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

A SEQUENT FORMULATION OF A LOGIC OF PREDICATES IN HOL (Revised of TR #920033)

C.-T. Chou

October 1992
CSD-920045

A Sequent Formulation of a Logic of Predicates in HOL*

Ching-Tsun Chou†
chou@cs.ucla.edu

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

October 6, 1992

Abstract

By a *predicate* we mean a term in the HOL logic of type `* -> bool`, where `*` can be any type. Boolean connectives, quantifiers and sequents in the HOL logic can all be *lifted* to operate on predicates. The lifted logical operators and sequents form a *Logic of Predicates* (LP) whose behavior resembles closely that of the unlifted HOL logic. Of the applications of LP we describe two in some detail: (1) a semantic embedding of Lamport's *Temporal Logic of Actions*, and (2) an alternative formulation of set theory. The main contribution of this paper is a systematic approach for *lifting* tactics that works in the unlifted HOL logic to ones that works in LP, so that one can enjoy the rich proof infrastructure of HOL when reasoning in LP. The power of this approach is illustrated by examples from modal and temporal logics. The implementation technique is briefly described.

Keyword Codes: F.4.1; I.2.3
Keywords: Mathematical Logic; Deduction and Theorem Proving.

1 A Logic of Predicates

By a *predicate* we mean a term in the HOL [3] logic of type `* -> bool`, where `*` is called the *domain* of the predicate and can be any type. Boolean connectives and quantifiers in the HOL logic can all be *lifted* to operate on predicates with the following definitions:

```
      (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))
    (!! R)(x)  =  ! i . (R i)(x)
    (?? R)(x)  =  ? i . (R i)(x)
```

*To appear in the Proceedings of the 1992 HOL Workshop.
†Supported by IBM Graduate Fellowship.

where `P` and `Q` are predicates of type `* -> bool` and `R : ** -> (* -> bool)` is an indexed family of predicates. Notice our notational convention of 'doubling' the symbols of the original operators to form those of the lifted operators.

The last two definitions above need a little more explanation. The lifted quantifiers `!!` and `??` are actually implemented as *binders* so that for any predicate `R[i] : * -> bool` which may contain `i : **` among its free variables, the following equations hold:

```
(!! i . R[i])(x)  =  ! i . R[i](x)
(?? i . R[i])(x)  =  ? i . R[i](x)
```

These equations have to be derived from the definitions of `!!` and `??` using β-conversion for each `R[i]`.

Just as logical operators can be lifted, so can sequents. The *lifted sequent* with *assumptions* `P1`, ..., `Pn` and *conclusion* `Q` is defined as:

```
[ P1 ; ... ; Pn ] |= Q  =  ! x . P1(x) /\ ... /\ Pn(x) ==> Q(x)
```

Notice that, while it does not make sense to say whether a predicate is true or false, a lifted sequent *is* either true or false.

The lifted logical operators and sequents form a *Logic of Predicates* (LP) whose behavior resembles closely that of the unlifted HOL logic, in the sense that theorems, inference rules and tactics in the unlifted HOL logic all have lifted counterparts in LP. For example, *Modus Ponens* can be lifted:

```
  [ ] |= (P ==>> Q)    [ ] |= P
 ---------------------------------
            [ ] |= Q
```

so can `DISCH_TAC`:

```
  [ ] ?= (P ==>> Q)
 =====================
     [ P ] ?= Q
```

where `?=` indicates that the lifted sequents containing it have not yet been proved. Indeed, the main contribution of this paper is a systematic approach for *lifting* tactics that works in the unlifted HOL logic to ones that works in LP, so that one can enjoy the rich proof infrastructure of HOL when reasoning in LP.

It should be noted that the success of our approach depends crucially upon our adopting *sequents* instead of *formulas* as the basis of our logic of predicates. Had we chosen to reason about the validities of individual predicates, instead of sequents of predicates, by defining:

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

we would have obtained a lifted logic obeying a Hilbert style calculus of formulas (*i.e.*, predicates), which is incompatible with the natural deduction style calculus of sequents upon which the unlifted HOL logic is based. Such a mismatch would render impossible the relatively straightforward lifting of HOL tactics described in this paper, thus incurring a great deal of unnecessary work in the process of mechanizing the lifted logic.

2 Applications of LP

Two applications of LP are discussed in this section: §2.1 contains a typical example of the semantic embeddings of other logics in HOL, and §2.2 briefly describes the 'predicates as sets' formulation of set theory using LP.

2.1 A Temporal Logic of Actions in HOL

A common method of semantically embedding various logics, such as programming logics [4] and modal logics [2], in HOL is to use *predicates* in the HOL logic to represent *propositions* in the embedded logic. The logical operators of the embedded logic are naturally represented by the lifted logical operators of LP. The non-logical operators, such as modal operators, are embedded by referring to the semantics of the domains of predicates. These ideas are nicely illustrated by the semantic embedding of Lamport's *Temporal Logic of Actions* (TLA) [5] described below, which is an on-going project of the author's.

In TLA there are three domains on which predicates are needed:

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

That is, a state can be anything (usually a tuple of 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 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 logical operators, there are two kinds of non-logical operators in TLA: (*type*) *coercion operators* and *temporal operators*. Coercion operators are all defined by specializing the *inverse image operator*. Let `f : *1 -> *2` be any mapping. For any predicate `P : *2 -> bool`, the inverse image of `P` under `f`, `(inv f P) : *1 -> bool`, is defined by:

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

(The more familiar notation for `(inv f P)` is $\mathtt{f}^{-1}(\mathtt{P})$.) For instance, let `map_b_s` be the projection that maps each behavior to the first state in that behavior:

```
map_b_s (b : num -> *state)  =  b(0) : *state
```

Then, by specializing the `f` in `(inv f)` to `map_b_s`:

```
b_s  =  inv map_b_s : (*state -> bool) -> ((num -> *state) -> bool)
```

we obtain a coercion operator that 'coerces' a state predicate into a temporal property by evaluating the predicate at the first state of a behavior:

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

There are several other coercion operators in TLA that allow one to view a predicate as an action, an action as a temporal property, and so on.

The advantage of defining all coercion operators by specializing `(inv f)` is that one can prove properties of coercion operators simply 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[1]:

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

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

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

[1] As a matter of fact, `(inv f)` distributes over *all* lifted logical operators.

and so do all other coercion operators. All these facts can be proved by specializing the above distributivity theorem for `(inv f)`.

In TLA there are two basic modal operators `[]` (read: *box*) and `<>` (read: *diamond*) on temporal properties which formalize the notions of, respectively, '*always*' and '*eventually*' (where 'eventually' includes 'now'):

```
([] 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 in TLA are 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 programs themselves are also 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 formalized as:[2]

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

The statement that program `Prog` satisfies temporal property `Spec` is expressed by:

```
[ ] |= Prog ==>> Spec
```

(Do not confuse `[]`, the modal operator, with `[ ]`, the empty list!) Such a statement is to be proved by a mixture of HOL and TLA reasoning. On the one hand, reasoning about individual predicates and actions, such as proving a particular action preserves a particular invariant, is application-specific and handled directly in HOL. On the other hand, temporal reasoning follows common patterns and is handled by TLA inference rules, which are actually proved as HOL theorems. For instance, the transitivity of `~~>`:

```
! P Q R : (num -> *state) -> bool .
  [ ] |= ((P ~~> Q) //\\ (Q ~~> R)) ==>> (P ~~> R)
```

can be proved as an HOL theorem and instantiated with particular temporal properties `P`, `Q` and `R` when needed.

Comparison with Another Embedding of TLA

In [7] and [8], von Wright and Långbacka describe another semantic embedding of TLA in HOL which is very similar to ours. As far as the main thrust of this paper is concerned, there are two major differences between their embedding and ours. Firstly, von Wright and Långbacka use curried functions to represent actions. For example, assuming the state consists only of two numeric variables `x` and `y`, the action `Inc_x` that increments `x` by 1 but keeps `y` unchanged would be formalized in their system as:

[2] This is an oversimplification, since TLA formulas takes the so-called *stuttering* into account; see [5].

```
Inc_x (x,y) (x',y')  =  (x' = x + 1) /\ (y' = y)
```

while in our system it would be formalized as:

```
Inc_x ((x,y), (x',y'))  =  (x' = x + 1) /\ (y' = y)
```

As a consequence, they do not have a uniform treatment of logical operators at predicate, action and temporal levels, nor can they define the coercion operators uniformly by specializing the inverse image operator. Thus their work suffers from an unnecessary proliferation of similar but slightly different definitions.

Secondly, von Wright and Långbacka's embedding is based on a Hilbert style calculus of the validities of single predicates, so their system does not enjoy the easy lifting of HOL reasoning to TLA reasoning which our system does (see the last paragraph of Section 1).

2.2 Predicates as Sets

By identifying sets with their characteristic functions, predicates can be viewed as sets. Thus viewed, a predicate `P : * -> bool` *is* the set of elements of type `*` which satisfies property `P`:

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

An extensive HOL library for predicates as sets, called the **pred_sets** library, has been written by Melham [6] (based on earlier work of Kalker). Many operations and relations on sets have logical interpretations if sets are identified with predicates. Using 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
DISJOINT P Q  =  [ ] |= (P //\\ Q) == FF
```

Furthermore, the lifted quantifiers `!!` and `??` provide, respectively, union and intersection over an indexed family of sets, which are not available in **pred_sets**. The upshot is that the proof technique described below can be used to reason about sets as well.

3 Lifting Tactics and Theorem Tactics

In principle one can always prove statements in an embedded logic, such as LP or the LP-based embedding of TLA, by expanding the embedded operators with their definitions 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? An embedded logic provides its user with not only more concise and elegant notations, but also (conceptually) 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. To the contrary, the technique described below involves a lot of translating back and forth between LP and plain HOL. The point is that the user should be shielded from the implementation details and be able to imagine that she or he is doing proofs directly in the embedded logic.

In the rest of this section we shall describe, by means of a series of examples drawn from propositional, modal and temporal logics, several *tactic transformers* which can lift tactics in plain HOL to tactics in LP, thus supporting the illusion of 'doing proofs directly in LP'. Due to limited space, the implementation technique can only be outlined.

In the HOL sessions shown below, in order not to have to supply explicit type information too often, we use anti-quotations with the following ML binding:

```
#let VALID = " ($|= [ ]) : (* -> bool) -> bool " ;;                    1
VALID = "$|= [ ]" : term
```

Also, output from HOL is edited for ease of reading.

3.1 Tautology Checking

The tactic `pred_TAUT_TAC` is the lifted tautology checking tactic for LP, which is based on Boulton's tautology checker [1] for HOL.

```
#g" ^VALID( ((X ==>> Y) ==>> X) ==>> X ) ";;                           2
"[ ] |= ( ((X ==>> Y) ==>> X) ==>> X )"

#e( pred_TAUT_TAC );;
OK..
goal proved
|- [ ] |= ( ((X ==>> Y) ==>> X) ==>> X )

#g" ^VALID( !! k : ** . (Z k) \\// ~~(Z k) ) ";;
"[ ] |= ( !! k. (Z k) \\// ~~(Z k) )"

#e( pred_TAUT_TAC );;
OK..
goal proved
|- [ ] |= ( !! k. (Z k) \\// ~~(Z k) )
```

Incidentally, the examples above are known as the *Peirce's Law* and the (indexed) *Law of Excluded Middle*, both of which are intuitionistically *in*valid.

3.2 Lifting Tactics

Consider the following proof in the unlifted HOL logic:

```
#g" p ==> q ==> (p /\ q) ";;                                    3
"p ==> q ==> p /\ q"

#e( REPEAT DISCH_TAC );;
OK..
"p /\ q"
    [ "p" ]
    [ "q" ]

#e( ASM_REWRITE_TAC [ ] );;
OK..
goal proved
.. |- p /\ q
|- p ==> q ==> p /\ q
```

Now consider the same proof *lifted*:

```
#g" ^VALID( P ==>> Q ==>> (P //\\ Q) ) ";;                     4
"[ ] |= ( P ==>> Q ==>> (P //\\ Q) )"

#e( pred_TCL (REPEAT DISCH_TAC) );;
OK..
"[ Q ; P ] |= ( P //\\ Q )"

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

The tactical `pred_TCL : tactic -> tactic` converts a tactic that works on unlifted sequents into one that works on lifted sequents in exactly the same way.

Admittedly the above proof is unnecessarily arduous: calling the tautology checker is much easier. But it is done to demonstrate, in as simple a setting as possible, how general tactics are lifted. The reason why the lifting of general tactics is desirable is that when LP is used to embed other logics, it may be necessary to mix logical reasoning with special-purpose, non-logical reasoning, such as modal or temporal reasoning. Also, one needs to reason about lifted quantifiers as well, which in general cannot be handled by tautology checking. Examples for all these appear in the next subsection, which also introduces the lifting of theorem tactics.

The last result is needed later, so we bind it with an ML identifier:

```
#let Lemma_0 = top_thm () ;;                                   5
Lemma_0 = |- [ ] |= ( P ==>> Q ==>> (P //\\ Q) )
```

3.3 Lifting Theorem Tactics

A broad class of modal logics are the so-called *normal logics* [2], which are characterized by *Schema K* and the *Rule of Necessitation*:

```
#K_Schema ;;                                                              6
|- !P Q. [ ] |= ( [](P ==>> Q) ==>> ([] P ==>> [] Q) )

#Necessitation_Rule ;;
|- !P. [ ] |= ( P ) ==> [ ] |= ( [] P )
```

For example, TLA is a normal logic and both `K_Schema` and `Necessitation_Rule` can be proved as theorems of our embedding. But for our present purpose, it suffices to regard them as axioms for the uninterpreted modal operator `[] : (* -> bool) -> (* -> bool)`.

Now consider the following goal:

```
#g" ^VALID( ([] P //\\ [] Q) ==>> [](P //\\ Q) ) ";;                      7
"[ ] |= ( ([] P //\\ [] Q) ==>> [](P //\\ Q) )"
```

The first step is to strip the antecedent:

```
#e( pred_TCL STRIP_TAC );;                                                8
OK..
"[ [] Q ; [] P ] |= ( [](P //\\ Q) )"
```

The following key lemma is derived from the result of the previous example using the Rule of Necessitation:

```
#let Lemma_1 = MATCH_MP Necessitation_Rule Lemma_0 ;;                      9
Lemma_1 = |- [ ] |= ( [](P ==>> Q ==>> (P //\\ Q)) )
```

To add `Lemma_1` to the assumption list of the current goal, we need to lift `ASSUME_TAC`. But `ASSUME_TAC` is a theorem tactic, which is lifted by `pred_TTCL` rather than `pred_TCL`:

```
#e( pred_TTCL ASSUME_TAC Lemma_1 );;                                      10
OK..
"[ [](P ==>> Q ==>> (P //\\ Q)) ; [] Q ; [] P ] |= ( [](P //\\ Q) )"
```

Finally, applying the lifted `IMP_RES_TAC` twice with `K_Schema` as the implicative theorem finishes the proof:

```
#e( pred_TTCL IMP_RES_TAC K_Schema );;                                    11
OK..
"[ [](Q ==>> (P //\\ Q)) ; [](P ==>> Q ==>> (P //\\ Q)) ; [] Q ; [] P ] |=
 ( [](P //\\ Q) )"

#e( pred_TTCL IMP_RES_TAC K_Schema );;
OK..
goal proved
|- [ [](Q ==>> (P //\\ Q)) ; [](P ==>> Q ==>> (P //\\ Q)) ; [] Q ; [] P ] |=
   ( [](P //\\ Q) )
|- [ [](P ==>> Q ==>> (P //\\ Q)) ; [] Q ; [] P ] |= ( [](P //\\ Q) )
|- [ [] Q ; [] P ] |= ( [](P //\\ Q) )
|- [ ] |= ( ([] P //\\ [] Q) ==>> [](P //\\ Q) )
```

The next example shows that lifted quantifiers can also be handled:

```
#Reflexivity ;;                                                    12
|- !P. [ ] |= ( P ~~> P )

#Transitivity ;;
|- !P Q R. [ ] |= ( ((P ~~> Q) //\\ (Q ~~> R)) ==>> (P ~~> R) )

#g" ^VALID( (!! n . R(SUC n) ~~> R(n)) ==>> (!! n . R(n) ~~> R(0)) ) ";;
"[ ] |= ( (!! n . R(SUC n) ~~> R(n)) ==>> (!! n . R(n) ~~> R(0)) )"

#e( pred_TCL (DISCH_TAC THEN INDUCT_TAC) );;
OK..
2 subgoals
"[ R(n) ~~> R(0) ; !! n . R(SUC n) ~~> R(n) ] |= ( R(SUC n) ~~> R(0) )"

"[ !! n . R(SUC n) ~~> R(n) ] |= ( R(0) ~~> R(0) )"

#e( pred_TTCL MATCH_ACCEPT_TAC Reflexivity );;
OK..
goal proved
|- [ !! n . R(SUC n) ~~> R(n) ] |= ( R(0) ~~> R(0) )

Previous subproof:
"[ R(n) ~~> R(0) ; !! n . R(SUC n) ~~> R(n) ] |= ( R(SUC n) ~~> R(0) )"

#e( pred_TCL (FIRST_ASSUM (ASSUME_TAC o SPEC "n:num")) );;
OK..
"[ R(SUC n) ~~> R(n) ; R(n) ~~> R(0) ; !! n . R(SUC n) ~~> R(n) ] |=
 ( R(SUC n) ~~> R(0) )"

#e( pred_TTCL IMP_RES_TAC Transitivity );;
OK..
goal proved
|- [ R(SUC n) ~~> R(n) ; R(n) ~~> R(0) ; !! n . R(SUC n) ~~> R(n) ] |=
   ( R(SUC n) ~~> R(0) )
|- [ R(n) ~~> R(0) ; !! n . R(SUC n) ~~> R(n) ] |= ( R(SUC n) ~~> R(0) )
|- [ ] |= ( (!! n . R(SUC n) ~~> R(n)) ==>> (!! n . R(n) ~~> R(0)) )
```

4 Outline of Implementation

Roughly speaking, the effect of applying the tactic `(pred_TCL tac)` to a goal is achieved in three steps:

1. *Unfold* to eliminate lifted operators in the goal;
2. Apply `tac`;
3. *Fold* to re-introduce lifted operators.

For instance, this is what happens when `(pred_TCL STRIP_TAC)` is applied to a goal:

```
#g" ^VALID( (P //\\ Q) ==>> (P \\// Q) ) ";;                                    13
"[ ] |= ( (P //\\ Q) ==>> (P \\// Q) )"

#e( pred_SEQ_UNFOLD_TAC "x:*" );;
OK..
"P x /\ Q x ==> P x \/ Q x"

#e( STRIP_TAC );;
OK..
"P x \/ Q x"
    [ "P x" ]
    [ "Q x" ]

#e( pred_SEQ_FOLD_TAC "x:*" );;
OK..
"[ Q ; P ] |= ( P \\// Q )"
```

The tactic (`pred_TTCL ttac th`) follows a similar pattern except that the theorem `th` must also be unfolded, while `pred_TAUT_TAC` does not need the folding phase at all.

Acknowledgements

The author is grateful to the members of HOL Seminar at UCLA for interesting discussions and to Dr. von Wright for sending him [7] and [8].

References

[1] R. J. Boulton, *The HOL* `taut` *Library*, Univ. of Cambridge Computer Lab., (1991).

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

[3] 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.

[4] 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.

[5] L. Lamport, "The Temporal Logic of Actions", DEC SRC technical report 79, (1991).

[6] T. F. Melham, *The HOL* `pred_sets` *Library*, Univ. of Cambridge Computer Lab., (1992).

[7] J. von Wright, "Mechanising the Temporal Logic of Actions in HOL", in *Proc. of the HOL Tutorial and Workshop*, (1991).

[8] J. von Wright and T. Långbacka, "Using a Theorem Prover for Reasoning about Concurrent Algorithms", in *Proc. of Workshop on Computer-Aided Verification*, (1992).