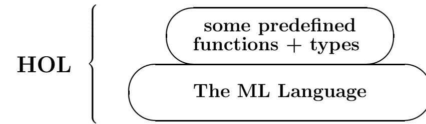
The HOL Mechanization of Higher Order Logic

Higher Order Logic and ML

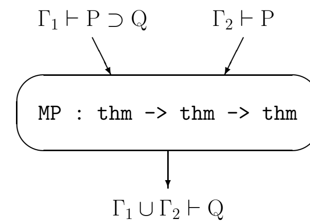
- The role of ML:



- How the logic is embedded in ML:

logic	ML data type
terms	`:term`
types	`:hol_type`
theorems	`:thm`

- Inference rules are ML functions:



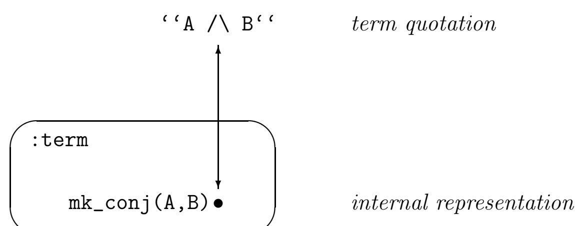
The ML Representation of Terms

- Terms are represented by values of the ML abstract data type `:term`.

- Term literals are written as follows:

```
- ``T /\ F ==> T``;
> val it = ``T /\ F ==> T`` : term

- ``!b. (b=T) \/ (b=F)``;
> val it = ``!b. (b = T) \/ (b = F)`` : term

- (``T``, ``3``);
> val it = (``T``,``3``) : term * term
```

- The quotation parser and prettyprinter:



Ascii Notation for Terms

- The following transliteration is used:

logical notation	HOL notation
$\lambda x.\ \mathrm{M}$	``` ``\x. M`` ```
$\forall x.\ \mathrm{P}$	``` ``!x. P`` ```
$\exists x.\ \mathrm{P}$	``` ``?x. P`` ```
$\exists! x.\ \mathrm{P}$	``` ``?!x. P`` ```
$\varepsilon x.\ \mathrm{P}$	``` ``@x. P`` ```
$\neg x$	``` ``~x`` ```
$\mathrm{P} \wedge \mathrm{Q}$	``` ``P /\ Q`` ```
$\mathrm{P} \vee \mathrm{Q}$	``` ``P \/ Q`` ```
$\mathrm{P} \supset \mathrm{Q}$	``` ``P ==> Q`` ```

- You can arrange to have logical notation on some workstations.

The ML Representation of Types

- Types are represented by values of the ML abstract data type `:hol_type`.
- Type literals in quotations:

```
- ``:bool``;
> val it = ``:bool`` : hol_type

- ``:num -> num``;
> val it = ``:num -> num`` : hol_type
```

- The function `type_of`:

```
- type_of  ``T \/ F``;
> val it = ``:bool`` : hol_type
```

- Displaying type information:

```
- show_types := true;
> val it = () : unit

- ``!b. b = T``;
> val it = ``!(b :bool). b = (T :bool)`` : term

- ``\x. x+1``;
> val it = ``\(x :num). x + (1 :num)`` : term
```


Type Inference and the Typechecker

- The system *infers* logical types:

```
- ``x + 1``;
> val it = ``(x :num) + (1 :num)`` : term
```

The `x` must be `:num`, since `+` is `:num->num->num`.

- Terms must be well-typed:

```
- ``7 + T``;

Type inference failure: unable to infer a type for
the application of

($+ :num -> num -> num) (7 :num)

which has type

num -> num

to

(T :bool)

unification failure message: structural difference
in types
!uncaught exception
!HOL_ERR
```


Warning: ML $\neq$ Logic

- Do not confuse terms with ML programs:

```
- ``7 + 3``;
> val it = ``7 + 3`` : term

- 7 + 3;
> val it = 10 : int

-  ``7`` + 3;
! Toplevel input:
! ``7`` + 3;
!         ^
! Type clash: expression of type
!  int
! cannot have type
!  term

-
```

- Do not confuse ML types and logical types:

Derived Syntactic Forms

- Conditional expressions:

```
- ``(b => 1 | 2)``;
> val it = ``b => 1 | 2`` : term
```

- Lists of logical type `('a)list`:

```
- ``[1;2;3;4]``;
> val it = ``[1; 2; 3; 4]`` : term
```

- Local bindings (`let`-terms):

```
- ``let x=1 and y=2 in x+y``;
> val it = ``let x = 1 and y = 2 in (x + y)`` : term
```

- Paired quantifications:

```
- ``?(q,r). (k = (q * n) + r) /\ r < n``;
> val it = ``?(q,r). (k = q * n + r) /\ r < n`` : term
```

Summary — Terms and Types

- **HOL has the two ML data types:**
  - `:term` — to represent logical terms
  - `:hol_type` — to represent logical types
- **Quotation parser for term and type literals:**
  - Term quotations: ``A ==> B``
  - Type quotations: ``:bool->bool``
- **Parser and pretty-printer support:**

conditionals	``(c => A | B)``
lists	``[1; 2; 3]``
let-terms	``let x = 1 in x+y``
paired quantifiers	``!(x,y). P``
sets	``{1,2,3}``, ``{x | P}``

- **Some built-in ML functions and values:**
  - `type_of : term -> hol_type`
  - `show_types : bool ref`

Syntax Functions

- **HOL provides *syntax functions* for**
  - constructing terms and types
  - taking terms and types apart
  - testing the syntactic classes of terms and types
- **For example:**

```
- mk_comb  ``f:bool -> bool``, ``x:bool``);
> val it = ``f x`` : term

- dest_forall  ``!x. x ==> F``;
> val it = (``x``,``x ==> F``) : term * term

- dest_forall  ``?x. x`` handle e => Raise e;

Exception raised at boolSyntax.dest_forall:
not a "!"

- is_var  ``x:bool``;
> val it = true : bool
```

- **The abstract data types `:term` and `:hol_type` are implemented by two sets of primitive syntax functions.**

Primitive Syntax Functions: Types

- **Syntax of types:**

$$\sigma \;::=\; \alpha \;\mid\; c \;\mid\; (\sigma_1,\ldots,\sigma_n)op$$

- **Syntax functions for types:**

mk_vartype "'a..." = :'a...
mk_type("op",[$\sigma_1,\ldots,\sigma_n$]) = :($\sigma_1,\ldots,\sigma_n$)op

dest_vartype :'a... = "'a..."
dest_type :($\sigma_1,\ldots,\sigma_n$)op = ("op",[$\sigma_1,\ldots,\sigma_n$])

is_vartype :σ **iff** :σ **is a type variable**

- **Examples:**

```
- dest_type (``:bool``);
> val it = ("bool",[]) : string * hol_type list

- mk_type ("prod",[``:bool``, mk_type("ind",[])]);
> val it = ``:bool # ind`` : hol_type

- is_vartype (``:'a``);
> val it = true : bool

- is_vartype (``:bool``);
> val it = false : bool
```


Primitive Syntax Functions: Terms

- **Term constructors:**

mk_abs(v,M) = \\v.M
mk_comb(M,N) = M N
mk_const("c",:σ) = c:σ
mk_var("v",:σ) = v:σ

- **Term destructors:**

dest_abs \\v.M = (v,M)
dest_comb M N = (M,N)
dest_const c:σ = ("c",:σ)
dest_var v:σ = ("v",:σ)

- **Term discriminators:**

is_abs M **iff** M **is an abstraction**
is_comb M **iff** M **is an application**
is_const M **iff** M **is a constant**
is_var M **iff** M **is a variable**

- **Examples:**

```
- dest_abs  ``\A. A ==> B``;
> val it = (``A``,``A ==> B``) : term * term

- mk_var ("x", ``:bool``);
> val it = ``x`` : term
```

Derived Syntax Functions

- There are derived syntax functions for some non-primitive terms:

```
- dest_forall   ``!A. A /\ B``;
> val it = (``A``,``A /\ B``) : term * term

- mk_conj  (``A:bool``, ``B:bool``);
> val it = ``A /\ B`` : term
```

- Naming convention:
  - prefixes: `mk`, `dest`, `is`
  - roots: `pair`, `list`, `cond`
    `eq`, `neq`, `conj`, `disj`, `imp`
    `forall`, `exists`, `select`

- Hence, for example:

```
mk_conj : term * term -> term
is_conj : term -> bool
dest_conj : term -> (term * term)
```


Derived Syntax Functions

- Derived syntax functions are just ordinary ML programs:

```
- fun free_vars t =
      if (is_var t) then [t] else
      if (is_const t) then [] else
      if (is_comb t) then
         let val (l,r) = dest_comb t in
             union (free_vars l) (free_vars r)
         end else
      let val (x,b) = dest_abs t in
          subtract (free_vars b) [x]
      end;
> val free_vars = fn : term -> term list

- free_vars  ``A /\ B /\ C``;
> val it = [``A``,``B``,``C``] : term list

- free_vars  ``!A. B /\ A``;
> val it = [``B``] : term list

- free_vars  ``let l = [x;y;z] in APPEND l [1;w]``;
> val it = [``w``,``x``,``y``,``z``] : term list
```

- As this shows, it is very convenient having such an economical primitive syntax.

Summary — Syntax Functions

- Primitive syntax functions:
  - For types: `mk_type`, `dest_type`, etc.
  - For terms: `mk_var`, `dest_var`, `is_var`, etc.

- Derived syntax functions for terms:

```
mk_   }     { pair, list, cond
dest_ }  +  { eq, neq, conj, disj, imp
is_   }     { forall, exists, select
```

- Other syntax functions:
  - `variant :  term list -> term -> term`
  - `free_in :  term -> term -> bool`
  - `free_vars :  term -> term list`
  - `aconv :  term -> term -> bool`

Sequents in HOL

- A sequent is a pair of the form (Γ, P) where
  - Γ is a set of formulas (assumptions) and
  - P is a formula (conclusion)

- In HOL, sequents are just pairs of ML type:

`term list * term`

(`term list`: assumptions; `term`: conclusion)

- Lists are used to simulate finite sets:

sequent in logic	sequent in HOL
$(\Gamma,\ \mathrm{P})$	`( G , P)`
set of assumptions	list of assumptions

Theorems

- Theorems are represented by values of the ML abstract data type `:thm`.

- How theorems are printed:

```
- BOOL_CASES_AX;
> val it = |- !t. (t = T) \/ (t = F) : thm

- ASSUME  ``A ==> B``;
> val it = . |- A ==> B : thm
```

The dot represents an assumption which has not been printed.

- To get assumptions to print:

```
- show_assums := true;
> val it = () : unit

- ASSUME  ``A ==> B``;
> val it = [A ==> B] |- A ==> B : thm
```


Syntax Functions for Theorems

- Syntax functions for destructing theorems:

```
dest_thm : thm -> term list * term
   concl : thm -> term
     hyp : thm -> term list
```

- Example:

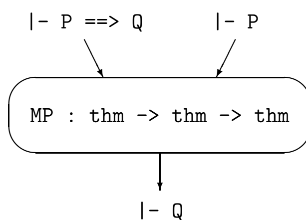
```
- val t1 = MP (ASSUME  ``A ==> B``)
              (ASSUME  ``A:bool``);
> val t1 = [A ==> B, A] |- B : thm

- dest_thm t1;
> val it = ([``A ==> B``,``A``],``B``) : term list * term
```

- There are no literals of type `thm`:
  - you can't just 'type in' arbitrary theorems
  - you have to prove them!

Inference Rules

- An inference rule is represented by an ML function that returns a theorem as a result.

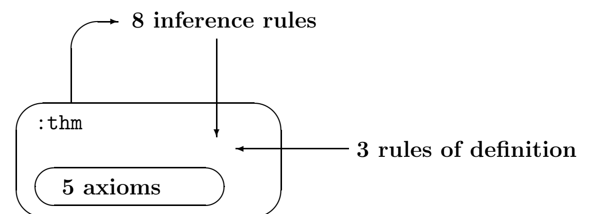
- For example, *modus ponens* in HOL:



- The function returns only objects of type `thm` that logically follow by the inference rule.

Theorems and Proof in HOL

- In the core of HOL we have:



- You can obtain a value of type `thm`:
  - directly — as an axiom.
  - by computation — using the built-in functions that represent the inference rules and rules of definition.

- ML typechecking ensures these are the *only* ways to generate a `thm`:

All theorems must be proved!

Summary — Theorems and Proofs

- **Sequents:**
  - the ML data type `goal` = `term list * term`.
- **Theorems:**
  - the ML data type `thm`.
- **Inference Rules:**
  - ML functions that return values of type `thm`.
- **Proof in HOL:**
  - must be done by ML programs that execute the appropriate sequence of inference rules.

© T F Melham 1993