αProlog User's Guide & Language Reference
Version 0.3
DRAFT

James Cheney

October 23, 2003

Contents

Chapter 1

Introduction

1.1 What αProlog Is For

Names, binding, and scope are perennial problems in many programming tasks, including implementing compilers and interpreters as well as symbolic mathematical tools and theorem proving systems. Few languages provide any assistance for programming with names, so programmers must reinvent the wheel every time a new system which makes use of names is built. This is often a tedious and error-prone process; also, the resulting programs tend to be more difficult to read and analyze. Programming language support for key data structures is crucial to writing clear, optimizable code in traditional domains such as matrix computations (via arrays), databases (via records), large-scale interactive systems (via objects), and algebraic symbolic computation (via ML-style algebraic datatypes or Prolog-style terms): these features have been standard in high-level programming languages for decades. The purpose of αProlog is to provide the same kind of built-in support for names in a practical logic programming language.

1.2 Background

αProlog is a logic programming language with a built-in notion of names, name binding, and equivalence up to renaming. αProlog is based on a logical theory of names and binding called nominal logic, that has its own semantics independent of logic programming. In this logic, it is possible to define relations that express precisely what we intuitively mean when we write programs that manipulate names, without giving up on determinism and ease of reasoning about programs. The clauses and goals allowed in αProlog are a simple subset of this logic, corresponding to Horn clauses and goals extended with a few additional operators. These operators include a *freshness* predicate $a \# t$, which checks that a name a does not occur in a term t, and a *new-quantifier* $\mathsf{N}a.\phi$, which expresses that a is a new name in ϕ (that is, a only appears in ϕ where it is explicitly mentioned; it cannot be "hidden" in any of ϕ's free variables).

αProlog also has a non-standard view of equality and unification. Terms in

αProlog include a variable-binding operator $a\backslash t$, which expresses that the name a is bound (or abstracted) in term t. This is not λ-abstraction; that is, the resulting term is not a function. Instead, however, terms are considered equal if corresponding abstractions can be renamed to be syntactically equal (without capturing any other variables). Thus, $a\backslash(a,b) = c\backslash(c,b)$ is true in αProlog; in fact, there is no way to distinguish the terms; on the other hand, $a\backslash(a,b) = b\backslash(b,b)$ does not hold, because the terms have different binding structure. This form of "equality up to (safe) renaming" is usually referred to as "α-equivalence", and that is where the α in αProlog comes from.

1.3 An Example Program

In this section we develop an example αProlog program which demonstrates how to encode a simple language with names, binding, and capture-avoiding substitution.

For our first example, consider the typed λ-calculus (a notation for anonymous function definitions). Its syntax is summarized by the following grammar

$$\begin{array}{rcl} e & ::= & x \mid e_1\ e_2 \mid \lambda x.e \\ \tau & ::= & \alpha \mid \tau \rightarrow \tau \end{array}$$

where expression variables x and type variables α are drawn from disjoint, countably infinite sets. We can encode the syntax of λ-terms using declarations

```
id  : name_type.
exp : type.

var : id -> exp.
app : (exp,exp) -> exp.
lam : id\exp -> exp.

tid   : name_type.
ty    : type.

tvar  : tid -> type.
arrow : (ty,ty) -> ty.
```

Here, *id* and *tid* are declared to be *name types*, tnat is, types containing a countable number of distinct names. As in Prolog identifiers starting with an upper-case letter are considered to be variables; undefined lower-case identifiers are considered to be names. The types *exp* and *ty* are plain (data)types. We can describe the inhabitants of datatypes (but not name types) via further declarations of constants and function symbols. In this case, there are three function symbols for expressions: variables, applications, and lambdas, and two for types: type variables and function types. The type $id\backslash exp$ that is used as the domain of *lam* is called an *abstraction type*, and it denotes the set of expressions with one identifier abstracted. These declarations

describe how well-formed terms can be constructed: for example, $lam(x\backslash var(x))$ is an encoding of the identity function and $arrow(tvar(a), tvar(a))$ is an encoding of a type for the identity function.

Since identifiers, expressions, and types are just terms, αProlog can be used to define interesting functions and relations among such terms. These definitions tend to be significantly closer to their "paper" presentations than is usual for either functional or logic languages. For example, the following rules express the syntax and typability relation for simply-typed λ-terms:

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

The equivalent αProlog program encoding these rules is

```
pred of([(id,ty)],exp,ty)
of(Gamma,var(X),A)              :- mem((X,A),Gamma).
of(Gamma,app(M,N),B)            :- of(Gamma,M,arrow(A,B)),
                                   of(Gamma,N,A).
of(Gamma,lam(x\M),arrow(A,B)) :- x # Gamma,
                                   of([(x,A)|Gamma],M,B).
```

The first line declares the argument types for the predicate of: the first argument is a list of identifier-type pairs representing a context, the second is an expression, and the third a type. The remaining lines are clauses that define the behavior of this predicate. As can be seen, each clause is almost a literal translation of the corresponding inference rule, with $x : A \in \Gamma$ written as `mem((X,A),Gamma)`, $\Gamma \rhd M{:}A$ written as `of(Gamma,M,A)`, and $x \notin FV(\Gamma)$ written `x # Gamma`.

In Prolog, we might instead encode a binding $\lambda x.e[x]$ as a term $\lambda(\text{“x”}, e[\text{“x”}])$, that is, using strings or some other data for variables. In αProlog, we think of λ as a term constructor mapping abstraction terms $x\backslash e$ to $\lambda x.e$. This view of the world is somewhat similar to that adopted in higher-order abstract syntax, where λ is (somewhat circularly) defined as a constructor with type $(exp \to exp) \to exp$. But although there are similarities (for example, both function variables and abstracted names admit equality up to α-equivalence), the abstraction type is quite distinct from the function type: for example, there is no built-in application of abstractions, and names are ground terms that can escape the scope of abstractions in limited ways (as does `x` in the last clause of `of`).

Another interesting relation (or function) on λ-terms is *substitution*. For example, $(x\ (\lambda y.y))[x/z] = z\ (\lambda y.y)$. In the λ-calculus, we ask that substitution not essentially change the "meaning" of a term (as a function). To illustrate this point, consider the following flawed definiton of substitution:

$$\begin{aligned} x[e/x] &= e \\ y[e/x] &= y \qquad (x \neq y) \\ (e_1\ e_2)[e/x] &= e_1[e/x]\ e_2[e/x] \\ (\lambda y.e_1)[e/x] &= \lambda y.e_1[e/x] \end{aligned}$$

Under this definition, substituting z for x in the term $\lambda x.z+x$ could result in $\lambda x.x+x$, which is a different function. This can be seen by α-renaming $\lambda x.z + x$ to $\lambda y.z + y$, in which substituting x for z results in $\lambda y.x + y$. The reason is that in the obvious approach to substitution, variables can be "captured" when substitution passes under a binding (as x is when we pass under the λx binder).

The classical approach to this problem is to assume that bound variables are always "renamed away" from the all free variables (this is called Barendregt's variable convention). Subject to this convention, the above naive definition becomes correct: equivalently, we can write the fourth clause as

$$(\lambda y.e_1)[e/x] \quad = \quad \lambda y.e_1[e/x] \qquad (y \notin FV(e) \cup \{x\})$$

However, this is not precise enough for a Prolog program to implement, because the implicit, highly nondeterministic "renaming away" step needs to be made explicit. Renaming itself also needs to avoid variable capture, so for a mechanical implementation of substitution it is necessary to write down explicitly how to rename/substitute without capture. A typical definition of "capture-avoiding" substitution uses a rule

$$(\lambda y.e_1)[e/x] \quad = \quad \lambda z.e_1[z/y][e/x] \qquad (z \notin FV(e) \cup \{x\})$$

This definition renames aggressively, picking a fresh variable z whenever a λ binding is encountered.

However, there are problems with turning this definition into a Prolog-style declarative program. For example, the choice of z is very open-ended, and to choose fresh variables z efficiently it is necessary to maintain a "store" of unused variable names. This store is passed as an extra argument and return value of *subst*. Also, this definition causes multiple passes to be made over the term because of the renaming. To reduce this to a single pass, we would have to add an additional argument mapping renamed variables to their renamings.

In αProlog, on the other hand, we are able to write a definition that is essentially the same as the declarative definition, yet correct:

```
func subst(exp,exp,id) = exp.
subst(var(x),E,x)     = E.
subst(var(y),E,x)     = var(y) :- y # x.
subst(app(E1,E2),E,x) = app(subst(E1,E,x),subst(E2,E,x)).
subst(lam(y\E1),E,x)  = lam(y\subst(E1,E,x)) :- y # E, y # x.
```

where we read $y \# x$ as $y \neq z$ and $y \# (E, x)$ as $y \notin FV(E) \cup (x)$. In fact, in αProlog, syntactically distinct name identifiers like x, y are assumed to denote distinct names, so the $y \# x$ constraints are redundant. On the other hand, we cannot assume that bound variables always are distinct from other variables in scope, so we must check $y \# E$ in the fourth clause. Because unification in αProlog is up to α-equivalence, the implicit renaming step from the declarative definition remains implicit.

Part I

User's Guide

Chapter 2

Tutorial

2.1 Running the Interpreter

The current implementation of αProlog is an interpreter written in Ocaml and using no libraries or extensions. Therefore, it runs on any computer system that Ocaml 3.06 runs on. If you haven't already, get the αProlog source distribution and install it according to the accompanying instructions.

After starting αProlog, you will see a banner followed by:

```
AlphaProlog 0.3
?-
```

The toplevel loop prompt `?-` indicates that αProlog is expecting a query. In the toplevel loop, you can't define any new types, predicates, or functions, or add any facts or clauses to the database. Since no external declarations have been loaded, we can only ask queries involving built-in predicates/functions.

Let's start with evaluation (`is`). Type

```
?- X is 1 + 2.
```

You should see

```
Yes.
X = 3
```

The interpreter now waits for further input from you. If you type `;` in response to this, the interpreter looks for another solution. In this case, there isn't one:

```
;
No.
```

2.2 Built-in Data and Types

Many of the built-in datatypes of αProlog are familiar from Prolog-like and ML-like languages. They include booleans (written `tt` and `ff`), integers (`-1`,`0`,`1`,`2`,...),

character constants (`'a','b','c'`), string constants (`"abc"`), the unit type (with the sole value `()`), pairs (`(1,2)`, `('a',1)`), and lists (`[]`, `1::[2]`, `[1,2,3]`, `[1|[2,3]]`).

In addition, αProlog supports name and abstraction types. There are no built-in name types, but new name types can be declared as follows:

```
id : name_type.
```

This says that `id` is a name type. Name types are inhabited by infinitely many distinct but "indeterminate" names. We can always come up with a new name of type `id`, just by writing down an unused (lower-case) identifier in a place where an `id` is expected. In addition to being placed in user-defined data structures, names can be used in transpositions, abstractions, and freshness tests.

A *transposition* is a term of the form `(a~b) T` where `a`, `b` are names (of the same type) and T is an arbitrary term. The result is a term with all occurrences of `a` and `b` swapped. Any variables in `T` will be annotated with the transposition as well, indicating that once the variable is instantiated the delayed transposition should be applied to it. For example,

```
?- X = (a~b) (a,b,c)
Yes.
X = (b,a,c)
```

An *abstraction* is a term of the form `a\T` where `a` is a name and `T` is a term. Abstractions describe a set of α-equivalent terms (that is, equivalent up to consistent renaming). Two abstractions are equal if they describe the same such set. For example,

```
?- a\a = b\b.
Yes.
?- a\b = b\b.
No.
?- a\b = c\b.
Yes.
```

It is important to point out that logic variables are not allowed on the left hand side of an abstraction, or as the names in a transposition. Thus, terms like `X\X` and `(X Y)X` are not well-formed.

2.3 Unification and Freshness

αProlog provides two special built-in predicates, = (unification) and # (freshness). For example, unification for ground terms is just syntactic equality:

```
?- 1 = 1.
Yes.
?- 1 = 0.
No.
```

Free variables can be instantiated to terms to solve the equations:

```
?- 1 = X, X = Y, Y = Z.
Yes.
X = 1
Y = 1
Z = 1
?- X = 1, Y = plus X 0, Y = X.
No.
```

In the last query, the `Y = X` subgoal fails because `plus 1 0` is syntactically different from `1`.

Freshness tests are a propositions of the form `a # T`, where `a` is a name and `T` is a term. They test that the name `a` does not occur *free* in `T` (that is, all occurrences are enclosed by an abstraction of `a`.) Any two syntactically different names are assumed to be distinct (in contrast to logic variables, which might eventually be identified through unification.) For example,

```
?- a # a.
No.
?- a # b.
Yes.
?- a # b\a.
No.
?= a # b\b.
Yes.
?- a # a\a.
Yes.
?- a # (a~b) a.
Yes.
```

As the last example illustrates, freshness checking occurs after any transpositions have been applied.

Testing whether terms involving abstractions are equal is more complex than ordinary first-order unification such as Prolog uses. For example:

```
?- x\y\(X1,y) = y\x\(x,X1).
No.

?- x\y\(X2,y) = y\x\(x,X3).
Yes.
X2 = x
X3 = y
```

The results of unification can contain delayed permutations as well as additional freshness constraints.

```
?- x\X = y\Y.
Yes.
X = (x~y)Y

?- x\X = y\X.
Yes.
x # X.
y # X.

?- x\y\(y,X6) = x\x\(x,X7).
Yes.
X6 = (x~y)X7
y # X7
```

In the last example, if `X7` were instantiated to `y`, then we would have `x\y\(x,x) = x\x\(x,x)`, but the `x`'s in the first term are bound to the outer `x` whereas in the second term they are bound to the inner `x`.

Similarly, freshness tests of non-ground terms might generate new constraints.

```
?- x # x\Y.
Yes.
?- x # y\Y.
Yes.
x # Y
?- x # y\Y, x = Y.
No.
```

2.4 Lists and Namespaces

αProlog has a small list library. To import the list library, do:

```
?- #use "list.apl".
```

This indicates that `list.apl` defines a namespace `List` which contains several predicates defining common list operations. To refer to list predicates within the namespace, we prefix them with `List.`.

For example, to append two lists we can do

```
?- List.append([1,2],[3,4],X).
Yes.
X = [1,2,3,4]
```

Of course, as usual in logic programming we can also run predicates "backwards" and nondeterministically:

```
?- List.append(X,Y,[1,2,3,4]).
Yes.
X = []
Y = [1,2,3,4]
;
Yes.
X = [1]
Y = [2,3,4]
;
...
```

or solve for unknowns:

```
?- List.append([1|Y],Y,[1,X,4]).
Yes.
X = 4
Y = [4]
```

Finally, we can "open" a namespace, which makes all of its identifiers (including other namespaces) bound locally so that we don't have to refer to them using explicitly qualified names. For example:

```
?- #open List.
?- append([1|Y],Y,[1,4,4]).
Yes.
Y = [4]
```

Warning: Using `open` incautiously can result in strange behavior. It shadows any previous identifier declarations in the current namespace with new ones; however, existing definitions that referred to the old identifiers will not change. Thus, had we already defined our own version of `append` (possibly having a different type signature), after opening `List` we can no longer refer to this definition by name. However had we also defined a predicate `foo` in terms of the shadowed `append`, `foo` would continue to work.

In the rest of this tutorial we assume that `List` has been opened.

2.5 Defining New Types

Let's define a type representing simple λ-expressions. Since you can't enter declarations at the toplevel, we need to put this into a file. Call it `tutorial.apl`.

To reflect the expression syntax, we need to define two types: a name type for variable identifiers, and the datatype of expressions.

```
id : name_type
exp : type.
```

This says that `id` is a name type and `exp` is a datatype.

Now we want to define the constructors for exp.

```
var : id -> exp.
app : (exp,exp) -> exp.
lam : (id\exp) -> exp.
```

This declares function symbols var, app, and lam with the corresponding types. Thus, given an identifier `v`, we can construct an expression by applying the constructor `var` to it; similarly given expressions `E1` and `E2` we can form expression `app(E1, E2)` and given an expression `E` and a name `a` we can form an expression `lam(a\E)`.

Now save the file and start αProlog as follows:

```
$ aprolog tutorial.apl
```

You should see:

```
alpha-Prolog 0.3
Reading file tutorial.apl...
?-
```

Try entering a few λ-terms (with or without meta-variables) and comparing them for equality or testing freshness.

```
?- lam (a\var a) = lam (b\var b).
Yes.
?- lam (a\var A) = lam (b\var A).
Yes.
a # A
b # A
```

2.6 Predicates

Now let's define a simple predicate on λ-terms: enumerating a list containing all free variables (though possibly with duplicates). Put the following in tutorial.apl:

```
pred fvs (exp,[var]).
fvs(var(v),[v]).
fvs(app(E1,E2),L) :- fvs(E1,L1),fvs(E2,L2),append(L1,L2,L).
fvs(lam(x\E),L) :- fvs(E,L'), remove(x,L',L).
```

Note that we use the list library function to remove (all occurrences of) `x` from the list in the third case.

Fire up αProlog again and try a few queries.

```
?- fvs(lam (x\var x),X).
Yes.
X = []
?- fvs(lam (x\var y),X).
Yes.
X = [y]
?- fvs(app (var y) (var y),X).
Yes.
X = [y,y]
```

Note that duplicate names in the term result in duplicate list entries.

2.7 Definitions

"cnst"

2.8 Functions

Now let's take on a harder problem: defining capture-avoiding substitution. We'll define substitution as a function since we usually want to run it "forward" (i.e., given a term and a substitution, construct the result) rather than "reverse".

To declare a function write (in tutorial.apl):

```
func subst(tm,[(id,tm)]) = tm.
```

This expresses that *subst* is a function symbol taking two arguments: a term and a list of identifier-term pairs (i.e., the subsitution), and producing a term. Now let's define cases. Clauses for function definitions are of the form $f(t_1, \ldots, t_n) = t :- G$, where f is the function being defined, the t_i are the inputs for which the clause defines a value, t is the value, and G is a subgoal that must be solved for the clause to apply. For example

```
subst(var(A),S)     = T                              :- mem((A,T),S).
subst(var(A),S)     = var A                          :- not(mem(A,_),S).
subst(app(T1,T2),S) = app(subst(T1,S),subst(T2,S)).
subst(lam(x\T),S)   = lam(x\subst(T,S))              :- x # S.
```

The first clause asserts that if A is bound to a term T in S then T is the result. The second, that if A is not so bound, then $var(A)$ is the result. The third clause just propagates *subst* past an application. The final clause propagates *subst* past a *lam*, *provided the bound variable x is not mentioned in S*. Since x appears only in an abstraction, it is *always* possible to satisfy this constraint by renaming x (although at times renaming may not be necessary). In fact, this is what αProlog does by default.

Is this definition really correct? Let's try some examples.

```
?- X = subst(var(a), [(a,var(b))]).
Yes.
X = var b
?- X = subst(lam(a\var(a)), [(a,var(b))]).
Yes.
X = lam (a\var(a))
?- X = subst(lam(b\var(a)), [(a,var(b))]).
Yes.
X = lam (b1\var(b))
;
No.
```

Note that in the final case, the bound variable b has been freshened to b_1. Remember that when a name is used in an abstraction, we lose control over its specific value since the abstraction really represents an equivalence class of α-equivalent terms. The last example also verifies that a simple form of variable capture is actually impossible. That is, it's not the case that *subst* happens to get the right answer the first time it succeeds; in fact, no alternative wrong answers are possible.

Now this definition of *subst* has some disadvantages: for example the first two clauses overlap and may repeat the membership test $mem((A, V), S)$. We can make this more efficient using αProlog's "if-then-else" construct:

```
subst(var A,S) = T :- mem ((A,V),S) -> T = V | T = var(A).
```

Similarly, we can collapse the other two clauses into one completely general case using disjunction:

```
subst(T,A) = T' :- ( T = var(A),
                     if mem((A,V),S)
                     then T = V
                     else T = var(A)
                   ; T = app(T1,T2),
                     T' = app(subst(T1,S),subst(T2,S)))
                   ; T = lam(x\T1),
                     x # S,
                     T' = lam(x\subst(T1,S)))
```

Another alternative would be to replace the call to *mem* with a substitution-specific *lookup* function:

```
func lookup(id,[(id,exp)]) = exp.
lookup(A,[]) = var A.
lookup(A,[(A,V)|S]) = V.
lookup(A,[(B,V)|S]) = lookup(A,S).
...
subst(var A,S) = lookup(A,S).
```

Both this and the earlier definition have the potential disadvantage that if duplicate bindings are present in a substitution (e.g., $s = [(a, t_1), (a, t_2)]$), then *lookup* or *mem* can succeed with multiple possible answers. We therefore regard substitutions to be well-formed only if there are no such duplicates.

Chapter 3

Running αProlog

3.1 Debugging Options

3.2 Command Line Options

3.3 Directives

αProlog provides several directives that control the interpreter. In this section we list them and drscribe how they work.

- `#quit` The `#quit` directive makes the interpreter stop running.
- `#help` The `#help` directive prints a help message explaining basic commands.
- `#type` *exp* The `#type` directive typechecks its input and, if the typechecking is successful, prints out a typing judgment that shows the most general types of the expression's variables and names, as well as the type of the expression itself.
- `#trace` *n* The `#trace` directive sets the *trace level* of the interpreter, which is a number between 0 and 3 indicating how much information is printed out during proof search. The default is trace level 0, no information. Trace level 1 prints out each atomic goal as it is solved, as well as backtracking information. Trace level 2 prints out all goals. Trace level 3 prints out all attempted resolutions.
- `#break` *sym* The `#break` directive sets a breakpoint at a given predicate named by *sym*. Execution will halt whenever a successful resolution step with *sym* as its subject occurs. In concert with `#trace`, this can be used for debugging. If no argument is given, then all breakpoints are listed.
- `#clear` *sym* The `#clear` directive clears a breakpoint. If no argument is given, all breakpoints are cleared.
- `#use` *filename* The `#use` directive instructs the intepreter to open and read an external αProlog source file, processing all the declarations, clauses and goals

therein as if they had been typed at the interpreter prompt. The main difference is that queries are printed out as they are executed (to make the output more intelligible) and that the interpreter only looks for at most one solution.

Part II

Language Reference

Chapter 4

Syntax

4.1 Lexical Structure

The text of an αProlog program is divided into contiguous tokens based on the following classifications. At every step, the longest matching rule applies.

4.1.1 Whitespace

Whitespace is a nonempty sequence of space, tab, or newline characters. Except where noted otherwise, whitespace is ignored.

4.1.2 Comments

There are three style of comments in αProlog:

- Prolog-style line comments: lines starting with `%`, as

```
% comment
% comment
noncomment
%comment
```

- C/Prolog-style nonnested block comments delimited with `/*` and `*/`, as

```
/* comment
   comment */
noncomment
/* comment */
```

- ML-style nested block comments delimited with `(*` and `*)`, as

```
(* comment
(* comment *) *)
noncomment
(* comment *)
```

Comments are always ignored in αProlog.

4.1.3 Directives

Directives are commands to the interpreter or compiler. Their behavior is implementation-dependent; implementations may simply ignore them. A directive is a string beginning with '`#`' and ending with '`.`'. Directives may occupy multiple lines. The first non-whitespace character on a line containing a directive must be the leading '`#`', and the last non-whitespace character on the line ending the directive must be the terminating '`.`'.

4.1.4 Symbols

Keywords

The following strings are reserved words (keywords) in αProlog:

```
cnst      exists     ff       forall    func       infixl
infixn    infixr     is       namespace name_type  new
not       pred       true     tt        type
```

Delimiters

αProlog recognizes the following delimiters:

```
( ) [ ] { }
```

Operators

αProlog reserves the following operator symbols:

```
= | , => -> --> :- ? ! : ; # \ ::
```

Identifiers

A textual identifier is a nonempty sequence of letters, digits, apostrophes, and underscores, starting with a letter or underscore, that does not form a reserved keyword. An identifier starting with an underscore or capital letter is called a variable. Otherwise, an identifier is called a symbol if it has been assigned some special meaning and a name if it has not.

The variable '`_`' has a special meaning: distinct instances are taken to refer to different, "anonymous" variables. Accordingly, '`_`' is called the anonymous variable.

An infix identifier is a sequence of the following symbols:

```
| * + < > = - & ^ $ @ ! ~ ?
```

that does not form a reserved operator. Infix identifiers (and their arity and precedence) can be user-defined.

Enclosing an infix identifier in parentheses (`(+)`) means it is treated as a (prefix) textual identifier. Conversely, enclosing a textual identifier in backquotes (`` `sym` ``) means it is treated as an infix operator.

A namespace-qualified identifier is a sequence of textual identifiers separated by `asdf` and terminated by either a textual or infix identifier. For example, `List.mem` and `Int.+` are legal namespace identifiers. A namespace identifier whose subject is infix is treated as prefix, not infix.

4.1.5 Literals

Booleans

There are two Boolean truth value literals, `tt` (true) and `ff` (false).

Integers

A (decimal) integer constant is a sequence of digits `0`–`9` possibly starting with a minus sign `-` insicating a negative number. Other bases are not supported.

Characters

A character constant is a printable ASCII character or an escape sequence enclosed in single quotes.

Strings

A string is a sequence of printable ASCII character or escape sequences enclosed in double quotes.

4.2 Abstract Syntax

4.2.1 Terms

The syntax of αProlog terms is summarized by the following grammar:

$$
\begin{array}{rcl}
t & ::= & () \mid \mathit{int} \mid \mathit{bool} \mid \mathit{char} \mid \mathit{string} \\
& \mid & c \mid f\ t_1 \cdots t_n \mid (t_1, \ldots, t_n) \mid [] \mid t{::}u \mid [t_1, \ldots, t_n] \mid [t_1, \ldots, t_n | t] \mid X \mid \_ \\
& \mid & \mathsf{n} \mid \mathsf{n}\backslash t \mid (\mathsf{n}\tilde{}\mathsf{m})t
\end{array}
$$

4.2.2 Types

The syntax of αProlog types is summarized by the following grammar:

$$\begin{array}{rcl} ty & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{char} \mid \mathsf{string} \\ & \mid & tc \mid tf\ ty_1 \cdots ty_n \mid (ty_1, \ldots, ty_n) \mid [ty] \mid A \mid \_ \\ & \mid & nty \backslash ty \end{array}$$

4.2.3 Formulas

Atomic formulas

$$A \ ::= \ q \mid p\ t_1 \cdots t_n$$

An atomic formula (or predicate) consists of either a single predicate symbol p or a symbol applied to a list of arguments $p\ t_1 \cdots t_n$.

Clauses

The syntax of αProlog program clauses is summarized by the following grammar:

$$P \ ::= \ A \mid A :- G \mid A \longrightarrow D$$

The first two forms are *Horn clauses* (expressed in implicational form). That is, they are either a single predicate (a *fact*) or a predicate qualified by a condition G (a *rule*). The third form is a *definite clause grammar rule*, which says that the nonterminal symbol A can be rewritten to the DCG body D.

Definite clause grammar bodies

$$D \ ::= \ A \mid D, D \mid D; D \mid char \mid string \mid \{G\}$$

A definite clause grammar rule body is either an atomic predicate (representing a nonterminal), sequential composition D, D, disjunciton $D; D$, a literal character or string (a terminal symbol), or a goal enclosed in braces.

Goals

Goals (or queries) are formed using the following syntax:

$$\begin{array}{rcl} G & ::= & \mathsf{true} \mid A \mid t_1\ \mathsf{is}\ t_2 \mid t_1 = t_2 \mid \mathsf{n} \# t \\ & \mid & G_1, G_2 \mid G_1; G_2 \mid ! \mid G_1 \rightarrow G_2 | G_3 \mid \mathsf{not}(G) \end{array}$$

They include the trivial goal true, atomic predicates, evaluation operator is, unification $=$, freshness $\#$, conjunction G, G, disjunction $G; G$, "cut" $!$, if-then-else $G_1 \rightarrow G_2 \mid G_3$, and negation $\mathsf{not}(G)$.

4.2.4 Declarations

`infix`***d*** *sym n* These declarations make an identifier *sym* into be a left-, right-, or non-associative infix operator (where ***d*** is `l`, `r`, or `n` respectively). Such symbols can be either plain infix identifiers or backquoted textual identifiers. The number *n* is a precedence between 1 and 9 indicating how strongly the operator binds relative to other operators.

Chapter 5

Type System

5.1 Basic Types

5.2 Compound Types

5.3 Polymorphism

Chapter 6

Language Extensions

Chapter 7

Libraries

Currently, there are three libraries: `Base` (containing base types and built-in functions), `List` (containing list operations) and `Option` (containing operations on optional values). For each prdicate or function, we summarize its type and expected modes of use (i.e., which arguments can be thought of as inputs or putputs). These modes are not currently part of or checked by αProlog.

7.1 Base

7.2 List

The `list` namespace includes several common list-manipulating predicates.

- `pred append ([A],[A],[A])`
  `mode append (in,in,out)`
  `mode append (out,out,in)`
  Holds if the third argument is the result of appending the first two arguments.

- `pred mem(A,[A])`
  `mode mem(out,in)`
  Holds if the first element is an element of the second.

- `pred concat([[A]],[A])`
  `mode concat(in,out)`
  Holds if the second list is the result of concatenating all the elements of the first list.

- `pred remove(A,[A],[A])`
  `mode remove(in,in,out)`
  Removes all occurrences of the first argument from the second and returns the result as the third.

- `pred reverse([A],[A])`
  `mode reverse(in,out)`
  Holds if the first list is the reverse of the second (and vice versa).

7.3 Option

- `opt : type -> type`
  The type constructor for optional values.

- `none : opt A`
  An optional value where a value is not present.

- `some : A -> opt A`
  An optional value where a value is present.

- `get_opt (opt A,A)`
  `mode get_opt(in,out)`
  Succeeds if the first argument is $some(A)$ and the second argument is A.