Dealing with the Choice Operator in HOL88[1]

by

Brian Graham

Computer Science Department, University of Calgary
Calgary, Alberta, Canada T2N 1N4
Tel: (403) 220 7691
Fax: (403) 284 4707
Net: grahamb@cpsc.ucalgary.ca

April 3, 1990

Abstract

This paper discusses the choice operator @ in HOL88. It gives an intuitive interpretation, as well as the definition, and presents proof strategies for goals that contain this operator, broken down into cases based on the number of values which satisfy the predicate argument. A working knowlege of the HOL system is assumed.

[1]supported in part by Contract No.W2213-8-6362/01-SS with the Department of National Defence

My recent work on the proof of the SECD microprocessor ([GB89], [SGB89], [BG90]) has suffered from close encounters with the choice operator ("`@`"). It is a powerful device for making definitions, but we have quite limited capabilities of manipulating this constant. This is unfortunate, as it shows up in many fundamental definitions, including the definitions of *REP* functions created when new types are defined. It is also used in the definition of the temporal abstraction function *TimeOf*[2], which is used when relating two granularities of time in hardware representation.

I thought it useful to draw together all the axioms, rules, conversions, and tactics concerned with it, and suggest some techniques to handle it in (backward) proofs.

1 What the HOL88 System Supplies

The constant "`$@`" is a binder, having the type `$@:(*->bool)->*`. In [Cam89a], it is described as follows:

> ...if t is a term having type σ`->bool`, then `@x.`t `x` (or, equivalently, `$@`t) denotes *some* member of the set whose characteristic function is t. If the set is empty, then `@x.`t `x` denotes an arbitrary member of the set denoted by σ. The constant `@` is a higher order version of Hilbert's ε-operator; it is related to the constant ι in Church's formulation of higher order logic. For more details, see Church's original paper [Chu40], Leisenring's book on Hilbert's ε-symbol [Lei69], or Andrews' textbook on type theory [And86].

When applied to a predicate, it returns an arbitrary value of the correct type that satisfies the predicate, and in the case where no such value exists, it returns an arbitrary but undetermined value of the correct type. It can be used to make a total function from a partial function.

The following listing includes all built-in tools in the HOL88 system that are specific to the choice operator.

1.1 Axioms

```
SELECT_AX : thm                    (in theory "bool")

|- !(P:* -> bool) (x:*). P x ==> P($@ P)
```

[2]This function was defined by Tom Melham in [Mel88] and Inder Dhingra in [Dhi88].

1.2 Forward Inference Rules

```
    SELECT_INTRO : (thm -> thm)

        A |- P t
      -----------------
       A |- P($@ P)


    SELECT_ELIM : (thm -> (term # thm -> thm)) (cases)

       A1 |- P($@ P)    ,    A2, "P v" |- t
      ------------------------------------------ (v occurs nowhere
                  A1 u A2 |- t    except in "P v")

    SELECT_RULE : (thm -> thm)

        A |- ?x. t[x]
      -----------------
       A |- t[@x.t[x]]


    SELECT_EQ : (term -> thm -> thm) (@ abstraction)

              A |- t1 = t2
         -----------------------
          A |- (@x.t1) = (@x.t2)
```

1.3 Conversions

```
    SELECT_CONV : conv

     "P [@x.P [x]]" ---> |- P [@x.P [x]] = ?x. P[x]
```

1.4 Tactics

```
    SELECT_TAC : (term -> tactic)          ( term = @x.P(x) )
(found in start_groups.ml in the "group" library)

        [A] P[@x.P(x)]
    ======================
         [A] ?x.P(x)
```

2 Manipulating the Choice Operator

We distinguish three cases for the values that the operator may return:

1. the predicate holds for no values, so the operator returns an arbitrary but unspecified value of the correct type.
2. the predicate holds for a single unique value.
3. the predicate holds for more than one value.

We deal with these cases individually in Sections 2.1 to 2.3.

2.1 No values satisfy

It is useful to observe the equivalence given by SELECT_CONV. The SELECT'ed value satisfies the predicate if and only if some value exists that satisfies the predicate. From this we can prove the following is false:

```
((@x. Q x) = y)  ==> Q y
```

(*i.e.* if the value at which Q holds equals y then Q holds at y.) By using AP_TERM *Q* on the lhs of the implication, we get:

```
(Q(@x. Q x) = Q y) ==> Q y
```

Using SELECT_CONV on the lhs gives:

```
((?x. Q x) = Q y) ==> Q y
```

If no value exists, then the equation simplifies to:

```
(F = F) ==> F
```

which is clearly false.

In short this means that it is not possible to prove anything about the specific value returned by the choice operator *unless* some value exists at which the predicate holds. This is relevent to the attempt to define an 'arbitrary' value of the form "`@x.F`". It is not possible to prove inequality of this term to any value of the appropriate type, but all such arbitrary terms are equal.

2.2 One value satisfies

In the case where a predicate is satisfied by a unique value, the choice operator applied to the predicate can be shown to be equivalent to that unique value. We can prove:

```
|- (?!x. Q x) ==> Q c ==> ((@x. Q x) = c)
```

The definition of "`$?!`" is:

```
|- $?! = (\P. $? P /\ (!x y. P x /\ P y ==> (x = y)))
```

From "`Q c`" one can derive "`?x. Q x`" (using EXISTS), so we can reduce the earlier theorem to :

```
|- (!x y. Q x /\ Q y ==> (x = y)) ==> Q c ==> ((@x. Q x) = c)
```

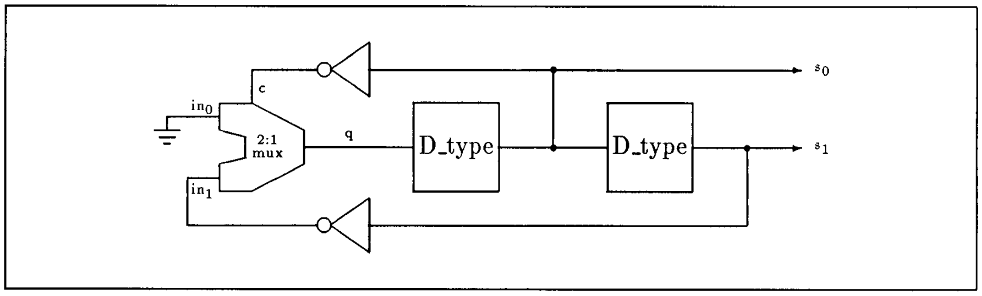
This has been implemented as a forward inference rule and tactic (see Appendix A for definitions). SELECT_UNIQUE_RULE takes a pair of terms, a theorem that the predicate holds at the second term, and a theorem that the predicate has a unique value that satisfies it, and generates a theorem that the value returned by the choice operator applied to the predicate is the unique value.

```
SELECT_UNIQUE_RULE : (term # term) -> thm -> thm -> thm

("x","y")   A1 |- Q[y]  A2 |- !x y.(Q[x] ∧ Q[y]) ==> (x=y)
=========================================================
 A1 U A2 |- (@x.Q[x]) = y


SELECT_UNIQUE_TAC : tactic

        [ A ] "(@x. Q[x]) = y"
====================================================
[ A ] "Q[y]"   [ A ] "!x y.(Q[x] ∧ Q[y]) ==> (x=y)"
```

An example problem using this tactic arose in the proof of the correctness of the initial state of the SECD machine. The essentials of the situation in that complex system are captured in the following simple example.

We define a circuit composed of a mux, 2 inverters, and 2 D-type flip flops. We sample all lines once per clock cycle, and assume that the D-types start up with the value F initially (a forced reset at a lower level of description). Definitions for all components are in Appendix B. Here we define only the circuit implementation:

```
let circuit_imp = new_definition
(`circuit_imp`,
 "circuit_imp s0 s1 =
  ? q c in0 in1:num->bool.
    (D_type q s0)  ∧
    (D_type s0 s1) ∧
    (inv s1 in1)   ∧
    (inv s0 c)     ∧
    (mux2 c in0 in1 q) ∧
    (gnd in0)"
 );;
```



This circuit implements a modulo 3 counter (convince yourself). In order to show that the first time that the s1 line is asserted is at time t = 2, we establish the goal:

```
"TimeOf s1 0 = 2"
    [ "circuit_imp s0 s1" ]
```

Note that this use of the TimeOf function defines a mapping from the clock cycle time granularity to one that is sampled only when the *s1* line is asserted.

The definition of TimeOf is:

```
|- !f n. TimeOf f n = (@t. IsTimeOf n f t).
```

After rewriting the goal with this definition, the choice operator is introduced.

```
"(@t. IsTimeOf 0 s1 t) = 2"
    [ "circuit_imp s0 s1" ]
```

Applying SELECT_UNIQUE_TAC reduces the goal to two subgoals, neither of which contains the choice operator.

```
"!t t'. IsTimeOf 0 s1 t ∧ IsTimeOf 0 s1 t' ==> (t = t')"
    [ "circuit_imp s0 s1" ]

"IsTimeOf 0 s1 2"
    [ "circuit_imp s0 s1" ]
```

The proof is simple. The first goal is proven by an existing theorem IsTimeOf_IDENTITY[3]:

```
|- !n f t1 t2. IsTimeOf n f t1 /\ IsTimeOf n f t2 ==> (t1 = t2)
```

[3]See Appendix C.

The second is solved by rewriting with the definition of IsTimeOf, and the various component definitions. (A complete proof is given in Appendix B.)

The use of SELECT_UNIQUE_TAC has permitted us to eliminate the choice operator entirely from the subgoals, and prove that the value it represents is equal to a specific value. Thus time *0* of the coarser granularity of time corresponds to time *2* at the finer granularity.

2.3 More than one value satisfies

When the choice operator is applied to a predicate that is satisfied at multiple different values, there is no way to determine which specific value the expression represents. One can only make use of the fact that the predicate is satisfied by the value. An approach to proof in this situation is supplied by Elsa Gunter, and contained within the *start_groups.ml* file in the *group* library[4]. The following tactic is provided:

```
SUPPOSE_TAC : term -> tactic  (term = t)

          [A] t1
   ======================
       [A;t] t1    [A] t
```

(There is also a REV_SUPPOSE_TAC which merely reverses the order of the subgoals.) The idea is to add an assumption that will be useful in solving the goal, and also requiring that the assumtion itself be proved. This approach is useful in splitting a goal into major subgoals, and proceeding along one branch, while deferring some of the proof steps to a more convenient time. This tactic can be useful on a goal of the form:

```
[ A ] "Q (@x. P x)"
```

`SUPPOSE_TAC "?x. P x"` gives the following subgoals:

```
[ A ; "?x. P x" ] "Q (@x. P x)"
[ A ] "?x. P x"
```

In the first subgoal, using SELECT_RULE on the new assumption gives:

```
[ A ; "P (@x. P x)" ] "Q (@x. P x)"
```

The properties given by the assumption `"P (@x. P x)"` may be helpful in solving the goal.

We revert to the previously given circuit for a concrete example. We start with the goal:

[4]See the Modular Arithmetic Case Study in [Cam89c].

```
"~s0(SUC(TimeOf s1 n))"
    [ "circuit_imp s0 s1" ]
```

We want to show that the line s0 is always F following the point corresponding to the coarser granularity of time. Using the definition of TimeOf to rewrite the goal,

```
"~s0(SUC(@t. IsTimeOf n s1 t))"
    [ "circuit_imp s0 s1" ]
```

followed by the tactic: `SUPPOSE_TAC "!n:num. ?t. IsTimeOf n s1 t"`, we get 2 subgoals:

```
"!n. ?t. IsTimeOf n s1 t"
    [ "circuit_imp s0 s1" ]

"~s0(SUC(@t. IsTimeOf n s1 t))"
    [ "circuit_imp s0 s1" ]
    [ "!n. ?t. IsTimeOf n s1 t" ]
```

The upper subgoal corresponds to proving a *liveness* property for the circuit: there is a finer grain time *t* that corresponds to every point in the coarser grain time. In effect, the predicate *s1 t* is satisfied infinitely often. The lower subgoal will permit us to use this property and the definition of the circuit in its proof. Thus we have effectively separated the goal into 2 distinct parts: the *liveness* of the circuit, and the essential property of the circuit that we are trying to prove. Tackling the bottom subgoal, we can apply SELECT_RULE to the bottom assumption to obtain

```
"IsTimeOf n s1(@t. IsTimeOf n s1 t)",
```

and resolve this with the theorem IsTimeOf_TRUE:

```
IsTimeOf_TRUE = |- !n f t.  IsTimeOf n f t ==> f t
```

to get the goal:

```
"~s0(SUC(@t. IsTimeOf n s1 t))"
    [ "circuit_imp s0 s1" ]
    [ "IsTimeOf n s1(@t. IsTimeOf n s1 t)" ]
    [ "s1(@t. IsTimeOf n s1 t)" ]
```

Resolving the first assumption with an unwound circuit definition:

```
circuit_unwound =
|- circuit_imp s0 s1 ==>
   (!t.  s0 0 = F) /\x
   (!t.  s0(SUC t) = (( s0 t) =>  s1 t | F)) /\
   (!t.  s1 0 = F) /\
   (!t.  s1(SUC t) = s0 t)
```

then rewriting with the theorem same_branches:

```
same_branches = |- !x (y:*). x => y | y = y,
```

solves this subgoal.

The previous subgoal can be solved using another application of SUPPOSE_TAC, with the assumption "Inf s1", giving the subgoals:

```
"Inf s1"
    [ "circuit_imp s0 s1" ]

"!n. ?t. IsTimeOf n s1 t"
    [ "circuit_imp s0 s1" ]
    [ "Inf s1" ]
```

The bottom subgoal is solved by resolving with the theorem IsTimeOf_EXISTS:

```
|- !f. Inf f ==> !n. ?t. IsTimeOf n f t.
```

The last subgoal requires proving that s1 is asserted infinitely often. This simple subgoal can be proven by a surprisingly lengthy proof using induction. It is provided in Appendix B for the interested reader.

It should be noted that SUPPOSE_TAC is more generally useful than just the cases given here, and it can be applied in the case where the predicate holds for only one value as well.

3 Conclusions

We have attempted to provide some useful examples of the approach to solving problems that involve the choice operator in HOL. An informal analysis based on the number of values satisfying the predicate to which the choice operator was applied, examined what is provable about goals containing the operator. A new tactic and inference rule for the case where a unique value satisfies the predicate were supplied. The techniques described were developed during the verification of the SECD microprocessor design.

Acknowledgements

This work was supported by Strategic, Operating, and Equipment Grants from the Natural Sciences and Engineering Research Council of Canada and The Canadian Microelectronics Corporation. The Strategic Grant was also supported by The Alberta Microelectronic Centre and LSI Canada Inc. The SECD verification effort was also supported by The Communication Research Establishment, Ottawa. We are indebted to Mike Gordon, Tom Melham, and Inder Dhingra for assistance in learning and using the HOL system, and to Elsa Gunter, whose initial input on this topic led to this report.

References

[And86] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Computer Science and Applied Mathematics Series. Academic Press, 1986.

[BG90] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Formal Methods for VLSI Design.* Proceedings of the 1990 IFIP WG 10.5 Summer School to be held at Lyngby, Denmark, 1990.

[Cam89a] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Description*, 1989.

[Cam89b] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Reference Manual*, 1989.

[Cam89c] Cambridge Research Center, SRI International, Cambridge, England. *The HOL System: Tutorial*, 1989.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Dhi88] I. S. Dhingra. *Formal Validation of an Integrated Circuit Design Style.* PhD thesis, University of Cambridge Computer Laboratory, 1988.

[GB89] B. Graham and G. Birtwistle. Formalising the Design of an SECD chip. In *Proceedings of the Cornell Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, New York, 1989. Springer-Verlag.

[Lei69] A. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol.* University Mathematical Series. Macdonald & Co. Ltd., London, 1969.

[Mel88] T. F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Norwell, Massachusetts, 1988. Kluwer.

[SGB89] T. Simpson, B. Graham, and G. Birtwistle. From LispKit to SECD Chip: Some Steps on the way to a Verified System. In *Proceedings of the Third Banff Verification Workshop*, 1989. submitted for publication.

A SELECT_UNIQUE.ml

```
%
SELECT_UNIQUE_RULE:
===================

("x","y")   A1 |- Q[y]  A2 |- !x y.(Q[x]/\Q[y]) ==> (x=y)
=========================================================
A1 U A2 |- (@x.Q[x]) = y

Permits substitution for values specified by the Hilbert Choice
operator with a specific value, if and only if unique existance
of the specific value is proven.
%

let SELECT_UNIQUE_RULE (x,y) th1 th2 =
  let Q = mk_abs (x, subst [x,y] (concl th1))
  in
  let th1' = SUBST [SYM (BETA_CONV "^Q ^y"), "b:bool"] "b:bool" th1
  in
  (MP (SPECL ["$@ ^Q"; y] th2)
      (CONJ (in1_conv_rule BETA_CONV (SELECT_INTRO th1')) th1));;


%
SELECT_UNIQUE_TAC:
==================

        [ A ] "(@x. Q[x]) = y"
=====================================================
        [ A ] "Q[y]"   [ A ] "!x y.(Q[x]/\Q[y]) ==> (x=y)"

Given a goal that requires proof of the value specified by the
Hilbert choice operator, it returns 2 subgoals:
  1. "y" satisfies the predicate, and
  2. unique existance of the value that satisfies the predicate.
%

let SELECT_UNIQUE_TAC:tactic (gl,g) =
  let Q,y = dest_eq g
  in
  let x,Qx = dest_select Q
  in
  let x' = variant (x.freesl(g.gl))x
  in
  let Qx' = subst [x', x] Qx
  in
  ([gl,subst [y,x]Qx;
    gl, "!^x ^x'. (^Qx /\ ^Qx') ==> (^x = ^x')"],
   (\thl. SELECT_UNIQUE_RULE (x,y) (hd thl) (hd (tl thl))));;
```

B hilbert.ml

```
new_theory `hilbert`;;

new_parent `when`;;

loadt `SELECT_UNIQUE`;;
loadf `start_groups`;;
load_all `when`;;

% *************************************************************** %
let mux2 = new_definition
(`mux2`,
 "!cnt in0 in1 out:num->bool.
   mux2 cnt in0 in1 out = !t:num. out t = cnt t => in1 t | in0 t"
 );;

let inv = new_definition
(`inv`,
 "!in out:num->bool. inv in out = !t. out t = ~in t"
 );;

let D_type = new_definition
(`D_type`,
 "!in out:num->bool.
   D_type in out =
     (out 0 = F) /\
     !t. out (SUC t) = in t"
 );;

let gnd = new_definition
(`gnd`,
 "!p:num->bool. gnd p = !t. p t = F"
 );;

let circuit_imp = new_definition
(`circuit_imp`,
 "circuit_imp s0 s1 =
  ? q c in0 in1:num->bool.
    (D_type q s0)  /\
    (D_type s0 s1) /\
    (inv s1 in1)   /\
    (inv s0 c)     /\
    (mux2 c in0 in1 q) /\
    (gnd in0)"
 );;

% *************************************************************** %
let lemma_0 = TAC_PROOF
(([], "circuit_imp s0 s1 ==> ~s1 0"),
 prt[circuit_imp; D_type]
 THEN DISCH_THEN (REPEAT_TCL CHOOSE_THEN (\th.rt[th]))
 );;
```

```
let lemma_1 = TAC_PROOF
(([], "circuit_imp s0 s1 ==> ~s1 1"),
 prt[circuit_imp; D_type; num_CONV "1"]
 THEN DISCH_THEN (REPEAT_TCL CHOOSE_THEN (\th.rt[th])));;

let lemma_2 = TAC_PROOF
(([], "circuit_imp s0 s1 ==> s1 2"),
 prt[circuit_imp; D_type; inv; mux2; num_CONV "2"; num_CONV "1"]
 THEN DISCH_THEN (REPEAT_TCL CHOOSE_THEN (\th. rt[th])));;

% ***************************************************************** %
set_goal(["circuit_imp s0 s1"], "TimeOf s1 0 = 2");;

expand (port[TimeOf]);;
expand (SELECT_UNIQUE_TAC);;
rotate 1;;
expand (rt[IsTimeOf_IDENTITY]);;
expand (port[IsTimeOf]);;
expand (IMP_RES_THEN (\th.rt[th]) lemma_2);;
expand (re_conv_tac num_CONV);;
expand (GEN_TAC THEN prt[LESS_THM] THEN STRIP_TAC);;

expand (art[]);;
expand (port[(SYM o num_CONV) "1"]);;
expand (IMP_RES_THEN ACCEPT_TAC lemma_1);;

expand (art[]);;
expand (IMP_RES_THEN ACCEPT_TAC lemma_0);;

expand (IMP_RES_TAC NOT_LESS_0);;
let initial_thm = save_top_thm `initial_thm`;;

% ***************************************************************** %
let same_branches = TAC_PROOF (([],"!x (y:*). x => y | y = y"),
GEN_TAC THEN GEN_TAC THEN COND_CASES_TAC THEN rt[]);;

let not_both = TAC_PROOF
((["circuit_imp s0 s1"], "!t:num. s1 t ==> ~s0 t"),
 INDUCT_TAC
 THENL
 [ IMP_RES_TAC lemma_0
   THEN art[]
 ; RULE_ASSUM_TAC (prr[circuit_imp; D_type; inv; mux2; gnd])
   THEN FIRST_ASSUM
     (\th.(CHOOSE_THEN (REPEAT_TCL CHOOSE_THEN ASSUME_TAC))th ?
      NO_TAC)
   THEN art[]
   THEN ASM_CASES_TAC "(s1:num->bool) t"
   THENL
   [ RES_TAC
     THEN art[]
   ; ASM_CASES_TAC "(s0:num->bool) t"
     THEN art[]
   ]
 ]);;

let circuit_unwound =
(fst o EQ_IMP_RULE o SPEC_ALL o (prr[D_type; inv; mux2; gnd]))
  circuit_imp;;

% ***************************************************************** %
set_goal (["circuit_imp s0 s1"], "~s0 (SUC(TimeOf s1 n))");;

expand (port[TimeOf]);;
expand (SUPPOSE_TAC "!n:num. ?t. IsTimeOf n s1 t");;

 expand (poke (SELECT_RULE o (SPEC "n:num")));;
 expand (IMP_RES_TAC IsTimeOf_TRUE);;
 expand (IMP_RES_THEN STRIP_ASSUME_TAC circuit_unwound);;
 expand (art[same_branches]);;

 expand (SUPPOSE_TAC "Inf s1");;

  expand (IMP_RES_THEN (\th.rt[th]) IsTimeOf_EXISTS);;

   expand (port[Inf]);;
   expand (INDUCT_TAC);;

    expand (EXISTS_TAC "2"
            THEN CONJ_TAC
    THENL
    [ prt[num_CONV "2"; LESS_0]
    ; IMP_RES_THEN ACCEPT_TAC lemma_2
    ]);;

    expand (poke (porr[SYM_RULE LESS_MONO_EQ]));;
    expand (poke (porr[LESS_THM]));;
    expand (POP_ASSUM STRIP_ASSUME_TAC);;

     expand (IMP_RES_THEN STRIP_ASSUME_TAC circuit_unwound);;
     expand (EXISTS_TAC "SUC (SUC (SUC t'))");;
     expand (art[LESS_THM; same_branches]);;
     expand (IMP_RES_TAC not_both);;

     expand (EXISTS_TAC "t':num" THEN art[]);;
let s0_thm = save_top_thm `s0_thm`;;

% ***************************************************************** %
```

C when.ml

```
%------------------------------------------------------------------------
| FILE : when.ml
|
| DESCRIPTION : Defines the predicates `Next`, `Inf`, `IsTimeOf`
|   and `TimeOf` and derives several major theorems
|   which provide a basis for temporal abstraction.
|
|   These predicates and theorems are taken from
|   T.Melham's paper, "Abstraction Mechanisms for
|   Hardware Verification", Hardware Verification
|   Workshop, University of Calgary, January 1987.
|
|                This file was written by I.S.Dhingra.
|
------------------------------------------------------------------------%

new_theory `when`;;

let Next = new_definition
  (`Next`,
   "Next t1 t2 f  =  (t1<t2)  /\
                     ( f t2)  /\
                 !t. (t1<t )  /\  (t<t2)  ==>  ~f t"
  );;

let IsTimeOf = new_prim_rec_definition
  (`IsTimeOf`,
  "(IsTimeOf      0  f t  =  f t  /\  !t'.  (t'<t) ==> ~f t') /\
   (IsTimeOf (SUC n) f t  =  ?t'.  IsTimeOf n f t' /\
                                   Next t' t f             )"
  );;

let TimeOf = new_definition
  (`TimeOf`,
   "TimeOf f n  =  @t. IsTimeOf n f t"
  );;

let when = new_infix_definition
  (`when`,
   "when (s:num->*) (p:num->bool)  =  \n. s (TimeOf p n)"
  );;

let Inf = new_definition
  (`Inf`,
   "Inf f =  !t. ?t'.  (t<t') /\ (f t')"
  );;

%------------------------------------------------------------------------
| Define "LEAST P" to represent that P has a smallest element.
------------------------------------------------------------------------%
let LEAST = new_definition
```

```
  (`LEAST`,
   "LEAST P  =  ?x. P x  /\  (!y. y<x ==> ~P y)"
 );;

close_theory();;

%------------------------------------------------------------------------
|   wop = |- !P.  (?n. P n)  ==>  LEAST P
------------------------------------------------------------------------%
let wop = prove_thm
  (`wop`,
   "!P. (?n. P n)  ==>  LEAST P",
   REWRITE_TAC [WOP; LEAST]
  );;

%------------------------------------------------------------------------
|   Inf_EXISTS = |- !f.  Inf f  ==>  ?n.  f n
------------------------------------------------------------------------%
let Inf_EXISTS = prove_thm
  (`Inf_EXISTS`,
   "!f.  Inf f  ==>  ?n.  f n",
   PURE_REWRITE_TAC [Inf]
   THEN REPEAT STRIP_TAC
   THEN FIRST_ASSUM (STRIP_ASSUME_TAC  o (SPEC "t:num"))
   THEN EXISTS_TAC "t':num"
   THEN FIRST_ASSUM ACCEPT_TAC
  );;

%------------------------------------------------------------------------
|   Inf_LEAST = |- !f.  Inf f  ==>  LEAST f
------------------------------------------------------------------------%
let Inf_LEAST = prove_thm
  (`Inf_LEAST`,
   "!f.  Inf f  ==>  LEAST f",
   REPEAT STRIP_TAC
   THEN IMP_RES_TAC Inf_EXISTS
   THEN IMP_RES_TAC wop
  );;

%------------------------------------------------------------------------
|   Inf_Next = |- !f. Inf f ==> !t. f t ==> ?t'. Next t t' f
------------------------------------------------------------------------%
let Inf_Next = prove_thm
  (`Inf_Next`,
   "!f. Inf f ==> !t. f t ==> ?t'. Next t t' f",
   PURE_REWRITE_TAC [Inf; Next]
   THEN REPEAT (X_GEN_TAC "v:num" ORELSE STRIP_TAC)
   THEN RULE_ASSUM_TAC (\th. SPEC "v:num" th ? th)
   THEN IMP_RES_THEN (X_CHOOSE_THEN "n:num" STRIP_ASSUME_TAC) wop'
   THEN EXISTS_TAC "n:num"
   THEN ASM_REWRITE_TAC []
   THEN REPEAT STRIP_TAC
   THEN RES_THEN (STRIP_ASSUME_TAC o (REWRITE_RULE[DE_MORGAN_THM]))
   THEN RES_TAC
  )
where wop' =
 CONV_RULE (DEPTH_CONV BETA_CONV) (SPEC (( mk_abs
                                         o dest_exists
                                         o snd
                                         o dest_forall
                                         o rhs
                                         o concl
                                         o SPEC_ALL
                                         ) Inf)
                                        WOP
                                    );;

%-----------------------------------------------------------------------
|   Next_ADD1 = |- !f t.  f (t+1)  ==>  Next t (t+1) f
-----------------------------------------------------------------------%
let Next_ADD1 = prove_thm
  (`Next_ADD1`,
   "!f t.  f (t+1)  ==>  Next t (t+1) f",
   REWRITE_TAC [ Next
               ; SYM (SPEC_ALL ADD1)
               ; LESS_SUC_REFL
               ]
   THEN REPEAT STRIP_TAC
   THENL [ FIRST_ASSUM ACCEPT_TAC
         ; IMP_RES_THEN
             (STRIP_ASSUME_TAC o (CONV_RULE (ONCE_DEPTH_CONV SYM_CONV)))
             LESS_NOT_EQ
          THEN IMP_RES_TAC LESS_SUC_IMP
          THEN IMP_RES_TAC LESS_ANTISYM
         ]
  );;

%-----------------------------------------------------------------------
|   Next_INCREAST = |- !f t1 t2.   ~f(t1+1)       ==>
|                               Next (t1+1) t2 f  ==>  Next t1 t2 f
-----------------------------------------------------------------------%
let Next_INCREASE = prove_thm
  (`Next_INCREASE`,
   "!f t1 t2.   ~f(t1+1)       ==>
             Next (t1+1) t2 f  ==>  Next t1 t2 f",
   PURE_REWRITE_TAC [Next; SYM (SPEC_ALL ADD1)]
   THEN REPEAT STRIP_TAC
   THENL [ IMP_RES_TAC SUC_LESS
         ; FIRST_ASSUM ACCEPT_TAC
         ; MATCH_UNDISCH_TAC "~^(genvar":bool")"
           THEN IMP_RES_TAC (PURE_REWRITE_RULE [LESS_OR_EQ] LESS_SUC_EQ)
           THEN RES_TAC
           THEN ASM_REWRITE_TAC []
        ]
  );;

%-----------------------------------------------------------------------
|   Next_IDENTITY = |- !t1 t2 f.   Next t1 t2 f  ==>
|                           !t3.   Next t1 t3 f  ==>  (t2 = t3)
-----------------------------------------------------------------------%
let Next_IDENTITY = prove_thm
  (`Next_IDENTITY`,
   "!t1 t2 f.   Next t1 t2 f  ==>
          !t3.  Next t1 t3 f  ==>  (t2 = t3)",
   PURE_REWRITE_TAC [Next]
   THEN REPEAT STRIP_TAC
   THEN PURE_ONCE_REWRITE_TAC
          [(SYM o SPEC_ALL o hd o CONJUNCTS) NOT_CLAUSES]
   THEN DISCH_TAC
   THEN STRIP_ASSUME_TAC
          (SPECL ["t2:num"; "t3:num"]
                 (REWRITE_RULE [DE_MORGAN_THM] LESS_ANTISYM))
   THENL [ ALL_TAC
         ; RULE_ASSUM_TAC (CONV_RULE (ONCE_DEPTH_CONV SYM_CONV))
         ]
   THEN IMP_RES_TAC LESS_CASES_IMP
   THEN RES_TAC
  );;

%-----------------------------------------------------------------------
|   IsTimeOf_TRUE = |- !n f t.  IsTimeOf n f t ==> f t
-----------------------------------------------------------------------%
let IsTimeOf_TRUE = prove_thm
  (`IsTimeOf_TRUE`,
   "!n f t.  IsTimeOf n f t ==> f t",
   INDUCT_TAC
   THEN REWRITE_TAC [IsTimeOf; Next]
   THEN REPEAT STRIP_TAC
  );;

%-----------------------------------------------------------------------
|   IsTimeOf_EXISTS = |- !f. Inf f ==> !n. ?t. IsTimeOf n f t
-----------------------------------------------------------------------%
let IsTimeOf_EXISTS = prove_thm
  (`IsTimeOf_EXISTS`,
   "!f. Inf f ==> !n. ?t. IsTimeOf n f t",
   GEN_TAC
   THEN DISCH_TAC
   THEN INDUCT_TAC
   THENL [ IMP_RES_TAC Inf_EXISTS
           THEN IMP_RES_THEN
                  (ASSUME_TAC o (PURE_REWRITE_RULE [LEAST]))
                  Inf_LEAST
           THEN ASM_REWRITE_TAC [IsTimeOf]
         ; FIRST_ASSUM STRIP_ASSUME_TAC
           THEN IMP_RES_TAC IsTimeOf_TRUE
           THEN IMP_RES_TAC Inf_Next
           THEN FIRST_ASSUM STRIP_ASSUME_TAC
           THEN REWRITE_TAC [IsTimeOf]
           THEN EXISTS_TAC "t':num"
           THEN EXISTS_TAC "t:num"
           THEN ASM_REWRITE_TAC []
```

```
       ]
  );;

%----------------------------------------------------------------------
|   TimeOf_DEFINED = |- !f. Inf f ==> (!n. IsTimeOf n f (TimeOf f n))
----------------------------------------------------------------------%
let TimeOf_DEFINED = save_thm
  (`TimeOf_DEFINED`, ( GEN_ALL
                     o DISCH_ALL
                     o GEN_ALL
                     o (REWRITE_RULE [SYM(SPEC_ALL TimeOf)])
                     o CONV_RULE (DEPTH_CONV BETA_CONV)
                     o (REWRITE_RULE [EXISTS_DEF])
                     o SPEC_ALL
                     o UNDISCH_ALL
                     o SPEC_ALL
                     ) IsTimeOf_EXISTS
  );;

%----------------------------------------------------------------------
|   TimeOf_TRUE = |- !f. Inf f ==> (!n. f (TimeOf f n))
----------------------------------------------------------------------%
let TimeOf_TRUE = save_thm
  (`TimeOf_TRUE`, ( GEN_ALL
                  o DISCH_ALL
                  o GEN_ALL
                  o (MATCH_MP IsTimeOf_TRUE)
                  o SPEC_ALL
                  o UNDISCH_ALL
                  o SPEC_ALL
                  ) TimeOf_DEFINED
  );;

%----------------------------------------------------------------------
|   IsTimeOf_IDENTITY =
|   |- !n f t1 t2. IsTimeOf n f t1 /\ IsTimeOf n f t2 ==> (t1 = t2)
----------------------------------------------------------------------%
let IsTimeOf_IDENTITY = prove_thm
  (`IsTimeOf_IDENTITY`,
   "!n f t1 t2. IsTimeOf n f t1 /\ IsTimeOf n f t2 ==> (t1 = t2)",
   INDUCT_TAC
   THEN PURE_REWRITE_TAC [IsTimeOf; Next]
   THEN X_GEN_TAC "f:num->bool"
   THEN X_GEN_TAC "t1:num"
   THEN X_GEN_TAC "t2:num"
   THEN REPEAT STRIP_TAC
   THENL [ ALL_TAC
         ; RES_THEN
             (\th. EVERY_ASSUM
                     (STRIP_ASSUME_TAC o (\thm. SUBS [th] thm? thm)))
         ]
   THEN STRIP_ASSUME_TAC
          (SPECL ["t2:num"; "t1:num"]
                 (REWRITE_RULE [LESS_OR_EQ] LESS_CASES))
   THEN RES_TAC
  );;

%-----------------------------------------------------------------------
|   TimeOf_INCREASING =
|   |- !f. Inf f  ==>  !n. (TimeOf f n) < (TimeOf f (n+1))
-----------------------------------------------------------------------%
let TimeOf_INCREASING = prove_thm
  (`TimeOf_INCREASING`,
   "!f. Inf f ==> (!n. (TimeOf f n) < (TimeOf f(n+1)))",
   GEN_TAC
   THEN DISCH_TAC
   THEN X_GEN_TAC "n:num"
   THEN IMP_RES_TAC Inf_Next
   THEN IMP_RES_THEN (STRIP_ASSUME_TAC o (SPEC "n:num")) TimeOf_DEFINED
   THEN IMP_RES_THEN
         ( CHOOSE_THEN ( STRIP_ASSUME_TAC
                       o (\thl. CONJ (el 1 thl) (el 2 thl))
                       o CONJUNCTS
                       )
         o (REWRITE_RULE [IsTimeOf; Next])
         o (SPEC "SUC n")
         ) TimeOf_DEFINED
   THEN MATCH_UNDISCH_TAC "x<y"
   THEN IMP_RES_TAC IsTimeOf_TRUE
   THEN IMP_RES_THEN
         (\th. ONCE_REWRITE_TAC[ADD1; th]) IsTimeOf_IDENTITY
  );;

%-----------------------------------------------------------------------
|   TimeOf_INTERVAL =
|   |- !f. Inf f ==>
|      !n t. (TimeOf f n)<t  /\  t<(TimeOf f (n+1)) ==> ~f t
-----------------------------------------------------------------------%
let TimeOf_INTERVAL = prove_thm
  (`TimeOf_INTERVAL`,
   "!f. Inf f ==>
    !n t. (TimeOf f n)<t  /\  t<(TimeOf f (n+1)) ==> ~f t",
   GEN_TAC
   THEN DISCH_TAC
   THEN X_GEN_TAC "n:num"
   THEN X_GEN_TAC "t:num"
   THEN IMP_RES_TAC Inf_Next
   THEN IMP_RES_THEN (STRIP_ASSUME_TAC o (SPEC "n:num")) TimeOf_DEFINED
   THEN IMP_RES_THEN
         ( CHOOSE_THEN ( STRIP_ASSUME_TAC
                       o (\thl.  CONJ (el 1 thl) (SPEC "t:num" (el 4 thl)))
                       o CONJUNCTS
                       )
         o (REWRITE_RULE [IsTimeOf; Next])
         o (SPEC "SUC n")
         ) TimeOf_DEFINED
   THEN FIRST_ASSUM (UNDISCH_TAC o concl)
   THEN IMP_RES_TAC IsTimeOf_TRUE
   THEN IMP_RES_THEN  (\th. ONCE_REWRITE_TAC[ADD1; th])  IsTimeOf_IDENTITY
  );;

%----------------------------------------------------------------------
|   TimeOf_Next = |- !f. Inf f ==> !n. Next (TimeOf f n) (TimeOf f (n+1)) f
----------------------------------------------------------------------%
let TimeOf_Next = prove_thm
  (`TimeOf_Next`,
   "!f. Inf f ==> !n. Next (TimeOf f n) (TimeOf f (n+1)) f",
   PURE_REWRITE_TAC [Next]
   THEN REPEAT STRIP_TAC
   THENL [ IMP_RES_THEN (\th. REWRITE_TAC [th]) TimeOf_INCREASING
         ; IMP_RES_THEN (\th. REWRITE_TAC [th]) TimeOf_TRUE
         ; IMP_RES_TAC TimeOf_INTERVAL THEN RES_TAC
         ]
  );;

print_theory `when`;;
%<----------------------------------------------------------------------
The Theory when
Parents --  HOL     wop
Constants --
  Next ":num -> (num -> ((num -> bool) -> bool))"
  IsTimeOf ":num -> ((num -> bool) -> (num -> bool))"
  TimeOf ":(num -> bool) -> (num -> num)"
  Inf ":(num -> bool) -> bool"
Curried Infixes --
  when ":(num -> *) -> ((num -> bool) -> (num -> *))"
Definitions --
  Next
    |- !t1 t2 f.
        Next t1 t2 f =
        t1 < t2 /\ f t2 /\ (!t. t1 < t /\ t < t2 ==> ~f t)
  IsTimeOf_DEF
    |- IsTimeOf =
       PRIM_REC
       (\f t. f t /\ (!t'. t' < t ==> ~f t'))
       (\g00004 n f t. ?t'. g00004 f t' /\ Next t' t f)
  TimeOf  |- !f n. TimeOf f n = (@t. IsTimeOf n f t)
  when    |- !s p. s when p = (\n. s(TimeOf p n))
  Inf     |- !f. Inf f = (!t. ?t'. t < t' /\ f t')
Theorems --
  IsTimeOf
    |- (!  f t. IsTimeOf 0 f t = f t /\ (!t'. t' < t ==> ~f t')) /\
       (!n f t. IsTimeOf(SUC n)f t = (?t'. IsTimeOf n f t' /\ Next t' t f))
  Inf_EXISTS  |- !f. Inf f ==> (?n. f n)
  Inf_LEAST   |- !f. Inf f ==> LEAST f
  Inf_Next    |- !f. Inf f ==> (!t. f t ==> (?t'. Next t t' f))
  Next_ADD1   |- !f t. f(t + 1) ==> Next t(t + 1)f
  Next_INCREASE
    |- !f t1 t2. ~f(t1 + 1) ==> Next(t1 + 1)t2 f ==> Next t1 t2 f
  Next_IDENTITY
    |- !t1 t2 f. Next t1 t2 f ==> (!t3. Next t1 t3 f ==> (t2 = t3))
  IsTimeOf_TRUE    |- !n f t. IsTimeOf n f t ==> f t
  IsTimeOf_EXISTS  |- !f. Inf f ==> (!n. ?t. IsTimeOf n f t)
  TimeOf_DEFINED   |- !f. Inf f ==> (!n. IsTimeOf n f(TimeOf f n))
  TimeOf_TRUE      |- !f. Inf f ==> (!n. f(TimeOf f n))
  IsTimeOf_IDENTITY
    |- !n f t1 t2. IsTimeOf n f t1 /\ IsTimeOf n f t2 ==> (t1 = t2)
  TimeOf_INCREASING
    |- !f. Inf f ==> (!n. (TimeOf f n) < (TimeOf f(n + 1)))
  TimeOf_INTERVAL
    |- !f. Inf f ==>
        (!n t. (TimeOf f n) < t /\ t < (TimeOf f(n + 1)) ==> ~f t)
  TimeOf_Next  |- !f. Inf f ==> (!n. Next(TimeOf f n)(TimeOf f(n + 1))f)
*************************************---------------------------------->%
```