The HOL theorem-proving system

Michael Norrish
Michael.Norrish@nicta.com.au

National ICT Australia

8 September 2004
Outline

Introduction
  History
  High-level description

Build a HOL kernel
  Design philosophy
  Basic types
  Logic
  Implementation
  Theories

Theorem-proving applications
  BDDs and symbolic model-checking
  TCP/IP trace-checking
What is HOL?

- A family of theorem-provers, stemming from University of Cambridge and work by Mike Gordon
- I will describe most recent implementation on the most active branch of development, HOL4
- HOLs on other branches of development include Harrison’s HOL Light, and ProofPower
- Ancestors of HOL4 are hol98, HOL90 and HOL88.
- Principal development of HOL is now done by me and Konrad Slind.
Where does HOL come from?

- Everything begins with LCF
  - Developed by Milner, Gordon and others in Stanford and Edinburgh starting in 1972. (One of the early developers was Malcolm Newey, now at ANU’s Dept. of Computer Science.)

- LCF is a theorem-proving system for proving theorems in the Logic of Computable Functions (due to Dana Scott).

- The Edinburgh LCF system introduced two crucial innovations:
  - Theorems as a protected abstract data type; and
  - Use of ML

- Isabelle, HOL, Coq and the Nuprl systems all acknowledge this ancestry: they embody the “LCF philosophy”
HOL evolved from LCF because Mike Gordon wanted to do hardware verification.

LCF is a logic for computable functions using denotational semantics, where every type is modelled via a *domain*.

Hardware’s demands are much simpler.
Birth of HOL

- HOL evolved from LCF because Mike Gordon wanted to do hardware verification.
- LCF is a logic for computable functions using denotational semantics, where every type is modelled via a *domain*.
- Hardware’s demands are much simpler.

- But naturally higher order:
  - Signals are functions from time to bool.
  - Devices are relations from signals to signals.
HOL since the 1980s

- First implementation effort was in “Classic ML” on top of Common Lisp — this led to HOL88 (described in book by Gordon and Melham)
- Konrad Slind wrote a version in Standard ML (SML/NJ implementation) — HOL90
- Slind also main author of hol98, which switched to Moscow ML, and a new representation for theories on disk
- Slind and I are the main authors of HOL4 (since June 2002). Other developers update the SourceForge repository from Cambridge, Oxford and the USA.
The core of HOL

The LCF design philosophy:

- **inference rules**
- **:thm**
- **axioms**

The ML inference rules both depend on the core type of `thm` and manipulate theorems to derive new ones.
How HOL is used in practice

- HOL is a programming environment
  - system command = a programming language
  - proof = computation of theorems
- Theory-creation in the HOL system

User \[ \rightarrow \]

ML source text:
  - specifications
  - proofs

HOL

HOL theory file:
  - definitions
  - theorems
Standard theorem-proving facilities

HOL4 comes with standard theorem-proving technology:

- **Definition tools:**
  - For *types*: inductive/algebraic, quotients, records and abbreviations
  - For *terms*: well-founded or primitive recursive function definition, inductive relations

- **Proof support:**
  - Simplifier (contextual rewriting with conditional rewrites, embedded decision procedures)
  - First-order reasoning (resolution and model elimination)
  - Arithmetic decision procedures (for $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$)
A hardware verification example

- Fragment of an adder circuit:

```
cin | i1 | i2 |
----|----|----|
     | p  |    |
```

- We wish to verify that

\[ o = (i1 + i2 + \text{cin}) \mod 2 \]

- There are three steps:
  - write a specification of the circuit in logic
  - formulate the correctness of the circuit
  - prove the correctness of the circuit
Specify the circuit

- Specification of an XOR gate:

  ![XOR gate diagram]

  \[
  \vdash \text{Xor}(i_1, i_2, o) = (o = \neg (i_1 = i_2))
  \]

- Specification of the adder circuit:

  ![Adder circuit diagram]

  \[
  \vdash \text{Add}(\text{cin}, i_1, i_2, o) = \exists p. \text{Xor}(\text{cin}, i_1, p) \land \text{Xor}(p, i_2, o)
  \]
Specify the circuit

- **ML source text:**

```plaintext
val Xor = 
  Define‘Xor(i1,i2,o) = (o = ¬(i1:bool = i2))‘;

val Add = 
  Define‘Add(cin,i1,i2,o) = 
     ∃p. Xor(cin,i1,p) ∧ Xor(i2,p,o)‘;
```
Formulate correctness

- Abstraction function from bool to num:

\[
\begin{array}{c}
\text{bool} \\
T \\
F \\
\end{array}
\rightarrow
\begin{array}{c}
\text{num} \\
1 \\
0 \\
\end{array}
\]

\[\vdash Bv(b) = \text{if } b \text{ then } 1 \text{ else } 0\]

- Logical formulation of correctness:

\[\vdash \forall \text{cin i1 i2 o.} \]
\[\text{Add(cin, i1, i2, o) } \Rightarrow \]
\[Bv \ o = (Bv \ i1 + Bv \ i2 + Bv \ cin) \ MOD \ 2\]
Formulate correctness

▶ ML source text:

```ml
val Bv = Define 'Bv b = if b then 1 else 0';

g '∀cin i1 i2 o.
   Add(cin,i1,i2,o) ⇒
   (Bv o = (Bv i1 + Bv i2 + Bv cin) MOD 2)';
```

▶ The g function establishes a formula as a goal that we wish to prove
Develop the proof interactively

In an interactive ML session, we have stated the ‘goal’:

∀cin i1 i2 o. 
Add (cin, i1, i2, o) ⇒ 
(Bv o = (Bv i1 + Bv i2 + Bv cin) MOD 2)'

Expand with definitions of the circuit:

- e(RW_TAC arith_ss [Add, Xor]);
OK..
1 subgoal:
> val it =
    Bv ¬(i2 = ¬(cin = i1)) =
        (Bv cin + (Bv i1 + Bv i2)) MOD 2

: goalstack
Develop the proof interactively

- Rewrite with the definition of Bv

```plaintext
- e (RW_TAC arith_ss [Bv]);
OK..

Goal proved.
|- Bv ¬(i2 = ¬(cin = i1)) =
   (Bv cin + (Bv i1 + Bv i2)) MOD 2
> val it =
   Initial goal proved.
|- ∀cin i1 i2 out.
   Add (cin,i1,i2,out) ⇒
   (Bv out = (Bv i1 + Bv i2 + Bv cin) MOD 2)
```

- Could combine two steps into one;
  RW_TAC arith_ss [Bv,Add,Xor] solves the goal.
The ML deliverable

```ml
val Xor = Define 'Xor(i1, i2, out) = (out = ¬(i1:bool = i2))';

val Add = Define 'Add(cin, i1, i2, out) =
   ∃p. Xor(cin, i1, p) ∧ Xor(i2, p, out)';

val Bv = Define 'Bv b = if b then 1 else 0';

val Add_CORRECT = store_thm("Add_CORRECT",
   "∀cin i1 i2 out.
   Add(cin, i1, i2, out) ⇒
   (Bv out = (Bv i1 + Bv i2 + Bv cin) MOD 2)'",
   RW_TAC arith_ss [Add, Xor, Bv]);
```
Other modes of use

- HOL as proof engine

  Example: TCP protocol trace-checking.

- Hybrid theorem-proving:

  Examples: links with Gandalf [Hurd], ACL2 [Staples], Voss [Joyce/Seger].
Introduction

History
High-level description

Build a HOL kernel

Design philosophy
Basic types
Logic
Implementation
Theories

Theorem-proving applications

BDDs and symbolic model-checking
TCP/IP trace-checking
Build your own HOL

- HOL is a relatively small system, built on a small kernel
- It's designed to be experimented with
- Numerous people have re-implemented significant parts of the kernel
- The kernel supports a narrow API, so it’s easy to provide new implementations
Build your own HOL

- HOL is a relatively small system, built on a small kernel
- It’s designed to be experimented with
- Numerous people have re-implemented significant parts of the kernel
- The kernel supports a narrow API, so it’s easy to provide new implementations

- In slides to come, I’ll present an idealised kernel’s API
Build your own HOL

- HOL is a relatively small system, built on a small kernel
- It’s designed to be experimented with
- Numerous people have re-implemented significant parts of the kernel
- The kernel supports a narrow API, so it’s easy to provide new implementations

- In slides to come, I’ll present an idealised kernel’s API
- The HOL4 kernel is a “distorted” version of this ideal
Design keywords

**Modularity:** To support custom applications, it must be possible to assemble different subsets of HOL functionality into real systems.

**Separability:** Custom applications should only link or include the code they use.

**Efficiency:** Code should perform as well as possible on big terms/theorems (thousands of conjuncts, lots of binders, &c).
Introduction

History
High-level description

Build a HOL kernel

Design philosophy

Basic types
Logic
Implementation
Theories

Theorem-proving applications

BDDs and symbolic model-checking
TCP/IP trace-checking
Types

Types are either *variables*, or an *operator* of arity $n$ applied to $n$ types.

```plaintext
eqtype hol_type
val mk_type : string * hol_type list -> hol_type
val mk_vartype : string -> hol_type
val dest_type : hol_type -> string * hol_type list
val dest_vartype : hol_type -> string
```

For example: $\alpha$, $(\alpha)\text{list}$, and $(())\text{num}\text{list}$
(where list has arity 1, and num has arity 0)
Operations on types

```ml
val type_subst : (hol_type,hol_type) subst -> hol_type -> hol_type
val new_type : string * int -> unit
```

- `type_subst` substitutes for type variables only
- `new_type` updates a global table of known types.
- `mk_type` fails if it fails to respect this table’s stored arities.
Terms

val mk_var : string * hol_type -> term
val mk_const : string * hol_type -> term
val mk_comb : term * term -> term
val mk_abs : term * term -> term

val new_const : string * hol_type -> unit

- Terms are either *variables*, *constants*, *applications* or *abstractions*.
- `mk_const(s,ty)` fails if the `ty` is not an instantiation of some `ty'`, where `new_const(s,ty')` was called earlier.
- `mk_comb` fails if the types are incompatible.
- `mk_abs(v,t)` fails if `v` is not a variable.
Operations on terms

val inst : (hol_type, hol_type) subst -> term -> term
val subst : (term, term) subst -> term -> term
val free_vars : term -> term set
val compare : term * term -> order
val match_term : hol_type set * term set ->
  term -> term ->
  ((hol_type, hol_type) subst *
   (term, term) subst)

(There are also dest inversions for all the mk functions.)
The only way to create theorems is through rules of inference!
Introduction

History
High-level description

Build a HOL kernel

Design philosophy
Basic types

Logic
Implementation
Theories

Theorem-proving applications

BDDs and symbolic model-checking
TCP/IP trace-checking
The kernel’s logic

- There at least as many different presentations of higher-order logic as there are HOL systems
- In slides to come, I will present one very idealised version, similar to that used in Harrison’s HOL Light system
The kernel’s logic

- There at least as many different presentations of higher-order logic as there are HOL systems
- In slides to come, I will present one very idealised version, similar to that used in Harrison’s HOL Light system
- HOL4 is not as purist as this, for (possibly misplaced) efficiency reasons, and because it gained all sorts of baggage as the system evolved
The primitive context

- Three types: bool (arity 0), ind (arity 0) and fun (arity 2). 
  \((\alpha, \beta)\text{fun} \text{ is written } \alpha \rightarrow \beta.\)

- Two constants:
  
  \(= : \alpha \rightarrow \alpha \rightarrow \text{bool}\)
  
  \(\varepsilon : (\alpha \rightarrow \text{bool}) \rightarrow \alpha\)
Rules of inference—I

\[
\frac{\vdash t = t}{\text{REFL}}
\]

\[
\frac{\Gamma \vdash f = g \quad \Delta \vdash x = y}{\Gamma \cup \Delta \vdash f \ x = g \ y} \quad \text{MK\_COMB}
\]

\[
\frac{\Gamma \vdash t = u}{\Gamma \vdash (\lambda x. \ t) = (\lambda x. \ u)} \quad \text{ABS}
\]

\[
\frac{\Gamma \vdash (\lambda x. \ t) x = t}{\text{BETA}}
\]

Side-conditions:
- In MK\_COMB, \( f \ x \) (and \( g \ y \)) must be valid terms (well-typed)
- In ABS, \( x \) must not be free in \( \Gamma \)
Rules of inference—II

- **ASSUME**
  \[
  \{t : \text{bool}\} \vdash t
  \]

- **EQ_MP**
  \[
  \Gamma \vdash t = u \quad \Delta \vdash (t : \text{bool})
  \quad \frac{}{\Gamma \cup \Delta \vdash u}
  \]

- **DED_ANTISYM**
  \[
  \Gamma \vdash (u : \text{bool}) \quad \Delta \vdash (v : \text{bool})
  \quad \frac{}{(\Gamma \setminus \{v\}) \cup (\Delta \setminus \{u\}) \vdash u = v}
  \]

- **INST_TYPE**
  \[
  \Gamma \vdash t
  \quad \frac{}{\Gamma[\tau_1/\alpha_1 \ldots \tau_n/\alpha_n] \vdash t[\tau_1/\alpha_1 \ldots \tau_n/\alpha_n]}
  \]

- **INST**
  \[
  \Gamma \vdash t
  \quad \frac{}{\Gamma[M_1/v_1 \ldots M_n/v_n] \vdash t[M_1/v_1 \ldots M_n/v_n]}
  \]
Rules of inference—III

\[ \Gamma \vdash (\lambda x. t \; x) = t \quad \text{ETA} \]

\[ \Gamma \vdash (P : \alpha \rightarrow \text{bool}) \; x \]
\[ \Gamma \vdash P \; (\varepsilon \; P) \quad \text{SELECT} \]

ETA could just as well be regarded as an axiom.

SELECT is equivalent to the Axiom of Choice.
Principles of definition

- **Terms:**
  \[ c = e \]
  is a legitimate definition of \( c \), if
  - \( e \) contains no free variables;
  - all the type variables that occur in \( e \) are in the type of \( c \)

- **Types:**
  \[
  \vdash (P : \tau \rightarrow \text{bool}) t \\
  \vdash \text{abs} \ (\text{rep} \ a) = (a : \tau') \\
  \vdash P \ r = (\text{rep} \ (\text{abs} \ r) = r)
  \]
  where \( \tau \) is an existing type, \( \tau' \) is the new type, \( P \) has no free variables, and \( \text{abs} \) and \( \text{rep} \) are new constants.
One last axiom

When $\forall$, $\exists$, $\neg$, $\land$ and $\Rightarrow$ have all been defined, the last axiom can be added:

$$\vdash \exists (f : \text{ind} \to \text{ind}).$$

$$ (\forall x_1 x_2. (f \; x_1 = f \; x_2) \Rightarrow (x_1 = x_2)) \land$$

$$\exists y. \forall x. \neg (y = f \; x)$$

This states that $\text{ind}$ is infinite (it forms the basis of the definition of $\mathbb{N}$)
More signature for Thm

val REFL : term -> thm (* could be an axiom *)
val MK_COMB : thm -> thm -> thm
val ABS : thm -> thm
val BETA : term -> thm (* can’t be an axiom *)
val ASSUME : term -> thm
val EQ_MP : thm -> thm -> thm
val DED_ANTISYM : thm -> thm -> thm
val INST_TYPE : (hol_type, hol_type) subst -> thm -> thm
val INST : (term, term) subst -> thm -> thm
val ETA : term -> thm (* could be an axiom *)
val SELECT : thm -> thm
More signature for Thm

val new_definition : term -> thm
val new_type_definition : thm -> thm * thm
val new_axiom : term -> thm (* eek *)
Derived rules

- The system is extended by providing derived rules; ML functions which use the kernel’s facilities to implement logical manipulations.

- For example,

\[
\Gamma \vdash x = y \\
\Gamma \vdash f \, x = f \, y \\
\]  

val AP_TERM : term -> thm -> thm
Derived rules

- The system is extended by providing derived rules; ML functions which use the kernel’s facilities to implement logical manipulations.
- For example,

\[
\begin{align*}
\Gamma & \vdash f = f \\
& \vdash x = y \\
\Gamma & \vdash f \cdot x = f \cdot y
\end{align*}
\]

\begin{align*}
\text{val AP_TERM : term -> thm -> thm} \\
\text{fun AP_TERM f th = MK_COMB (REFL f) th}
\end{align*}

Introduction
   History
   High-level description

Build a HOL kernel
   Design philosophy
   Basic types
   Logic
   Implementation
   Theories

Theorem-proving applications
   BDDs and symbolic model-checking
   TCP/IP trace-checking
Implementing the HOL API

- The basic API has been specified; how do we implement it?
- Experimentation in this area is only just beginning
- Harrison’s HOL Light system demonstrates that a “naïve” implementation can do well
Implementing the HOL API

- The basic API has been specified; how do we implement it?
- Experimentation in this area is only just beginning
- Harrison’s HOL Light system demonstrates that a “naïve” implementation can do well
- ... but even there, the implementation of substitution is fine-tuned, and complicated!
Implementing types

- Types are straightforward:
  
  ```haskell
  datatype hol_type = Varty of string
                   | Tyop of string * hol_type list
  ```

- Could almost expose this to the user
  - But: to insist that types are well-formed, must check calls to `mk_type`
    - `mk_type("list", [alpha, beta])` must fail

- Implementation must include a global “symbol table”, linking types to arities
Implementing terms

- Terms are much more complicated than types:
  - They can be large (a CNF propositional formula of 1000s of variables is very large, but its largest type is `bool → bool → bool`)
  - They include bound variables
- Approaches to implementing terms include
  - name-carrying terms
  - use of de Bruijn indices
  - free variable caching
  - explicit substitutions
- All of these have been tried in HOL’s history
Name-carrying terms

The “naïve” approach:

```haskell
datatype term = Var of string * hol_type
               | Const of string * hol_type
               | App of term * term
               | Abs of term * term
```

The first argument to Abs is a term and not a string because

\[
(\lambda x : \text{num}. \ x \land x > 4) \neq (\lambda x : \text{bool}. \ x \land x > 4)
\]

- Conceptually simple
- Efficient construction/destruction:
  - Building App terms requires a type-check
  - Making a Const requires a check with the global symbol table for constants
  - Abs and Var construction is \(O(1)\)
Name-carrying terms—the problems

- Implementing comparison is complicated:

  \[(\lambda x \cdot x (\lambda y \cdot f y x) y) = (\lambda u \cdot u (\lambda w \cdot f w u) v)\]

- Substitution is worse:
  - When performing \((\lambda u \cdot N)[v \mapsto M]\), must check if \(u \in \text{FV}(M)\), and if so do \(N[u \mapsto u']\), with \(u'\) “fresh”
  - Done poorly, easy to create an exponential cost algorithm.
de Bruijn terms

- Core idea: represent bound variables as numbers "pointing" back to binding site. Names for bound variable disappear.

\[(\lambda \backslash x \cdot x \times (\lambda \backslash y \cdot f \; y \; x) \; y) \leadsto (\lambda \lambda. \; 2 \; (\lambda \cdot f \; 1 \; 3) \; 1)\]

- In ML

```ml
datatype term = FVar of string * hol_type
  | BVar of int
  | Const of string * hol_type
  | App of term * term
  | Abs of hol_type * term
```

- Advantages: substitution, matching and free variable calculations are easy.
de Bruijn terms—the problems

- Users can’t cope with \((\lambda. \lambda. 2 \ (\lambda. f \ 1 \ 3) \ 1)\); they want names to look at:
  - Data type declaration for Abs constructor changes to Abs of term \* term
  - Very nice canonicity property disappears
- Construction and destruction of abstractions takes time linear in size of term:
  - \(mk_{\text{abs}}(\ x, \ t)\) must traverse \(t\) looking for occurrences of \(x\), turning them into de Bruijn indices
  - Conversely \(dest_{\text{abs}}\) must undo this
  - Term traversals happen a lot (though abstractions are comparatively rare)
Explicit substitutions

- When asked to calculate $N[v \rightarrow M]$ as part of $\beta$-reduction, it can be efficient to defer the work (like laziness in a language like Haskell)
- HOL4 provides a library for doing efficient “applicative” or “call-by-value” rewriting, written by Bruno Barras
- The CBV code uses lazy-substitution to merge pending substitutions and to avoid doing unnecessary work
- Implemented with an extra constructor:
  
  $$\text{LzSub} : (\text{term} \times \text{int}) \text{ list} \times \text{term} \rightarrow \text{term}$$
Free variable caching

► One of the most frequently called operations in HOL is the free variable calculation:

\[ \text{FV} : \text{term} \rightarrow \text{term set} \]

► A classic time-space tradeoff is to cache the results of calls to free variable calculations (*memoisation*)

► Extend Abs and App constructors with extra arguments. E.g.:

\[ \text{App of term} * \text{term} * \text{term set option ref} \]

► The reference initially points to the NONE value

► After a free variable calculation, it’s updated to point to SOME(s) where s is the result

► Experiments continue...
Introduction
  History
  High-level description

Build a HOL kernel
  Design philosophy
  Basic types
  Logic
  Implementation
  Theories

Theorem-proving applications
  BDDs and symbolic model-checking
  TCP/IP trace-checking
Theories on disk (persistence)

- Users save work to disk as theory files
- Theories can be
  - independently reloaded into interactive sessions (extending the logical context)
  - independently included in custom applications
- Before hol98, theory files were data files, with their own format
- Konrad Slind and Ken Friis Larsen realised that theories looked just like SML modules:
  - they link names to values
- Now, HOL theories (generated by export_theory) are SML source code
Theories as SML modules

- Logical dependencies can now be analysed statically:
  - Before:
    ```sml
    val th = theorem "arithmetic" "ADD_CLAUSES"
    ```
    The theorem function looks up theorem values in a dynamically updated database; static analysis impossible
  - Now:
    ```sml
    val th = arithmeticTheory.ADD_CLAUSES
    ```
    Dependency on arithmeticTheory is clear.

- Moscow ML linker automatically resolves theory references and includes theory object code in custom applications
Introduction
History
High-level description

Build a HOL kernel
Design philosophy
Basic types
Logic
Implementation
Theories

Theorem-proving applications
BDDs and symbolic model-checking
TCP/IP trace-checking
Linking to the Buddy BDD package

- Buddy is an efficient C implementation of BDDs (Binary Decision Diagrams)
- BDDs are at the heart of important hardware theorem-proving techniques:
  - Equivalence checking: determining if two combinational circuits are equivalent on all inputs
  - Symbolic model-checking: checking temporal properties of transition systems
- Buddy is linked to Moscow ML through the Muddy package:
  - BDDs become a type manipulable in ML programs
- Gordon’s $\text{HOLBdd}$ package allows linked BDD and $\text{HOL}$ reasoning.
Using BDDs in HOL

- Use of *tagged oracles*, allows BDD theorems to be treated as HOL theorems
- Standard BDD algorithms can be implemented
Using BDDs in HOL

- Use of *tagged oracles*, allows BDD theorems to be treated as HOL theorems
- Standard BDD algorithms can be implemented
- But more interesting to investigate combination of styles
- When you can do proofs by induction *and* analyse finite systems in the same environment, what is possible?
- Much current research in this area
Verified model-checking in HOL

- BDDs are at the core of the standard model-checking algorithm
- Hasan Amjad implemented model-checking algorithm for propositional $\mu$-calculus on top of HolBdd
- This algorithm is implemented as a derived rule
- The core algorithm is simple enough, but in this framework
  - embeddings of other logics
  - abstraction optimisations
  
  can also be implemented and known to be correct.

- HOL becomes a framework for the development of high-assurance model-checking algorithms
- Efficiency is not necessarily bad either
TCP/IP trace-checking

- [Joint work with Peter Sewell, Keith Wansbrough and others at the University of Cambridge]
- Have developed a detailed specification of the TCP/IP protocol, and the accompanying sockets API
  - all written in HOL
- This is a post hoc specification:
  - if it and current implementations disagree, the spec. is likely wrong
- How to spot if specification is wrong?
- NB: Without a specification, you certainly can’t tell if an implementation is wrong
Specification validation

- Use experimental infrastructure to generate detailed traces of socket/TCP activity
- Test: does the formal model agree that the observed behaviour is possible?
- Instance of
  
  ![Diagram](image)

  - Users have a command-line tool (distributes work over multiple hosts), and do not interact with HOL directly