Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A SEQUENT FORMULATION OF THE PROPOSITIONAL LOGIC OF PREDICATES IN HOL

C.-T. Chou

June 1992
CSD-920033

A Sequent Formulation of the Propositional Logic of Predicates in HOL
(Preliminary Version)

*Ching-Tsun Chou**
chou@cs.ucla.edu

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, U.S.A.

June 14, 1992

1 Propositional logic of predicates

By a *predicate* we mean an HOL [2, 4] term of type `* -> bool`, where `*` is called the *domain* of the predicate and can be any HOL type. Propositional operators on Boolean terms can be *lifted* to form propositional operators on predicates over the same domain. In the following `P` and `Q` are predicates of type `* -> bool` and `x` ranges over `*`. Also notice the notational convention of 'doubling' the symbols of the original operators to form those of the lifted operators.

```
      (TT)(x) = T
      (FF)(x) = F
    (~~ P)(x) = ~ P(x)
(P //\\ Q)(x) = P(x) /\ Q(x)
(P \\// Q)(x) = P(x) \/ Q(x)
(P ==>> Q)(x) = P(x) ==> Q(x)
(P  ==  Q)(x) = (P(x) = Q(x))
```

Not only can propositional operators be lifted, but the notion of *sequents* can also be lifted to apply to predicates:

```
(P |= Q) = (!x. P(x) ==> Q(x))
```

where `P` and `Q` are called, respectively, the *assumption* and the *conclusion* of the lifted sequent. Notice that `==>>` and `|=` have different types:

```
==>> : (* -> bool) -> (* -> bool) -> (* -> bool)
  |= : (* -> bool) -> (* -> bool) -> bool
```

*Supported by IBM Graduate Fellowship.

Hence it does not make sense to say whether `P ==>> Q` is true or false, but `P |= Q` *is* either true or false.

The logic of sequents of predicates (with respect to propositional operators on predicates) behaves exactly like the logic of sequents of Boolean terms (with respect to propositional operators on Boolean terms). For instance, the lifted version of *Modus Ponens* is still valid:

```
    TT |= (P ==>> Q)     TT |= P
-----------------------------------
               TT |= Q
```

so is the lifted version of 'Deduction Theorem':

```
    (P //\\ Q) |= R
----------------------
    P |= (Q ==>> R)
```

To be sure, there *is* a difference between a lifted sequent and an HOL sequent: the former has a single assumption while the latter has a list of assumptions:

`P |= Q` *versus* `p1, p2, p3 |- q`

But since the latter is equivalent to `p1 /\ p2 /\ p3 |- q`, we will view the assumption of a lifted sequent as a list of its individual conjuncts and `TT` as an empty assumption list:

```
P1 //\\ P2 //\\ P3 |= Q     is viewed as     P1, P2, P3 |= Q
                TT |= Q     is viewed as                |= Q
```

(Of course, only the two on the left have a formal meaning; the two on the right don't.) The significance of this view will become clear in Section 4. But before that, let us justify our interest in the propositional logic of predicates by examining some of its applications.

2 Semantic embeddings of propositional logics

A common method of semantically embedding in HOL various propositional logics, such as programming logics [5] and modal logics [3], is to use *predicates* in the HOL logic to represent *propositions* in the embedded logic. Since, as explained in Section 1, the propositional logic of predicates behaves just like the ordinary propositional logic, propositional reasoning in the embedded logic can be adequately modeled by the lifted propositional reasoning in HOL. Furthermore, the higher-orderness of predicates make it possible to semantically embed various non-propositional operators, such as modal operators, by using the semantics of the domains of predicates. These ideas are illustrated by the semantic embedding in HOL of Lamport's *Temporal Logic of Actions*[1] (TLA) [6], which is an on-going project of the author's [1].

2.1 TLA in HOL

In TLA, there are three types of objects on which one wishes to have predicates:

[1] Only part of the features of TLA is mentioned in this paper; see Lamport's report for a complete description [6].

```
    states   : *state
transitions  : *state # *state
  behaviors  : num -> *state
```

In other words, a state can be anything (usually a tuple representing the values of program variables), a transition is a pair of states (representing a step of program execution), and a behavior is an infinite sequence of states (representing an infinite history of program execution). Predicates on states are called *state predicates* or simply *predicates*, predicates on transitions *actions*, and predicates on behaviors *temporal properties*.

In addition to the lifted propositional operators, there are two kinds of special operators on various predicates in TLA: (*type*) *coercion operators* and *temporal operators*. Coercion operators are defined by specializing the *inverse image operator*, as follows. Let `f : * -> **` be any mapping. For each predicate `P : ** -> bool`, the inverse image of `P` under `f`, `(inv f)(P) : * -> bool`, is:

```
(inv f)(P)(x) = P(f(x))
```

(In mathematical literature `(inv f)(P)` is usually written as $\mathtt{f}^{-1}(\mathtt{P})$.) In TLA, there are (among others) four mappings between domains of predicates:

```
map_t_s  (s : *state, s' : *state) = s
map_t_s' (s : *state, s' : *state) = s'
map_b_s  (b : num -> *state)       = b(0)
map_b_t  (b : num -> *state)       = (b(0), b(SUC(0)))
```

Their corresponding coercion operators are:

```
t_s  = inv map_t_s  : (*state           -> bool) -> ((*state # *state) -> bool)
t_s' = inv map_t_s' : (*state           -> bool) -> ((*state # *state) -> bool)
b_s  = inv map_b_s  : (*state           -> bool) -> ((num -> *state)   -> bool)
b_t  = inv map_b_t  : ((*state # *state) -> bool) -> ((num -> *state)   -> bool)
```

Coercion operators allow one to view predicates on one domain as those on the other. For example, a little rewriting shows that:

```
(b_s P)(b) = P(b(0))
```

So `b_s` coerces a state predicate `P` into a temporal property by evaluating `P` at the first state of a behavior. Similarly, `b_t` coerces an action `A` into a temporal property by evaluating `A` at the first two states of a behavior, and `t_s` (`t_s'`) coerces state predicate `P` into an action by evaluating `P` at the first (second) state of a transition.

The advantage of defining coercion operators by specializing `inv` is that one can prove properties of coercion operators by specializing properties of `inv`, which have to be proved only once. For instance, one can prove that `(inv f)` distributes over the lifted implication[2]:

```
!f. !P Q. (inv f)(P ==>> Q) = (inv f)(P) ==>> (inv f)(Q)
```

As a special case, `b_s` distributes over the lifted implication as well:

```
!P Q. b_s(P ==>> Q) = b_s(P) ==>> b_s(Q)
```

[2]In fact, `(inv f)` distributes over all lifted propositional operators.

In TLA, there are two modal operators [] (read: *box*) and <> (read: *diamond*) on temporal properties which express the notions of, respectively, *always* and *eventually*:

```
([] G)(b : num -> *state) = ! n. G(suffix n b)
(<> G)(b : num -> *state) = ? n. G(suffix n b)
```

where

```
suffix n b = \ m. b(m + n)
```

denotes the n-th suffix of behavior b. In other words, a temporal property G is always (eventually) true of a behavior b if and only if it is true of the n-th suffix of b for all (some) n. Other temporal operators can be defined in terms of [] and <>. For example,

```
(G ~> H) = [](G ==>> <> H)
```

(read: G *leads to* H) expresses the notion that whenever G is true, H will eventually be true.

In TLA, not only the temporal properties of programs, but also programs themselves, are expressed as predicates on behaviors. A program Prog which starts in a state satisfying the initial condition Init, henceforth takes only steps allowed by action Next, and meets the fairness condition Fair, is expressed by[3]:

```
Prog = b_s(Init) //\\ [](b_t(Next)) //\\ Fair
```

The statement that program Prog satisfies the temporal specification Spec is expressed by:

```
Prog |= Spec
```

It turns out that much of the reasoning involved in proving programs correct in TLA is propositional, so it is important to have powerful tools in HOL to do propositional reasoning on predicates.

3 Predicates as sets

By identifying sets with their characteristic functions, predicates can viewed as sets. In this view, a predicate P : * -> bool *is* the set of elements of type * having property P:

```
P = { x : * | P(x) }
```

An extensive HOL88 library for predicates as sets, called the **pred_sets** library, has been written by Melham [7] (based on earlier work by Kalker). Many operations and relations on sets have logical interpretations if sets are viewed as predicates. With the notation of pred_sets, all of the following are theorems:

```
        UNIV = TT
       EMPTY = FF
   P INTER Q = (P //\\ Q)
   P UNION Q = (P \\// Q)
   P DIFF  Q = (P //\\ ~~ Q)
 P  SUBSET Q = (P |= Q)
 P PSUBSET Q = (P |= Q) /\ ~(P = Q)
DISJOINT P Q = (P //\\ Q) |= FF
```

This means that the proof technique presented below can be used to reason about sets as well, especially the Boolean algebra of sets.

[3]This is actually over-simplified, since TLA formulas takes the so-called *stuttering* into account; see [6].

4 Lifting tactics for propositional reasoning

At least in principle, one can always prove the validity of statements in an embedded logic by expanding the definitions of operators and reasoning directly in HOL. But doing so defeats the very purpose of embedding: if all the reasoning is to be done directly in HOL, then why bother with the embedding in the first place? It is important to observe that an embedded logic provides its user with not only more concise and elegant notations, but potentially also larger inference steps, than available in plain HOL. Hence it seems reasonable to accept as a general principle that the user of an embedded logic should perform as much reasoning as possible in the embedded logic. This is not to say that the actual inference steps executed by the HOL system should contain few expansions of embedded operators. Indeed, the technique described below involves a lot of translating back and forth between the embedded operators and their HOL definitions. But the user should be shielded from the implementation details and be able to imagine that she or he is doing proofs in the embedded logic.

In HOL, there are extensive facilities for propositional reasoning on Boolean terms. Since we have lifted propositional operators and sequents to apply to predicates, the natural next step is to lift *tactics* to work on sequents of predicates. More specifically, we would like to have a tactical

```
pseq_TCL : tactic -> tactic
```

(the prefix 'pseq_' stands for 'predicate sequent') such that, when applied to (say) the tactic DISCH_TAC:

```
    ?- p ==> q
==================  DISCH_TAC
       p ?- q
```

it produces a new tactic which performs:

```
    ?= P ==>> Q
==================  pseq_TCL DISCH_TAC
       P ?= Q
```

Namely, (pseq_TCL DISCH_TAC) is 'identical' to DISCH_TAC except that |- is replaced by |= and ==> by ==>>. This idea is illustrated in the following HOL session[4]:

```
#g " ( TT : * -> bool ) |= ( (P //\\ Q //\\ R) ==>> (R //\\ P) ) " ;;                1
"TT |= ((P //\\ (Q //\\ R)) ==>> (R //\\ P))"

#e( pseq_TCL DISCH_TAC );;
OK..
"(P //\\ (Q //\\ R)) |= (R //\\ P)"
```

Recall that the assumption can be viewed as a list of its conjuncts (with TT being the empty list). So the last goal can be solved by rewriting with the assumption list:

```
#e( pseq_TCL (ASM_REWRITE_TAC [ ]) );;                                              2
OK..
goal proved
|- (P //\\ (Q //\\ R)) |= (R //\\ P)
|- TT |= ((P //\\ (Q //\\ R)) ==>> (R //\\ P))
```

[4]HOL sessions are displayed in rectangular windows, each of which is labelled by a sequence number shown at the upper-right corner. Side-effects produced in lower-numbered windows persist into higher-numbered ones until the number is reset to 1, which marks the beginning of a new session.

Similarly, we can define a theorem tactical

```
pseq_TTCL : thm_tactic -> thm_tactic
```

which lifts a theorem tactic on ordinary HOL sequents to a theorem tactic on predicate sequents, where the theorem argument to the resulting theorem tactic must also be a predicate sequent. The implementations of `pseq_TCL` and `pseq_TTCL` are described in the Appendix. Here we conclude this section with an (only slightly) bigger example.

Consider the antisymmetry of the lifted implication:

1

```
#g " !P Q : * -> bool.
#    (TT |= ( P ==>> Q )) /\ (TT |= ( Q ==>> P )) ==> (TT |= ( P == Q )) " ;;
"!P Q. TT |= (P ==>> Q) /\ TT |= (Q ==>> P) ==> TT |= (P == Q)"

#e( REPEAT STRIP_TAC );;
OK..
"TT |= (P == Q)"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]
```

Since the goal is a lifted equivalence, the lifted `EQ_TAC` is the appropriate tactic:

2

```
#e( pseq_TCL EQ_TAC );;
OK..
2 subgoals
"TT |= (Q ==>> P)"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]

"TT |= (P ==>> Q)"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]
```

Now push the antecedent `P` onto the assumption list of the lifted sequent:

3

```
#e( pseq_TCL DISCH_TAC );;
OK..
"P |= Q"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]
```

Now `ASSUME` the first of the two assumptions of the goal:

4

```
#let asm1 = ASSUME " ( TT : * -> bool ) |= ( P ==>> Q ) " ;;
asm1 = TT |= (P ==>> Q) |- TT |= (P ==>> Q)
```

Now a little lifted resolution solves the first subgoal:

```
#e( pseq_TTCL IMP_RES_TAC asm1 );;                                    5
OK..
goal proved
. |- P |= Q
.. |- TT |= (P ==>> Q)

Previous subproof:
"TT |= (Q ==>> P)"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]
```

The second subgoal can be solved analogously:

```
#e( pseq_TCL DISCH_TAC );;                                            6
OK..
"Q |= P"
    [ "TT |= (P ==>> Q)" ]
    [ "TT |= (Q ==>> P)" ]

#let asm2 = ASSUME " ( TT : * -> bool ) |= ( Q ==>> P ) " ;;
asm2 = TT |= (Q ==>> P) |- TT |= (Q ==>> P)

#e( pseq_TTCL IMP_RES_TAC asm2 );;
OK..
goal proved
. |- Q |= P
.. |- TT |= (Q ==>> P)
.. |- TT |= (P == Q)
|- !P Q. TT |= (P ==>> Q) /\ TT |= (Q ==>> P) ==> TT |= (P == Q)
```

Finally, the above session can be condensed into one single tactic:

```
#g " !P Q : * -> bool.                                                7
#    (TT |= ( P ==>> Q )) /\ (TT |= ( Q ==>> P )) ==> (TT |= ( P == Q )) " ;;
"!P Q. TT |= (P ==>> Q) /\ TT |= (Q ==>> P) ==> TT |= (P == Q)"

#e( REPEAT GEN_TAC THEN
#   DISCH_THEN \asm1.
#     let (asm1, asm2) = CONJ_PAIR asm1 in
#     pseq_TCL (EQ_TAC THEN DISCH_TAC) THENL
#     map (pseq_TTCL IMP_RES_TAC) [asm1; asm2] );;
OK..
goal proved
|- !P Q. TT |= (P ==>> Q) /\ TT |= (Q ==>> P) ==> TT |= (P == Q)
```

Appendix: Implementations

Assume the following rewrite rules are available:

```
    tt_pseq_THM = |- !P. (TT |= P) = (!x. P(x))
rev_tt_pseq_THM = |- !P. (!x. P(x)) = (TT |= P)

       pseq_THM = |- !P Q. (P |= Q) = (!x. P(x) ==> Q(x))
   rev_pseq_THM = |- !P Q. (!x. P(x) ==> Q(x)) = (P |= Q)

      logic_THM = |- (!x.           (TT)(x) = T           ) /\
                     (!x.           (FF)(x) = F           ) /\
                     (!P x.       (~~ P)(x) = ~ P(x)      ) /\
                     (!P Q x. (P //\\ Q)(x) = P(x) /\ Q(x) ) /\
                     (!P Q x. (P \\// Q)(x) = P(x) \/ Q(x) ) /\
                     (!P Q x. (P ==>> Q)(x) = P(x) ==> Q(x)) /\
                     (!P Q x. (P  ==  Q)(x) = (P(x) = Q(x)))
  rev_logic_THM = |- !x P Q.
                     (           T = (TT)(x)       ) /\
                     (           F = (FF)(x)       ) /\
                     (        ~P(x) = (~~ P)(x)    ) /\
                     (P(x) /\  Q(x) = (P //\\ Q)(x)) /\
                     (P(x) \/  Q(x) = (P \\// Q)(x)) /\
                     (P(x) ==> Q(x) = (P ==>> Q)(x)) /\
                     ((P(x) = Q(x)) = (P == Q)(x)  )
```

`pseq_goal_var(_, "P |= (Q : ** -> bool)")` returns a fresh variable of type `**`.

```
let pseq_goal_var : goal -> term =
  genvar o (\ t . (fst o hd) t ? ": *") o snd
         o match "$|= : (* -> bool) -> (* -> bool) -> bool"
         o rator o rator o snd
;;
```

`unfold_pseq_RULE "x" (P //\\ Q |= R)` returns the Boolean sequent `(P(x), Q(x) |- R(x))`; `unfold_pseq_RULE "x" (TT |= R)` returns `(|- R(x))`.

```
let unfold_pseq_RULE (x : term) (th : thm) =
  let (vl, th') = SPEC_VAR_ALL th
  in
  ( if (can (match "TT |= (P : * -> bool)") (concl th)) then
        GENL vl o
        PURE_REWRITE_RULE [logic_THM] o
        SPEC x o
        PURE_ONCE_REWRITE_RULE [tt_pseq_THM]
      else
        GENL vl o
        S MP (LIST_CONJ o map ASSUME o conjuncts o fst o dest_imp o concl) o
        PURE_REWRITE_RULE [logic_THM] o
        SPEC x o
        PURE_ONCE_REWRITE_RULE [pseq_THM]
  ) th'
;;
```

unfold_pseq_TAC "x" (P //\\ Q ?= R) returns the Boolean sequent (P(x), Q(x) ?- R(x)); unfold_pseq_TAC "x" (TT ?= R) returns (?- R(x)).

```
let unfold_pseq_TAC (x : term) (asl, g) =
  ( if (can (match "TT |= (P : * -> bool)") g) then
      PURE_ONCE_REWRITE_TAC [tt_pseq_THM] THEN
      X_GEN_TAC x THEN
      PURE_REWRITE_TAC [logic_THM]
    else
      PURE_ONCE_REWRITE_TAC [pseq_THM] THEN
      X_GEN_TAC x THEN
      PURE_REWRITE_TAC [logic_THM] THEN
      DISCH_THEN (MAP_EVERY ASSUME_TAC o rev o CONJUNCTS)
  ) (asl, g)
;;
```

fold_pseq_TAC "x" (P(x), Q(x) ?- R(x)) returns the predicate sequent (P //\\ Q ?= R); fold_pseq_TAC "x" (?- R(x)) returns (TT ?= R). Notice the use of FREEZE_THEN to prevent any instantiation of x.

```
let fold_pseq_TAC (x : term) (asl, g) =
  let rev_logic_THM_INST = ISPEC x rev_logic_THM
  in
  ( if (null asl) then
      FREEZE_THEN (\ th . PURE_REWRITE_TAC [th])
                  rev_logic_THM_INST THEN
      SPEC_TAC (x, x) THEN
      PURE_ONCE_REWRITE_TAC [rev_tt_pseq_THM]
    else
      POP_ASSUM_LIST (MP_TAC o LIST_CONJ) THEN
      FREEZE_THEN (\ th . GEN_REWRITE_TAC (SUB_CONV o TOP_DEPTH_CONV) [ ] [th])
                  rev_logic_THM_INST THEN
      SPEC_TAC (x, x) THEN
      PURE_ONCE_REWRITE_TAC [rev_pseq_THM]
  ) (asl, g)
;;
```

pseq_TCL (pseq_TTCL) removes and saves all assumptions of the goal, unfolds the definitions of predicate sequent and lifted propositional operators, calls the argument tactic (theorem tactic), folds the definitions, and finally puts back the original assumptions.

```
let pseq_TCL (tac : tactic) =
  POP_ASSUM_LIST ( \ asl .
    ( \ g . let x = pseq_goal_var g
            in
            ( unfold_pseq_TAC x THEN tac THEN fold_pseq_TAC x ) g
    ) THEN MAP_EVERY ASSUME_TAC (rev asl) )
;;

let pseq_TTCL (ttac : thm_tactic) (th : thm) =
  POP_ASSUM_LIST ( \ asl .
    ( \ g . let x = pseq_goal_var g
            in
            let th' = unfold_pseq_RULE x th
            in
            ( unfold_pseq_TAC x THEN ttac th' THEN fold_pseq_TAC x ) g
    ) THEN MAP_EVERY ASSUME_TAC (rev asl) )
;;
```

Acknowledgements

The author is grateful to members of the HOL Seminar at UCLA for interesting discussions: Vernon Austel, Peter Homeier, Dinh Le, Professor David Martin, Brad Pierce, David Quam, Ray Toal, and Yih-Kuen Tsay.

References

[1] Ching-Tsun Chou, "A Semantic Embedding of Lamport's Temporal Logic of Actions in HOL", work in progress, (1992).

[2] DSTO and SRI International, *The HOL System: DESCRIPTION*, (1991).

[3] R. Goldblatt, "Logics of Time and Computation", CSLI Lecture Notes 7, (1987).

[4] M. J. C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", in *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73–128.

[5] M. J. C. Gordon, "Mechanizing Programming Logics in Higher-Order Logic", in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam, (Springer-Verlag, 1989), pp. 387–439.

[6] L. Lamport, "The Temporal Logic of Actions", DEC Systems Research Center technical report 79, (1991).

[7] T. F. Melham, *The HOL pred_sets Library*, University of Cambridge Computer Laboratory, (1992).