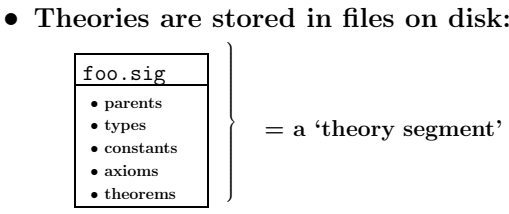


Theories in HOL

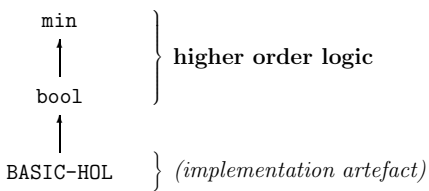
Theories in HOL

- Conceptually, a theory looks this:
 - vocabulary
 - type constants
 - type operators
 - term constants
 - axioms
 - constant definitions
 - constant specifications
 - type definitions
 - arbitrary postulates
 - (saved) theorems
- You always work within some *current theory*:
 - in *draft mode* when creating a theory,
 - in *working mode* otherwise.



Core HOL theories

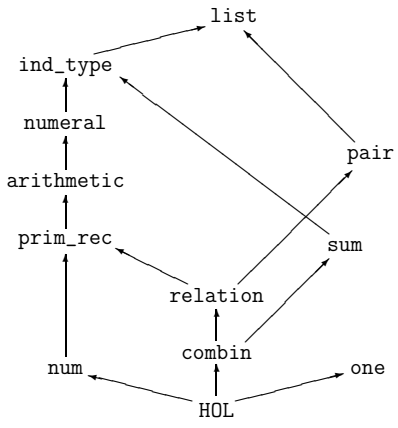
- The core theory hierarchy:



- In principle, only bool needs to contain non-definitional axioms!

Hierarchy of Built-in Theories

- The built-in HOL theories are:



- These are all definitional theories built on top of the core theory bool

Creating Theories

- There are two modes of working:
 - *draft mode*: used when building a new theory by making definitions.
 - *working mode*: used when proving theorems in the current theory but not adding new definitions
- A typical theory-creating source text:

```
new_theory "foo";      (* create the theory foo *)
new_parent "bar";
make_definitions      } system in draft mode
  :
                        }
                        } (* finished definitions *)
prove_theorems        } system in working mode
  :
export_theory();      } (* save theory on disk *)
exit();
```

Adding Axioms

- Functions for extending the vocabulary:

```
new_type : int -> string -> void
new_constant : (string * hol_type) -> unit
new_infix : (string * hol_type * int) -> unit
new_binder : (string * hol_type) -> unit
```
- Function for postulating an axiom:

```
new_axiom : (string * term) -> thm
```
- `new_axiom ("name", Q)`;
 - makes Q into an axiomatic theorem $\vdash Q$
 - stores the axiom under the name "name"
- Warning:

This function can be used to add inconsistent axioms to a theory!

Constant Definitions

- The rule of constant definition:

```
new_definition: (string * term) -> thm
```

 - The form of the term is restricted, as follows.
 - The constant name comes from the form of the term.
- `new_definition("name", 'c = Q')`;
 - makes c a constant defined by $\vdash c = Q$
 - stores the definition $\vdash c = Q$ in the current theory under the name "name"
- Derived definitions:
 - `new_definition("name", 'f x1 ... xn = Q')`;
 - `new_definition("name", 'f(x1, ..., xn) = Q')`;
- Mutually dependent constants
 - `new_specification` defines related constants.
 - The constants may be infix, binding or plain.

Definition Examples

- A constant definition:

```
- new_definition("foo", 'foo x = x+1');
> val it = |- !x. foo x = x + 1 : thm
```
- Defining an infix:

```
- val th1 = TAC_PROOF
  (([], 'f. !x L. $f x L = MEM x L'),
   EXISTS_TAC 'MEM'
   THEN REPEAT STRIP_TAC
   THEN REFL_TAC);
> val th1 = |- ?f. !x L. f x L = MEM x L : thm

- new_specification {consts=[{const_name="In",
                             fixity=Infixr 451}],
                    name = "inDef",
                    sat_thm = th1};
> val it = |- !x L. x In L = MEM x L : thm
```

 - Satisfiability theorem guarantees consistency.
- Defining a binder:
 - In this case fixity is Binder.

Common Errors

- Recursive definition attempted:

```
- new_definition
  ("fact",
   'fact n = ((n=0) => 1 | n * fact(n-1))')
  handle e => Raise e;
Exception raised at Definition.new_definition:
Free variables in rhs of definition: "fact"
! Uncaught exception: ....
```

- Free variables on the right-hand side:

```
- new_definition("bar", 'bar x y = x + y + z')
  handle e => Raise e;
Exception raised at Definition.new_definition:
Free variables in rhs of definition: "z"
! Uncaught exception: ....
```

Loose Specification of Constants

- The constant specification function:

```
- new_specification;
> val it = fn :
  {consts : {const_name : string,
             fixity : fixity} list,
   name : string,
   sat_thm : thm} -> thm
```

- Logically, the rule is:

$$\frac{|- ?x_1 \dots x_n. Q[x_1, \dots, x_n]}{|- Q[c_1, \dots, c_n]}$$

So you can introduce constants whose meaning is 'loosely specified' by a designated property.

- Example of use in HOL:

```
- ethm;
> val it = |- ?x y. x < y : thm

- new_specification {
  consts = [{const_name = "HI", fixity=Closefix},
            {const_name = "LO", fixity=Closefix}],
  name = LOHI",
  sat_thm = ethm};
> val it = |- LO < HI : thm
```

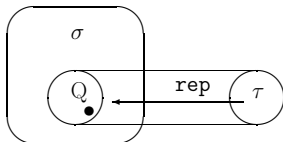
Type Definitions

- The rule of type definition:

```
new_type_definition : (string * thm) -> thm
```

- New types are defined by taking 'subsets' of existing types:

```
|- TYPE_DEFINITION (Q:σ->bool) (rep:τ->σ)
```



- Example in the system:

```
- eth;
> val it = |- ?x. (\b. b) x : thm

- new_type_definition("unit", eth);
> val it = |- ?rep. TYPE_DEFINITION (\b. b) rep : thm

- 'x:unit';
> val it = 'x' : term
```

Primitive Recursion

- Definition of functions by primitive recursion is a *derived* principle of definition in HOL.

- A primitive recursive definition has the form:

```
f(0) = E0 (base case)
f(SUC n) = E1[f(n), n] (recursive case)
```

- Example – definition of addition:

```
|- plus 0 = \m.m
|- plus (SUC n) = \m. SUC(plus n m)
```

- In more familiar notation:

```
|- 0 + m = m
|- (SUC n) + m = SUC(n + m)
```

Primitive Recursion in HOL

- The derived rule is:

```
new_recursive_definition:
  {def: term, name: string, rec_axiom: thm} -> thm
```

- the string is the name under which the definition will be stored in the current theory,
- the theorem is the appropriate primitive recursion theorem,
- the term gives the desired definition, and
- the result is the definition as a theorem.

- Example:

```
- prim_recTheory.num_Axiom;
> val it = |- !e f. ?fn. (fn 0 = e) /\
              !n. fn (SUC n) = f n (fn n) : thm

- new_recursive_definition{
  name = "PLUS_DEF",
  def = '(PLUS 0 m = m) /\
         (PLUS (SUC n) m = SUC(PLUS n m))',
  rec_axiom = prim_recTheory.num_Axiom};
> val it = |- (!m. PLUS 0 m = m) /\
              (!n m. PLUS (SUC n) m = SUC(PLUS n m))
              : thm
```

Other Recursive Definitions

- There is a recursion theorem for lists:

```
- listTheory.list_Axiom;
> val it =
  |- !f0 f1. ?fn. (fn [] = f0) /\
                  !a0 a1. fn (a0::a1)
                        = f1 a0 a1 (fn a1) : thm
```

- The MAP function can be defined by:

```
- new_recursive_definition{
  name = "map_DEF",
  def = '(map (f:'a->'b) ([]) = []) /\
         (map f (h :: t) = (f h)::(map f t))',
  rec_axiom = listTheory.list_Axiom};
> val it =
  |- (!f. map f [] = []) /\
     (!f h t. map f (h :: t) = (f h) :: (map f t))
     : thm
```

- For any inductively defined type

- Prove a corresponding recursion theorem
- Similarly define recursive functions
- Trees (of various sorts) provide many more such examples

Adding and Retrieving Theorems

- The function for saving theorems:

```
save_thm: (string * thm) -> thm
```

- Example in the system:

```
- thm;
> val it = |- !m n. m + n = n + m

- save_thm("ADD_COMM", thm);
> val it = |- !m n. m + n = n + m : thm
```

- Fetching a theorem from current theory:

```
theorem: string -> thm
```

```
- theorem("ADD_COMM");
> val it = |- !m n. m + n = n + m : thm
```

- Fetching basis theorems of current theory:

```
axiom: string -> thm
definition: string -> thm
```

Inspecting a Theory

- The function for inspecting a theory

```
print_theory: string -> unit
```

- Example:

```
- print_theory "one";
Theory: one

Parents:
  bool

Type constants:
  one 0

Term constants:
  one :one

Definitions:
  one_TY_DEF |- ?rep. TYPE_DEFINITION (\b. b) rep
  one_DEF   |- () = @x. T

Theorems:
  one_axiom |- !f g. f = g
  one      |- !v. v = ()
  one_Axiom |- !e. ?!fn. fn () = e
```

Inspecting a Theory by Component

- Various functions get the various parts:

- parents: string -> string list

```
- parents "prim_rec";  
> val it = ["relation", "num"] : string list
```

- types: string -> (string * int) list

```
- parents "pair";  
> val it = [("prod", 2)] : (string * int) list
```

- constants: string -> term list

```
- constants "sum";  
> val it = [‘‘ABS_sum‘‘, ‘‘REP_sum‘‘, ‘‘ISR‘‘, ‘‘ISL‘‘,  
           ‘‘INR‘‘, ‘‘INL‘‘, ‘‘case‘‘, ‘‘OUTR‘‘, ‘‘OUTL‘‘,  
           ‘‘IS_SUM_REP‘‘] : term list
```

- axioms: string -> (string * thm) list
- definitions: string -> (string thm) list
- theorems: string -> (string * thm) list

Summary

- Creating a theory:

- new_theory

- Adding types, constants and axioms:

- new_axiom, new_type, new_constant

- Rules of definition:

- new_definition, new_specification,
new_type_definition

- Saving theorems:

- save_thm

- Theory-accessing functions:

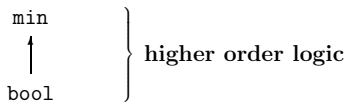
- axiom, definition, theorem
- parents, types, constants, axioms, definitions,
theorems

- Inspecting a theory:

- print_theory

The Primitive HOL Theories

- The core HOL theories:



- The theory min contains:

- the type constants: ind, bool
- the type operator: -> (i.e. 'fun')
- constants: =, ==>, @

- The theory bool contains:

- the type constant: bool
- constants: T, F, ~, /\, \/, !, ?, ?!
- definitions of the basic logical constants
- the five axioms of the logic

plus a few 'non-logical' extras:

- definitions of constants for syntactic abbreviations

Standard (Built-in) Theories

- Standard theories built on top of bool and ind:

pair	defines products
num	defines non-negative integers
prim_rec	theory of primitive recursion
arithmetic	theory of arithmetic
list	defines the type of lists
combin	defines the combinators S, K, I, o
sum	defines disjoint sum
one	defines a type with only one value

- These theories are all purely definitional.

- These theories are already loaded but not opened.

- Each theory is an ML module (with .sig file)
- The names of the modules are pairTheory, numTheory, prim-recTheory, etc.
- Theorems are accessed by giving the module name and the theorem name.
e.g. arithmeticTheory.MULT_SYM

Summary

- HOL implementation of theories:
 - Functions for creating theories.
 - Functions for accessing theory components.
- The core HOL theories `min` and `bool`.
- The hierarchy of built-in derived theories.