The HOL theorem-proving system

Michael Norrish

Michael.Norrish@nicta.com.au

National ICT Australia

8 September 2004

Outline

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.
- See http://hol.sourceforge.net for downloads &c.

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"

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

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

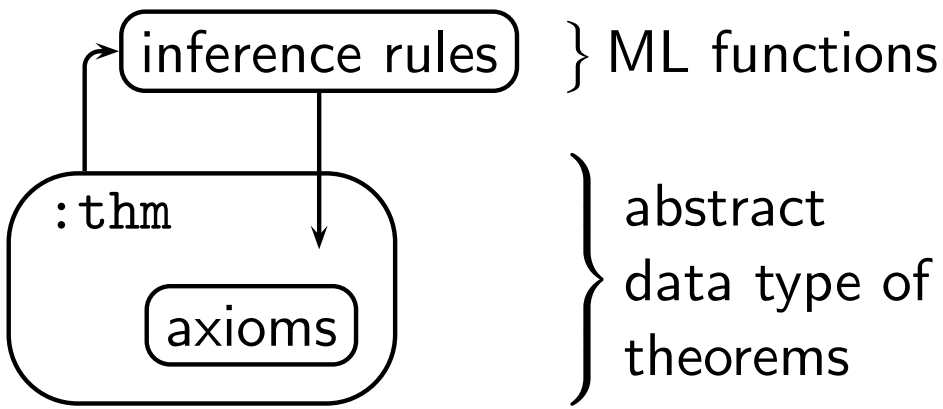
- But naturally higher order
  - Signals are functions from time to bool
  - Devices are relations from signals to signals

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

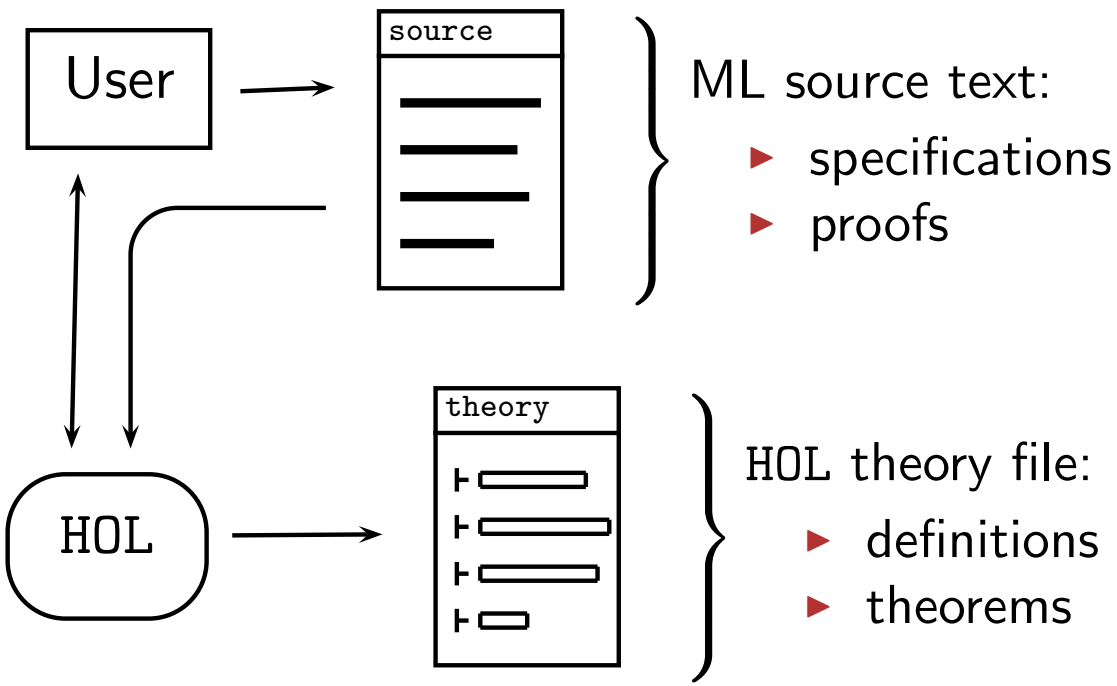
The LCF design philosophy:



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



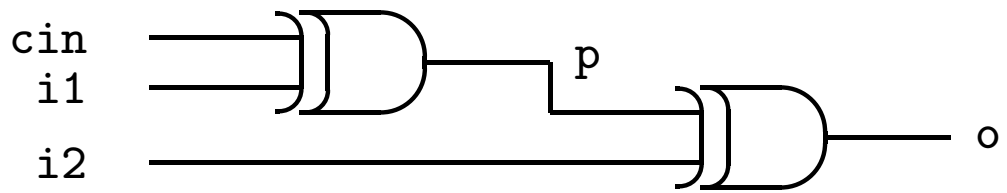
Standard theorem-proving facilities

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:



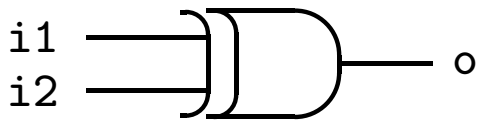
- We wish to verify that

$$\texttt{o} = (\texttt{i1} + \texttt{i2} + \texttt{cin})\ \texttt{MOD}\ 2$$

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

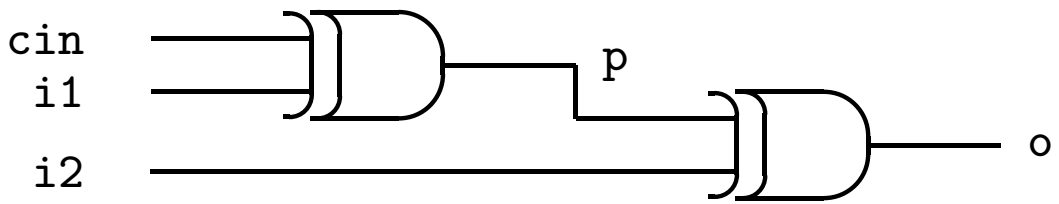
Specify the circuit

- Specification of an XOR gate:



$$\vdash \texttt{Xor}(\texttt{i1}, \texttt{i2}, \texttt{o}) = (\texttt{o} = \neg(\texttt{i1} = \texttt{i2}))$$

- Specification of the adder circuit:



$$\vdash \texttt{Add}(\texttt{cin}, \texttt{i1}, \texttt{i2}, \texttt{o}) = \exists \texttt{p}.\ \texttt{Xor}(\texttt{cin}, \texttt{i1}, \texttt{p}) \wedge \texttt{Xor}(\texttt{p}, \texttt{i2}, \texttt{o})$$

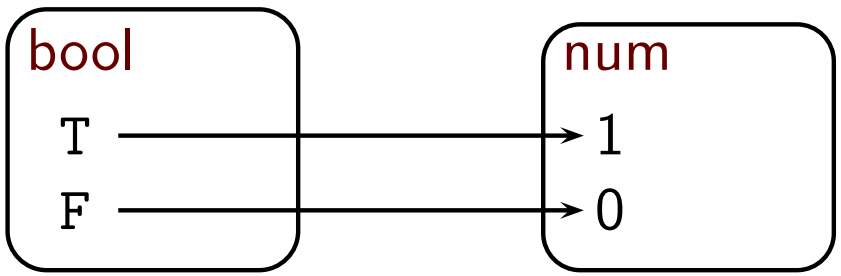
Specify the circuit

- ML source text:

```
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:



$$\vdash \texttt{Bv}(b) = \text{if } b \text{ then } 1 \text{ else } 0$$

- Logical formulation of correctness:

$$\begin{aligned}\vdash\ & \forall \texttt{cin i1 i2 o}.\\ & \texttt{Add(cin, i1, i2, o)} \Rightarrow\\ & \texttt{Bv o} = (\texttt{Bv i1} + \texttt{Bv i2} + \texttt{Bv cin})\ \texttt{MOD 2}\end{aligned}$$

Formulate correctness

- ML source text:

```
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`

```
- 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)
```

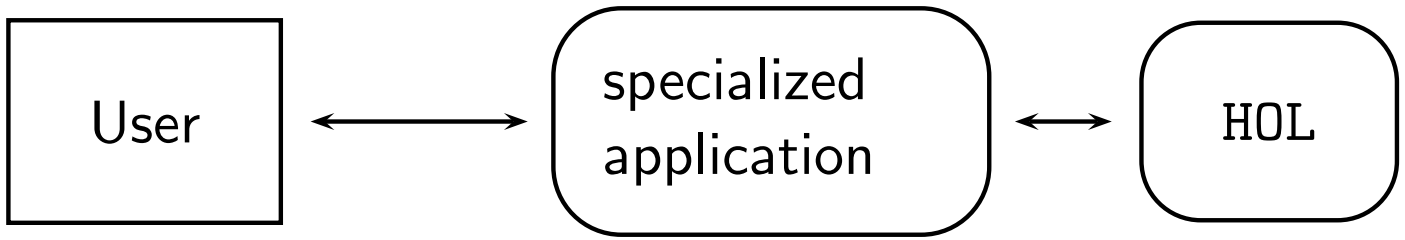
- Could combine two steps into one;
  `RW_TAC arith_ss [Bv,Add,Xor]` solves the goal.

The ML deliverable

```
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]);
```

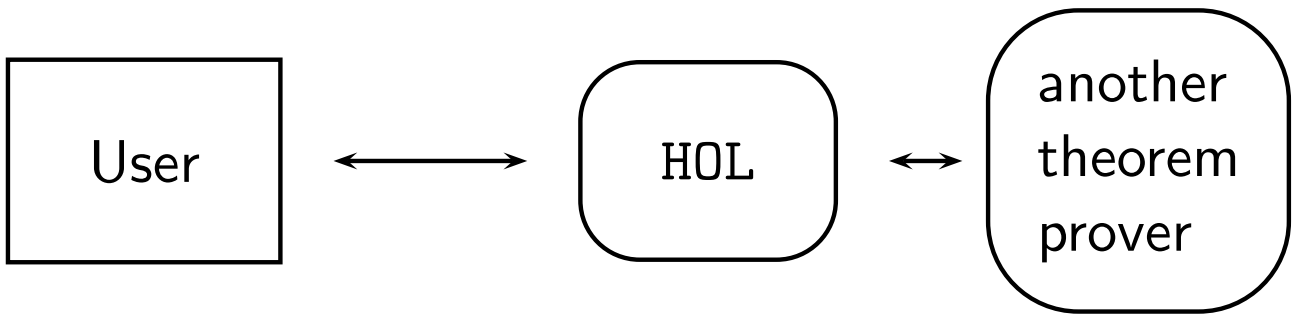
Other modes of use

- HOL as proof engine



Example: TCP protocol trace-checking.

- Hybrid theorem-proving:



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

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)

Types

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

```
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: α, (α)list, and (()num)list
(where list has arity 1, and num has arity 0)

Operations on types

```
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.)

Theorems

```
val dest_thm : thm -> term set * term
```

The only way to *create* theorems is through rules of inference!

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). ((α, β)fun is written $\alpha \to \beta$.)
- Two constants:

$$
\begin{aligned}
= &\quad : \alpha \to \alpha \to \text{bool} \\
\varepsilon &\quad : (\alpha \to \text{bool}) \to \alpha
\end{aligned}
$$

Rules of inference—I

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash f = g \quad \Delta \vdash x = y}{\Gamma \cup \Delta \vdash f\ x = g\ y}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash t = u}{\Gamma \vdash (\lambda x.\ t) = (\lambda x.\ u)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \texttt{BETA}$$

Side-conditions:

- In MK_COMB, $f\ x$ (and $g\ y$) must be valid terms (well-typed)
- In ABS, x must not be free in Γ

Rules of inference—II

$$\frac{}{\{t : \mathsf{bool}\} \vdash t}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash t = u \quad \Delta \vdash (t : \mathsf{bool})}{\Gamma \cup \Delta \vdash u}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash (u : \mathsf{bool}) \quad \Delta \vdash (v : \mathsf{bool})}{(\Gamma \backslash \{v\}) \cup (\Delta \backslash \{u\}) \vdash u = v}\ \texttt{DED\_ANTISYM}$$

$$\frac{\Gamma \vdash t}{\Gamma[\tau_1/\alpha_1 \ldots \tau_n/\alpha_n] \vdash t[\tau_1/\alpha_1 \ldots \tau_n/\alpha_n]}\ \texttt{INST\_TYPE}$$

$$\frac{\Gamma \vdash t}{\Gamma[M_1/v_1 \ldots M_n/v_n] \vdash t[M_1/v_1 \ldots M_n/v_n]}\ \texttt{INST}$$

Rules of inference—III

$$\frac{}{\vdash (\lambda x.\ t\ x) = t}\ \texttt{ETA}$$

$$\frac{\Gamma \vdash (P : \alpha \to \mathsf{bool})\ x}{\Gamma \vdash P\ (\varepsilon\ P)}\ \texttt{SELECT}$$

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:

$$\frac{\vdash (P : \tau \to \mathsf{bool})\ t}{\vdash \mathit{abs}\ (\mathit{rep}\ a) = (a : \tau') \quad \vdash P\ r = (\mathit{rep}\ (\mathit{abs}\ r) = r)}$$

  where τ is an existing type, τ' is the new type, P has no free variables, and abs and rep are new constants.

One last axiom

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

$$\begin{aligned}\vdash \exists(f : \mathsf{ind} \longrightarrow \mathsf{ind}).\\ (\forall x_1\, x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2)) \wedge\\ \exists y.\ \forall x.\ \neg(y = f\ x)\end{aligned}$$

This states that 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,

$$\frac{\Gamma \vdash x = y}{\Gamma \vdash f\ x = f\ y}\ \texttt{AP\_TERM}$$

```
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,

$$\frac{\dfrac{}{\vdash f = f}\ \texttt{REFL} \qquad \Gamma \vdash x = y}{\Gamma \vdash f\ x = f\ y}\ \texttt{MK\_COMB}$$

```
val AP_TERM : term -> thm -> thm
```

```
fun AP_TERM f th = MK_COMB (REFL f) th
```

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:

```
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 $\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$)
  - 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:

```
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 $(\lambda x : \text{num}.\ x \wedge x > 4) \neq (\lambda x : \text{bool}.\ x \wedge x > 4)$

- 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:

$$(\lambda x\, y.\ x\ (\lambda y.\ f\ y\ x)\ y) = (\lambda u\, v.\ u\ (\lambda w.\ f\ w\ u)\ v)$$

- Substitution is worse:
  - When performing $(\lambda u.\ N)[v \mapsto M]$, must check if $u \in \mathtt{FV}(M)$, 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.

$$(\lambda x\, y.\ x\ (\lambda y.\ f\ y\ x)\ y) \rightsquigarrow (\lambda.\lambda.\ 2\ (\lambda.\ f\ 1\ 3)\ 1)$$

- In 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_abs(`x`,` t`)` must traverse t looking for occurrences of x, turning them into de Bruijn indices
  - conversely `dest_abs` must undo this
  - term traversals happen a lot (though abstractions are comparatively rare)

Explicit substitutions

- When asked to calculate $N[v \mapsto M]$ as part of β-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:

```
LzSub : (term * int) list * term -> term
```

Free variable caching

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

```
FV : term -> 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.:

```
App of term * term * 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...

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:

    ```
    val th = theorem "arithmetic" "ADD_CLAUSES"
    ```

    The `theorem` function looks up theorem values in a dynamically updated database; static analysis impossible
  - Now:

    ```
    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

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 `HolBdd` package allows linked BDD and `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 sytems 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 μ-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

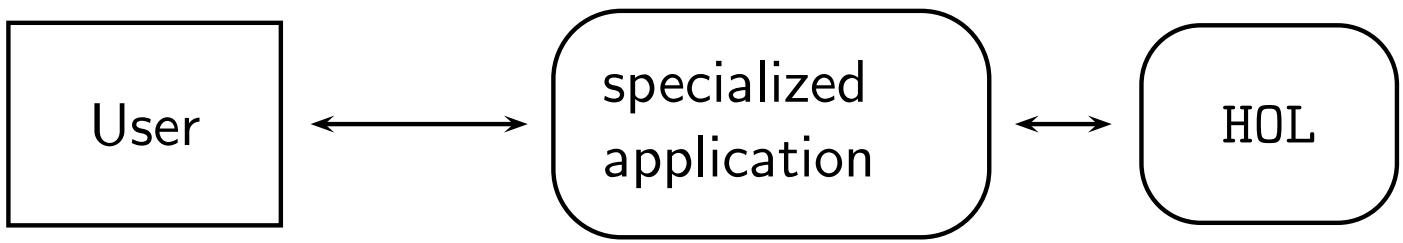
  can also be implemented and known to be correct.
- `HOL` becomes a framework for the development of high-assurance model-checking algorithms
- Efficiency is not necessarily bad either

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 behavour is possible?
- Instance of



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