Interactive Proofs Of Programs Using High-Order Logic (HOL)

Stephen Motty
Rutter Inc. (Technologies Division)
and Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University smotty@{rutter.ca, mun.ca}

Theodore Norvell
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University
theo@mun.ca

***Abstract*—Programs can be treated as mathematical objects and thus proofs of programs are simply mathematical proofs. Software that handles proofs in various areas of mathematics such as HOL4/Kananaskis can therefore be used for verification that programs meet their specifications. User interaction will be simplified by adding a graphical editor for entry of specifications, programs, and proofs; the editor is coupled to the proof-checker using a client/server interface.**

This paper describes progress being made in support of the above software design methodology. Demonstrations of some essential proofs and core theories written in the HOL4/Kananaskis language are presented.

***Index Terms*—SIMPPLE project, Mathematics of programming, Computer programs, Software development.**

I. Introduction

This paper presents preliminary results on the use of HOL4 [2] to mechanize the checking of the refinement steps in proofs that programs meet their specification. We use Hehner's [1] predicative programming approach to reduce programs to terms in a higher-order logic. Some background understanding of logic is a pre-requisite.

II. Predicative Programming

In [1] commands are interpreted as boolean expressions relating initial to final values of state variables. For example the specification $x' = y + 1$ requires that the final value of state variable x be one more than the initial value of state variable y. We can understand commands (for non looping code) to be boolean expressions as follows:

$$\textbf{skip} \triangleq x' = x \wedge y' = y \wedge z' = z \tag{1}$$

$$x := E \triangleq x' = E \wedge y' = y \wedge z' = z \tag{2}$$

$$F; G \triangleq \exists \dot{x}, \dot{y}, \dot{z} \cdot F^{x',y',z'}_{\dot{x},\dot{y},\dot{z}} \wedge G^{x,y,z}_{\dot{x},\dot{y},\dot{z}} \tag{3}$$

$$\textbf{if } E \textbf{ then } F \textbf{ else } G \triangleq F \triangleleft E \triangleright G \tag{4}$$

Here E is an expression with (potentially) free variables x, y, and z, while F, and G are specifications in the form of boolean expression in which the free variables are x, y, z, x', y', and z'. In (4) we use the three argument operator $a \triangleleft b \triangleright c$, where b is boolean, to mean a, if b is true, and c, if b is false. Note that operators on commands such as sequential composition and if-then-else can be applied to any specification; this is essential for top-down step-wise refinement. We can now define refinement as a relation between boolean expressions

$$(F \sqsubseteq G) \triangleq \forall x, y, z, x', y', z' G \Rightarrow F \tag{5}$$

Two particularly useful programming laws are the forward and backward substitution laws given by

$$F^x_E = (x := E; F) \tag{6}$$

$$F \sqsubseteq F^{x'}_{E'}; x := E \tag{7}$$

Representing specifications and commands by boolean expressions is concise and convenient for manual derivation, but suffers from some of drawbacks for mechanization. First, in equations such as (3), (6) and (7), we have to be careful rewrite the expressions F and G without using programming notations such as assignment or sequential composition before doing any substitutions. Second, the definitions are relative to a particular set of variables; if we were to introduce another variable, the definitions of assignment, sequential composition, and refinement would have to be rewritten. Third, the laws for assignment, forward substitution, and backward substitution as stated apply only to the variable x; technically we should restate each of these laws for each variable.

For the above reasons, it is better, for the purposes of formalizing and mechanizing predicative programming, to take the point of view that specification are boolean functions (predicates) of two states, that expressions are functions of one state, and that states are functions from some set of variable names to values [3]. This leads to the following definitions

$$\textbf{skip}\, s\, s' \triangleq (s = s') \tag{8}$$

$$(x := e)\, s\, s' \triangleq \forall v \cdot\ s'\, v = e\, s \triangleleft v = x \triangleright s'\, v = s\, v \tag{9}$$

$$(f; g)\, s\, s' \triangleq \exists \dot{s} \cdot f\, s\, \dot{s} \wedge g\, \dot{s}\, s' \tag{10}$$

$$(\textbf{if } e \textbf{ then } f \textbf{ else } g)\, s\, s' \triangleq f\, s\, s' \triangleleft e\, s \triangleright g\, s\, s' \tag{11}$$

Refinement is expressed as

$$(f \sqsubseteq g) \triangleq \forall s, s' \cdot g\, s\, s' \Rightarrow f\, s\, s' \tag{12}$$

The substitution laws are expressed in terms of a substitution operator on states

$$\mathit{sub}\, s\, x\, z \triangleq \lambda y \cdot z \lhd x = y \rhd s\, y \tag{13}$$

$$\mathit{initSub}\, f\, x\, e \triangleq \lambda s \cdot \lambda s' \cdot f\, (\mathit{sub}\, s\, x\, (e\, s))\, s' \tag{14}$$

$$\mathit{finalSub}\, f\, x\, e \triangleq \lambda s \cdot \lambda s' \cdot f\, s\, (\mathit{sub}\, s'\, x\, (e\, s')) \tag{15}$$

$$\mathit{initSub}\, f\, x\, e = (x := e; f) \tag{16}$$

$$f\ \sqsubseteq \mathit{finalSub}\, f\, x\, e; x := e \tag{17}$$

All these definitions and laws are polymorphic. Consider the definition of assignment; suppose the type of x is *var* and the type of s and s' is $\mathit{var} \rightarrow \mathit{val}$, then the type of $:=$ can be inferred to be $\mathit{var} \rightarrow \mathit{exp} \rightarrow \mathit{state} \rightarrow \mathit{state} \rightarrow \mathit{bool}$, where $\mathit{state} = \mathit{var} \rightarrow \mathit{val}$ and $\mathit{exp} = \mathit{state} \rightarrow \mathit{val}$. The point is that *var* and *val* can be any types at all and the variable x ranges over all values of type *var*.

Since we quantify over states, which are functions, we are dealing with higher-order logic. This suggests that, in mechanizing predicative programming, we turn to a theorem proving system for higher-order logic.

III. HOL

The HOL system is an interactive theorem proving system for higher-order logic written in ML. HOL allows ML style parametric polymorphism, which is good, as our definitions and laws from the previous section are polymorphic. Indeed the definitions of the previous section can be transliterated into HOL, essentially as they are.

The logic of HOL is defined by a small set of inference laws, also known in HOL as rules. Rules are ML functions that take one or more theorems and return a theorem. Thus HOL is at heart based on forward reasoning, i.e., computing new theorems from old.

As described in [2], the proof manager relies upon tactics. Tactics are functions which take a primary goal (i.e. a potential theorem) and returns a list of subsidiary goals which, if proved, would suffice to prove the primary goal. As a second output, tactics provide a derived inference rule that can be used once the subsidiary goals have been turned into theorems. Tactics, thus support goal-directed reasoning, i.e. reasoning that starts from a goal and works backward toward the axioms and previously proved theorems.

There are a number of libraries in HOL which permit management of proofs. One library in particular, known as 'bossLib', provides a suite of basic automated proving tools. A number of other libraries provide type syntaxes which make it possible to extend HOL's native data types to include numbers, strings and lists. By using HOL as a tool for theorem proving, we eliminate the need to develop such theorems on our own.

Given a list of existing theorems that can serve as lemmas, the following strategy can be used to derive a proof:

1) Introduce the lemma as an assumption of the proof.
2) Rewrite the goal using propositional logic until a term is of the same form as the lemma.
3) If the lemma contains polymorphic or abstract types, instantiate the types to be consistent with the proof.
4) If the lemma and the sub-goal are alpha-convertible, the proof can be completed by the HOL tactic, ACCEPT_TAC. Otherwise, show that the sub-goal evaluates to true using EVAL_TAC. If the evaluation depends on formulaic simplifications, the HOL tactic PROVE_TAC may be used to automatically complete the proof from HOL's built-in libraries of logic theory.

IV. Predicative Programming in HOL

Over the past months, an initiative was undertaken to formalize some of the basic premises of predicative programming using the HOL proof manager. A primary goal was to achieve a proof of the forward substitution law, and use the law to prove that two imperative programs both satisfy the specification: $s'\, x = s\, y \wedge s'\, y = s\, x$. The first program satisfies the specification in the general case, while the second satisfies it in the case where values retrieved through x and y are numeric.

The following sections present the development of the program which was successful in achieving these aims. The program will be used as part of a client-server based system for interactive proofs of programs using a web-enabled editor.

A. Refinement

We define refinement following (12). For notational convenience, definition 1 is an infix operator that takes its first argument on the left-hand side. (HOL uses the exclamation point (!) for universal quantification.)

Definition 1 ($v \sqsubseteq u$ (refinement)):

```
set_fixity "[=." (Infixl 500);
val DefinitionOfRefinement = xDefine "Refinement"
  `v [=. u <=> !s s'. u s s' ==> v s s'`
;
```

Having defined refinement in this fashion, the notational form of the left-hand side requires a conversion to and from its corresponding propositional form on the right-hand side. The proof of Theorem 2 is a function that converts a goal from the form of '$\lambda s\, s' \cdot v\, s\, s' \sqsubseteq u$ to the form $u\, s\, s' \Rightarrow v\, s\, s'$. The derivation relies on four predefined tactics/rules of the HOL logic system. (Recall that rules operate on theorems, while tactics operate on goals.)

- REPEAT
  Modifies a tactic so that it repeats until it is no longer applicable to the goal. For example, REPEAT modifies GEN_TAC so that every universal quantifier is eliminated from a goal.
- PURE_ONCE_REWRITE_TAC/ PURE_ONCE_REWRITE_RULE
  Performs a limited rewrite of terms in an existing theorem; it is used here for removing custom notation and abbreviations, replacing them with formal expressions.
- GEN_TAC/GEN_ALL
  GEN_TAC removes the outermost universal quantifier from a goal. GEN_ALL is the rule corresponding to REPEAT GEN_TAC.

- BETA_TAC/BETA_RULE
  This tactic evaluates lambda expressions embedded within the goal.

(HOL uses the backslash (\) for λ.)

Theorem 2 (Rewrite Refinement):

$$(\lambda s\, s' \cdot v\, s\, s' \sqsubseteq u) \Leftrightarrow (u\, s\, s' \Rightarrow v\, s\, s')$$

Proof:

```
val REFINEMENT_TAC =
(*  \s s' .v s s' [=. u *)
  (PURE_ONCE_REWRITE_TAC
    [DefinitionOfRefinement])
(*  !s s'. u s s' ==> \ s s'. v s s' *)
  THEN (REPEAT GEN_TAC)
(*  u s s' ==> (\s s'. v s s') s s' *)
  THEN (BETA_TAC)
(*  u s s'   ==>  v s s' *)
;

fun REFINEMENT_RULE th =
BETA_RULE (
  GEN_ALL (
    PURE_ONCE_REWRITE_RULE [DefinitionOfRefinement]
      (th)
  )
);
```

■

B. Definition of Assignment

Next we define assignment, following 9.

Definition 3 ($x := e$ (assign)):

```
Define `assign x e s s' =
  !y. if x=y then (s' y = e s) else (s' y = s y)
;
```

When a function is defined in HOL, its expansion is treated as an axiom of the logic. Any term that contains the expression can be promoted to a theorem by using the HOL ASSUME function. This places the term as both the hypothesis and the conclusion of a new theorem. The EVAL_RULE function can be used to evaluate the conclusion of the theorem while leaving the hypothesis untouched. This replaces the custom notation with the equivalent expansion on the right-hand side. In the case of Definition 3, the right-hand side expansion is that of a universally quantified expression; the HOL SPEC tactic can therefore instantiate a specific value using universal specialization. In this way, we can test the definition by verifying that $s'x = es$ is implied whenever y is instantiated by x in definition of assignment. (HOL encloses assumptions in square brackets, and uses $\vdash$ to separate the hypotheses from the conclusion of a theorem.)

Theorem 4 ($(x := e)\, s\, s' \vdash s'\, x = e\, s$):

```
val xForyImplies_sx_Is_es =
  EVAL_RULE (SPEC ``x:'a`` (
    (EVAL_RULE (ASSUME ``assign x e s s'``))
  )
  (*  [
      assign x e s s'
    ] |-
        s' x  = e s  : thm
  *)
;
```

C. Sequential Composition

The modelling of sequential composition in HOL is made possible by using an existential specification. Specifically, we assert that there exists an intermediate state, s'', such that initial specification f provides a path from s to s'', and the final specification g provides a path from s'' to s'. In the user interface we wish to implement this as two consecutive blocks separated by a semicolon: hence $f;g$. In the HOL proof checker, this corresponds to the definition introduced next. (HOL uses the question mark (?) for existential quantification.)

Definition 5 ($f;g$ (sequential composition)):

```
Define `sc f g s s' = ? s'' . f s s'' /\ g s'' s'` ;
```

D. Definition of Substitution

Definitions 3 and 5 yield tests on two states. Definition 6 is somewhat different in that it first transforms a reference state space or model, and then shows that the transformed state space passes the specification.

Definition 6 (f^x_e (initsub)):

```
Define `initsub f x e s s' =
  let s'' = \y. if x=y then e s else s y
  in  f s'' s'
`;
```

E. Proof of One-Point Lemma

Logical proofs in HOL are largely about transformation. HOL works best when it can be demonstrated that the proof at hand is nothing more than a type-instantiation, alpha-conversion, specialization or generalization of an existing theorem. Lemmas play an important role in this process. They act as waypoints along the path of a proof; by navigating from an existing form to the form of an lemma, large proofs can be solved as a series of sub-goals.

The one-point law is that for a variable x, $x = e \wedge A$ is the same as $x = e \wedge A^x_e$ (and similarly for implication). We only need a special case.

```
fun EXHAUSTIVELY x  = (REPEAT (CHANGED_TAC x));

val REP_EVAL_TAC  =   (EXHAUSTIVELY EVAL_TAC);

(* if v is in the assumption list, replace v with T \\
     in the goal *)
val thmAcceptInPlace =
  UNDISCH (prove (``(v:bool) ==> (v <=> T)``,\\
       REP_EVAL_TAC))
;
(* undo DISCH_ALL *)
fun USE_CONTEXT (asl:term list) (th:thm)  =
  if (null asl) then th else (UNDISCH (USE_CONTEXT (\\
       tl(asl)) th))
;
(* VSUB rewrites free variable v with an expression \\
     *)
fun VSUB (v:term) (e:term) (th:thm) =
  USE_CONTEXT (hyp th) (SPEC e (GEN v (DISCH_ALL th))\\
       )
;
(* if th is an expression in the assumption list,
  simplify the expression to T in the goal *)
fun MAKE_IT_SO (th:thm) = (
    (SUBST_TAC [(VSUB ``v:bool`` (concl th) \\
         thmAcceptInPlace)])
  THEN
```

```
      EVAL_TAC
  ) ;
```

Theorem 7 ($x = x \wedge f\,x\,t \Leftrightarrow f\,x\,t$):
Proof:

```
  val thmOnePointLemma=
    prove (
      `` (x = x) /\ (f x t ) <=> f x t``,
      (
        EQ_TAC  THENL [(
          (*  []  |- ((x =x) /\ (f  x t )) ==> f x t *)
          (ASSUME_TAC (REFL ``x``))
            (* [x=x] |- ((x=x) /\ (f x t)) ==> f x t *)
          THEN
            (FIRST_ASSUM  MAKE_IT_SO)
        ),(
          (*  []  |- (f x t) ==> (x=x) /\ f x t *)
          (DISCH_TAC)
            (*  [ f x t ] |- (x=x) /\ f x t *)
          THEN
            (FIRST_ASSUM  MAKE_IT_SO)
        )]
      )
    )
  ;
```

■

The preceding proof involves a number of manipulations using the goal's assumption list. The value of assumptions in logic must not be under-estimated. In automated proving systems such as HOL, assumption lists serve as a stack of known truths for specific instances, cases and states. This stack primes the theorem proving engine. HOL tactics such as FIRST_ASSUM can then test every known assumption until one is found that progresses the proof towards its conclusion.

V. Program Verification using HOL

Having set up the definitions, we are ready to prove a fundamental programming law, the forward substitution law (16). This particular proof is rather lengthy and is presented in outline form only. The full proof is available upon request.

Theorem 8: Forward substitution

```
  val thmAbstractSpecification =
  let val
    SPEC_EQ_THM =
      prove (
        ``((!s s').f s s' = g s s') <=> (f = g)``,
        (* Proof available upon request *)
      )
  in
    INST_TYPE [
      alpha |-> ``:'a -> 'b``,
      beta  |-> ``:'a -> 'b``,
      gamma |-> ``:bool``
    ] SPEC_EQ_THM;
  end;
  val thmConditionalFunction =
    prove (
      ``
        !(t a b c ).
          (!y .if a y then t y = b y else t y = c y)
          <=>
          (t = (\y. if a y then b y else c y))
      ``,
      (* Proof available upon request *)
    )
  val thmForwardSubstitution =
    prove (
      ``!f e x s s'. sc (assign x e) f s s' = initsub f\\
           x e s s' ``,
      (* Proof available upon request *)
    )
```

The right hand side of `thmForwardSubstitution` can be shown to be a existential witness to the left-hand side. The proof relies on Theorem 7.

We can use the theory thus developed to formally verify two algorithms that swap values held in two variables. Theorem 8 is used in the verification to transform the initial goal to an "if.. then... else if.." structure. Upon doing so, the assumption list can be used to evaluate terms held in the state space at various times. The `EvaluateFor` (below) tactic will be used in the proof to perform the evaluation. In order for the tactic to work, every assumption initially in the assumption list must be of the form $\forall y \cdot f(s\,y)(s'\,y)$ where s and s' are state spaces. The function accepts a list of values to assign to the bound variable y of each assumption. The tactic works by evaluating $f(s\,y)(s'\,y)$ for the instance where $y = \mathtt{hd}\,\mathtt{valList}$ ($\mathtt{hd}\,\mathtt{valList}$ is HOL's method for retrieving the pending value from the list). After doing so for each value in the list, REPEAT DISCH_TAC pushes these evaluated results onto the assumption list. The automated theorem prover then uses these evaluated terms to perform repeated substitutions until no further simplification results.

```
  fun EvaluateFor valList =
    if(null valList) then (
      (REPEAT DISCH_TAC)
    THEN
      (REPEAT (FIRST_ASSUM
        (fn th => (CHANGED_TAC (SUBST_TAC [th])))
      ))
    ) else (
      (EVERY_ASSUM
        (fn th => let val instance = (SPECL [(hd \\
             valList)] th)
          in
            (
              (ASSUME_TAC instance) THEN
              (UNDISCH_TAC (concl instance))  THEN
              (REP_EVAL_TAC)
            )
          end
        )
      )
      THEN
        (EvaluateFor (tl valList))
    )
  ;
```

A. Swapping: General Case

The general case of the swapping program is handled by considering the simple command given by $t := x; x := y; y := t$. Using the theorems thus created, the aim is to interactively prove that $s'x = sy \wedge s'y = sx$ is refined by this command.

Theorem 9: Swap

```
  val GeneralSwap = let val
    conversion =
      PURE_ONCE_REWRITE_RULE
        [thmAbstractSpecification]
        (
          SPECL   [
            ``"t"``,
            ``(assign "x" (\(s:string->'b). s "y"))``,
            ``(\(s:string->'b).s "x")``
          ]
          ( INST_TYPE [
              alpha |-> ``:string``,
            gamma |-> ``:string->'b``
          ] (
            REFINEMENT_RULE (
              SPECL [
                ``f:('a->'b)->'c->bool``,
                ``e:('a->'b)->'b``,
                ``x:'a``
              ]
              thmForwardSubstitution
            )
          )
        )
      )
  in
    prove (
      ``(
        ( \(s:string->'b) (s':string->'b) .
            ((s' "x")=(s "y")) /\ ((s' "y")=(s "x"))
        )
        [=.
        (sc
          (sc
            (assign ("t") (\ (s:string->'b).s "x"))
            (assign "x" (\ (s:string->'b). s "y"))
          )
          (assign "y" (\ (s:string->'b).s "t"))
        )
      )``
      ,
      (SUBST_TAC [conversion])
    THEN
      (REP_EVAL_TAC)
    THEN
      (REPEAT STRIP_TAC)
    THEN
      (EvaluateFor [``"t"``,``"x"``,``"y"``])
    THEN
      REP_EVAL_TAC
    )
  end;
```

B. Swapping: Numeric Types

In the case where numeric values are retrieved from each state space via x and y, it is possible to avoid the allocation of the intermediate variable t. Apart from additional type information, the swapping specification remains the same, but in this case we wish to verify that it also applies to the program given by $x := x + y; y := x - y; x := x - y$.

Theorem 10: Numeric swap

```
  val NumericSwap = let val
    conversion =
      PURE_ONCE_REWRITE_RULE
        [thmAbstractSpecification] (
        SPECL [
          ``"x"``,
          ``(assign "y" (\ (s:string->num). ((s "x") - \\
              (s "y"))))``,
          ``(\ (s:string->num).((s "x") + (s "y")))``
        ](
          INST_TYPE [
              alpha |-> ``:string``,
              beta |-> ``:num``,
              gamma |-> ``:string->num``
          ](
            REFINEMENT_RULE (
              SPECL [
                ``f:('a->'b)->'c->bool``,
                ``e:('a->'b)->'b``,
                ``x:'a``
              ]
              thmForwardSubstitution
            )
          )
        )
      )
  and
    lemma = prove (
        ``!(a:num) (b:num). (a + b -(a + b -b)) = (b + \\
            a - a)``,
        (PROVE_TAC [LESS_EQ_REFL, LESS_EQ_ADD_SUB, \\
            SUB_EQ_0,ADD_0,ADD_SYM])
      )
  in
    prove (
      ``
        (\(s:string->num) (s':string->num). ((s' "x")=(\\
            s "y")) /\ ((s' "y")=(s "x")))
        [=.
        (sc
          (sc
            (assign "x" (\ (s:string->num).((s "x") + (\\
                s "y"))))
            (assign "y" (\ (s:string->num). ((s "x") - \\
                (s "y"))))
          )
          (assign "x" (\(s:string->num).((s "x") - (s "\\
              y"))))
        )
      ``
      ,
        (SUBST_TAC [conversion])
      THEN
        (REP_EVAL_TAC)
      THEN
        (REPEAT STRIP_TAC)
      THEN
        (EvaluateFor [``"x"``,``"y"``])
      THEN
        (PROVE_TAC [LESS_EQ_REFL, LESS_EQ_ADD_SUB, \\
            SUB_EQ_0,ADD_0,lemma])
      )
  end;
```

As with Theorem 9, the proof of Theorem 10 relies on the substitution law and uses `EvaluateFor`. Since arithmetic plays a significant role in the proof, the derivation benefits greatly from the built-in libraries of HOL. The implementation accesses these libraries by means of the HOL tactic PROVE_TAC and the automated model elimination algorithm (MesonLib) invoked by it.

VI. Conclusions

We have presented a number of proofs based on predicative programming as implemented in HOL. These proofs have led to a number of custom tactics and utility functions that find frequent application during a derivation. The knowledge gained will assist in the design of an interactive editor that can assist in the entry of these proofs using a web-based client interface. Future work will attempt to capture the proof finding strategies used as a small number of tactics. Each refinement step should be automatically checked by HOL or a similar automated theorem prover. That is, we do not expect the user to be involved in any interactions with HOL. The user should simply present the proof at a sufficient, but not onerous, level of detail and the tactics should be able to verify each step of the derivation.

References

[1] Eric C.R. Hehner, a Practical Theory of Programming, Springer-Verlag, 1993.

[2] Michael Norrish and Konrad Slind, ”The HOL System Tutorial”, Cambridge Research Center of SRI International

[3] Theodore S. Norvell, “The Predicative Semantics of Loops”, in Algorithmic Languages and Calculi, Richard Bird and Lambert Mertens Eds, Chapman-Hall, 1997.