System Description: αProlog, a Fresh Approach to Logic Programming Modulo α-Equivalence

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. αProlog is a prototype logic programming language with a built-in notion of binders and unification modulo α-equivalence. It is based on a mild extension of first-order Horn formulae: instead of the usual first-order terms and first-order unification, αProlog uses nominal terms and nominal unification introduced in [3]. In this paper, we give three examples that demonstrate the advantages of αProlog and describe our current implementation.

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

$$\frac{x:A \in \Gamma}{\Gamma \rhd x:A} \qquad \frac{\Gamma \rhd M:A \to B \quad \Gamma \rhd N:A}{\Gamma \rhd MN:B} \qquad \frac{\{x:A\} \cup \Gamma \rhd M:B}{\Gamma \rhd \lambda x.M:A \to B}$$

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 λ-abstraction typing rule with the clause

```
type Gamma (lam X M) (arrow A B) :- type (pair X A)::Gamma M B.
```

where `X` is a (logical) variable, then we get incorrect answers for λ-terms such as $\lambda x.\lambda x.(x\ x)$. For this untypeable 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 [1]), 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.

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

```
type Gamma (var X) A :- member (pair X A) Gamma.
type Gamma (app M N) B :- type Gamma M (arrow A B), type Gamma N A.
type Gamma (lam x.M) (arrow A B) / x#Gamma :- type (pair x A)::Gamma M B.
member A A::Tail.
member A B::Tail :- member A Tail.
```

Two novel features of αProlog are illustrated by the third clause. First, λ-abstractions are translated to *nominal terms* which include an *abstraction* construct `a.t`. Thus, `(lam x.M)` is intended to unify with any λ-abstraction, for example `(lam x.var x)`, `(lam y.app (var x) (var y))` and so on. Note that binders in nominal terms have concrete values that are not restricted to the scope of the abstraction. These values can be therefore used in the body of the clause, for example to add `(pair x A)` 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 behavior is given by: choose fresh names for `Gamma`, `x`, `M`, `A` and `B` (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`.

To calculate unifiers, αProlog employs the nominal unification algorithm of Urban et al. (see [3] for the details). Nominal unification retains the main advantages of first-order unification: all unification problems are decidable and solvable problems possess most general unifiers. Moreover, the nominal unification algorithm produces unifiers that make two terms equal modulo α-equivalence. For example if we unify `(lam x.M)`, where `M` is a logical variable, with any term of the form `(lam y.t)`—`t` being the representation of an arbitrary λ-term—then `M` will be instantiated such that all occurrences bound by `y` in `t` are bound in `M` by `x`. This is achieved by two means: First, bound names are renamed via a swapping operation. Unlike renaming via substitutions, swapping preserves

α-equivalence: consider for instance the α-equivalent terms $\lambda y.x\ y$ and $\lambda z.x\ z$. Swapping all occurrences of x and y in these terms gives $\lambda x.y\ x$ and $\lambda z.x\ z$, which are again α-equivalent, whereas renaming free occurrences of x to y results in capture. Second, unification may generate freshness constraints that ensure that possible solutions respect α-equivalence. For example if one wishes to solve the unification problem `(lam x.t)` $\approx ?$ `(lam  x'.t')`, then one needs to solve

$$\texttt{t} \approx ?\ (\texttt{x}\,\texttt{x'})\cdot\texttt{t'} \quad \text{and} \quad \texttt{x}\ \#?\ \texttt{t'}$$

where the first problem asks whether `t` and $(\texttt{x}\,\texttt{x'})\cdot\texttt{t'}$ unify, where $(\texttt{x}\,\texttt{x'})\cdot\texttt{t'}$ is `t'` with all occurrences of `x` and `x'` swapped. The second problem asks whether `x` is fresh for `t'`, i.e. `t'` cannot contain any free occurrence of `x`. Posing the queries

(a) `type nil (lam x.(lam x. (var x))) T` and
(b) `type nil (lam x.(lam x. (app (var x) (var x))) T'`

to αProlog produces the unique answer `T -> arrow A (arrow B B)` for (a), and no answer for (b).

One important point in using nominal unification is that binders have explicit names. This has the pleasant consequence that manipulations involving terms with binders can be expressed very elegantly in αProlog. 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 synchronize 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.

Another 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, y#X :- subst M X T M'.
```

Given this program, posing the query `subst (lam (x. var y)) y (var x) R` yields the result `R -> lam x43.var x`.

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

$$
\begin{array}{ll}
\tau ::= \delta \mid \alpha \mid \tau \times \cdots \times \tau \mid \langle\alpha\rangle\tau & t ::= c \mid f\ t_1 \cdots t_n \mid (t_1, ..., t_n) \mid X \mid a \mid a.t \\
\sigma ::= \mathsf{prop} \mid \delta \mid \tau \to \sigma & C ::= t \ / \ \nabla :- G \\
\kappa ::= \mathsf{atom} \mid \mathsf{data} \mid \sigma & G ::= t \mid \top \mid G \wedge G \mid G \vee G
\end{array}
$$

Fig. 1. αProlog syntax summary

2 Syntax

Figure 1 summarizes the abstract syntax of αProlog. In αProlog, the syntactic entities that can be bound and renamed are called *atoms*, as opposed to (logical) *variables*, which can be instantiated with other terms via substitution. An αProlog program is a sequence of signature declarations of data types, program clauses, and queries. A signature declaration is of the form $sym : \kappa$, where κ is either atom or data (declaring sym to be an atom type α or data type δ), or a "symbol type" σ (declaring sym to be a constant or function symbol). Propositions and relation symbols have types of the form $\tau_1 \to \ldots \to \tau_n \to \mathsf{prop}$. Types τ consist of base atom types, data types, n-tuples, and abstraction types $\langle\alpha\rangle\tau$.

The following signature introduces data types `exp` and `type` and an atom type `id`, and defines function symbols representing λ-terms and simple types:

```
id   : atom.    var   : id -> exp.
exp  : data.    lam   : <id>exp -> exp.
type : data.    app   : exp -> exp -> exp.
                arrow : type -> type -> type.
```

The terms include constants, function applications, tuples, atoms a and atom-abstractions $a.t$. The program clauses are of the form $t \ / \ \nabla :- G$, where G is a goal, t is a term of type prop, and ∇ is a set of *freshness constraints* $a \# X$ which must be satisfied in order for the rule to apply. If ∇ is empty, we omit it and the / symbol; similarly, if G is $\top$, we omit it and the $:-$ symbol. Goals include $\top$ (true), conjunctions, disjunctions, and atomic propositions t.

As usual, clauses are implicitly universally quantified over all free logical variables `X`. Variables can be used in place of atom occurrences but not in abstractions: i.e., `var X` is a legal term whereas `lam X.(var X)` is not. Instead, we write `lam x.(var x)`, which will unify with any α-equivalent λ-term.

3 Proof search

The main differences between αProlog and standard logic programming languages are (1) it uses nominal unification to decide whether a goal and a clause head are unifiable, (2) it interprets freshness constraints in clauses as obligations to generate fresh atoms, and (3) it answers queries with a substitution plus a set of freshness constraints on some variables in the substitution. We shall present a nondeterministic transition semantics of proof search in αProlog.

A program state is a tuple $(\Psi, G, \nabla, \theta)$, where Ψ is a set of program clauses, G a goal, ∇ a set of freshness constraints, and θ a substitution. The result of proof search is a pair $\langle\theta, \nabla\rangle$, where θ is a substitution instantiating some variables of G to terms, and ∇ is a set of freshness constraints of the form $a \# X$.

If G is a non-atomic formula, then proof search is relatively standard, as shown below:

1. $(\Psi, \top, \theta, \nabla)$ succeeds with answer $\langle\theta, \nabla\rangle$.
2. $(\Psi, G_1 \wedge G_2, \theta, \nabla)$ steps to $(\Psi, \theta' G_2, \theta' \circ \theta, \nabla')$, provided $(\Psi, G_1, \theta, \nabla)$ succeeds with answer $\langle\theta', \nabla'\rangle$.
3. $(\Psi, G_1 \vee G_2, \theta, \nabla)$ steps to $(\Psi, G_i, \theta, \nabla)$ for $i \in \{1, 2\}$.

Otherwise, if the the goal formula is atomic, then

1. Let $t' \ / \ \nabla' \ {:-}\ G'$ be an element of Ψ.
2. Let $t^* \ / \ \nabla^* \ {:-}\ G^*$ be the 'freshened' version of $t' \ / \ \nabla' \ {:-}\ G'$, i.e. the clause where all atoms mentioned in ∇' and all variables are renamed with new names (the latter is standard in logic programming)
3. If t^* unifies with t subject to the constraints ∇^*, resulting in $\langle\theta'', \nabla''\rangle$, then $(\Psi, t, \theta, \nabla)$ steps to $(\Psi, \theta'' G^*, \theta'' \circ \theta, \nabla'')$.

That is, to solve an atomic goal by resolving it with a clause, we first replace atoms mentioned in ∇ with fresh atoms, then unify the goal and the head of the clause, while also checking that all the constraints are satisfied. Unification may produce new constraints that cannot be resolved yet; if so, these constraints are propagated through proof search until they can be resolved.

4 Conclusions

We have developed a proof-of-concept implementation of αProlog and are in the process of scaling it up to realistic examples. There are many directions for future work. For example, the complexity of nominal unification remains to be investigated. On the theoretical side, we intend to relate proof search in αProlog to a suitable notion of uniform proofs in Nominal Logic [2].

Acknowledgments: Dominik Wee implemented a preliminary version of αProlog as a thesis project.

References

1. G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Oxford University Press, 1998.
2. A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Proceedings of the 2001 International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan*, volume 2215 of *LNCS*, pages 219–242. Springer-Verlag, 2001.
3. C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. Submitted. `http://www.cl.cam.ac.uk/~cu200/Unification/`.