Work in Progress: Logic Programming with Names and Binding

James Cheney[1] and Christian Urban[2]

[1] Cornell University
jcheney@cs.cornell.edu
[2] University of Cambridge
Christian.Urban@cl.cam.ac.uk

Abstract. In this paper we describe work in progress on αProlog, a logic programming language with a built-in notion of names, binding and unification modulo α-equivalence. αProlog is based on a mild extension of first-order Horn formulae: instead of the usual first-order terms and first-order unification, it uses nominal terms and nominal unification introduced in [3]. **Note:** This paper is a revised version of [1].

1 Introduction

Logic programming is particularly suited for implementing inference rules defining relations over terms. Many interesting examples, however, involve terms with bound variables and α-equivalence. For example the typing relation for λ-terms is often formulated with the inference rules

$$\frac{x : T \in \Gamma}{\Gamma \triangleright x : T} \qquad \frac{\Gamma \triangleright M : T \to T' \quad \Gamma \triangleright N : T}{\Gamma \triangleright M N : T'} \qquad \frac{\{x : T\} \cup \Gamma \triangleright M : T'}{\Gamma \triangleright \lambda x.M : T \to T'}$$

with the implicit side-condition that in Γ (a set of type assignments for variables) no variable has more than one assignment. This side-condition is violated if the third rule is applied bottom-up and Γ in the conclusion already contains an assignment for x. In this case the bound variable x in $\lambda x.M$ needs to be renamed to a new x' so that a type assignment for x' can be safely added to Γ. It is understood that this renaming does not affect the 'meaning' of the term, because terms are identified modulo α-equivalence.

However, this intuitive account of α-equivalence does not carry over to a correct declarative implementation in Prolog. For example, if we naïvely implement the typing rules with the clauses

```
member X X::L.
member X Y::L                    :- member X L.

type Gamma (var X) T             :- member (pair X T) Gamma.
type Gamma (app M N) T'          :- type Gamma M (arrow T T'),
                                    type Gamma N T.
type Gamma (lam X M) (arrow T T') :- type (pair X T)::Gamma M T'.
```

where `X` is a (logical) variable, then we get incorrect answers for λ-terms such as $\lambda x.\lambda x.(x\ x)$. For this untypable term, the naïve implementation erroneously returns two possible types: $\tau \to (\tau \to \sigma) \to \sigma$ and $(\tau \to \sigma) \to \tau \to \sigma$. Although this problem can be fixed by judicious use of cut, first-order terms are unwieldy for implementing relations about syntax with binders correctly.

The traditional solution for this problem is to use a higher-order logic programming language, such as λProlog (see for example [2]), with higher-order terms. Such languages provide elegant mechanisms for dealing with binders, but they also introduce new problems: unification for higher-order terms is in general undecidable and if unifiers exist for a problem they may not be subsumed by a single most general unifier. Furthermore, inference rules such as the ones given above are specified using meta-variables (often also referred to as context-variables, which can be substituted with any term); using higher-order unification one needs to replace these meta-variables with function variables and moreover has to explicitly specify the parameters the function variables depend on. This leads to encodings that are quite different in form from their intuitive pen-and-paper counterparts since there dependences are left implicit as much as possible.

In this paper we shall present a logic programming language called αProlog in which one can implement the inference rules given above correctly as follows:

```
type Gamma (var X) T              :- member (pair X T) Gamma.
type Gamma (app M N) T'           :- type Gamma M (arrow T T'),
                                     type Gamma N T.
type Gamma (lam x.M) (arrow T T') / x#Gamma
                                  :- type (pair x T)::Gamma M T'.
```

Posing the queries

(a) `type nil (lam x.(lam x. (var x))) A` and
(b) `type nil (lam x.(lam x. (app (var x) (var x))) B`

to αProlog produces the unique answer [`A` := `arrow T (arrow T' T')`] for (a), and no answer for (b).

Two novel features of αProlog are illustrated by the third clause of `type`. First, *abstractions* are a built-in notion of αProlog; thus we can write `(lam x.M)` in which `x` is taken from the syntactic category of names, `M` is a variable and `lam` a function symbol. This term is intended to unify with any λ-abstraction, for example `(lam x.var x)`, `(lam y.app (var x) (var y))` and so on, where in the first case the unifier is the capturing substitution [`M` := `var x`]. Note that if we had used higher-order unification, then `(lam x.M)` does *not* unify with `(lam x.var x)`, because there is no capture-avoiding substitution that makes both terms equal. Also note that in αProlog binders have concrete names, which are not restricted to the scope of the abstraction. These names can be therefore used in the body of the clause, for example to add `(pair x T)` to the context `Gamma`. In contrast, in higher-order abstract syntax binders are anonymous and cannot escape their scope. So, in λProlog, a binder from the head cannot be used in the body of a clause.

Second, *freshness constraints*, here `x#Gamma`, can be attached to the heads of clauses; the freshness constraint `x#Gamma` ensures that the variable `Gamma` is not instantiated with a term containing `x` freely. Since the third clause is intended to implement the type inference rule for λ-abstractions, its operational behaviour is given by: choose fresh names for `Gamma`, `x`, `M`, `T` and `T'` (this is standard in Prolog-like languages), unify the head of the clause with the goal formula, apply the resulting unifier to the body of the clause and make sure that `Gamma` is not substituted with a term that contains free occurrences of the fresh name we have chosen for `x`.

2 Nominal terms and nominal unification

To calculate unifiers, αProlog employs the nominal unification algorithm of Urban et al. [3], which produces unifiers that make two terms equal modulo α-equivalence. Nominal unification retains the main advantages of first-order unification: all unification problems are decidable and solvable problems possess most general unifiers. It makes use of the notion of possibly capturing substitutions, i.e. substitutions that allow `M` to be substituted with `(var x)` in the term `(lam x.M)` without introducing a new name for the binder `x`, and the notion of name swapping. A name swapping is a pair of the form (`x y`), and if applied for example to the term

```
lam x. app (lam x. var x)(app (var y) (var z))
```

gives

```
lam y. app (lam y. var y)(app (var x) (var z)) .
```

As can be seen the name swapping acts on all occurrences of `x` and `y` regardless whether they are bound, free or binding. The crucial point about name swapping is that it preserves α-equivalence. For instance, consider `lam y.var x` and `lam z.var x`, which are α-equivalent. Swapping `x` and `y` results in `lam x.var y` and `lam z.var y`, which are still α-equivalent, but naïvely substituting `y` for `x` results in variable capture in the first term.

Swappings are employed in the nominal unification algorithm when abstractions are unified. For example if one unifies the term `lam x.M` with `lam y.var y`, then `M` will be instantiated with the term `var x` which can be obtained by applying the swapping (`x y`) to `var y` (written as (`x y`)$\cdot$`var y`). However care needs to be taken if `x` occurs freely in the term to which the swapping is applied. For example the unification problem

$$\texttt{lam}\,\texttt{x.M} \approx ?\ \texttt{lam}\,\texttt{y.app}\,(\texttt{var}\,\texttt{y})\,(\texttt{var}\,\texttt{x})$$

does not have any solution (just replacing `M` by the 'swapped' version of `app` (`var y`) (`var x`) does not yield two α-equivalent terms). This problem is solved in nominal unification by freshness constraints, which ensure that possible solutions respect α-equivalence. So for solving the unification problem given above one needs to solve

$$\texttt{M} \approx ?\ (\texttt{x}\,\texttt{y}) \cdot \texttt{app}\,(\texttt{var}\,\texttt{y})\,(\texttt{var}\,\texttt{x}) \quad \text{and} \quad \texttt{x}\ \#?\ \texttt{app}\,(\texttt{var}\,\texttt{y})\,(\texttt{var}\,\texttt{x})$$

where the first problem asks whether `M` and `app` (`var x`) (`var y`) unify, and the second problem asks whether `x` is fresh for `app` (`var y`) (`var x`), i.e. `app (var y) (var x)` cannot contain any free occurrence of `x`. Since the latter does not hold, the unification problem fails—as expected.

While in the examples above the variable `M` is unified with ground terms only, nominal unification also solves more general unification problems involving open terms. For this the notion of freshness constraint needs some slight extension—for the details we refer the reader to [3] which also includes references about a formal verification of the correctness of the nominal unification algorithm using the proof assistant Isabelle.

3 αProlog examples

Considering the first experience we gained with αProlog there is good evidence that abstractions and freshness constraints from nominal unification provide useful programming constructs. Consider for example two of the transformations required for calculating a prenex-normal form of a formula

$$\forall x.P \wedge \forall x.Q \Longrightarrow \forall x.(P \wedge Q) \quad \text{and} \quad (\forall x.P) \wedge Q \Longrightarrow \forall x.(P \wedge Q)$$

where in the second transformation Q must not contain any free occurrence of x. These transformations can be implemented as the following clauses.

```
prenex (and (forall x.P) (forall x.Q)) (forall x.(and P Q)).
prenex (and (forall x.P) Q) (forall x.(and P Q)) / x#Q.
```

In the first clause, the use of nominal unification allows, roughly speaking, to synchronise the two binders. In effect this clause is applicable for any term of the form `(and (forall y....) (forall z....)`. What is pleasant about αProlog is that in this clause no explicit renaming of the binders is required: it will be done implicitly by the swapping operation of the unification algorithm. In the second clause, the freshness constraint `x#Q` makes sure that the side-condition placed upon the second transformation holds. In contrast, in a higher-order abstract syntax encoding, we cannot explicitly say that a variable must not occur in a term; instead we can only say that a variable may occur in a term. So, for example, we would have to make the potential dependences on `x` explicit everywhere in λProlog. In case of the prenex transformation, this results into the following λProlog clauses:

```
prenex (and (forall x\P x) (forall x\Q x)) (forall x\ (and (P x) (Q x))).
prenex (and (forall x\P x) Q) (forall x\(and (P x) Q)).
```

Another αProlog example is given below. It implements the capture-avoiding substitution of a variable in a λ-term[3].

```
id X X.
subst (var X) X T T.
subst (var Y) X T (var Y)      :- 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 y.M) X T (lam y.M')  /  y#T,X
                               :- subst M X T M'.
```

Given this program, posing the query `subst (lam (x. var y)) y (var x) M` yields the result [`M` := `lam x'.var x`] where `x'` is chosen freshly during proof search.

Our final example is the definition of evaluation for λ-terms. We first define a relation which expresses β-reduction:

```
beta (app (lam (x.E)) E') R :- subst E x E' R.
```

Now we define relations `step` that express one-step reduction of λ-terms, and `steps`, the reflexive, transitive closure of `step`:

```
step M M'                        :- beta M M'.
step (app M N) (app M' N)        :- step M M'.
step (app M N) (app M N')        :- step N N'.
step (lam (x.M)) (lam (x.M'))    :- step M M'.

steps M M.
steps M M''                      :- step M M', steps M' M''.
```

From the experience we gained so far, all inference rules involving λ-terms can in αProlog be implemented nearly 'one-to-one'. We wish to stress, however, that αProlog is not limited to defining relations on λ-terms. We have chosen the λ-calculus as an example because it well-known and the issues raised by binding and capture-avoiding substitution are widely appreciated. However, αProlog can also be used to write programs about languages with other forms of binding, provided these forms admit α-equivalence. Some examples include concurrency calculi such as the π-calculus and the Calculus of Mobile Ambients, in which names can be bound but are not subject to substitution by other (process) terms, only renaming. Another example is modeling imperative languages like C, in which variables serve a dual role as references to memory cells and their contents. Also, freshness constraints and fresh-name generation may be of interest in their own right, for example in modeling fresh-nonce generation in security protocols.

4 Conclusion

Currently, we have implemented a prototype interpreter for αProlog that permits user-defined datatypes and polymorphically typed Horn clauses. Work on verifying the correctness of αProlog's proof search algorithm relative to Nominal Logic and on optimizing the nominal unification algorithm is in progress. In the future, we plan to adapt more advanced logic program analyses for αProlog programs, such as mode, determinism, and termination analyses. Many interesting properties of languages can be framed in terms of such properties on relations; for example unicity and decidability of typechecking can be expressed in terms of determinism and termination of an appropriate moding of the typing relation. Eventually we wish to support general inductive theorem

[3] `not` is the usual negation-by-failure.

proving about αProlog programs. We believe that this will lead to better tools for computer-assisted reasoning about properties of languages.

Acknowledgements: Dominik Wee implemented a preliminary version of αProlog as a thesis project. We thank Marinos Georgiades who invited us to write this paper for the CoLogNet newsletter.

References

1. J. Cheney and C. Urban. System description: alpha-Prolog, a fresh approach to logic programming modulo alpha-equivalence. In J. Levy, M. Kohlhase, J. Niehren, and M. Villaret, editors, *Proceedings of the 17th International Workshop on Unification, UNIF'03*, pages 15–19, Valencia, Spain, June 2003. Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia. Technical Report DSIC-II/12/03.
2. Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming.* Oxford, 1986.
3. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Godel Colloquium (CSL'03 & KGC)*, Lecture Notes in Computer Science, Vienna, Austria, 2003. Springer-Verlag. To appear.