αProlog: A Logic Programming Language with Names, Binding and α-Equivalence

James Cheney[1], Christian Urban[2]

[1] Cornell University (jcheney@cs.cornell.edu)
[2] University of Cambridge (cu200@cl.cam.ac.uk)

Abstract. There are two well-known approaches to programming with names, binding, and equivalence up to consistent renaming: representing names and bindings as concrete identifiers in a first-order language (such as Prolog), or encoding names and bindings as variables and abstractions in a higher-order language (such as λProlog). However, both approaches have drawbacks: the former often involves stateful name-generation and requires manual definitions for α-equivalence and capture-avoiding substitution, and the latter is semantically very complicated, so reasoning about programs written using either approach can be very difficult. Gabbay and Pitts have developed a new approach to encoding abstract syntax with binding based on primitive operations of name-swapping and freshness. This paper presents αProlog, a logic programming language that uses this approach, along with several illustrative example programs and an operational semantics.

1 Introduction

Names, binding, α-equivalence, and capture-avoiding substitution are endemic phenomena in logics and programming languages. The related concepts of name freshness, fresh name generation and equivalence up to consistent renaming also appear in many other domains, including state identifiers in finite automata, nonces in security protocols, and channel names in process calculi. Dealing with names is therefore an important practical problem in meta-programming, and there are a variety of approaches to doing so, involving different tradeoffs [2, 3, 7, 12, 15, 19]. The following are important desiderata for such techniques:

- *Convenience*: Basic operations including substitution, α-equivalence, and fresh name generation should be built-in.
- *Simplicity*: The semantics of the meta-language should be as simple as possible in order to facilitate reasoning about programs.
- *Abstraction*: Low-level implementation details concerning names should be taken care of by the meta-language and hidden from the programmer.
- *Faithfulness/Adequacy*: Object terms should be in bijective correspondence with the values of some meta-language type.

There are several established techniques for programming with names and binding, which we categorize as *first-order abstract syntax* (FOAS) and *higher-order abstract syntax* (HOAS).

In first-order abstract syntax, object languages are encoded using first-order terms (e.g. Prolog terms or ML datatypes). Names are encoded using a concrete datatype var such as strings, and binders are encoded using first-order function symbols like $\mathit{lam} : \mathit{var} \times \mathit{exp} \to \mathit{exp}$. FOAS has several disadvantages: the encoding does not respect α-equivalence, damaging adequacy; fresh names are often generated using side-effects, complicating the semantics; and operations like α-equivalence and substitution must be implemented manually. Nameless encodings like de Bruijn indices [2] ameliorate some, but not all, of these problems.

In higher-order abstract syntax [15], object languages are encoded using higher-order terms (e.g. λ-terms in λProlog [13]). HOAS comes in “deep” and “shallow” flavors. In deep HOAS, names are encoded as meta-language variables and binders are encoded with meta-language λ-abstraction using higher-order function symbols like $\mathit{lam} : (\mathit{exp} \to \mathit{exp}) \to \mathit{exp}$. Capture-avoiding substitution and α-equivalence need only be implemented once, in the meta-language, and can be inherited by all object languages. However, because of the presence of types like exp that are defined via negative recursion, the semantics of deep HOAS is complex [8] and inductive proof and recursive definition principles are very difficult to develop. In shallow HOAS [3], names are encoded using a concrete type var, but binders are encoded as λ-abstractions using constructors like $\mathit{lam} : (\mathit{var} \to \mathit{exp}) \to \mathit{exp}$. In this approach, α-equivalence is still built-in, but substitution must be defined. Also, shallow HOAS encodings may not be adequate because of the presence of *exotic* terms, or closed terms of type exp which do not correspond to any object term; additional well-formedness predicates are needed to recover adequacy.

Recently, Gabbay and Pitts developed a novel approach to encoding names and binding [5], based on taking *name-swapping* and *freshness* as fundamental operations on names. This approach has been codified by Pitts as a theory of first-order logic called *nominal logic* [16], in which names are a first-order abstract data type admitting only swapping, binding, and equality and freshness testing operations. Object language variables and binding can be encoded using names x, y and *name-abstractions* $\mathsf{x}\backslash t$, which are considered equal up to α-equivalence. For example, object variables x and binders $\lambda x.t$ can be encoded as nominal terms $\mathit{var}(\mathsf{x})$ and abstractions $\mathit{lam}(\mathsf{x}\backslash t)$ where $\mathit{var} : \mathit{id} \to \mathit{exp}$ and $\mathit{lam} : \mathit{id}\backslash\mathit{exp} \to \mathit{exp}$. Unlike FOAS encodings, the use of name abstractions ensures faithfulness and adequacy; for example, $\mathit{lam}(\mathsf{x}\backslash\mathit{var}(\mathsf{x})) = \mathit{lam}(\mathsf{y}\backslash\mathit{var}(\mathsf{y}))$.

We refer to this approach to programming with names and binding as *nominal abstract syntax* (NAS). NAS provides convenient operations such as α-equivalence and fresh name generation for free, while remaining semantically simple, requiring neither recursive types nor state. Furthermore, names are sufficiently abstract that the low-level details of name generation can be hidden from the programmer, and exotic terms are not possible in NAS encodings.

This paper presents αProlog, a logic programming language which supports nominal abstract syntax. In the rest of this paper, we describe αProlog, show how several interesting, nontrivial languages (including the π-calculus) can be encoded using NAS in αProlog, and discuss briefly its unification algorithm (due to Urban, Pitts, and Gabbay [20]) and operational semantics. We conclude with a discussion of related languages and future work.

2 Language Overview

2.1 Syntax

The term language of αProlog consists of *(nominal) terms*, constructed according to the grammar

$$t ::= X \mid \mathsf{n} \mid \mathsf{n}\backslash t \mid (\mathsf{n}\ \mathsf{m})t \mid f(\bar{t})$$

where X is a *(logic) variable*, f is a *function symbol* (we write $\bar{t}$ to denote a (possibly empty) sequence $(t_1, \ldots, t_n)$), and n, m are *names*. By convention, function symbols are lower case, logic variables are capitalized, and names are printed using the sans-serif typeface. We shall refer to a term of the form $f(\bar{t})$ as an atomic term. Terms of the form $\mathsf{n}\backslash t$ are called *abstractions*, and terms of the form $(\mathsf{n}\ \mathsf{m})t$ are called *(suspended) transpositions*. Transpositions have higher precedence than abstractions. Variables cannot be used to form abstractions and transpositions, i.e., $X\backslash t$ and $(X\ Y)t$ are not legal terms.

αProlog has a ML-like polymorphic type system. Types are classified into two *kinds*: **type**, the kind of all types, and **name_type**, the kind of types inhabited only by names. Types classify terms, and include atomic type constructor applications $c(\sigma_1, \ldots, \sigma_n)$ as well as type variables α and abstraction types $\nu\backslash\sigma$. In an abstraction, the kind of ν must be **name_type**. Type constructor and uninterpreted function symbol declarations are of the form as $c : (\overline{\kappa}) \to \kappa'$ and $f : (\overline{\sigma}) \to \sigma'$, where κ and σ indicate kinds and types respectively. Atomic formulas are declared as **pred** $p(\overline{\sigma})$ and interpreted function symbols as **func** $f(\overline{\sigma}) = \sigma'$. Type abbreviations can be made with the declaration **type** $c(\overline{\alpha}) = \sigma$. The latter three declaration forms are loosely based on Mercury syntax [18]. We assume built-in type and function symbols $\cdot \times \cdot : (\mathbf{type}, \mathbf{type}) \to \mathbf{type}$, $(\cdot, \cdot) : (\alpha, \beta) \to \alpha \times \beta$ for pairs and $[\cdot] : \mathbf{type} \to \mathbf{type}$, $[] : [\alpha]$, and $[\cdot|\cdot] : (\alpha, [\alpha]) \to [\alpha]$ for lists. The result type of an uninterpreted function symbol may not be a built-in type or a **name_type**.

Atomic formulas A are terms of the form $q(t_1, \ldots, t_n)$, where p, q are *relation symbols*. *Goals* (or *queries*) are given by the grammar

$$G ::= \mathbf{true} \mid G_1 \wedge G_2 \mid A \mid t \# u \mid t = u$$

where $t \# u$ is a freshness formula, and $t = u$ is an equality formula. In $t \# u$, the term t must be of some name type $\nu : \mathbf{name\_type}$, whereas u may be of any type; in $t = u$, both t and u must be of the same type. Program clauses include Horn clauses of the form $A :- G$ and function-definition clauses of the form $a = t :- G$, which introduce a (conditional) rewrite rule for an atomic term $f(t_1, \ldots, t_n)$ with an interpreted head symbol f. We abbreviate clauses $A :- \mathbf{true}$ and $a = t :- \mathbf{true}$ as simply A and $a = t$. We write conjunctions as 'G_1, G_2' instead of '$G_1 \wedge G_2$' within program text.

By convention, constant symbols are function symbols applied the empty argument list; we write c instead of $c()$, and $c : \tau$ instead of $c : () \to \tau$. This also applies to propositional and type constants. We write $V(\cdot)$ and $N(\cdot)$ for the variables or names of a term or formula. Observe that $N(\cdot)$ includes all occurrences of names in t, even abstracted ones, hence $N(\mathsf{x}\backslash\mathsf{x}) = \{\mathsf{x}\}$. We say a nominal term e is *ground* when $V(e) = \emptyset$; names may appear in ground terms, so $f(X, Y)$ is not ground but $f(\mathsf{x}, \mathsf{y})$ is.

2.2 Equality and freshness

Figure 1 shows the axioms of equality and freshness for ground nominal terms (based on [16, 20]). The swapping axioms (S_1)–(S_5) describe the behavior of swapping. From now on, we assume that all terms are normalized with respect to these axioms (read right-to-left as rewrite rules), so that swaps are not present in ground terms and are present only surrounding variables in non-ground terms.

The next two axioms (A_1), (A_2) define equality for abstractions. The first axiom is a simple congruence property. The second guarantees that abstractions are equal "up to renaming". Two abstractions of different names $\mathsf{x}\backslash t, \mathsf{y}\backslash u$ are equal just in case their bodies are equal up to swapping the names $(t = (\mathsf{x}\ \mathsf{y})u)$ and $\mathsf{x} \mathrel{\#} u$ (or symmetrically, $\mathsf{y} \mathrel{\#} t$; the two conditions are equivalent if $t = (\mathsf{x}\ \mathsf{y})u$). For example, $\mathsf{x}\backslash g(\mathsf{x}) = \mathsf{y}\backslash g(\mathsf{y})$ and $\mathsf{x}\backslash f(\mathsf{x}, \mathsf{y}) = \mathsf{z}\backslash f(\mathsf{z}, \mathsf{y})$, but $\mathsf{x}\backslash f(\mathsf{x}, \mathsf{y}) \neq \mathsf{y}\backslash f(\mathsf{y}, \mathsf{x})$ because $\mathsf{x} \mathrel{\#} f(\mathsf{y}, \mathsf{x})$ fails.

The freshness axioms (F_1)–(F_5) describe the freshness relation. Intuitively, $\mathsf{x} \mathrel{\#} t$ means "name x does not appear unbound in t". For example, it is never the case that $\mathsf{x} \mathrel{\#} \mathsf{x}$, whereas any two distinct names are fresh $(\mathsf{x} \mathrel{\#} \mathsf{y})$. Moreover, freshness passes through function symbols (in particular, any name is fresh for any constant). The abstraction freshness rules are more interesting: $\mathsf{x} \mathrel{\#} \mathsf{x}\backslash t$ is unconditionally valid because any name is fresh for a term in which it is immediately abstracted, whereas if x and y are different names, then $\mathsf{x} \mathrel{\#} \mathsf{y}\backslash t$ just in case $\mathsf{x} \mathrel{\#} t$.

3 Example: the λ-calculus

The prototypical example of a language with variable binding is the λ-calculus. In αProlog, the syntax of λ-terms may be described with the following type and constructor declarations:

$(S_1)\ (\mathsf{n}\ \mathsf{m})\mathsf{n} = \mathsf{m} \quad (S_2)\ (\mathsf{n}\ \mathsf{m})\mathsf{m} = \mathsf{n}$	$(S_3)\ \mathsf{x} \mathrel{\#} \mathsf{n}, \mathsf{x} \mathrel{\#} \mathsf{m} \Rightarrow (\mathsf{n}\ \mathsf{m})\mathsf{x} = \mathsf{x}$
$(S_4)\ (\mathsf{n}\ \mathsf{m})f(\bar{t}) = f((\mathsf{n}\ \mathsf{m})\bar{t})$	$(S_5)\ (\mathsf{n}\ \mathsf{m})(\mathsf{x}\backslash t) = (\mathsf{n}\ \mathsf{m})\mathsf{x}\backslash(\mathsf{n}\ \mathsf{m})t$
$(A_1)\ t = u \Rightarrow \mathsf{n}\backslash t = \mathsf{n}\backslash u$	$(A_2)\ t = (\mathsf{n}\ \mathsf{m})u \wedge \mathsf{n} \mathrel{\#} u \Rightarrow \mathsf{n}\backslash t = \mathsf{m}\backslash u$
$(F_1)\ \mathsf{n} \neq \mathsf{m} \Rightarrow \mathsf{n} \mathrel{\#} \mathsf{m} \quad (F_2)\ \neg(\mathsf{n} \mathrel{\#} \mathsf{n})$	$(F_3)\ \bigwedge_{i=1}^{n} \mathsf{n} \mathrel{\#} t_i \Rightarrow \mathsf{n} \mathrel{\#} f(\bar{t})$
$(F_4)\ \mathsf{n} \mathrel{\#} \mathsf{n}\backslash t$	$(F_5)\ \mathsf{n} \mathrel{\#} t \Rightarrow \mathsf{n} \mathrel{\#} \mathsf{m}\backslash t$

Fig. 1. Ground equational and freshness theory

$$\begin{array}{lll} id : \textbf{name\_type}. & exp : \textbf{type}. & \\ var : id \to exp. & app : (exp, exp) \to exp. & lam : id\backslash exp \to exp. \end{array}$$

We make the simplifying assumption that the variables of object λ-terms are constants of type id. Then we can translate λ-terms as follows:

$$\ulcorner \mathsf{x} \urcorner = var(\mathsf{x}) \qquad \ulcorner e_1\ e_2 \urcorner = app(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \qquad \ulcorner \lambda \mathsf{x}.e \urcorner = lam(\mathsf{x}\backslash \ulcorner e \urcorner)$$

It is not difficult to verify that e is a λ-term if and only if $\ulcorner e \urcorner$ is a *closed* nominal term, i.e. $FV(\ulcorner e \urcorner) = \emptyset$, and we have that $e \equiv_\alpha e'$ if and only if $\ulcorner e \urcorner \approx \ulcorner e' \urcorner$.

Although capture-avoiding substitution is not a built-in operator in αProlog, it is easy to define via the conditional rewriting rules (where we use function-like notation inspired from the language Mercury):

$$\begin{array}{ll} \textbf{func}\ subst(exp, exp, id) & = exp. \\ subst(var(X), E, X) & = E. \\ subst(var(Y), E, X) & = var(Y) \coloneq X \# Y. \\ subst(app(E_1, E_2), E, X) & = app(subst(E_1, E, X), subst(E_2, E, X)). \\ subst(lam(\mathsf{y}\backslash E'), E, X) & = lam(\mathsf{y}\backslash subst(E', E, X)) \coloneq \mathsf{y} \# (X, E). \end{array}$$

Note the two freshness side-conditions: the constraint $X \# Y$ prevents the first and second clauses from overlapping; the constraint $\mathsf{y} \# (X, E)$ ensures capture-avoidance, by restricting the application of the fourth clause to when y is fresh for X and E. Despite these side-conditions, this definition is total and deterministic. Determinism is immediate: no two clauses overlap. Totality follows because, by nominal logic's *freshness principle*, the bound name y in $lam(\mathsf{y}\backslash E')$ can always be renamed to a fresh z chosen so that $\mathsf{z} \# (X, E)$. It is straightforward to prove that $subst(\ulcorner t \urcorner, \ulcorner t' \urcorner, \mathsf{x})$ coincides with the traditional capture-avoiding substitution on λ-terms.

We may define β-reduction and η-expansion relations:

$$\begin{array}{ll} \textbf{pred}\ beta(tm, tm). & beta(app(lam(\mathsf{x}\backslash E), E'), E'') \coloneq E'' = subst(E, E', \mathsf{x}). \\ \textbf{pred}\ eta(tm, tm). & eta(E, lam(\mathsf{x}\backslash app(E, var(\mathsf{x})))) \coloneq \mathsf{x} \# E. \end{array}$$

The usual side-condition on η-expansion is encoded in αProlog as $\mathsf{x} \# E$. Next we consider the problem of typechecking λ-terms. The syntax of types can be encoded as follows:

$$tid : \textbf{name\_type}. \qquad ty : \textbf{type}. \qquad tvar : tid \to ty. \qquad arr : (ty, ty) \to ty.$$

We define contexts ctx as lists of pairs of identifiers and types, and the 3-ary relation typ relating a context, term, and type:

$$\begin{array}{ll} \textbf{type}\ ctx = [id \times ty]. & \\ \textbf{pred}\ typ(ctx, tm, ty). & \\ typ(C, var(X), T) & \coloneq mem((X, T), C). \\ typ(C, app(E_1, E_2), T') & \coloneq typ(C, E_1, arr(T, T')), typ(C, E_2, T). \\ typ(C, lam(\mathsf{x}\backslash E), arr(T, T')) & \coloneq \mathsf{x} \# C, typ([(\mathsf{x}, T)|C], E, T'). \end{array}$$

The predicate $mem(\alpha, [\alpha])$ is the usual predicate for testing list membership. The side-condition $x \notin Dom(\Gamma)$ is translated to the freshness constraint $\mathsf{x} \# C$.

Two example queries demonstrating the behavior of this program are as follows (αProlog gives unique answers to all queries).

$$
\begin{array}{l}
?\!-\ X = subst(lam(\mathsf{y}\backslash var(\mathsf{x})), var(\mathsf{y}), \mathsf{x}).\\
X = lam(\mathsf{x}_{13}\backslash var(\mathsf{y}))\\
?\!-\ typ([], lam(\mathsf{x}\backslash lam(\mathsf{y}\backslash var(\mathsf{x})), T).\\
T = arr(arr(T_{112}, T_{125}), T_{125}),\ \ \mathsf{x}_{89} \# T_{112}\\
?\!-\ typ([], lam(\mathsf{x}\backslash lam(\mathsf{x}\backslash app(var(\mathsf{x}), var(\mathsf{x})))), T).\\
No.
\end{array}
$$

The constraint $\mathsf{x}_{89} \# T_{112}$ in the second answer indicates that a name x_{89} generated during execution must be fresh for T_{112}. Since x_{89} does not appear elsewhere in the query or answer, this constraint is trivially satisfiable; however, our implementation of αProlog does not yet take this into account.

3.1 Extending to the $\lambda\mu$-calculus

The $\lambda\mu$-calculus, invented by Parigot [14], extends the λ-calculus with *continuations* α; terms may be passed to continuations ($[\alpha]e$) and continuations may be bound ($\mu\alpha.e$). Intuitively, $\lambda\mu$-terms are proof terms for classical natural deduction, and μ-abstractions represent proofs by double negation. In addition to capture-avoiding substitution of terms for variables, the $\lambda\mu$-calculus introduces a capture-avoiding *replacement* operator $e'\{e/\alpha\}$ which replaces each occurrence of the pattern $[\alpha]e_0$ in e' with $[\alpha](e_0\ e)$. We give a variant of the $\lambda\mu$-calculus in Figure 2. In contexts Γ, the bar over the type of α indicates that it is not a value of type τ, but a continuation accepting the type τ.

We may extend the λ-calculus encoding with new types and term constructors for the $\lambda\mu$-calculus:

$$con : \mathbf{name\_type} \qquad pass : (con, exp) \to exp \qquad mu : con\backslash exp \to exp$$

Terms, Types, and Contexts

$$
\begin{array}{l}
e ::= x \mid (e\ e') \mid \lambda x.e \mid [\alpha]e \mid \mu\alpha.e\\
\tau ::= b \mid \tau \to \tau' \mid \bot\\
\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha : \overline{\tau}
\end{array}
$$

Replacement Operation

$$
\begin{array}{rcl}
x\{e/\alpha\} &=& x\\
(e_1\ e_2)\{t/\alpha\} &=& (e_1\{e/\alpha\}\ e_2\{e/\alpha\})\\
(\lambda y.e')\{e/\alpha\} &=& \lambda y.e'\{e/\alpha\}\\
([\alpha]e')\{e/\alpha\} &=& [\alpha](e'\{e/\alpha\}\ e)\\
([\beta]e')\{e/\alpha\} &=& [\beta](e'\{e/\alpha\}) \quad (\beta \neq \alpha)\\
(\mu\beta.e')\{e/\alpha\} &=& \mu\beta.e'\{e/\alpha\} \quad (\beta \notin FN(e, \alpha))
\end{array}
$$

Some Typing-Rules

$$
\frac{\alpha{:}\overline{\tau} \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash [\alpha]e : \bot}
\qquad
\frac{\Gamma \vdash e_1 : \bot \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \bot}
\qquad
\frac{\Gamma, \alpha{:}\overline{\tau} \vdash e : \bot \quad (\alpha \notin Dom(\Gamma))}{\Gamma \vdash \mu\alpha.e : \tau}
$$

Fig. 2. A slight variant of Parigot's $\lambda\mu$-calculus.

Again, it is very easy to show that ground *exp*-terms are in one-to-one correspondence with $\lambda\mu$-terms. Capture-avoiding substitution can be extended to $\lambda\mu$-terms easily. For replacement, we show the interesting cases for continuation applications and μ-abstractions:

$$\begin{array}{ll}
\textbf{func}\ \mathit{repl}(\mathit{exp}, \mathit{exp}, \mathit{con}) & = \mathit{exp}.\\
\mathit{repl}(\mathit{pass}(A, E'), E, A) & = \mathit{pass}(A, \mathit{app}(\mathit{repl}(E', E, A), E)).\\
\mathit{repl}(\mathit{pass}(B, E'), E, A) & = \mathit{pass}(B, \mathit{repl}(E', E, A)) \mathrel{:-} A \mathrel{\#} B.\\
\mathit{repl}(\mathit{mu}(\mathsf{b}\backslash E'), E, A) & = \mathit{mu}(\mathsf{b}\backslash \mathit{repl}(E', E, A)) \mathrel{:-} \mathsf{b} \mathrel{\#} (A, E).
\end{array}$$

Then the relevant β-contraction and η-expansion laws for μ can be stated as:

$$\begin{array}{l}
\mathit{beta}(\mathit{app}(\mathit{mu}(\mathsf{a}\backslash E'), E), \mathit{mu}(\mathsf{a}\backslash E'')) \mathrel{:-} \mathsf{a} \mathrel{\#} E, E'' = \mathit{repl}(E', E, \mathsf{a}).\\
\mathit{eta}(E, \mathit{mu}(\mathsf{a}\backslash \mathit{pass}(\mathsf{a}, E))) \mathrel{:-} \mathsf{a} \mathrel{\#} E.
\end{array}$$

The standard approach to typechecking $\lambda\mu$-terms is to use two contexts, Γ and Δ, for variable- and continuation-bindings respectively. We instead consider a single context with variable-bindings $x : \tau$ and continuation-bindings $\alpha : \overline{\tau}$. Therefore we modify the encoding of contexts slightly as follows:

$$\mathit{bind} : \textbf{type}. \quad \mathit{vb} : (\mathit{id}, \mathit{ty}) \to \mathit{bind} \quad \mathit{cb} : (\mathit{con}, \mathit{ty}) \to \mathit{bind} \quad \textbf{type}\ \mathit{ctx} = [\mathit{bind}].$$

Then the typechecking rules from the previous section may be adapted by replacing bindings (x, T) with $\mathit{vb}(\mathsf{x}, T)$, and adding three new rules:

$$\begin{array}{ll}
\mathit{typ}(C, \mathit{pass}(X, E), \mathit{bot}) & \mathrel{:-} \mathit{mem}(\mathit{cb}(X, T), C), \mathit{typ}(C, E, T).\\
\mathit{typ}(C, \mathit{app}(E, E'), \mathit{bot}) & \mathrel{:-} \mathit{typ}(C, E, \mathit{bot}), \mathit{typ}(C, E', T).\\
\mathit{typ}(C, \mathit{mu}(\mathsf{a}\backslash E), T) & \mathrel{:-} \mathsf{a} \mathrel{\#} C, \mathit{typ}([\mathit{cb}(\mathsf{a}, T)|C], E, \mathit{bot}).
\end{array}$$

The following query illustrates the typechecking for the term $\lambda x.\mu\alpha.(x\ (\lambda y.[\alpha]y))$ whose principal type corresponds to the classical double negation law.

$$\begin{array}{l}
?\!-\ \mathit{typ}([], \mathit{lam}(\mathsf{x}\backslash \mathit{mu}(\mathsf{a}\backslash \mathit{app}(\mathit{var}(\mathsf{x}), \mathit{lam}(\mathsf{y}\backslash \mathit{pass}(\mathsf{a}, \mathit{var}(\mathsf{y})))))), T).\\
T = \mathit{arr}(\mathit{arr}(\mathit{arr}(T_{162}, \mathit{bot}), \mathit{bot}), T_{162})
\end{array}$$

4 Example: the π-calculus

The π-calculus is a calculus of concurrent, mobile processes. Its syntax (following Milner, Parrow, and Walker [11], but omitting definitions) is described by the grammar rules shown in Figure 3. The symbols $x, y, \ldots$ are *channel names*. The inactive process $\mathsf{0}$ is inert. The $\tau.p$ process performs a *silent action* τ and then does p. Parallel composition is denoted $p|q$ and nondeterministic choice by $p+q$. The process $x(y).p$ inputs a channel name from x, binds it to y, and then does p. The process $\overline{x}y.p$ outputs y to x and then does p. The match operator $[x = y]p$ is p provided $x = y$, but is inactive if $x \neq y$. The restriction operator $(y)p$ restricts y to p. Parenthesized names (e.g. y in $x(y).p$ and $(y)p$) are binding, and $fn(p)$, $bn(p)$ and $n(p)$ denote the sets of free, bound, and all names occurring in p. Capture-avoiding renaming is written $t\{x/y\}$.

$$\begin{array}{ll} \text{Process terms} & p ::= 0 \mid \tau.p \mid p|q \mid p+q \mid x(y).p \mid \overline{x}y.p \mid [x=y]p \mid (x)p \\ \text{Actions} & a ::= \tau \mid x(y) \mid \overline{x}y \mid \overline{x}(y) \end{array}$$

$$\tau.p \xrightarrow{\tau} p \qquad \frac{p \xrightarrow{a} p' \quad bn(a) \cap fn(q) = \emptyset}{p|q \xrightarrow{a} p'|q} \qquad \frac{p \xrightarrow{\overline{x}y} p' \quad q \xrightarrow{x(z)} q'}{p|q \xrightarrow{\tau} p'|q'\{y/z\}}$$

$$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \overline{x}y.p \xrightarrow{\overline{x}y} p \qquad \frac{w \notin fn((z)p)}{x(z).p \xrightarrow{x(w)} p\{w/z\}} \qquad \frac{p \xrightarrow{a} p'}{[x=x]p \xrightarrow{a} p'}$$

$$\frac{p \xrightarrow{\overline{x}(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \frac{p \xrightarrow{a} p' \quad y \notin n(a)}{(y)p \xrightarrow{a} (y)p'} \qquad \frac{p \xrightarrow{\overline{x}y} p' \quad y \neq x \quad w \notin fn((y)p)}{(y)p \xrightarrow{\overline{x}(w)} p'\{w/y\}}$$

Fig. 3. The π-calculus

Milner et al.'s original operational semantics (shown in Figure 3, symmetric cases omitted) is a labeled transition system with relation $p \xrightarrow{a} q$ indicating "p steps to q by performing action a". Actions τ, $\overline{x}y$, $x(y)$, $\overline{x}(y)$ are referred to as *silent*, *free output*, *input*, and *bound output* actions respectively; the first two are called *free* and the second two are called *bound* actions. For an action a, $n(a)$ is the set of all names appearing in a, and $bn(a)$ is empty if a is a free action and is $\{y\}$ if a is a bound action $x(y)$ or $\overline{x}(y)$.

Much of the complexity of the rules is due to the need to handle *scope extrusion*, which occurs when restricted names "escape" their scope because of communication. In $((x)\overline{a}x.p)|(a(z).z(x).0) \xrightarrow{\tau} (x')(p|x'(x).0))$, for example, it is necessary to "freshen" x to x' in order to avoid capturing the free x in $a(z).z(x).0$. Bound output actions are used to lift the scope of an escaping name out to the point where it is received.

These rules can be translated directly into αProlog (see Figure 4). Processes and actions can be encoded using the following syntax:

$$\begin{array}{llll} \mathit{chan} : \textbf{name\_type}. & \mathit{proc} : \textbf{type}. & \mathit{ina} : \mathit{proc}. & \mathit{tau} : \mathit{proc} \to \mathit{proc}. \\ \multicolumn{2}{l}{\mathit{par}, \mathit{sum} : (\mathit{proc}, \mathit{proc}) \to \mathit{proc}.} & \multicolumn{2}{r}{\mathit{in} : (\mathit{chan}, \mathit{chan}\backslash\mathit{proc}) \to \mathit{proc}.} \\ \multicolumn{2}{l}{\mathit{out}, \mathit{match} : (\mathit{chan}, \mathit{chan}, \mathit{proc}) \to \mathit{proc}.} & \multicolumn{2}{l}{\mathit{res} : (\mathit{chan}\backslash\mathit{proc}) \to \mathit{proc}.} \\ \mathit{act} : \textbf{type}. & \mathit{tau\_a} : \mathit{act}. & \multicolumn{2}{l}{\mathit{in\_a}, \mathit{fout\_a}, \mathit{bout\_a} : (\mathit{chan}, \mathit{chan}) \to \mathit{act}.} \end{array}$$

The function $\mathit{ren\_p}(P, Y, X)$ performing capture-avoiding renaming is not shown, but easy to define.

We can check that this implementation of the operational semantics produces correct answers for the following queries:

$?\!-\ \mathit{step}(\mathit{res}(\mathsf{x}\backslash\mathit{par}(\mathit{res}(\mathsf{y}\backslash\mathit{out}(\mathsf{x},\mathsf{y},\mathit{ina})),\mathit{in}(\mathsf{x},\mathsf{z}\backslash\mathit{out}(\mathsf{z},\mathsf{x},\mathit{ina})))),A,P).$
$A = \mathit{tau\_a}, P = \mathit{res}(\mathsf{y}_{58}\backslash\mathit{res}(\mathsf{z}_{643}\backslash\mathit{par}(\mathit{ina},\mathit{out}(\mathsf{z}_{643},\mathsf{y}_{58},\mathit{ina}))))$
$?\!-\ \mathit{step}(\mathit{res}(\mathsf{x}\backslash\mathit{out}(\mathsf{x},\mathsf{y},\mathit{ina})),A,P).$
No.

```
func ren_p(proc, chan, chan) = proc. (* definition omitted *)
pred safe(act, pr).          (* tests bn(A) ∩ fn(P) = ∅ *)
safe(tau_a, P).
safe(fout_a(X, Y), P).
safe(bout_a(X, Y), P)  :- Y # P.
safe(in_a(X, Y), P)    :- Y # P.
pred step(pr, act, pr).                       (* encodes p --a--> p' *)
step(tau(P), tau_a, P).
step(par(P, Q), A, par(P', Q))               :- step(P, A, P'), safe(A, Q).
step(par(P, Q), tau_a, par(P', Q''))         :- step(P, fout_a(X, Y), P'),
                                                step(Q, in_a(X, Z), Q'),
                                                Q'' = ren_p(Q', Y, Z).
step(sum(P, Q), A, P')                       :- step(P, A, P').
step(out(X, Y, P), fout_a(X, Y), P).
step(in(X, z\P), in_a(X, W), P')             :- W # z\P, P' = ren_p(P, W, z).
step(match(X, X, P), A, P')                  :- step(P, A, P').
step(par(P, Q), tau_a, res(z\par(P', Q'))) :- step(P, bout_a(X, z), P'),
                                                step(Q, in_a(X, z), Q').
step(res(y\P), A, res(y\P'))                 :- y # A, step(P, A, P').
step(res(y\P), bout_a(X, W), P'')            :- step(P, fout_a(X, y), P'), y # X,
                                                W # y\P, P'' = ren_p(P', W, y).
```

Fig. 4. π-calculus transitions in αProlog

This αProlog session shows that $(x)((y)\overline{x}y.0 \mid x(y).\overline{y}x.0) \xrightarrow{\tau} (x)(y)(0 \mid \overline{y}x.0)$, but $(x)(x(y).0)$ cannot make any transition. Moreover, the answer to the first query is unique (up to renaming).

Given that a wide variety of alternative formulations of the π-calculus have appeared since Milner et al.'s work, why have we chosen to encode the original π-calculus rules? We want to show αProlog is capable of coping with the original version without help. Many of the π-calculus variants were developed in order to simplify the treatment of names; some were designed specifically to be encoded within HOAS. αProlog is also quite capable of handling these HOAS-friendly versions, but this is not surprising. However, as we have shown, αProlog is equally capable of encoding the HOAS-unfriendly original version.

5 Semantics

In this section we present an operational semantics for αProlog programs. We describe briefly nominal unification and αProlog's execution algorithm, emphasizing the main novelties relative to standard unification and logic programming execution.

5.1 Nominal Unification

Nominal unification is unification of nominal terms up to α-equivalence (as formalized by the axioms of Figure 1). For ground terms, nominal unification coincides with α-equivalence: for example, the term $\mathsf{n}\backslash\mathsf{n}$ unifies with $\mathsf{m}\backslash\mathsf{m}$, but $\mathsf{n}\backslash\mathsf{m}$ and $\mathsf{n}\backslash\mathsf{n}$ do not unify. However, seemingly α-equivalent non-ground terms such as $\mathsf{n}\backslash X$ and $\mathsf{m}\backslash X$ unify only subject to the *freshness constraints* $\mathsf{n} \# X$ and $\mathsf{m} \# X$. A freshness constraint of the form $\mathsf{n} \# X$ states that X may not be instantiated with a term containing a free occurrence of n. The problem $\mathsf{n}\backslash X \approx ?\ \mathsf{m}\backslash Y$ is unified by substitution $X = (\mathsf{n}\ \mathsf{m})Y$ subject to the constraint $\mathsf{n} \# X$; that is, X must be identical to Y with n and m swapped, and n must be fresh for X. The nominal unification algorithm therefore also must solve freshness (or *disunification*) subproblems of the form $\mathsf{a} \#?\ t$. In αProlog, this is generalized slightly to allow variables of name type on the left side ($A \#?\ t$).

Technically, a *constraint* is a set ∇ of freshness formulas that cannot be solved any further, such as $t \# Y$, where t is a name or variable. A *substitution* is a function θ mapping logic variables to terms, which we overload for the application of a substitution to a variable, term, formula, and constraint. Substitution is *not* capture-avoiding with respect to abstraction. For example, $\theta(\mathsf{n}\backslash X) = \mathsf{n}\backslash\mathsf{n}$ if $\theta(X) = \mathsf{n}$. Nominal unification is decidable and produces unique most general unifiers: pairs of the form $\langle\theta, \nabla\rangle$, such that for every unification problem $t \approx ?\ u$ we have that $\nabla \vdash \theta(t) \approx \theta(u)$. We refer the reader to [20] for the definition of this judgment and for the details of the nominal unification algorithm. We write $t \approx ?\ u \Downarrow \langle\theta, \nabla\rangle$ and $t \#?\ u \Downarrow \nabla$ to indicate that terms t, u unify with result $\langle\theta, \nabla\rangle$ or disunify with result ∇.

5.2 Operational semantics

We now present the operational semantics of αProlog programs. A *program* consists of a set of clauses $\mathcal{P}$, a goal G, and a constraint ∇; we shall write $\mathcal{P}$?– $G \mid \nabla$ for a query. An *answer* to this query is a pair $\langle\theta, \nabla\rangle$ of a substitution and some freshness constraints. We define the operational semantics of an αProlog query using judgments of the form $\mathcal{P}$?– $G \mid \nabla \Downarrow \langle\theta, \nabla'\rangle$, which means "$\langle\theta, \nabla'\rangle$ is a possible answer for the program $\mathcal{P}$?– $G \mid \nabla$". The rules of this judgment are listed below:

$\mathcal{P}$?– **true** $\mid \nabla \Downarrow \langle\epsilon, \nabla\rangle$ holds always (where ϵ stands for the identity substitution).

$\mathcal{P}$?– $G_1 \wedge G_2 \mid \nabla \Downarrow \langle\theta_2 \circ \theta_1, \nabla_2\rangle$ holds if $\mathcal{P}$?– $G_1 \mid \nabla \Downarrow \langle\theta_1, \nabla_1\rangle$ and $\mathcal{P}$?– $\theta_1(G_2) \mid \nabla_1 \Downarrow \langle\theta_2, \nabla_2\rangle$.

$\mathcal{P}$?– $\mathsf{n} \# t \mid \nabla \Downarrow \langle\epsilon, \nabla \cup \nabla'\rangle$ holds if $\mathsf{n} \#?\ t \Downarrow \nabla'$.
This case solves a freshness formula using disunification. Any resulting constraints must be added to the constraint set.

$\mathcal{P}$?– $t_1 = t_2 \mid \nabla \Downarrow \langle\theta, \nabla' \cup \nabla''\rangle$ holds if $t_1 \approx ?\ t_2 \Downarrow \langle\theta, \nabla'\rangle$ and $\theta(\nabla) \Downarrow \nabla''$.
This case solves an equality formula using unification. We must check that the unifying substitution θ is consistent with the pre-existing constraints ∇ by disunifying the problem $\theta(\nabla)$ to a new constraint ∇''.

$\mathcal{P}$?– $A \mid \nabla \Downarrow \langle \theta, \nabla' \rangle$ holds if $(A' \mathrel{:\!-} G) \in \mathcal{P}$, $\mathit{fresh}((\nabla, A), (A' \mathrel{:\!-} G))$ and $\mathcal{P}$?– $A = A' \wedge G \mid \nabla \Downarrow \langle \theta, \nabla' \rangle$
where $\mathit{fresh}(X, Y)$ means $V(X) \cap V(Y) = \emptyset$ and $N(X) \cap N(Y) = \emptyset$.
This case performs backchaining. We first unify A and A' and then solve G subject to the result of unification.

$\mathcal{P}$?– $A[a] \mid \nabla \Downarrow \langle \theta, \nabla' \rangle$**:** holds if $(a' = t \mathrel{:\!-} G) \in \mathcal{P}$, $\mathit{fresh}((\nabla, A[a]), (a' = t \mathrel{:\!-} G))$ and $\mathcal{P}$?– $a = a' \wedge G \wedge A[t] \mid \nabla \Downarrow \langle \theta, \nabla' \rangle$.
This case applies a rewriting rule. We write $A[]$ to indicate an atomic formula with a "hole" that can be filled by an atomic term. Assuming the goal is $A[a]$ for an atomic term a, we unify a with the rule's left-hand side a', solve the rule's side condition G, and replace a with t in $A[]$.

If we assume that $\mathcal{P}$, G, and ∇ are given, these rules can be viewed as a (nondeterministic) program for computing answers. In αProlog, as usual in logic programming, variables in program clauses or rewriting rules are renamed to new variables prior to unification; in addition, names are freshened to new names. This is enforced by the side condition $\mathit{fresh}(\cdot, \cdot)$.

Example 1. Consider the goal $X = \mathit{subst}(\mathit{lam}(\mathsf{x} \backslash \mathit{var}(\mathsf{y})), \mathit{var}(\mathsf{x}), \mathsf{y})$. The substitution on the right-hand side is in danger of capturing the free variable $\mathit{var}(\mathsf{x})$. How is capture avoided in αProlog? The freshened rewrite rule

$$\mathit{subst}(\mathit{lam}(\mathsf{y}_1 \backslash E_1'), E_1, X_1) = \mathit{lam}(\mathsf{y}_1 \backslash \mathit{subst}(E_1', E_1, X_1)) \mathrel{:\!-} \mathsf{y}_1 \mathrel{\#} E_1$$

matches with substitution $[E_1' = \mathit{var}(\mathsf{y}), X_1 = \mathsf{y}, E_1 = \mathit{var}(\mathsf{x})]$. The freshness constraint $\mathsf{y}_1 \mathrel{\#} \mathit{var}(\mathsf{x})$ guarantees that $\mathit{var}(\mathsf{x})$ cannot be captured. It is easily verified, so the term reduces to $\mathit{lam}(\mathsf{y}_1 \backslash \mathit{subst}(\mathit{var}(\mathsf{y}), \mathit{var}(\mathsf{x}), \mathsf{y}))$. Using the freshened rule $\mathit{subst}(\mathit{var}(X_2), E_2, X_2) = E_2$ with matching substitution $[X_2 = \mathsf{y}, E_2 = \mathit{var}(\mathsf{x})]$, we can rewrite to $\mathit{lam}(\mathsf{y}_1 \backslash \mathit{var}(\mathsf{x}))$, so the final solution is $X = \mathit{lam}(\mathsf{y}_1 \backslash \mathit{var}(\mathsf{x}))$.

Example 2. Consider the problem of inferring a type for the λ-term $\lambda x.\lambda y.x$. We start with the query ?– $\mathit{typ}([], \mathit{lam}(\mathsf{x} \backslash \mathit{lam}(\mathsf{y} \backslash \mathit{var}(\mathsf{x}))), T)$. We can reduce this goal by backchaining against the suitably freshened rule

$$\mathit{typ}(C_1, \mathit{lam}(\mathsf{x}_1 \backslash E_1), \mathit{arr}(T_1, U_1)) \mathrel{:\!-} \mathsf{x}_1 \mathrel{\#} C_1, \mathit{typ}([(\mathsf{x}_1, T_1) | C_1], E_1, U_1)$$

with matching unifier $[C_1 = [], E_1 = \mathit{lam}(\mathsf{y} \backslash \mathit{var}(\mathsf{x}_1)), T = \mathit{arr}(T_1, U_1)]$. This yields subgoal $\mathsf{x}_1 \mathrel{\#} [] \wedge \mathit{typ}([(\mathsf{x}_1, T_1) | C_1], E_1, U_1)$. The first conjunct is trivially valid since C_1 is a constant. The second is solved by backchaining against the third typ-rule again:

$$\mathit{typ}(C_2, \mathit{lam}(\mathsf{x}_2 \backslash E_2), \mathit{arr}(T_2, U_2)) \mathrel{:\!-} x_2 \mathrel{\#} C_2, \mathit{typ}([(\mathsf{x}_2, T_2) | C_2], E_2, U_2)$$

with unifier $[C_2 = [(\mathsf{x}_1, T_1)], E_2 = \mathit{var}(\mathsf{x}_1), U_1 = \mathit{arr}(T_2, U_2)]$ and subgoal $\mathsf{x}_2 \mathrel{\#} [(\mathsf{x}_1, T_1)] \wedge \mathit{typ}([(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], \mathit{var}(\mathsf{x}_1), U_2)$. The freshness subgoal reduces to the constraint $\mathsf{x}_2 \mathrel{\#} T_1$, and the typ subgoal can be solved by backchaining against

$$\mathit{typ}(C_3, \mathit{var}(X_3), T_3) \mathrel{:\!-} \mathit{mem}((X_3, T_3), C_3)$$

using unifier $[C_3 = [(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], X_3 = \mathsf{x}_1, T_3 = U_2]$. Finally, the remaining subgoal $mem((\mathsf{x}_1, U_2), [(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)])$ clearly has most general solution $[U_2 = T_1]$. Solving for T, we have

$$T = arr(T_1, U_1) = arr(T_1, arr(T_2, U_2)) = arr(T_1, arr(T_2, T_1))$$

This solution corresponds to the principal type of $\lambda x.\lambda y.x$.

5.3 Equivariance

It is straightforward to show that our operational semantics is sound with respect to first-order logic extended with the axioms of Figure 1. The proof of this fact relies on the soundness of nominal unification for nominal equational satisfiability ([20, Thm. 2]).

Completeness is more elusive, because we have not taken into account *equivariance*, an important property of nominal logic that guarantees that validity is preserved by name-swapping [16]. Formally, the equivariance axiom asserts that $p(\bar{t}) \Rightarrow p((\mathsf{n}\ \mathsf{m})\bar{t})$ is valid in nominal logic for any atomic formula $p(\bar{t})$ and names n, m. For example, for any binary relation **pred** $p(\nu, \nu)$ for ν : **name_type**, we have $p(\mathsf{x}, \mathsf{y}) \iff p(\mathsf{y}, \mathsf{z})$ valid in both directions because $(\mathsf{x}\ \mathsf{y})(\mathsf{y}\ \mathsf{z})$ translates between them. But many-to-one renamings may not preserve validity: for example, $\mathsf{x} \# \mathsf{y} \Rightarrow \mathsf{z} \# \mathsf{z}$ is not valid.

Because of equivariance, resolution based on nominal unification is incomplete for nominal logic. For example, given program clause $p(\mathsf{n})$ where n is a name, the goal $p(\mathsf{n})$ cannot be solved. Even though $p(\mathsf{n}) \vdash p(\mathsf{n})$ is obviously valid, proof search fails because the program clause $p(\mathsf{n})$ must be freshened to $p(\mathsf{n}')$, and $p(\mathsf{n}')$ and $p(\mathsf{n})$ do not unify. However, by equivariance these formulas are equivalent in nominal logic, since $p(\mathsf{n}') \Rightarrow p((\mathsf{n}\ \mathsf{n}')\mathsf{n}') \Rightarrow p(\mathsf{n}) \Rightarrow p((\mathsf{n}\ \mathsf{n}')\mathsf{n}) \Rightarrow p(\mathsf{n}')$.

Therefore, for complete nominal resolution, it is necessary to extend nominal unification to *equivariant unification* or "unification up to a permutation". However, even deciding whether an equivariant unification problem has a solution is **NP**-complete [1]. This does not necessarily mean that equivariant unification is impractical. Developing a practical approach to equivariant unification is the subject of current research, however, and the current version of αProlog opts for efficiency over completeness. We have experimented with brute-force search and more advanced techniques for equivariant unification but have yet to find a satisfactory solution.

Nevertheless, αProlog is useful even without equivariance. Equivariant unification does not seem necessary for many interesting αProlog programs, including all purely first-order programs and all the examples in this paper. In fact, such programs can be shown to be equivariant without using the equivariance axiom. We speculate that nominal unification-based resolution is complete for such *intrinsically equivariant* programs; we are currently working on proving this and developing a program analysis for checking that programs are intrinsically equivariant.

6 Concluding Remarks

6.1 Related work

Several existing languages are closely related to αProlog.

FreshML [17]: an extension of the ML programming language with Gabbay-Pitts names, name-binding with pattern matching, and fresh name generation. αProlog is reminiscent of FreshML in many ways, and it is fair to say that αProlog is to logic programming what FreshML is to functional programming. We believe however that the differences between FreshML and αProlog are more than cosmetic. αProlog lends itself to a declarative style of nameful programming which is refreshingly close to informal declarative presentations of operational semantics, type systems and logics, in contrast to FreshML which remains procedural (and effectful) at heart.

Qu-Prolog [19]: an extension of Prolog with built-in names, binding, and explicit capture-avoiding substitutions and unification up to both α-equivalence and substitution evaluation. Qu-Prolog includes "not free in" constraints corresponding to our freshness constraints. Nevertheless, there are significant differences; αProlog is not a reinvention of Qu-Prolog. First, αProlog is a strongly typed polymorphic language, in contrast to Qu-Prolog, which is untyped in the Prolog tradition. Second, αProlog is based on a simpler unification algorithm that unifies up to α-equivalence but not up to substitution. Finally, Qu-Prolog lacks a logical semantics, and because of its internalized treatment of capture-avoiding substitution, developing one would likely be difficult. In contrast, αProlog's semantic foundations have already been developed in the setting of nominal logic [16].

Logic programming with binding algebras: Hamana [7] has formalized a logic programming language based on Fiore, Turi, and Plotkin's *binding algebras* [4]. No implementation of this language appears to be available. However, since binding algebras are a formalization of HOAS, we believe that this approach will also share the semantic complexity of HOAS.

6.2 Status and Future Work

We have implemented an interpreter for αProlog based on nominal unification as outlined in this paper, along with many additional example programs, such as translation to a small typed assembly language, evaluation for a core object calculus, and modeling a cryptographic authentication protocol. The implementation is available online.[1] In addition, we are experimenting with support for equivariance. This makes additional applications possible, such as type inference for a small ML-like language and translations from regular expressions to finite automata. Therefore we are very interested in developing techniques for equivariant unification and resolution.

Following Miller et al. [10], we have formulated a uniform proof theoretic semantics for *nominal hereditary Harrop formulas* based on a sequent calculus

[1] `http://www.cs.cornell.edu/People/jcheney/aprolog/`

for nominal logic [6]. A more traditional model-theoretic semantics is in development. We also plan to develop equivariance, mode, and determinism analyses. Another interesting direction is relating αProlog's freshness constraint solving with a standard constraint logic programming framework [9].

The only deficiency of αProlog relative to HOAS systems and Qu-Prolog is that capture-avoiding substitution is not built-in, but must be written by hand when needed. This is easier in αProlog than in most languages, but is still tedious work we would rather avoid. Because binding is made explicit in αProlog's abstraction terms and types, it should be possible to define capture-avoiding substitution as a generic built-in function. Doing this safely in a strongly typed language requires care: for example Qu-Prolog's approach of blindly replacing object-variables with terms would be unsound in αProlog (if $\mathsf{x} : id, t : exp$, then $var(\mathsf{x})$ is well-formed but $var(t)$ is not). The solution we are pursuing is to allow the programmer to declare specific primitive function symbols such as $v : \nu \to \tau$ to be "substitutable variables"; then, αProlog can provide a polymorphic substitution action $\cdot[\cdot/\cdot] : (\alpha, \tau, \nu) \to \alpha$ that replaces all free occurrences of $v(\mathsf{x})$ with t' in t (where $t : \alpha$, $t' : \tau$ and $\mathsf{x} : \nu$). However, this does not help with unusual substitution-like operations such as $\lambda\mu$-calculus replacement $t\{t'/\alpha\}$, which we would also like to be able to derive generically.

6.3 Summary

Though still a work in progress, αProlog shows great promise. Although αProlog is not the first language to include special constructs for dealing with variable binding, αProlog allows programming much closer to informal "paper" definitions than any other extant system. We have given several examples of languages that can be defined both declaratively and concisely in αProlog. We have also described the operational semantics for core αProlog, and have proved that it is sound with respect to nominal logic, but complete only for a class of well-behaved *equivariant* programs. Additional work is needed to develop practical techniques for *equivariant unification* necessary for complete nominal resolution, and to develop static analyses and other forms of reasoning about αProlog programs. More broadly, we view αProlog as the first step toward a *nominal logical framework* for reasoning about programming languages, logics, and type systems encoded in nominal abstract syntax.

References

1. James Cheney. The complexity of equivariant unification. Submitted.
2. N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34(5):381–392, 1972.
3. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Int. Conf. on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag LNCS 902.

4. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. 14th Symp. on Logic in Computer Science (LICS 1999)*, pages 193–202. IEEE, 1999.
5. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
6. Murdoch Gabbay and James Cheney. A proof theory for nominal logic. Submitted.
7. Makoto Hamana. A logic programming language based on binding algebras. In *Proc. Theoretical Aspects of Computer Science (TACS 2001)*, number 2215 in Lecture Notes in Computer Science, pages 243–262. Springer-Verlag, 2001.
8. Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. 14th Symp. on Logic in Computer Science*, pages 204–213. IEEE, July 1999.
9. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 19/20:503–581, 1994.
10. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
11. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I-II. *Information and Computation*, 100(1):1–77, September 1992.
12. A. Momigliano, S. J. Ambler, and R. L. Crole. A comparison of formalizations of the meta-theory of a language with variable bindings in Isabelle. In *Informatics Research Report EDI-INF-RR-0046, Supplemental Proceedings of TPHOLs 2001*, pages 267–282. University of Edinburgh, 2001.
13. 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.
14. Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of the 1992 International Conference on Logic Programming and Automated Reasoning (LPAR '92)*, number 624 in LNAI, pages 190–201, 1992.
15. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89)*, pages 199–208. ACM Press, 1989.
16. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.
17. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.
18. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1–3):17–64, October-December 1996.
19. J. Staples, P. J. Robinson, R. A. Paterson, R. A. Hagen, A. J. Craddock, and P. C. Wallis. Qu-prolog: An extended prolog for meta level programming. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 23. MIT Press, 1996.
20. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC)*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527, Vienna, Austria, 2003. Springer-Verlag.