Boolean Expressions

```
(true,_) -->> true :- !.               % constants

(false,_) -->> false :- !.             % constants

(eq(A,B),State) -->> Val :-            % equality
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA =:= ValB),!.

(le(A,B),State) -->> Val :-            % le
    (A,State) -->> ValA,
    (B,State) -->> ValB,
    Val xis (ValA =< ValB),!.
```

Boolean Expressions

```
% Note: we introduce new terms for the Prolog conjunction,
% disjunction, and negation.  We could have used the built-in ',' and ';'
% operators but this would make terms difficult to read.

(not(A),State) -->> Val :-              % not
    (A,State) -->> ValA,
    Val xis (not ValA),!.

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

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

Expression Equivalence

Many of the proofs for Boolean expressions are analogous to proofs for arithmetic expressions.

Of course our notion of equivalence can be applied to boolean expressions:

$$b_0 \sim b_1 \text{ iff } \forall s, \exists B_0, B_1 \left[(b_0, s) \twoheadrightarrow B_0 \wedge (b_1, s) \twoheadrightarrow B_1 \wedge =(B_0, B_1) \right],$$

where $b_0, b_1 \in \mathbf{Bexp}$.

Expression Equivalence

```
% proof-equiv-bool.pl

:- ['preamble.pl'].

:- >>> 'prove that true ~ not(false)'.

% show that
% (forall s)(exists B0,B1)
%          [(true,s)-->>B0 ^ (not(false)s)-->>B1 ^ =(B0,B1)]

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

% proof
:- (true,s)-->>B0,(not(false),s)-->>B1,B0=B1.
```

Commands

```
% semantics of commands

(skip,State) -->> State :- !.          % skip

(assign(X,A),State) -->> OState :-     % assignment
    (A,State) -->> ValA,
    put(X,ValA,State,OState),!.

(seq(C0,C1),State) -->> OState :-      % composition, seq
    (C0,State) -->> S0,
    (C1,S0) -->> OState,!.

(if(B,C0,_),State) -->> OState :-     % if
    (B,State) -->> true,
    (C0,State) -->> OState,!.

(if(B,_,C1),State) -->> OState :-     % if
    (B,State) -->> false,
    (C1,State) -->> OState,!.

(whiledo(B,_),State) -->> OState :-    % while
    (B,State) -->> false,
    State=OState,!.

(whiledo(B,C),State) -->> OState :-    % while
    (B,State) -->> true,
    (C,State) -->> SC,
    (whiledo(B,C),SC) -->> OState,!.
```

Put Predicate

```
% the predicate 'put(+Variable,+Value,+State,-FinalState)' adds
% a binding term to the state.
:- dynamic put/4.                   % modifiable predicate

put(X,Val,S,state([bind(Val,X)],S)) :-
    atom(S).

put(X,Val,state(L,S),state([bind(Val,X)|L],S)).
```

Commands

```
% proof-loop.pl

:- ['preamble.pl'].

:- >>> 'prove that the value of y is equal to 6 when'.
:- >>> 'the following program p:'.
:- >>> '       assign(x,3) seq'.
:- >>> '       assign(y,1) seq'.
:- >>> '       whiledo(le(2,x),'.
:- >>> '               assign(y,mult(y,x)) seq'.
:- >>> '               assign(x,sub(x,1))'.
:- >>> '       )'.
:- >>> 'is run in the context of any state'.

% We need to prove
%            (forall s)(exists SF)[(p,s)-->>SF ^ lookup(y,SF,6)]

% proof
:- ['sem.pl'].

% A nice coding trick for long proofs:
%   'program' is a predicate that holds our program
program(assign(x,3) seq
        assign(y,1) seq
        whiledo(le(2,x),
            assign(y,mult(y,x)) seq
            assign(x,sub(x,1)))).

:- program(P),(P,s)-->>SF,lookup(y,SF,6).
```

Commands

It is interesting to run this proof interactively rather than just as a proof script in order to see the variable unifications:

```
?- ['proof-loop.pl'].
%   xis.pl compiled 0.00 sec, 6,920 bytes
%  preamble.pl compiled 0.00 sec, 8,108 bytes
>>> prove that the value of y is equal to 6 when
>>> the following program p:
>>>        assign(x,3) seq
>>>        assign(y,1) seq
>>>        whiledo(le(2,x),
>>>                assign(y,mult(y,x)) seq
>>>                assign(x,sub(x,1))
>>>        )
>>> is run in the context of any state
%    xis.pl compiled 0.00 sec, 136 bytes
%   preamble.pl compiled 0.00 sec, 264 bytes
%   xis.pl compiled 0.00 sec, 136 bytes
%  sem.pl compiled 0.00 sec, 5,960 bytes
% proof-loop.pl compiled 0.00 sec, 16,380 bytes
true.

?- program(P),(P,s)-->>SF,lookup(y,SF,6).
P = (assign(x, 3)seq assign(y, 1)seq whiledo(le(2, x), (assign(y, mult(y, x))seq assign(x, sub( x, 1))))),
SF = state([bind(1, x), bind(6, y), bind(2, x), bind(3, y), bind(1, y), bind(3, x)], s)

?-
```

Command Equivalence

We can consider the semantic equivalence of commands as follows:

$$\begin{array}{l} c_0 \sim c_1 \text{ iff} \\ \forall s, x, \exists S_0, S_1, K_0, K_1 \\ \left[(c_0, s) \longrightarrow\!\!\!\gg S_0 \wedge (c_1, s) \longrightarrow\!\!\!\gg S_1 \wedge \text{lookup}(x, S_0, K_0) \wedge \text{lookup}(x, S_1, K_1) \wedge =\!(K_0, K1)\right], \end{array}$$

where $s, S_0, S_1 \in \mathbb{S}$, $x \in \mathbf{Loc}$, $K_0, K_1 \in \mathbb{N}$, and $c_0, c_1 \in \mathbf{Com}$.

NOTE: Two commands are equivalent if they result in states that are indistinguishable under variable evaluations.

Command Equivalence

```
% proof-equiv-command.pl

:- ['preamble.pl'].

:- >>> 'prove that'.
:- >>> '   assign(x,1) seq assign(y,2) ~ assign(y,2) seq assign(x,1)'.

% We need to show that
% (forall s,z)(exist S0,S1,V0,V1)
%     [(assign(x,1) seq assign(y,2),s)-->>S0 ^
%      (assign(y,2) seq assign(x,1),s)-->>S1 ^
%      lookup(z,S0,V0) ^ lookup(z,S1,V1) ^ =(V0,V1)]
% assume lookup(z,s,vz)

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

% assumption
:- asserta(lookup(z,s,vz)).
```

Command Equivalence

```
:- >>> 'we show equivalence by case analysis on z' .
:- >>> 'case z = x'.
:- ((assign(x,1) seq assign(y,2)),s)-->>S0,
   ((assign(y,2) seq assign(x,1)),s)-->>S1,
   lookup(x,S0,V0),
   lookup(x,S1,V1),
   V0=V1.

:- >>> 'case z = y'.
:- ((assign(x,1) seq assign(y,2)),s)-->>S0,
   ((assign(y,2) seq assign(x,1)),s)-->>S1,
   lookup(y,S0,V0),
   lookup(y,S1,V1),
   V0=V1.

:- >>> 'case z =/= x and z =/= y'.
:- ((assign(x,1) seq assign(y,2)),s)-->>S0,
   ((assign(y,2) seq assign(x,1)),s)-->>S1,
   lookup(z,S0,V0),
   lookup(z,S1,V1),
   V0=V1.
```

Command Equivalence

Running the proof score

```
?- ['proof-equiv-command.pl'].
%  xis.pl compiled 0.01 sec, 6,920 bytes
% preamble.pl compiled 0.01 sec, 8,108 bytes
>>> prove that
>>>    assign(x,1) seq assign(y,2) ~ assign(y,2) seq assign(x,1)
%   xis.pl compiled 0.00 sec, 136 bytes
%  preamble.pl compiled 0.00 sec, 264 bytes
%  xis.pl compiled 0.00 sec, 136 bytes
% sem.pl compiled 0.00 sec, 5,960 bytes
>>> we show equivalence by case analysis on z
>>> case z = x
>>> case z = y
>>> case z =/= x and z =/= y
% proof-equiv-command.pl compiled 0.01 sec, 15,988 bytes
true.

?-
```

Inductive Proofs

We can use Prolog to construct proofs that perform induction over the syntax ... structural induction.

Example. Prove that all arithmetic expressions a terminate. Formally,

$$\forall a, \forall s, \exists K \left[(a, s) \longrightarrow\!\!\!\gg K \right]$$

Inductive Proofs

```
% proof-aexp-induction.pl

:- ['preamble.pl'].

:- >>> 'show that all arithmetic operations terminate'.
:- >>> '  (forall a)(forall s)(exists K)[(a,s)-->>K]'.
:- >>> 'we prove this by structural induction on arithmetic expressions'.

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

:- >>> 'base case: variables'.
%%% assumption: lookup will always produce a value
:- asserta(lookup(x,s,vx)).
%%% proof
:- (x,s)-->>vx.
%%% clean up
:- retract(lookup(x,s,vx)).

:- >>> 'base case: integer values'.
%%% assumption: n is an integer
:- asserta(int(n)).
%%% proof
:- (n,s)-->>n.
%%% clean up
:- retract(int(n)).
```

Inductive Proofs

```
:- >>> 'inductive step: add(a0,a1)'.
%%% inductive hypotheses
:- asserta((a0,s)-->>va0).
:- asserta((a1,s)-->>va1).
%%% induction step
:- (add(a0,a1),s)-->>va0+va1.
%%% clean up
:- retract((a0,s)-->>va0).
:- retract((a1,s)-->>va1).

:- >>> 'the remaining operators can be proved similarly'.
```

Inductive Proofs

```
?- ['proof-aexp-induction.pl'].
%   xis.pl compiled 0.00 sec, 6,920 bytes
%  preamble.pl compiled 0.00 sec, 8,108 bytes
>>> show that all arithmetic operations terminate
>>>   (forall a)(forall s)(exists K)[(a,s)-->>K]
>>> we prove this by structural induction on arithmetic expressions
%    xis.pl compiled 0.00 sec, 136 bytes
%   preamble.pl compiled 0.00 sec, 264 bytes
%   xis.pl compiled 0.00 sec, 136 bytes
%  sem.pl compiled 0.00 sec, 5,960 bytes
>>> base case: variables
>>> base case: integer values
>>> inductive step: add(a0,a1)
>>> the remaining operators can be proved similarly
% proof-aexp-induction.pl compiled 0.01 sec, 16,188 bytes
true.

?-
```

Extending the Language

Problem: Extend the language defined so far with a *modulo* operator, say

$$A ::= \mathbf{mod}(A, A)$$

The behavior of this operation is defined as follows:

$$\mathbf{mod}(3, 2) = 1$$
$$\mathbf{mod}(4, 2) = 0$$

The operations of the type $\mathbf{mod}(3, 0)$ are not defined.

Hint: Prolog has a builtin operation called `rem` for *remainder*,

```
+IntExpr1 rem +IntExpr2                                              [ISO]
    Remainder of integer division.  Behaves as if defined by
    Result is IntExpr1 - (IntExpr1 // IntExpr2)  * IntExpr2
    where // is integer division (the result is truncated).
```

Use this operator to implement the semantics of **mod**.

Extending the Language

It is straight forward to write the semantics of this modulo operator based on other arithmetic operators

```
(mod(A,B),State) -->> Val :-        % modulo
     (A,State) -->> ValA,
     (B,State) -->> ValB,
     Val xis ValA rem ValB,!.
```

Pragmatics

- Always use `asserta` instead of `assert` for stating assumptions. The former puts your assumption at the beginning of the definition of the predicate you are asserting whereas the latter puts your assumption at the end. Given that assumptions are typically more specific than the rules already in your database you want your assumptions at the beginning!
- Always `retract` your assumptions when no longer needed.
- Never evaluate an integer term with the `is`, always use the `xis` predicate.
- Make use of the `trace` facility in order to debug your proofs.
- Use the `listing` predicate to see the state of your database.
- When debugging proofs it is often helpful to break it into its individual sub-goals and try to prove the individual sub-goals interactively.

Assignment

see website for assignment