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 [3, 4, 9, 12, 15, 18, 21, 24] . 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.

In *first-order abstract syntax* (FOAS), 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 $lam : var \times exp \to exp$. 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 (and painstakingly). Nameless encodings like de Bruijn indices [3] ameliorate some, but not all, of these problems.

In *higher-order abstract syntax* (HOAS) [18], object languages are encoded using higher-order terms (e.g. λ-terms in λProlog [16]). In HOAS, names are encoded as meta-language variables and binders are encoded with meta-language λ-abstraction using higher-order function symbols like $lam : (exp \to exp) \to exp$. 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 HOAS is complex [10] and induction principles are difficult to develop. Moreover, HOAS cannot deal with open object terms, that is terms containing free variables. In *weak HOAS* [4], induction principles are recovered by encoding names using a concrete type var, and encoding binders encoded as λ-abstractions using constructors like $lam : (var \to exp) \to exp$. In this approach, α-equivalence is still built-in, but substitution must be defined. Also, weak 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 [8], 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* [19], 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* $\mathsf{x}.t$, which are considered equal up to α-equivalence. For example, object variables x and binders $\lambda x.t$ can be encoded as nominal terms $var(\mathsf{x})$ and abstractions $lam(\mathsf{x}.t)$ where $var : id \to exp$ and $lam : \langle id \rangle exp \to exp$.

We refer to this approach to programming with names and binding as *nominal abstract syntax* (NAS). NAS provides α-equivalence and fresh name generation for free, while remaining semantically simple, requiring neither recursive types nor stateful name-generation. 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. However, names are still sufficiently concrete that there is no problem working with open terms. Therefore NAS makes possible a distinctive new style of meta-programming.

This paper presents αProlog, a logic programming language based on the Horn clause fragment of nominal logic which supports nominal abstract syntax. In the rest of this paper, we describe αProlog, and discuss its unification and constraint solving algorithm (due to Urban, Pitts, and Gabbay [25]) and operational semantics. We also discuss an important open issue: our current implementation is incomplete, because complete proof search in αProlog requires solving *equivariant unification* problems, which are **NP**-complete in general [2]. However, this problem can often be avoided, and we give several examples of interesting languages and relations that can be encoded using NAS in αProlog. We conclude with a discussion of related languages and future work.

2 Syntax

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

$$t ::= X \mid \mathsf{n} \mid \mathsf{n}.t \mid (\mathsf{n}\ \mathsf{m}) \cdot t \mid f(\overline{t})$$

where X is a *(logic) variable*, f is a *function symbol* (we write $\overline{t}$ 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 $\mathsf{sans\text{-}serif}$ typeface. We shall refer to a term of the form $f(\overline{t})$ as an atomic term. Terms of the form $\mathsf{n}.t$ are called *abstractions*, and terms of the form $(\mathsf{n}\ \mathsf{m}) \cdot t$ are called *swapping*, which intuitively denote the result of swapping names n, m within t. Swapping takes higher precedence than abstraction, i.e., $(\mathsf{a}\ \mathsf{b}) \cdot \mathsf{c}.t = ((\mathsf{a}\ \mathsf{b}) \cdot \mathsf{c}).t$. Variables cannot be used to form abstractions and transpositions, i.e., $X.t$ and $(X\ Y) \cdot t$ 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 $.\nu\sigma$. 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. The result type of an uninterpreted function symbol may not be a built-in type or a **name_type**. Relation symbols 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 [23]. We assume built-in and self-explanatory type and function symbols for pairs $((x, y) : \sigma \times \sigma')$ and lists $([], x :: y, [x|y] : [\sigma])$.

Atomic formulas A are terms of the form $p(t_1, \ldots, t_n)$, where p is a *relation symbol*. *Constraints* C include freshness formulas $t \# u$, and equality formulas $t = u$. In $t \# u$, the term t must be of some name type ν : **name_type**, whereas u may be of any type; in $t = u$, both t and u must be of the same type. *Goals* G consist of sequences of constraints and atomic formulas. *Program clauses* include Horn clauses of the form $A :- G$ and function-definition clauses of the form $f(\overline{t}) = t' :- G$, which introduce a (conditional) rewrite rule for an atomic term with an interpreted head symbol f.

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 abbreviate clauses $A :- ()$ and

$f(\bar{t}) = t' :- ()$, where $()$ denotes the empty sequence, as A and $f(\bar{t}) = t'$. 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 $N(\mathsf{x}.\mathsf{x}) = \{\mathsf{x}\}$. 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. We write $VN(t)$ for $V(t) \cup N(t)$.

3 Semantics

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

3.1 Equality, freshness, and unification

Figure 1 shows the axioms of equality and freshness for ground nominal terms (based on [19, 25]). 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 $\mathsf{x}.t, \mathsf{y}.u$ are equal just in case their bodies are equal up to swapping the names (i.e., $t = (\mathsf{x}\ \mathsf{y}) \cdot u$) and x does not appear free in u (i..e., $\mathsf{x} \# u$) Symmetrically, it suffices to check $\mathsf{y} \# t$; the two conditions are equivalent if $t = (\mathsf{x}\ \mathsf{y}) \cdot u$. For example, $\mathsf{x}.g(\mathsf{x}) = \mathsf{y}.g(\mathsf{y})$ and $\mathsf{x}.f(\mathsf{x}, \mathsf{y}) = \mathsf{z}.f(\mathsf{z}, \mathsf{y})$, but $\mathsf{x}.f(\mathsf{x}, \mathsf{y}) \neq \mathsf{y}.f(\mathsf{y}, \mathsf{x})$ because $\mathsf{x} \# f(\mathsf{y}, \mathsf{x})$ fails.

The freshness axioms (F_1)–(F_5) describe the freshness relation. Intuitively, $\mathsf{x} \# t$ means "name x does not appear unbound in t". For example, it is never the case that $\mathsf{x} \# \mathsf{x}$, whereas any two distinct names are fresh ($\mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{x} \# \mathsf{y}$). Moreover, freshness passes through function symbols (in particular, any name is fresh for any constant). The abstraction freshness rules are more interesting: $\mathsf{x} \# \mathsf{x}.t$ 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 $\mathsf{x} \# \mathsf{y}.t$ just in case $\mathsf{x} \# t$.

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

Fig. 1. Ground equational and freshness theory

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 $\mathsf{n}.\mathsf{n}$ unifies with $\mathsf{m}.\mathsf{m}$, but $\mathsf{n}.\mathsf{m}$ and $\mathsf{n}.\mathsf{n}$ do not unify. However, non-ground terms such as $\mathsf{n}.X$ and $\mathsf{m}.X$ unify only subject to the *freshness constraints* $\mathsf{n} \mathrel{\#} X$ and $\mathsf{m} \mathrel{\#} X$. A freshness constraint of the form $\mathsf{n} \mathrel{\#} X$ states that X may not be instantiated with a term containing a free occurrence of n. The problem $\mathsf{n}.X \approx ?\ \mathsf{m}.Y$ is unified by substitution $X = (\mathsf{n}\ \mathsf{m}) \cdot Y$ subject to the constraint $\mathsf{n} \mathrel{\#} X$; 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 $t \mathrel{\#?} u$, where t is of name type.

Urban, Pitts, and Gabbay [25] developed an algorithm for solving nominal unification and freshness constraint problems of the form encountered in αProlog. A modified form of this algorithm is used in our current implementation. For space reasons, we omit a fuller discussion of the details of the algorithm, and note that any other constraint solving procedure for nominal equality and freshness constraints could be used instead.

3.2 Operational semantics

We now present the operational semantics of αProlog programs. This semantics is based loosely on that of *constraint logic programming* [11], over the domain of nominal terms as axiomatized in Figure 1. A *program* is a set $\mathcal{P}$ of program clauses, closed under permutative renaming of names and variables. A *program state* consists a goal G and a set ∇ of equality and freshness constraints; we shall write $\langle G \mid \nabla \rangle$ for such a state. An *answer* to this query is a set of constraints ∇'. We define the operational semantics of an αProlog query using transitions of the form $\langle G \mid \nabla \rangle \longrightarrow \langle G' \mid \nabla' \rangle$ and write $G \longrightarrow^* \nabla$ if $\langle G \mid \emptyset \rangle \longrightarrow^* \langle \emptyset \mid \nabla \rangle$. The rules for the transitions are as follows:

$$
\begin{array}{lll}
\langle C, G \mid \nabla \rangle & \longrightarrow \langle G \mid \{C\} \cup \nabla \rangle & \text{if } \{C\} \cup \nabla \text{ satisfiable} \\
\langle p(\overline{t}), G \mid \nabla \rangle & \longrightarrow \langle \overline{t} = \overline{u}, G', G \mid \nabla \rangle & \text{if } (p(\overline{u}) :- G') \in \mathcal{P} \\
 & & \text{and } VN(p(\overline{u}) :- G') \cap VN(p(\overline{t}), G, \nabla) = \emptyset \\
\langle G \mid \nabla \rangle & \longrightarrow \langle G \mid \nabla' \rangle & \text{if } \nabla' \vdash \nabla
\end{array}
$$

In the last rule, $\nabla' \vdash \nabla$ is constraint entailment in the theory of Figure 1; this rule permits constraint simplification via unification. In αProlog, as usual in logic programming, variables in program clauses or rewriting rules are renamed to new variables during backchaining; in addition, names are freshened to new names. Rewriting rules $f(\overline{t}) = u :- G$ defining function symbols are translated to a clausal form $p_f(\overline{t}, u) :- G'$ via *flattening*, as in many Prolog systems.

It is straightforward to show that our operational semantics is sound with respect to an appropriate variant of nominal logic. The proof of this fact relies on the soundness of nominal unification for nominal equational satisfiability ([25, Thm. 2]). However, completeness fails, because we have not taken into account *equivariance*, an important property of nominal logic that guarantees that validity is preserved by name-swapping [19]. Formally, the equivariance axiom asserts that $p(\overline{t}) \Rightarrow p((\mathsf{n}\ \mathsf{m}) \cdot \overline{t})$ is valid in nominal logic for any atomic formula $p(\overline{t})$ 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 the swapping $(\mathsf{x}\ \mathsf{y})(\mathsf{y}\ \mathsf{z})$ 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, backchaining based on nominal unification is incomplete. For example, given program clause $p(\mathsf{n})$ where n is a name, the goal $p(\mathsf{n})$ cannot be solved. Even though $p(\mathsf{n}) \Rightarrow p(\mathsf{n})$ 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 $p(\mathsf{n}') \Rightarrow p((\mathsf{n}\ \mathsf{n}') \cdot \mathsf{n}') \Rightarrow p(\mathsf{n}) \Rightarrow p((\mathsf{n}\ \mathsf{n}') \cdot \mathsf{n}) \Rightarrow p(\mathsf{n}')$.

This can be fixed by adding a transition rule

$$\langle p(\overline{t}), G \mid \nabla \rangle \longrightarrow \langle p((\mathsf{a}\ \mathsf{b}) \cdot \overline{t}), G \mid \nabla \rangle\ .$$

However, this rule introduces nondeterminism. To perform goal-directed proof search, it seems preferable to replace the naive backchaining rule above with one based on unifying up to nominal equality modulo a permutation:

$$\begin{array}{l} \langle p(\overline{t}), G \mid \nabla \rangle \longrightarrow \langle \pi \cdot \overline{t} = \overline{u}, G', G \mid \nabla \rangle \\ \qquad \text{if } (p(\overline{u}) :- G') \in \mathcal{P} \text{ and } VN(p(\overline{u}) :- G') \cap VN(p(\overline{t}), G, \nabla) = \emptyset \end{array}$$

where π is a sequence of transpositions $(\mathsf{a}_1\ \mathsf{b}_1) \cdots (\mathsf{a}_n\ \mathsf{b}_n)$. We use the term *equivariant unification* for the problem of unifying up to nominal equality modulo a permutation. However, even deciding whether an equivariant unification problem has a solution is **NP**-complete [2]. This does not necessarily mean that equivariant unification is impractical. Developing a practical approach to equivariant unification or constraint solving 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, our incomplete implementation of αProlog is still useful. Equivariant unification does not seem necessary for many interesting αProlog programs, including all purely first-order programs and all the example programs in this paper (capture-avoiding substitution, typing, etc.). In fact, the semantics we have presented is complete for such programs. In a separate paper, we identify (and prove correct) a condition on program clauses that ensures that nominal unification-based backchaining is complete [26].

4 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:

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

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:

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

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 $\ulcorner e \urcorner \approx \ulcorner e' \urcorner$.

Example 1 (Typechecking and inference). First, we consider the problem of typechecking λ-terms. The syntax of types can be encoded as follows:

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

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

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

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 $\mathsf{x} \# C$.

Consider the query ?– $typ([], lam(\mathsf{x}.lam(\mathsf{y}.var(\mathsf{x}))), T)$. We can reduce this goal by backchaining against the suitably freshened rule

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

which unifies with the goal with $[C_1 = [], E_1 = lam(\mathsf{y}.var(\mathsf{x}_1)), T = arr(T_1, U_1)]$. This yields subgoal $\mathsf{x}_1 \# [], typ([(\mathsf{x}_1, T_1)|C_1], E_1, U_1)$. The first conjunct is trivially valid since C_1 is a constant. The second is solved by backchaining against the third typ-rule again, producing unifier $[C_2 = [(\mathsf{x}_1, T_1)], E_2 = var(\mathsf{x}_1), U_1 = arr(T_2, U_2)]$ and subgoal $\mathsf{x}_2 \# [(\mathsf{x}_1, T_1)], typ([(\mathsf{x}_2, T_2), (\mathsf{x}_1, T_1)], var(\mathsf{x}_1), U_2)$. The freshness subgoal reduces to the constraint $\mathsf{x}_2 \# T_1$, and the typ subgoal can be solved by backchaining against

$$typ(C_3, var(X_3), T_3) :-\ mem((X_3, T_3), C_3)$$

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 $\lambda x.\lambda y.x$.

Example 2 (Capture-avoiding substitution). Although capture-avoiding substitution is not a built-in operator in αProlog, it is easy to define via the clauses:

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

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 $lam(\mathsf{y}.E')$ 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 $t[t'/x]$.

Consider the goal $X = subst(lam(\mathsf{x}.var(\mathsf{y})), var(\mathsf{x}), \mathsf{y})$. The substitution on the right-hand side is in danger of capturing the free variable $var(\mathsf{x})$. How is capture avoided in αProlog? First, note that function definitions are translated to a flattened clausal form in αProlog, so we must solve the equivalent goal $subst'(lam(\mathsf{x}.var(\mathsf{y})), var(\mathsf{x}), \mathsf{y}, X)$ subject to an appropriately translated definition of $subst'$. The freshened, flattened clause

$$subst'(lam(\mathsf{y}_1.E_1'), E_1, X_1, lam(\mathsf{y}_1, E_1'')) :- \mathsf{y}_1 \# E_1, subst'(E_1', E_1, X_1, E_1'')$$

unifies with substitution $[E_1' = var(\mathsf{y}), X_1 = \mathsf{y}, E_1 = var(\mathsf{x}), X = lam(\mathsf{y}_1.E_1'')]$. The freshness constraint $\mathsf{y}_1 \# var(\mathsf{x})$ guarantees that $var(\mathsf{x})$ cannot be captured. It is easily verified, so the goal reduces to $subst'(var(\mathsf{y}), var(\mathsf{x}), \mathsf{y}, E_1'')$. Using the freshened rule $subst'(var(X_2), E_2, X_2, E_2)$ with unifying substitution $[X_2 = \mathsf{y}, E_2 = var(\mathsf{x}), E_1'' = var(\mathsf{x})]$, we obtain the solution $X = lam(\mathsf{y}_1.var(\mathsf{x}))$.

Example 3 (Parsing). Logic programming languages often provide advanced support for parsing using definite clause grammars (DCGs). DCG parsing can be implemented in αProlog by translating DCG rules $h \longrightarrow t$ to ordinary αProlog programs. We assume familiarity with DCG syntax in this example. We assume that $v(string), ws\_opt, ws, token(string)$ are predefined nonterminals recognizing variable names, (optional) whitespace, and string tokens. Here is a small example of parsing λ-terms from strings. (Here, $mem1(S, X, M)$ holds when S, X is the first binding of S in M.)

$$\begin{array}{ll}
\mathbf{type}\ map = [(string, id)] & \\
ltm(M, var(X)) & \longrightarrow v(S), \{mem1(S, X, M)\}. \\
ltm(M, app(E_1, E_2)) & \longrightarrow token(\text{“(”}), ws_opt, ltm(M, E_1), ws, \\
& \qquad ltm(M, E_2), ws_opt, token(\text{“)”}) \\
ltm(M, lam(\mathsf{x}.E)) & \longrightarrow \{\mathsf{x} \# M\}, token(\text{“\textbackslash”}), ws_opt, v(X), ws_opt, \\
& \qquad token(\text{“.”}), ws_opt, ltm([(S, \mathsf{x})|M], E).
\end{array}$$

This program parses “$\backslash x.\backslash y.x$” to $lam(\mathsf{x}.lam(\mathsf{y}.\mathsf{x}))$.

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

The $\lambda\mu$-calculus, invented by Parigot [17], extends the λ-calculus with *continuations* α; terms may be “named” by continuations ($[\alpha]e$) and continuations may be introduced with μ-binding ($\mu\alpha.e$). Intuitively, $\lambda\mu$-terms are proof terms

Terms, Types, and Contexts

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

Replacement Operation

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

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.

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 a new name type con for continuations and term constructors for $\lambda\mu$-terms:

$$con : \textbf{name\_type} \qquad pass : (con, exp) \to exp \qquad mu : \langle con \rangle exp \to exp$$

and encoding $\ulcorner[\alpha]t\urcorner = pass(\alpha, \ulcorner t\urcorner)$ and $\ulcorner\mu\alpha.t\urcorner = mu(\alpha.\ulcorner t\urcorner)$. Again, it is easy to show that ground exp-terms are in bijective correspondence with $\lambda\mu$-terms.

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:

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

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

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

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.

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

Capture-avoiding substitution can be extended to $\lambda\mu$-terms easily. For replacement, we show the interesting cases for continuation applications and μ-abstractions:

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

This first-order NAS encoding is quite different in flavor from a HOAS encoding of the $\lambda\mu$-calculus due to Abel [1]. There, $\mu\alpha.t$ is encoded as $mu(\lambda\ulcorner\alpha\urcorner.\ulcorner t\urcorner)$, where $mu : ((tm\ A \to nam) \to nam) \to tm\ A$, and $tm\ A$ is the type of terms of type A and nam the type of named terms. Continuations are encoded as variables $\ulcorner\alpha\urcorner : (tm\ A \to nam)$ and named terms $[\alpha]t$ are encoded as applications $\ulcorner\alpha\urcorner\ \ulcorner t\urcorner$. While elegant, this third-order encoding is semantically complex, making proofs of interesting properties difficult.

5 Example: the π-calculus

The π-calculus is a calculus of concurrent, mobile processes. Its syntax (following Milner, Parrow, and Walker [14]) 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 $t\{x/y\}$.

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

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

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

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

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

Fig. 3. The π-calculus

$$
\begin{array}{ll}
\textbf{func}\ ren\_p(proc, chan, chan) = proc. & \text{(* definition omitted *)} \\
\textbf{pred}\ safe(act, pr). & \text{(* tests } bn(A) \cap fn(P) = \emptyset \text{ *)} \\
safe(tau\_a, P). & \\
safe(fout\_a(X,Y), P). & \\
safe(bout\_a(X,Y), P)\ \ :-\ Y \ \#\ P. & \\
safe(in\_a(X,Y), P)\ \ :-\ Y \ \#\ P. & \\
\textbf{pred}\ step(pr, act, pr). & \text{(* encodes } p \xrightarrow{a} p' \text{ *)} \\
step(tau(P), tau\_a, P). & \\
step(par(P,Q), A, par(P',Q)) & :-\ step(P,A,P'), safe(A,Q). \\
step(par(P,Q), tau\_a, par(P',Q'')) & :-\ step(P, fout\_a(X,Y), P'), \\
 & \quad step(Q, in\_a(X,Z), Q'), \\
 & \quad Q'' = ren\_p(Q', Y, Z). \\
step(sum(P,Q), A, P') & :-\ step(P, A, P'). \\
step(out(X,Y,P), fout\_a(X,Y), P). & \\
step(in(X, z.P), in\_a(X,W), P') & :-\ W \ \#\ z.P, P' = ren\_p(P, W, z). \\
step(match(X,X,P), A, P') & :-\ step(P, A, P'). \\
step(par(P,Q), tau\_a, res(z.par(P',Q'))) & :-\ step(P, bout\_a(X,z), P'), \\
 & \quad step(Q, in\_a(X,z), Q'). \\
step(res(y.P), A, res(y.P')) & :-\ y \ \#\ A, step(P, A, P'). \\
step(res(y.P), bout\_a(X,W), P'') & :-\ step(P, fout\_a(X,y), P'), y \ \#\ X, \\
 & \quad W \ \#\ y.P, P'' = ren\_p(P', W, y).
\end{array}
$$

Fig. 4. π-calculus transitions in αProlog

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 $\overline{x}(y)$. Processes and actions can be encoded using the following syntax:

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

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. The rules can be translated directly into αProlog (see Figure 4). The function $ren\_p(P,Y,X)$ 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:

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

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).

Röckl [20] and Gabbay [6] have also considered encodings of the π-calculus using nominal abstract syntax. Röckl considered only modeling the syntax of terms up to α-equivalence using swapping, whereas Gabbay went further, encoding transitions and the bisimulation relation and proving basic properties thereof. By [6, Thm 4.5], Gabbay's version of the π-calculus is equivalent to our conventional representation. In fact, Gabbay's presentation is a bit simpler and easier to reason about, but we have chosen Milner et al.'s original presentation to emphasize that informal "paper" presentations (even for fairly complicated calculi) can be translated directly to αProlog programs.

6 Concluding Remarks

6.1 Related work

FreshML [22]: an extension of the ML programming language with Gabbay-Pitts names, name-binding with pattern matching, and fresh name generation. αProlog is related to 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. There are also major differences from a technical point of view: in FreshML much research went into designing an expressive type-system, while the problems we face in αProlog concern the design of an efficient proof search procedure (see [26]).

Qu-Prolog [24]: 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, nominal logic [19] provides a solid semantic foundation for αProlog.

Logic programming with binding algebras: Hamana [9] has formalized a logic programming language based on Fiore, Turi, and Plotkin's *binding algebras* [5]. 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.

L_λ: Miller [12] discovered a restricted form of higher-order logic programming called L_λ in which unification is efficiently decidable and HOAS encodings are possible, but built-in capture-avoiding substitution is not available. There are several interesting parallels between L_λ and αProlog (and nominal unification and L_λ unification); relating these languages is future work.

Delphin: Schürmann et al. [21] are developing a functional programming language called Delphin which supports advanced meta-programming with recursion over higher-order abstract syntax. This approach seems very powerful, but also very complex because it is based on 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] Some additional applications of interest, such as type inference for a small ML-like language and translations from regular expressions to finite automata, do not work properly because of αProlog's current incomplete implementation. Therefore we are very interested in developing techniques for equivariant unification and resolution.

Following Miller et al. [13], we have formulated a uniform proof theoretic semantics for *nominal hereditary Harrop formulas* based on a sequent calculus for nominal logic [7]. A more traditional model-theoretic semantics is in development. We also plan to develop mode and determinism analyses for αProlog. Another interesting direction is the possibility of integrating αProlog's constraint domain into an existing constraint logic programming system.

One 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. We are currently experimenting with an operation $\cdot\{\cdot/\cdot\} : (\alpha, \beta, \beta) \to \alpha$, such that $t\{u/v\}$ denotes the result of replacing the term v with u everywhere in t, renaming abstractions to avoid capture. Thus, $subst(t, u, \mathsf{x})$ for λ or $\lambda\mu$-terms can be written as $t\{u/var(\mathsf{x})\}$, and $ren\_p(p, \mathsf{a}, \mathsf{b})$ as $p\{\mathsf{a}/\mathsf{b}\}$. Currently there are *ad hoc* restrictions such as that t and v must be ground when $t\{u/v\}$ is evaluated. Also, this does not help with unusual substitution-like operations

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

such as $\lambda\mu$-calculus replacement. Developing a logical account of substitution in nominal logic and αProlog is an important area for future work.

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, which is sound with respect to nominal logic, but complete only for a class of well-behaved 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 a modest first step toward a *nominal logical framework* for reasoning about programming languages, logics, and type systems encoded in nominal abstract syntax.

References

1. Andreas Abel. A third-order representation of the $\lambda\mu$-calculus. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, volume 58(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
2. James Cheney. The complexity of equivariant unification. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, 2004. To appear.
3. N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34(5):381–392, 1972.
4. 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.
5. 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.
6. M. J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*. Kluwer, 2003.
7. M. J. Gabbay and J. Cheney. A proof theory for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, 2004. To appear.
8. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
9. 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.

10. Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. 14th Symp. on Logic in Computer Science*, pages 204–213. IEEE, July 1999.
11. Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
12. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.
13. 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.
14. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I-II. *Information and Computation*, 100(1):1–77, September 1992.
15. 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.
16. 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.
17. 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.
18. 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.
19. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.
20. Christine Röckl. A first-order syntax for the pi-calculus in isabelle/hol using permutations. In S.J. Ambler, R.L. Crole, and A. Momigliano, editors, *MERLIN 2001: Mechanized Reasoning about Languages with Variable Binding*, volume 58(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
21. C. Schrmann, R. Fontana, and Y. Liao. Delphin: Functional programming with deductive systems. Available at `http://cs-www.cs.yale.edu/homes/carsten/`, 2002.
22. 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.
23. 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.
24. 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.
25. 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.
26. Christian Urban and James Cheney. Avoiding equivariant unification. Submitted.