised as follows: Section 2 describes nominal terms, formulae and the inference rules of αProlog's proof-search procedure. Section 3 introduces a well-formedness condition for clauses and shows that the π-rule can be eliminated from proofs involving only well-formed clauses. Section 4 describes how the well-formedness condition can be automatically verified. Section 5 concludes and describes future work.

2 Terms, Formulae and Proof-Search Rules

The terms used in αProlog are *nominal terms* (see [9] for more details) as defined by the grammar:

$$t ::= a \mid \pi{\cdot}X \mid \langle\rangle \mid \langle t, t\rangle \mid a.t \mid \mathtt{f}(t)$$

where a is a name, X a variable, $\mathtt{f}$ a function symbol and π a permutation expressed as a list of swappings $(a_1\, b_1)\cdots(a_n\, b_n)$. We have the operations $-@-$ and $(-)^{-1}$ for composing (list concatenation) two permutations and inverting (list reversal) a permutation, respectively. Constants are encoded as function symbols with unit arguments $\mathtt{f}(\langle\rangle)$, and n-tuples are encoded by iterated pairs $\langle t_1, \cdots \langle t_{n-1}, t_n\rangle\rangle$. Following [9], we refer to terms of the form $\pi{\cdot}X$ as *suspensions*, because the permutation π is suspended in front of a variable waiting to be applied to a term substituted for X.

Formulae are divided into goal formulae G and definite (or program) clauses D defined as

$$G ::= p(t) \mid G{\wedge}G \mid G{\vee}G \mid \top \qquad B ::= G \supset p(t) \qquad D ::= \text{И}as.\forall Xs.\nabla/B$$

where $p(t)$ stands for an atomic predicate with the argument t (we shall also write A for such formulae whenever the argument is unimportant); $\top, \wedge, \vee, \supset$ are standard connectives; and ∇ is a set of freshness constraints of the form $a_1 \# X_1, \ldots, a_n \# X_n$ (X_i and a_i being variables and names, respectively). The intended meaning of ∇ in D-formulae is that a clause is applicable only if its freshness constraints are satisfied. For freshness constraints and quantifier-free formulae we shall use the notation $Q_{as,Xs}$ ($Q ::= \nabla|G|B$) to indicate that the terms of Q are built up from names as and variables Xs (we have the usual convention that as stands for lists of names and Xs for lists of variables; similarly $\text{И}as$ stands for $\text{И}a_1 \ldots \text{И}a_n$ and $\forall Xs$ for $\forall X_1 \ldots \forall X_n$). We call a D-formula *closed* when it has no free variables and free names, that is the formula must be of the form $\text{И}as.\forall Xs.\nabla_{as,Xs}/B_{as,Xs}$. Fig. 1 shows two examples illustrating how D-formulae relate to the αProlog-clauses given at the beginning.

There is a delicate point with respect to binding: while in nominal terms the constructor $a.(-)$ is *not* a binder in the traditional sense (it only *acts* as a binder), in formulae the quantifiers $\text{И}a.(-)$ and $\forall X.(-)$ do bind a and X, respectively. Therefore we have the usual convention that formulae are identified if they only differ in the names of binders (i.e. $\forall X.(-)$ and $\text{И}a.(-)$), and operations on formulae need to respect this convention. As a result the definition of the permutation operation introduced for nominal terms in [9] needs to be extended. We define a generalised permutation operation $\pi{\cdot}_{\mathcal{B}}(-)$ that depends on a set of variables $\mathcal{B}$. The permutation π only acts upon variables *not* in $\mathcal{B}$. Whenever a permutation is "pushed" under a $\forall X.(-)$-quantifier, then X is added to the set of variables the permutation does not affect. The definition of the

```
subst(var(X),X,T,T).
```
$\forall X,T.\ \varnothing\,/\,\top \supset s(\mathtt{var}(X),X,T,T)$

```
subst(lam(a.M),X,T,lam(a.M')) :- a#T, a#X, subst(M,X,T,M').
```
$\text{И}a.\forall M,X,T,M'.\ \{a\,\#T, a\,\#X\}\,/\,s(M,X,T,M') \supset s(\mathtt{lam}(a.M),X,T,\mathtt{lam}(a.M'))$

Fig. 1. Two examples showing how αProlog-clauses relate to D-formulae (s is a predicate symbol standing for `subst`). We have the usual convention that clauses stand for closed D-formulae.

Terms:

$$[]{\cdot_\mathcal{B}}a \stackrel{\text{def}}{=} a \qquad \pi{\cdot_\mathcal{B}}(\pi'{\cdot}X) \stackrel{\text{def}}{=} \pi@\pi'{\cdot}X$$

$$((a_1\,a_2)::\pi){\cdot_\mathcal{B}}a \stackrel{\text{def}}{=} \begin{cases} a_1 & \text{if } \pi{\cdot_\mathcal{B}}a = a_2 \\ a_2 & \text{if } \pi{\cdot_\mathcal{B}}a = a_1 \\ \pi{\cdot_\mathcal{B}}a & \text{otherwise} \end{cases} \qquad \begin{aligned} \pi{\cdot_\mathcal{B}}(\langle\rangle) &\stackrel{\text{def}}{=} \langle\rangle \\ \pi{\cdot_\mathcal{B}}(\langle t_1,t_2\rangle) &\stackrel{\text{def}}{=} \langle \pi{\cdot_\mathcal{B}}t_1, \pi{\cdot_\mathcal{B}}t_2\rangle \\ \pi{\cdot_\mathcal{B}}(\mathtt{f}(t)) &\stackrel{\text{def}}{=} \mathtt{f}(\pi{\cdot_\mathcal{B}}t) \end{aligned}$$

$$\pi{\cdot_\mathcal{B}}X \stackrel{\text{def}}{=} \begin{cases} X & \text{if } X \in \mathcal{B} \\ \pi{\cdot}X & \text{otherwise} \end{cases} \qquad \pi{\cdot_\mathcal{B}}(a.t) \stackrel{\text{def}}{=} (\pi{\cdot_\mathcal{B}}a).(\pi{\cdot_\mathcal{B}}t)$$

Formulae:

$$\begin{aligned} \pi{\cdot_\mathcal{B}}(\top) &\stackrel{\text{def}}{=} \top & \pi{\cdot_\mathcal{B}}(G\supset A) &\stackrel{\text{def}}{=} (\pi{\cdot_\mathcal{B}}G)\supset(\pi{\cdot_\mathcal{B}}A) \\ \pi{\cdot_\mathcal{B}}(p(t)) &\stackrel{\text{def}}{=} p(\pi{\cdot_\mathcal{B}}t) & \pi{\cdot_\mathcal{B}}(\nabla/B) &\stackrel{\text{def}}{=} \nabla/(\pi{\cdot_\mathcal{B}}B) \\ \pi{\cdot_\mathcal{B}}(G_1\star G_2) &\stackrel{\text{def}}{=} (\pi{\cdot_\mathcal{B}}G_1)\star(\pi{\cdot_\mathcal{B}}G_2) & \pi{\cdot_\mathcal{B}}(\text{И}a.D) &\stackrel{\text{def}}{=} \text{И}a.\pi{\cdot_\mathcal{B}}D \\ \text{for } \star &::= \wedge|\vee & \pi{\cdot_\mathcal{B}}(\forall X.D) &\stackrel{\text{def}}{=} \forall X.\pi{\cdot_{\{X\}\cup\mathcal{B}}}D \end{aligned}$$

Fig. 2. Definition of the permutation operation $\pi{\cdot_\mathcal{B}}(-)$ for terms and formulae. In the clause for the new-quantifier, it is assumed that a is renamed, so that the permutation π can safely be pushed under the binder without capture.

permutation operation is given in Fig. 2. We use the shorthand notation $\pi{\cdot}(-)$ in case $\mathcal{B}$ is the empty set. This is a generalisation of the permutation action given in [9]; however, when a permutation acts on a formula with quantifiers, it acts only on the free names and free variables.

Similar problems arise in the definition of the substitution operation—with respect to the abstractions $a.(-)$ substitution is possibly-capturing, whereas with respect to the И- and $\forall$-quantifier it must be capture-avoiding. For terms we can use the definition given in [9]: a substitution σ is a function from variables to terms with the property that $\sigma(X) = X$ for all but finitely many variables X. If the domain of σ consists of distinct variables $X_1,\ldots,X_n$ and $\sigma(X_i) = t_i$ for $i = 1\ldots n$, we sometimes write σ as $[X_1 := t_1,\ldots,X_n := t_n]$. Moreover, we shall write $\sigma(t)$ for the result of applying a substitution σ to a term t; this is the term obtained from t by replacing each variable X by the term $\sigma(X)$ and each suspension $\pi{\cdot}X$ in t by the term $\pi{\cdot}\sigma(X)$ got by letting π act on the term $\sigma(X)$. This definition is extended to formulae as follows:

$$\begin{aligned} \sigma(\top) &\stackrel{\text{def}}{=} \top & \sigma(G\supset A) &\stackrel{\text{def}}{=} \sigma(G)\supset\sigma(A) \\ \sigma(p(t)) &\stackrel{\text{def}}{=} p(\sigma(t)) & \sigma(\nabla/B) &\stackrel{\text{def}}{=} \sigma(\nabla)/\sigma(B) \\ \sigma(G_1\star G_2) &\stackrel{\text{def}}{=} \sigma(G_1)\star\sigma(G_2) & \sigma(\text{И}a.D) &\stackrel{\text{def}}{=} \text{И}a.\sigma(D) \\ \text{for } \star &::= \wedge|\vee & \sigma(\forall X.D) &\stackrel{\text{def}}{=} \forall X.\sigma(D) \end{aligned}$$

with the proviso that the quantified names and variables are suitably renamed so that no capturing is possible. For example, if $\sigma = [X := \langle a, Y\rangle]$ and $t = a.X$, then

$$\frac{}{\nabla \vdash \langle\rangle \approx \langle\rangle}\,(\approx\text{-unit}) \quad \frac{\nabla \vdash t_1 \approx t'_1 \quad \nabla \vdash t_2 \approx t'_2}{\nabla \vdash \langle t_1, t_2\rangle \approx \langle t'_1, t'_2\rangle}\,(\approx\text{-pair}) \quad \frac{\nabla \vdash t \approx t'}{\nabla \vdash \mathtt{f}\, t \approx \mathtt{f}\, t'}\,(\approx\text{-fun. symbol})$$

$$\frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\,(\approx\text{-abs-1}) \qquad \frac{a \neq a' \quad \nabla \vdash t \approx (a\, a')\cdot t' \quad \nabla \vdash a \# t'}{\nabla \vdash a.t \approx a'.t'}\,(\approx\text{-abs-2})$$

$$\frac{}{\nabla \vdash a \approx a}\,(\approx\text{-name}) \qquad \frac{(a \# X) \in \nabla \text{ for all } a \in ds(\pi, \pi')}{\nabla \vdash \pi\cdot X \approx \pi'\cdot X}\,(\approx\text{-suspension})$$

$$\frac{}{\nabla \vdash a \# \langle\rangle}\,(\#\text{-unit}) \quad \frac{\nabla \vdash a \# t_1 \quad \nabla \vdash a \# t_2}{\nabla \vdash a \# \langle t_1, t_2\rangle}\,(\#\text{-pair}) \quad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# \mathtt{f}\, t}\,(\#\text{-fun. symbol})$$

$$\frac{}{\nabla \vdash a \# a.t}\,(\#\text{-abs-1}) \qquad \frac{a \neq a' \quad \nabla \vdash a \# t}{\nabla \vdash a \# a'.t}\,(\#\text{-abs-2})$$

$$\frac{a \neq a'}{\nabla \vdash a \# a'}\,(\#\text{-name}) \qquad \frac{(\pi^{-1}\cdot a \# X) \in \nabla}{\nabla \vdash a \# \pi\cdot X}\,(\#\text{-suspension})$$

Fig. 3. Inductive definitions for $\approx$ and $\#$. The reader is referred to [9] for more details.

$\sigma(t) = a.\langle a, Y\rangle$, but if $D = \text{И}a.\forall Y.\varnothing/\top \supset A(a, Y, X)$ then forming $\sigma(D)$ gives the formula $\text{И}a'.\forall Y'.\varnothing/\top \supset A(a', Y', \langle a, Y\rangle)$. We use the notation $\sigma(\nabla)$ to mean that every freshness constraint $a \# X$ in ∇ is replaced by $a \# \sigma(X)$.

It is crucial for programming in αProlog that abstractions $a.(-)$ have concrete names. This allows us to formulate the `type`-clause for lambda-abstractions in the usual fashion whereby the abstracted name `x` and its type is just added to the context `Gamma`. Furthermore, the work reported in [9] provides us with a simple algorithm for unifying nominal terms. This unification algorithm does not calculate unifiers to make nominal terms syntactically equal, but equal modulo an equivalence relation $\approx$. For example when unifying the two terms $a.a \approx ?\ b.X$, the nominal unification algorithm produces the unifier $[X := b]$. While the relation $\approx$ is intended to capture the (traditional) notion of α-equivalence, it is in fact a more general relation. For example, $\approx$ is not just a relation between two nominal terms, but a relation that depends on some freshness constraints ∇. Figure 3 gives a syntax-directed inductive definition for judgements of the form $\nabla \vdash (-) \approx (-)$, which asserts that two terms are $\approx$-equal under the hypotheses ∇; the definition depends on the auxiliary relation $\nabla \vdash (-) \# (-)$, which defines when a name is *fresh* for a term under some hypotheses. This definition depends on the auxiliary notion of a disagreement set, ds, between two permutations (the set of names on which the permutations disagree) given by: $\{a \mid \pi_1\cdot a \neq \pi_2\cdot a\}$.

We can extend $\approx$ to *quantifier-free* G-formulae as follows:

$$\frac{}{\nabla \vdash \top \approx \top} \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash p(t) \approx p(t')} \qquad \frac{\nabla \vdash G_1 \approx G_3 \quad \nabla \vdash G_2 \approx G_4}{\nabla \vdash G_1 \star G_2 \approx G_3 \star G_4} \quad \text{for } \star ::= \wedge | \vee$$

The advantage of setting up the formalism in this way is that the $\approx$-equivalence has a number of good properties, which will play an important rle in our proof for showing that the π-rule can be eliminated. For example, $\approx$ is preserved under (possibly-capturing) substitutions and behaves well with respect to the permutation operation. This is made precise in the following lemma.

$$\frac{}{\nabla;\Gamma\Rightarrow\top}\top_R \quad \frac{\nabla;\Gamma\Rightarrow G \quad \nabla;\Gamma\Rightarrow G'}{\nabla;\Gamma\Rightarrow G\wedge G'}\wedge_R \quad \frac{\nabla;\Gamma\Rightarrow G_i}{\nabla;\Gamma\Rightarrow G_1\vee G_2}\vee_{Ri} \quad \frac{\nabla;D,\Gamma\xrightarrow{D}p(t)}{\nabla;D,\Gamma\Rightarrow p(t)}\,\mathrm{Sel}$$

$$\frac{\nabla\vdash t'\approx t}{\nabla;\Gamma\xrightarrow{p(t')}p(t)}\,\mathrm{Ax} \quad \frac{\nabla\vdash\nabla' \quad \nabla;\Gamma\Rightarrow G \quad \nabla;\Gamma\xrightarrow{p(t')}p(t)}{\nabla;\Gamma\xrightarrow{\nabla'/G\supset p(t')}p(t)}\supset_L$$

$$\frac{\nabla;\Gamma\xrightarrow{D[X:=t']}p(t)}{\nabla;\Gamma\xrightarrow{\forall X.D}p(t)}\forall_L \quad \frac{b\# Xs,\nabla;\Gamma\xrightarrow{(a\,b)\cdot D}p(t)}{\nabla;\Gamma\xrightarrow{\text{И}a.D}p(t)}\text{И}_L \quad \frac{\nabla;\Gamma\xrightarrow{\pi\cdot D}p(t)}{\nabla;\Gamma\xrightarrow{D}p(t)}\pi$$

Fig. 4. Proof-search rules of αProlog. In the new-left rule it is assumed that b is a fresh name not occurring in the conclusion and Xs are all free variables in Γ and $p(t)$.

Lemma 1. *The permutation and substitution operations preserve $\approx$ in the sense that*

(i) if $\nabla\vdash t\approx t'$ then $\nabla\vdash\pi\cdot t\approx\pi\cdot t'$ for all permutations π and

(ii) if $\nabla\vdash t\approx t'$, then $\nabla'\vdash\sigma(t)\approx\sigma(t')$ for all substitutions σ with $\nabla'\vdash\sigma(\nabla)$ (whereby $\nabla'\vdash\sigma(\nabla)$ means that $\nabla'\vdash a\#\sigma(X)$ holds for each $(a\# X)\in\nabla$).

The proof of these two facts are a minor extension of the proofs given for [9]; they hold because permutations are bijections on names, and substitutions act on variables only (not names). The properties stated in Lemma 1 should be compared with the notion of α-equivalence we imposed (on the meta-level) on quantified D-formulae. There, whenever a permutation or substitution is pushed under a binder, we might have to rename its binder in order to avoid possible capture.

Next we introduce the inference rules on which proof-search is based in αProlog (see Figure 4). Sequents are of the form $\nabla;\Gamma\Rightarrow G$ or $\nabla;\Gamma\xrightarrow{D}p(t)$ where the former models *goal-directed proof-search* and the latter models *focused backchaining* (the formula above the sequent arrow is usually called the *stoup*-formula). These inference rules are adapted from a standard focusing approach to first-order logic programming (for example [4]).[4] The main novelty of these rules is the presence of the freshness-constraints ∇. Traditionally axiom rules are formulated as

$$\frac{}{p(t'),\Gamma\Rightarrow p(t)}\,\mathrm{Ax}, \quad \text{or in focusing proofs as} \quad \frac{}{\Gamma\xrightarrow{p(t')}p(t)}\,\mathrm{Ax},$$

where the terms t and t' need to be syntactically equal. In αProlog this requirement is relaxed: terms only need to be being equal modulo $\approx$. However, $\approx$ only makes sense in the context of some freshness constraints. Consequently, in αProlog, the axiom-rule takes the form

$$\frac{\nabla\vdash t'\approx t}{\nabla;\Gamma\xrightarrow{p(t')}p(t)}\,\mathrm{Ax}$$

where the context ∇ explicitly records all freshness constraints in a sequent. The only inference rule which adds new freshness-constraints to this context is the И-rule; that

[4] The question of establishing the precise relation between the inference rules given here and nominal logic introduced in [8] is beyond the scope of this paper, but will appear in a full version (some results concerning this question have already been presented in [5]).

is whenever a И-quantifier is analysed, a new name is chosen and some freshness-constraints are added to ∇ in order to enforce the "freshness" of this name.

The $\supset_L$-rule includes the judgement $\nabla \vdash \nabla'$ where ∇' is the set of freshness constraints associated with the D-formula in the stoup. This judgement requires that all constraints in ∇' (being of the form $a \;\#\; t$) are satisfied by the ∇, that is for all $a \;\#\; t$ the judgement $\nabla \vdash a \;\#\; t$ defined in Fig. 3 holds.

Of most interest in this paper is the π-rule. In a "root-first" proof-search, this rule is a source of non-determinism. For example, if we want to prove the sequent $\varnothing; \xrightarrow{p(a)} p(b)$, we need the π-rule in order to make the terms a and b $\approx$-equivalent—in this case, only after applying a permutation such as $(a\,b)$ to a may the axiom-rule be used. Prima facie the π-rule is innocuous, however, the problem of simultaneously unifying nominal terms and finding a π is, as mentioned earlier, an NP-hard decision problem. In the next section we shall show that such problems never need to be solved provided the program clauses are well-formed.

3 Elimination of the π-Rule

We implemented αProlog using the nominal unification algorithm. With this implementation we were able to calculate the expected results for programs such as `subst` and `type`. The reason for this is, roughly speaking, that the name we used for specifying the clauses dealing with λ-abstractions does not matter. When using nominal unification, the following renamed clauses (where `a` and `x` are renamed to `b` and `y`, respectively)

```
subst(lam(b.M),X,T,lam(b.M')):- b#T, b#X, subst(M,X,T,M').
type(Gamma,lam(y.M),arr(S,T)):- y#Gamma, type([(y,S)|Gamma],M,T).
```

behave just the same as the original clauses, in the sense that all queries successfully solved by the original versions are solved by the renamed versions. In contrast, the name `a` in the clauses `p(a)`, `q(a.X,X)` and `r(X) :- id(X,a)` determines which queries can be solved successfully using nominal unification and which cannot: given our inference rules, which choose a fresh name for `a`, there are some queries whose answers can only be found using the π-rule, and this means they cannot be solved using nominal unification. Consider for example the following deduction.

$$\dfrac{\dfrac{\dfrac{\ldots \qquad \dfrac{\varnothing \vdash c \approx c}{\varnothing; \xrightarrow{p(c)} p(c)}\,\text{Ax}}{\varnothing; \xrightarrow{\varnothing/\top \supset p(c)} p(c)}\,\supset_L}{\varnothing; \xrightarrow{\varnothing/\top \supset p(b)} p(c)}\,\pi \;\; (c\,b)}{\varnothing; \xrightarrow{\text{И}a.\varnothing/\top \supset p(a)} p(c)}\,\text{И}_L$$

In this deduction the π-rule, applying the permutation $(c\,b)$ (annotated to the π-rule), is crucial for the sequent being provable and it will turn out that it is impossible to eliminate it from this deduction. Consequently, a proof-search procedure based on nominal unification will not find this proof.

If we impose the following well-formedness condition on D-formulae, we can ensure that the π-rule can always be eliminated from corresponding deductions and hence the nominal unification algorithm alone is sufficient for solving queries.

Definition 1. *A closed D-formula* $\reflectbox{\textsf{N}}as.\forall Xs.\nabla_{as,Xs}/G_{as,Xs} \supset A_{as,Xs}$ *is* well-formed *if there exists a substitution σ and a permutation π such that*

(i) $bs \# Xs, \nabla_{as,Xs} \vdash \sigma(A_{bs,Xs}) \approx A_{as,Xs}$ *and*
(ii) $bs \# Xs, \nabla_{as,Xs} \vdash \sigma(\pi{\cdot}G_{bs,Xs}) \approx G_{as,Xs}$

where the bs are some fresh names (different from as).

Let us illustrate this condition with some examples. Clauses without names clearly satisfy the condition. For example the first `subst`-clause in Fig. 1

$$\varnothing \vdash \sigma(S(\mathtt{var}(X), X, T, T)) \;\approx\; S(\mathtt{var}(X), X, T, T) \quad \text{and} \quad \varnothing \vdash \sigma(\pi{\cdot}\top) \;\approx\; \top$$

trivially satisfies the condition by taking for σ the identity substitution and for π the empty permutation. More complicated is the case of the second `subst`-clause in Fig. 1

$$\nabla \vdash \sigma(S(\mathtt{lam}(b.M), X, T, \mathtt{lam}(b.M')) \;\approx\; S(\mathtt{lam}(a.M), X, T, \mathtt{lam}(a.M'))$$
$$\nabla \vdash \sigma(\pi{\cdot}S(M, X, T, M')) \;\approx\; S(M, X, T, M')$$

where ∇ is $\{b \# M, b \# X, b \# T, b \# M', a \# X, a \# T\}$. In this case

$$\sigma = [M := (a\,b){\cdot}M, X := (a\,b){\cdot}X, T := (a\,b){\cdot}T, M' := (a\,b){\cdot}M'] \quad \text{and} \quad \pi = (a\,b)$$

verify that the clause is well-formed.

Before we formally show that all π-rules can be eliminated from deductions consisting of well-formed clauses only, we outline our proof-plan with some examples. Consider the following deduction, which has a π-rule on the top right-hand side. The corresponding permutation $(e\,d)$ transforms $p(b.d)$ into $p(b.e)$ so that the axiom-rule is applicable.

$$\dfrac{\dfrac{\dfrac{\dots \qquad \dfrac{\dfrac{\vdash b.e \approx d.e}{\varnothing; \xrightarrow{p(b.e)} p(d.e)}\ \text{Ax}}{\varnothing; \xrightarrow{p(b.d)} p(d.e)}\ \pi\ (e\,d)}{\varnothing; \xrightarrow{\varnothing/\top\supset p(b.d)} p(d.e)}\ \supset_L}{\varnothing; \xrightarrow{\forall X.\varnothing/\top\supset p(b.X)} p(d.e)}\ \forall_L\ \ [X := d]}{\varnothing; \xrightarrow{\reflectbox{\textsf{N}}a.\forall X.\varnothing/\top\supset p(a.X)} p(d.e)}\ \reflectbox{\textsf{N}}_L\ (a\,b)$$

Observe that the "choice" of the fresh name (namely b) introduced by the $\reflectbox{\textsf{N}}_L$-rule has no effect on whether this sequent is derivable, since this binder will not bind anything inside the abstraction. The annotated substitution $[X := d]$ however is important with respect to the π-rule we are trying to eliminate. If we had instead substituted e for X, then the axiom is applicable *without* the π-rule.

Note, however, that changing the instantiation of $\forall$-quantifiers might have some "non-local" consequences in deductions. Consider for example the deduction in Fig. 5. In this deduction, the π-rule (marked by $\bullet$) swaps the names z and y. If we eliminate this π-rule by applying the swapping to the terms instantiated for the variables M, X, T and M', then the π-rule is not needed, but at the same time the subgoal (marked by $\star$) is changed. The well-formedness condition ensures that the modification of the terms introduced by the $\forall_L$-rules does not affect the provability of the sequent.

To show that π-rules can be eliminated from derivations involving well-formed programs, we first prove some auxiliary facts.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        ..\quad \dfrac{\vdots}{\Rightarrow s(\mathtt{v}(z),z,\mathtt{v}(y),\mathtt{v}(y))}\,\star
        \qquad
        \dfrac{
          \dfrac{..\vdash\langle \mathtt{l}(b.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(b.\mathtt{v}(z))\rangle\approx\langle \mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z))\rangle}
          {..;\ \xrightarrow{s(\mathtt{l}(b.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(b.\mathtt{v}(z)))} s(\mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z)))}\,\mathrm{Ax}
        }
        {..;\ \xrightarrow{s(\mathtt{l}(b.\mathtt{v}(z)),z,\mathtt{v}(y),\mathtt{l}(b.\mathtt{v}(y)))} s(\mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z)))}\,\pi^\bullet
      }
      {..;\ \xrightarrow{\nabla/s(\mathtt{v}(z),z,\mathtt{v}(y),\mathtt{v}(y))\supset s(\mathtt{l}(b.\mathtt{v}(z)),z,\mathtt{v}(y),\mathtt{l}(b.\mathtt{v}(y)))} s(\mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z)))}\,{\supset_L}
    }
    {\vdots \qquad [M:=\mathtt{v}(z),X:=z,T:=\mathtt{v}(y),M':=\mathtt{v}(y)]}\,\forall_L
  }
  {..;\ \xrightarrow{\forall M,X,T,M'.\nabla/s(M,X,T,M')\supset s(\mathtt{l}(b.M),X,T,\mathtt{l}(b.M'))} s(\mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z)))}\,\forall_L
}
{..;\ \xrightarrow{\text{И} a.\forall M,X,T,M'.\nabla/s(M,X,T,M')\supset s(\mathtt{l}(a.M),X,T,\mathtt{l}(a.M'))} s(\mathtt{l}(x.\mathtt{v}(y)),y,\mathtt{v}(z),\mathtt{l}(x.\mathtt{v}(z)))}\,\text{И}_L
$$

Fig. 5. Deduction proving the fact `s(l(x.v(y)),y,v(z),l(x.v(z)))` where `l` and `v` stand for lambda-abstractions and variables, respectively.

Lemma 2. *For all permutations π, the sequent $\nabla;\Gamma \Rightarrow \pi{\cdot}G$ is derivable only if the sequent $\nabla;\pi^{-1}{\cdot}\Gamma \Rightarrow G$ is derivable (where we use the notation $\pi{\cdot}\Gamma$ to indicate that π is applied to every formula in Γ).*

Proof. By induction on the structure of deductions. It makes use of the property of $\approx$ that $\nabla \vdash t \approx \pi{\cdot}t'$ holds only if $\nabla \vdash \pi^{-1}{\cdot}t \approx t'$ holds. By inspection we can further see that no additional π-rule is necessary to show the provability in both directions. $\square$

The following corollary is a simple consequence of this lemma by the fact that for closed D-formulae $\pi{\cdot}D = D$ holds.

Corollary 1. *For all permutations π and contexts Γ consisting of closed D-formulae only, $\nabla;\Gamma \Rightarrow \pi{\cdot}G$ is derivable only if $\nabla;\Gamma \Rightarrow G$ is derivable.*

Lemma 3. *If the sequent $\nabla;\Gamma \Rightarrow G$ is derivable and $\nabla \vdash G \approx G'$, then the sequent $\nabla;\Gamma \Rightarrow G'$ is derivable.*

Proof. Since $\nabla \vdash G \approx G'$ is inductively defined extending the $\approx$-equality of the terms occurring in G and G', we can prove this lemma by inspection of the inference-rules, noting that in the $(-)_L$-rules the right-hand side of sequents is always of the form $p(t)$ and the lemma for axioms follows from the transitivity of $\approx$.

For showing our main result, it is convenient to restrict attention to some specific instances of the π-rule. The next lemma shows that we only need to consider unmovable instances of the π-rule.

Definition 2. *A π-rule is* movable *provided it is not directly under an axiom, otherwise it is said to be* unmovable.

Lemma 4. *All movable instances of the π-rules can be replaced by unmovable instances.*

Proof. We call a derivation π-normalised if all instances of the π-rule are unmovable. We first show that if $\Gamma \xrightarrow{\pi \cdot D} A$ has a π-normalised derivation, then we can construct a π-normalised derivation of $\Gamma \xrightarrow{D} A$. Using this construction, we can eliminate the movable π-rules from any derivation one at a time.

There is one case for each left-rule. For Ax, we have

$$\dfrac{}{\nabla; \Gamma \xrightarrow{\pi \cdot A} \pi \cdot A}\,Ax \;\longrightarrow\; \dfrac{\dfrac{}{\nabla; \Gamma \xrightarrow{\pi \cdot A} \pi \cdot A}\,Ax}{\nabla; \Gamma \xrightarrow{A} \pi \cdot A}\,\pi^{-1}$$

since $\pi^{-1} \cdot \pi \cdot A = A$. A π-normalised derivation ending in a π'-rule must be immediately followed by Ax, we can derive

$$\dfrac{\dfrac{}{\nabla; \Gamma \xrightarrow{\pi' \cdot \pi \cdot A} \pi' \cdot \pi \cdot A}\,Ax}{\nabla; \Gamma \xrightarrow{\pi \cdot A} \pi' \cdot \pi \cdot A}\,\pi' \;\longrightarrow\; \dfrac{\dfrac{}{\nabla; \Gamma \xrightarrow{\pi' @ \pi \cdot A} \pi' \cdot \pi \cdot A}\,Ax}{\nabla; \Gamma \xrightarrow{A} \pi' \cdot \pi \cdot A}\,\pi' @ \pi$$

since $\pi' \cdot \pi \cdot A = \pi' @ \pi \cdot A$. For $\forall_L$, since $\pi \cdot \forall X.D = \forall X.\pi \cdot_{\{X\}} D$ and $(\pi \cdot_{\{X\}} D)[X := t] = \pi \cdot (D[X := \pi^{-1} \cdot t])$, so we have

$$\dfrac{\nabla; \Gamma \xrightarrow{(\pi \cdot_{\{X\}} D)[X:=t]} A}{\nabla; \Gamma \xrightarrow{\forall X.\pi \cdot_{\{X\}} D} A}\,\forall_L \;\longrightarrow\; \dfrac{\nabla; \Gamma \xrightarrow{D[X:=\pi^{-1} \cdot t]} A}{\nabla; \Gamma \xrightarrow{\forall X.D} A}\,\forall_L$$

where by induction $\nabla; \Gamma \xrightarrow{D[X:=\pi^{-1} \cdot t]} A$ has a π-normalised derivation obtained from that of $\nabla; \Gamma \xrightarrow{\pi \cdot D[X:=\pi^{-1} \cdot t]} A$. The cases for $\wedge_L$ and И_L are straightforward since $\pi \cdot_{\mathcal{B}} (-)$ commutes with $\wedge$ and И. For $\supset_L$ we have

$$\dfrac{\nabla \vdash \nabla \quad \nabla; \Gamma \Rightarrow \pi \cdot G \quad \nabla; \Gamma \xrightarrow{\pi \cdot A} A'}{\nabla; \Gamma \xrightarrow{\nabla / (\pi \cdot G) \supset (\pi \cdot A)} A'}\,\supset_L \;\longrightarrow\; \dfrac{\nabla \vdash \nabla \quad \nabla; \Gamma \Rightarrow G \quad \nabla; \Gamma \xrightarrow{A} A'}{\nabla; \Gamma \xrightarrow{\nabla / G \supset A} A'}\,\supset_L$$

using Lemma 3 to derive $\nabla; \Gamma \Rightarrow G$ from $\nabla; \Gamma \Rightarrow \pi \cdot G$ and the induction hypothesis to obtain a π-normalised derivation of $\nabla; \Gamma \xrightarrow{A} A'$ from that of $\nabla; \Gamma \xrightarrow{\pi \cdot A} A'$.

Theorem 1. *If Γ consists of well-formed clauses and the sequent $\nabla; \Gamma \Rightarrow G$ is derivable, then it is derivable without using the π-rule.*

Proof. Since Γ consists of well-formed clauses only, all Γ's in the deduction consist of well-formed clauses (formulae on the left-hand side are analysed only if they are selected to be in the stoup-position). By Lemma 4, we can replace this deduction by one in which all π-rules are unmovable. So we need to consider how unmovable π-rules can be eliminated. Recall that unmovable π-rules occur in segments of the form

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots \\ \nabla''_{bs}, \nabla' \vdash \nabla_{ts}\end{matrix} \quad \begin{matrix}\vdots \\ \nabla''_{bs}, \nabla'; \Gamma \Rightarrow G_{bs,ts}\end{matrix} \quad \dfrac{\dfrac{\nabla''_{bs}, \nabla' \vdash \pi \cdot s_{bs,ts} \approx t}{\nabla''_{bs}, \nabla'; \Gamma \xrightarrow{\pi \cdot p(s_{bs,ts})} p(t)}\,\mathrm{Ax}}{\nabla''_{bs}, \nabla'; \Gamma \xrightarrow{p(s_{bs,ts})} p(t)}\,\pi}{\begin{matrix}\nabla''_{bs}, \nabla'; \Gamma \xrightarrow{\nabla_{ts}/G_{bs,ts} \supset p(s_{bs,ts})} p(t) \\ \vdots \qquad\qquad \}\,\text{И}\forall\end{matrix}}\,\supset_L}{\nabla'; \Gamma \xrightarrow{\text{И}as.\forall Xs.\nabla_{Xs}/G_{as,Xs} \supset p(s_{as,Xs})} p(t)}}{\nabla'; \Gamma \Rightarrow p(t)}\,\mathrm{Sel} \tag{4}$$

where the И-quantifier introduces the names bs and the $\forall$-quantifiers replace the variables Xs with the terms ts. We indicate this by using the notation $G_{as,Xs}$ and $G_{bs,ts}$. The freshness constraints ∇''_{bs} stand for the constraints introduced by the И-quantifiers, that is $b \# FV(t)$ for each b in bs. Let σ be the substitution of the terms ts for the variables Xs, that is the terms introduced by the $\forall$-quantifiers.

Below we give a deduction without the π-rule where the bs and ts are suitably changed. For this we choose first some fresh names cs with the proviso that $\pi{\cdot}cs = cs$, which means they are unaffected by the permutation introduced by the π-rule (such fresh names always exist). From the well-formedness of the clause in the stoup-position, we know that there is a substitution σ' and a permutation π' such that

$$\begin{array}{l} cs \# Xs, \nabla_{Xs} \vdash \sigma'(p(s_{cs,Xs})) \approx p(s_{bs,Xs}) \\ cs \# Xs, \nabla_{Xs} \vdash \sigma'(\pi'{\cdot}G_{cs,Xs}) \approx G_{bs,Xs} \end{array} \tag{5}$$

hold where we use the short-hand notation $cs \# Xs$ to refer the sets of freshness constraints $c_i \# X_1, \ldots, c_i \# X_n$ for all names c_i in cs. By Lemma 1*(ii)*, $\approx$ is preserved under substitutions, so we can infer from (5) that

$$\begin{array}{l} cs \# Xs \vdash \sigma \circ \sigma'(p(s_{cs,Xs})) \approx \sigma(p(s_{bs,Xs})) \\ cs \# Xs \vdash \sigma \circ \sigma'(\pi'{\cdot}G_{cs,Xs}) \approx \sigma(G_{bs,Xs}) \end{array} \tag{6}$$

hold where the right-hand sides are $p(s_{bs,ts})$ and $G_{bs,ts}$, respectively. Note that the ∇_{Xs} "vanish" because we have that $cs \# Xs \vdash \sigma(\nabla_{Xs})$. From (6) we can further infer that

$$cs \# Xs \vdash \pi{\cdot}\sigma \circ \sigma'(p(s_{cs,Xs})) \approx \pi{\cdot}(p(s_{bs,ts})) \tag{7}$$

$$cs \# Xs \vdash \pi{\cdot}\sigma \circ \sigma'(\pi'{\cdot}G_{cs,Xs}) \approx \pi{\cdot}(G_{bs,ts}) \tag{8}$$

hold by Lemma 1*(i)* asserting that $\approx$ is preserved under permutations. Recall that we chosen the cs so that π does not affect them. So if we apply the substitution $\sigma \circ \sigma'$ and the permutation π to the left-hand side of (7) we have $\pi{\cdot}\sigma \circ \sigma'(p(s_{cs,Xs})) = p(s_{cs,ts'})$ for some terms ts'. Moreover we have $cs \# Xs \vdash s_{cs,ts'} \approx \pi{\cdot}s_{bs,ts}$ which means we can replace in the deduction (4) $\pi{\cdot}s_{bs,ts}$ by $s_{cs,ts'}$ and get by transitivity of $\approx$ a correct instance of the axiom. Thus we can form the deduction:

$$\dfrac{\dfrac{\begin{array}{c} \dfrac{\begin{array}{c}\vdots\\ \nabla''_{cs}, \nabla' \vdash \nabla_{ts'}\end{array} \quad \begin{array}{c}\vdots\\ \nabla''_{cs}, \nabla'; \varGamma \Rightarrow G_{cs,ts'}\end{array} \quad \dfrac{\nabla''_{cs}, \nabla' \vdash s_{cs,ts'} \approx t}{\nabla''_{cs}, \nabla'; \varGamma \xrightarrow{p(s_{cs,ts'})} p(t)}\,\mathrm{Ax}}{\nabla''_{cs}, \nabla'; \varGamma \xrightarrow{\nabla_{ts'}/G_{cs,ts'} \supset p(s_{cs,ts'})} p(t)}\,\supset_L \\ \vdots \qquad\qquad \}\,И\forall \end{array}}{\nabla'; \varGamma \xrightarrow{Иas.\forall Xs.\nabla_{Xs}/G_{as,Xs} \supset p(s_{as,Xs})} p(t)}}{\nabla'; \varGamma \Rightarrow p(t)}\,\mathrm{Sel}$$

without the π-rule. We still need to ensure that $\nabla''_{cs}, \nabla'; \varGamma \Rightarrow G_{cs,ts'}$ and $\nabla''_{cs}, \nabla' \vdash \nabla_{ts'}$ are derivable. The second sequent is derivable because $cs \# Xs \vdash \sigma(\nabla_{Xs})$. For the first sequent we can infer from the original (sub)deduction $\nabla''_{bs}, \nabla'; \varGamma \Rightarrow G_{bs,ts}$ by Corollary 1 that $\nabla''_{bs}, \nabla'; \varGamma \Rightarrow \pi{\cdot}G_{bs,ts}$ is derivable (this deduction does not introduce any new π-rules). In (8) we can pull out the permutation π' and we have $\pi{\cdot}\sigma \circ \sigma'(\pi'{\cdot}G_{cs,Xs}) = \pi @ \pi'{\cdot}(\sigma \circ \sigma'(G_{cs,Xs}))$. Therefore applying the substitution to $G_{cs,Xs}$ gives $\pi @ \pi'{\cdot}(\sigma \circ \sigma'(G_{cs,Xs})) = \pi @ \pi'{\cdot}G_{cs,ts'}$ (taking the ts' we introduced for $s_{cs,ts'}$ earlier). Thus by Lemma 3 we can show that $\nabla''_{cs}, \nabla'; \Gamma \Rightarrow G_{cs,ts'}$ is derivable.

Each transformation decreases the number of π-rules in a deduction by one and thus by repeated application we will eventually end up with a π-free proof. $\square$

We have shown that when all the formulas in Γ are well-formed, every deduction of $\Gamma \Rightarrow G$ containing π-rules can be replaced by one without π-rules. Consequently, nominal unification is sufficient for executing well-formed αProlog-programs.

4 Verification of Well-Formedness Using Nominal Matching

In this section we consider the question of how to verify the well-formedness condition given in Definition 1. For a clause $\text{И} as.\forall Xs.\nabla_{as,Xs}/G_{as,Xs}\supset A_{as,Xs}$, we need to find a substitution σ and permutation π which make the two judgements

$$bs \;\#\; Xs, \nabla_{as,Xs} \vdash \sigma(A_{bs,Xs}) \approx A_{as,Xs} \quad \text{and} \quad bs \;\#\; Xs, \nabla_{as,Xs} \vdash \sigma(\pi{\cdot}G_{bs,Xs}) \approx G_{as,Xs}$$

hold. For the first judgement, σ can be found by nominal matching. But for the second judgement, finding both substitution σ and permutation π requires solving (NP-hard) equivariant matching problems. This seemingly negative result should, however, be seen in the context that well-formedness only needs to be verified once per clause, rather than repeatedly during proof-search. Thus, the one-time cost of performing equivariant unification in checking well-formedness is negligible compared to the cost of performing equivariant unification throughout computation. Furthermore, as can be seen from the examples, the number of names in a clause is usually small. Taking the following proposition (whose proof we omit)

Proposition 1. *If $G_{bs,Xs}$ equivariantly matches with $G_{as,Xs}$, then a matching exists in which the permutation π consists of swappings $(a_i\, b_i)$ only.*

into account, we can just enumerate all possible cases (2^n given n names) and solve each of the nominal matching problems. If one problem can be solved, then we have a σ and a π as required by the condition.

5 Conclusion

We have shown that for well-formed αProlog programs, all instances of the π-rule can be removed from deductions. As a result, proof search using only nominal unification is complete for such programs, which coincides with our experimental results gained from our implementation of αProlog. This is a significant result, because the alternative is to use an NP-hard equivariant unification algorithm for proof search.

In order to be well-formed, the `type`-program given in the Introduction needs to be stated as follows

```
type(Gamma,lam(x.M),arr(S,T)) :-
          x # Gamma, x#S, x#T, type([(x,S)|Gamma],M,T).
```

explicitly giving the freshness constraints `x#S` and `x#T`. These constraints do not affect the meaning of the program because term variables are expected (by programmer convention) not to appear in types. In fact, our implementation of αProlog is strongly typed and therefore can determine automatically from type information that lambda-term variables can never occur in types. Thus, our analysis could be made more precise by taking type information into account.

Let us briefly mention whether our result can be strengthened. The logic programming language λProlog [7] has convincingly demonstrated the usefulness of implications in G-formulae (that is extending logic programming to the setting of Hereditary Harrop formulae). In αProlog we would like to allow implications in G-formulae as well. Whether our result extends to such formulae is still open. It seems that our definition of G-formulae can be extended to include existential and universal formulae. However, our proving technique for showing this would require some subtle modifications—for example we would need to define when two formulae with quantifiers are $\approx$-equal, which is non-trivial. However, we expect that this can be done. What is impossible is to allow И-quantifiers in goal-formulae. Such formulae really need equivariant unification.

Acknowledgements: This research was supported by a fellowship for Urban from the Alexander-von-Humboldt foundation.

References

1. J. Cheney. The complexity of equivariant unification. In *Proc. of International Colloquium on Automata, Languages and Programming*, volume 3142 of *LNCS*, pages 332–344, 2004.
2. J. Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, 2004.
3. J. Cheney and C. Urban. Alpha-prolog: A logic programming language with names, binding, and α-equivalence. In B. Demoen and V. Lifschitz, editors, *Proc. of International Conference on Logic Programming*, volume 3132 of *LNCS*, pages 269–283, 2004.
4. R. Dyckhoff and L. Pinto. Proof Search in Constructive Logic. In S. Barry Cooper and John K. Truss, editors, *Proc. of the Logic Colloquium 1997*, volume 258 of *London Mathematical Society Lecture Note Series*, pages 53–65. Cambridge University Press, 1997.
5. M. J. Gabbay and J. Cheney. A proof theory for nominal logic. In *Proc. of Annual IEEE Symposium on Logic in Computer Science*, pages 139–148, 2004.
6. J. C. Mitchell. *Concepts in Programming Languages*. CUP Press, 2003.
7. G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Clarendon Press, 1998.
8. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
9. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-2):473–497, 2004.