Executable Specifications

Given the similarity between natural deduction,

Inference Step

$$\frac{premise_1 \quad \cdots \quad premise_n}{conclusion}\ (condition)$$

Assumption

$$\frac{}{conclusion}$$

and Horn clause logic,

Inference Step P_0 :- $P_1, \ldots, P_n$.

Fact P_0 :- **true**.

it is natural to assume that we can implement our operational semantics in Prolog.

New Syntax

In order to do this we have to rewrite our syntax specification in terms of Prolog terms.

Arithmetic expressions:

$$A ::= n \mid x \mid \mathbf{add}(A, A) \mid \mathbf{sub}(A, A) \mid \mathbf{mult}(A, A)$$

where n is any valid Prolog integer value and x any valid Prolog object name.
Boolean expressions:

$$B ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{eq}(A, A) \mid \mathbf{le}(A, A) \mid \mathbf{not}(B) \mid \mathbf{and}(B, B) \mid \mathbf{or}(B, B)$$

Commands:

$$C ::= \mathbf{skip} \mid \mathbf{assign}(x, A) \mid \mathbf{seq}(C, C) \mid \mathbf{if}(B, C, C) \mid \mathbf{whiledo}(B, C)$$

Note: the **seq** operator is made infix for convenience - just remember that it is left-associative - under certain circumstances you will need to help the Prolog parser out with parentheses around the appropriate terms.

Programs

Example 1:

$$v := 1; \textbf{ if } v \leq 0 \textbf{ then } v := (-1) * v \textbf{ else skip end}$$

$$\textbf{assign}(v, 1) \textbf{ seq } \textbf{if}(\textbf{le}(v, 0), \textbf{assign}(v, \textbf{mult}(-1, v)), \textbf{skip})$$

Example 2:

$$n := 5; y := 1; \textbf{while } 2 \leq n \textbf{ do } (y := n * y; n := n - 1) \textbf{ end}$$

$$\textbf{assign}(n, 5) \textbf{ seq } \textbf{assign}(y, 1) \textbf{ seq } \textbf{whiledo}(\textbf{le}(2, n), \textbf{assign}(y, \textbf{mult}(n, y)) \textbf{ seq } \textbf{assign}(n, \textbf{sub}(n, 1)))$$

State

Prolog does not allow us to pass functions around (it is a first-order language), therefore, we cannot use the definition of state from our natural deduction operational semantics.

However, consider the following,

$$\sigma[m/x][n/y][k/z],$$

This could be interpreted as a list of variable bindings applied to the state σ if we interpret the $[\ldots]$ as list constructors and juxtaposition as a list append operation,

$$\sigma[m/x, n/y, k/z].$$

In Prolog we can model this as the term structure

$$\mathbf{state}([\mathbf{bind}(k, z), \mathbf{bind}(n, y), \mathbf{bind}(m, x)], \mathbf{s})$$

where $k, n, m \in \mathbb{I}$ and $x, y, z \in \mathbf{Loc}$ and s represents an arbitrary state.

State

If the state is the initial state, where

$$\sigma_0[m/x, n/y, k/z]$$

then

$$\textbf{state}([\textbf{bind}(k, z), \textbf{bind}(n, y), \textbf{bind}(m, x)], \textbf{s}0)$$

where **s**0 is a reserved symbol for the initial state in our representation.

Given our representation we have an interesting equivalence, given a state s, then

$$s \equiv \textbf{state}([\,], s)$$

We will make use of this equivalence when encoding our semantic predicates.

State

Since we turned the state representation from a function into a list, we now have to adjust the variable lookup mechanism. We do this via the predicate *lookup*,

```
% the predicate 'lookup(+Variable,+State,-Value)' looks up
% the variable in the state and returns its bound value.
:- dynamic lookup/3.                    % modifiable predicate

lookup(_,s0,0).

lookup(X,state([],S),Val) :-
    lookup(X,S,Val).

lookup(X,state([bind(Val,X)|_],_),Val).

lookup(X,state([_|Rest],S),Val) :-
    lookup(X,state(Rest,S),Val).
```

Arithmetic Expression Summary

Recall our natural deduction definition for arithmetic expressions:

$$\frac{}{(n, \sigma) \mapsto \text{eval}(n)} \quad \text{for } n \in \mathbf{D} \text{ and } \sigma \in \Sigma$$

$$\frac{}{(x, \sigma) \mapsto \sigma(x)} \quad \text{for } x \in \mathbf{Loc} \text{ and } \sigma \in \Sigma$$

$$\frac{(a_0, \sigma) \mapsto k_0 \qquad (a_1, \sigma) \mapsto k_1}{(a_0 + a_1, \sigma) \mapsto k} \quad \text{where } k = k_0 + k_1$$

$$\frac{(a_0, \sigma) \mapsto k_0 \qquad (a_1, \sigma) \mapsto k_1}{(a_0 - a_1, \sigma) \mapsto k} \quad \text{where } k = k_0 - k_1$$

$$\frac{(a_0, \sigma) \mapsto k_0 \qquad (a_1, \sigma) \mapsto k_1}{(a_0 * a_1, \sigma) \mapsto k} \quad \text{where } k = k_0 \times k_1$$

NOTE: k, k_0, $k_1 \in \mathbb{I}$, a_0, $a_1 \in \mathbf{Aexp}$, and $\sigma \in \Sigma$.

Prolog Aexp Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% semantics of arithmetic expressions

(C,_) -->> C :-                         % constants
    int(C),!.

(X,State) -->> Val :-                   % variables
    atom(X),
    lookup(X,State,Val),!.

(add(A,B),State) -->> Val :-            % addition
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA + ValB,!.

(sub(A,B),State) -->> Val :-            % subtraction
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA - ValB,!.

(mult(A,B),State) -->> Val :-           % multiplication
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis ValA * ValB,!.
```

Note: The cut predicate is necessary to make sure that these rules are interpreted as *state transitions*, i.e., once a state transition has occurred in an abstract machine it cannot be undone.

The -->> Predicate

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the predicate '(+Syntax,+State) -->> -SemanticValue' computes
% the semantic value for each syntactic structure

:- op(700,xfx,-->>).
:- dynamic (-->>)/2.                % modifiable predicate
:- multifile (-->>)/2.
```

The ‘xis’ Predicate

```
bash-3.2$ prolog -f prolog-semantics/preamble.pl
%  xis.pl compiled 0.00 sec, 6,960 bytes
% /Users/lutz/Documents/csc501/prolog-semantics/preamble.pl compiled 0.00 sec, 9,532 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.1)
Copyright (c) 1990-2010 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- X is 1 + 2.
X = 3.

?- X xis 1 + 2.
X = 3.

?- X is y + 2.
ERROR: is/2: Arithmetic: ‘y/0’ is not a function

?- X xis y + 2.
X = y+2.

?-
```

Note: both the -->> and the xis predicate are defined in the preamble.pl file.

Suggestion: put the preamble.pl file in a known place such as the super-directory of all your projects, then you can load it as 'prolog -f ../preamble.pl'.

Evaluation of Arithmetic Expressions

Let $ae \equiv (2 * 3) + 5$, prove that the semantic value of this expression in some state s is equal to 11 using the Prolog semantics (assume that the semantics is given in the file 'sem.pl').

```
?- ['sem.pl'].
%  preamble.pl compiled 0.00 sec, 900 bytes
%  xis.pl compiled 0.00 sec, 6,788 bytes
% sem.pl compiled 0.00 sec, 14,164 bytes
true.

?- (add(mult(2,3),5),s) -->> V, V = 11.
V = 11.

?-
```

Evaluation of Arithmetic Expressions

Now, let $ae \equiv x + 1$, where $x \in$ **Loc**, prove that the semantic value of this expression in some state s is equal to $vx + 1$ where `lookup(x,s,vx)`.

```
?- ['sem.pl'].
%  preamble.pl compiled 0.00 sec, 900 bytes
%  xis.pl compiled 0.00 sec, 6,788 bytes
% sem.pl compiled 0.00 sec, 14,164 bytes
true.

?- asserta(lookup(x,s,vx)).
true.

?- (add(x,1),s) -->> vx+1.
true.

?-
```

Note: the predicate 'asserta' is preferable because it inserts the clause at the *top* of the rule database.

Evaluation of Arithmetic Expressions

Now, let $ae \equiv x + 1$, where $x \in$ **Loc**, prove that the semantic value of this expression in the initial state $s0$ equal to 1.

```
?- ['sem.pl'].
%  preamble.pl compiled 0.00 sec, 900 bytes
%  xis.pl compiled 0.00 sec, 6,788 bytes
% sem.pl compiled 0.00 sec, 14,164 bytes
true.

?- (add(x,1),s0) -->> 1.
true.

?-
```

Evaluation of Arithmetic Expressions

Consider the following proof: Let $ae \equiv x + 3 * 5$, where $x \in$ **Loc**, prove that the semantic value of this expression in some state s is equal to $vx + 15$ where `lookup(x,s,vx)`.

```
?- ['sem.pl'].
%  preamble.pl compiled 0.00 sec, 900 bytes
%  xis.pl compiled 0.00 sec, 6,788 bytes
% sem.pl compiled 0.00 sec, 14,164 bytes
true.

?- asserta(lookup(x,s,vx)).
true.

?- (add(x,mult(3,5)),s) -->> vx+15.
true.

?-
```

Theorem Proving with Prolog

Using Prolog as a theorem prover:

1. *The Prolog Meta-Language* – we can **consult** programs, **assert** assumptions, **retract** assumptions, and **query** a program in order to prove a theorem.
2. *Universally Quantified Variables in Queries* – consider the proof,

   ```
   ?- (mult(3,5),s) -->> V, V = 15.
   ```

   can be interpreted in standard FOL as,

   $$\forall s \exists V[(\mathbf{mult}(3,5),s) \longrightarrow\!\!\gg V \wedge =(V,15)].$$

   $\Rightarrow$ We use symbolic **constants** in queries to express universally quantified variables.
3. *Proof Scores* – we can write a proof as a Prolog meta-language program.
4. *Soundness* – under certain circumstances the default resolution rule can be unsound, to avoid this insert the following code into your proof scores:

   ```
   :- set_prolog_flag(occurs_check,true).
   ```

   If you use the 'preamble.pl' file this is done automatically for you.

Theorem Proving with Prolog

The second point on the previous slide deserves some additional attention. Consider for the moment that we would like to prove

$$\forall s[(\mathbf{mult}(3,5),s) \longrightarrow\!\!\gg 15].$$

If we write this blindly as a query using standard Prolog variables, then

```
?- (mult(3,5),S) -->> 15.
```

Now interpreting this query according to Prolog, then

$$\exists s[(\mathbf{mult}(3,5),s) \longrightarrow\!\!\gg 15].$$

That means, this query does *not* prove our intended proof goal.

Theorem Proving with Prolog

From FOL quantification theory we have the following axiom (Universal Generalization). Let p be a predicate and let $y \in U$ be some arbitrarily chosen element of some universe U, then

$$\frac{p(y)}{\therefore \forall x[p(x)]}$$

with $x \in U$. In plain English,

> *If I can show that a predicate holds for an arbitrarily chosen element of some universe, then I can infer that this predicate holds for all elements of that universe.*

With this we can rewrite our query as

```
?- (mult(3,5),y) --> 15.
```

Here the lowercase y is an element of some universe, in this case the States, and therefore, if Prolog can prove this goal, we can conclude that

$$\forall x[(\mathbf{mult}(3,5),x) \dashrightarrow\!\!\gg 15].$$

with $x \in$ States.

We have to be careful with universal generalization; the statement "*some arbitrarily chosen element of some universe*" has a specific meaning:

> *The element is not allowed to reveal its structure or internal state.*

Consequently, the predicate we want to generalize *is not allowed to investigate the structure or state of the element*.

Example: Let Σ be the set of all states and let $\sigma' \in \Sigma$ be some arbitrarily chosen element of that set. Let $p(\sigma') \equiv \sigma'(x) = 20$. Now, applying universal generalization we have,

$$\frac{p(\sigma')}{\therefore \forall\sigma[p(\sigma)]}$$

with $\sigma \in \Sigma$. This is clearly a fallacious argument, there will be many states in which $\sigma(x) \neq 20$. This argument failed because the predicate p investigated the internal structure of σ'.

Example: Let Σ be the set of all states and let $\sigma' \in \Sigma$ be some arbitrarily chosen element of that set. Let $p(\sigma') \equiv \sigma'[20/x](x) = 20$. Now, applying universal generalization we have,

$$\frac{p(\sigma')}{\therefore \forall\sigma[p(\sigma)]}$$

with $\sigma \in \Sigma$. This argument clearly holds because we did not look at the internal structure of the element.

Example: Let $\mathbb{N}$ be the set of all natural numbers and let $7 \in \mathbb{N}$ be some arbitrarily chosen element of that set. Let $p(7) \equiv 7 \leq 100$. Now, applying universal generalization we have,

$$\frac{p(7)}{\therefore \forall k[p(k)]}$$

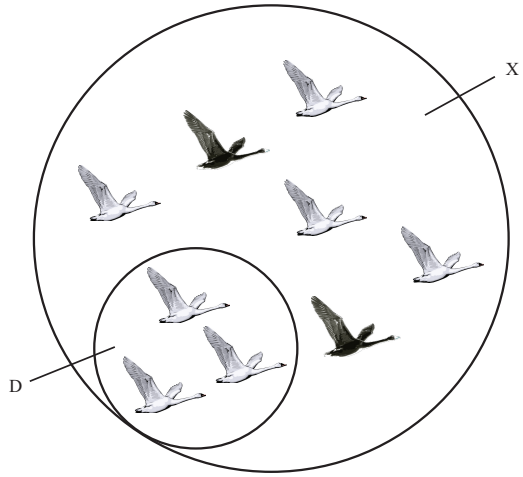
with $k \in \mathbb{N}$. Again, this argument fails because we allowed the predicate to investigate the structure of the element (the value 7).

Example: Let $\mathbb{N}$ be the set of all natural numbers and let $n \in \mathbb{N}$ be some arbitrarily chosen element of that set. Let $p(n) \equiv n \leq n + 100$. Now, applying universal generalization we have,

$$\frac{p(n)}{\therefore \forall k[p(k)]}$$

with $k \in \mathbb{N}$. This argument succeeds because we did not look at the structure (specific value) of the element.

Theorem Proving with Prolog



The *black swan problem* is a classic problem from machine learning theory set forward by mathematician and logician Karl Popper at the beginning of the past century: looking at all the swans in Europe and North America one would conclude that *all swans are white* (set D). However, if you look at the worldwide population of swans (set X) you will also find black swans (in Australia) and therefore the conclusion based on set D is not valid. To avoid these invalid inferences, in machine learning theory this often framed as a probabilistic statement:

> *Based on set D it is* **most likely** *that all swans are white.*

Coming back to our universal generalization problem.

Example: Let X be the set of all swans worldwide, let $x' \in X$ be some arbitrarily selected element of that set. Let,

$$p(x') \equiv x' \text{ is a white swan.}$$

Now, applying universal generalization we have,

$$\frac{p(x')}{\therefore \forall x[p(x)]}$$

with $x \in X$. Again, this argument fails because we allowed the predicate to investigate the structure of the element x' directly by looking at its color.

Proof Scores

```
% load preamble
:- ['preamble.pl'].

% proof1.pl
% Proof score:
%
:- >>> 'Show that'.
:- >>> '(forall x)(forall s)(forall vx)(exists V)[(add(x,mult(3,5)),s)-->>V ^ =(V,vx+15)]'.
:- >>> ' assuming lookup(x,s,vx)'.

% load semantics
:- ['sem.pl'].

% state our assumption
:- asserta(lookup(x,s,vx)).

% run the proof
:- (add(x,mult(3,5)),s)-->>V, V = vx + 15.
```

Expression Equivalence

In our Prolog framework semantic equivalence between arithmetic expression can be formulated as follows:

$$a_0 \sim a_1 \text{ iff } \forall s, \exists V_0, V_1 \left[(a_0, s) \dashrightarrow\!\!\gg V_0 \wedge (a_1, s) \dashrightarrow\!\!\gg V_1 \wedge =(V_0, V_1) \right],$$

for $a_0, a_1 \in \mathbf{Aexp}$.

Expression Equivalence

```
% load preamble
:- ['preamble.pl'].

% proof-equiv.pl
:- >>> ' prove that mult(2,3) ~ add(3,3)'.
%
% show that
% (forall s)(exists V0,V1)
%         [(mult(2,3),s)-->>V0 ^ (add(3,3),s)-->>V1 ^ =(V0,V1)]

% load semantics
:- ['sem.pl'].

% proof
:- (mult(2,3),s)-->>V0 , (add(3,3),s)-->>V1 , V0 = V1.
```

Expression Equivalence

```
?- ['proof-equiv.pl'].
%  preamble.pl compiled 0.00 sec, 924 bytes
>>>  prove that mult(2,3) ~ add(3,3)
%   preamble.pl compiled 0.00 sec, 128 bytes
%   xis.pl compiled 0.00 sec, 6,788 bytes
%  sem.pl compiled 0.00 sec, 12,548 bytes
% proof-equiv.pl compiled 0.01 sec, 14,876 bytes
true.

?-
```

Expression Equivalence

```
% proof-comm.pl
:- ['preamble.pl'].

:- >>> 'prove that add(a0,a1) ~ add(a1,a0)'.
%
% show that
% (forall s,a0,a1)(exists V0,V1)
%         [sem(add(a0,a1),s,V0)^sem(add(a1,a0),s,V1)^=(V0,V1)]
% assuming
% (a0,s) -->> va0.
% (a1,s) -->> va1.

% load semantics
:- ['sem.pl'].

% assumptions on semantic values of expressions
:- asserta((a0,s)-->>va0).
:- asserta((a1,s)-->>va1).

% assumption on integer addition commutativity
:- asserta(comm(A + B, B + A)).

% proof
:- (add(a0,a1),s)-->>V0, (add(a1,a0),s)-->>V1,comm(V0,VC0),VC0=V1.
```