Interactive Theorem Proving in HOL4

Course 07: Core Types

Dr Chun TIAN
chun.tian@anu.edu.au

12 September 2024

Australian National University

Acknowledgement of Country

We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present.

More information about Acknowledgement of Country can be found here and here

More operators in `boolTheory`

If-then-else

In HOL, the term "`if` t `then` t_1 `else` t_2" abbreviates the application "`COND` $t\ t_1\ t_2$", where `COND` is defined by:

[COND_DEF]
$\vdash$ COND =
$(\lambda t\ t_1\ t_2.\ \varepsilon x.\ ((t \iff \mathtt{T}) \Rightarrow x = t_1) \land ((t \iff \mathtt{F}) \Rightarrow x = t_2))$

LET binding

The concrete syntax "`let v = M in N`" is translated by the parser to the term "`LET (\v.N) M`", where `LET` is defined by:

[LET_DEF]
$\vdash$ LET = $(\lambda f\ x.\ f\ x)$

Combinators (combinTheory)

Combinatory Logic (**CL**) is a formal system equivalent to (untyped) λ-calculus. Most (but not all, e.g. Ω) combinators are available in HOL.

Most important combinators in practice

$\vdash \forall f\ g.\ f \circ g = (\lambda x.\ f\ (g\ x))$	[o_DEF]
$\vdash \mathtt{I} = \mathtt{S}\ \mathtt{K}\ \mathtt{K}$	[I_DEF]
$\vdash \mathtt{K} = (\lambda x\ y.\ x)$	[K_DEF]
$\vdash \mathtt{flip} = (\lambda f\ x\ y.\ f\ y\ x)$	[C_DEF]

Important theorems about combinators

$\vdash \forall f\ g\ x.\ (f \circ g)\ x = f\ (g\ x)$	[o_THM]
$\vdash \forall f\ g\ h.\ f \circ g \circ h = (f \circ g) \circ h$	[o_ASSOC]
$\vdash \forall x.\ \mathtt{I}\ x = x$	[I_THM]
$\vdash \forall x\ y.\ \mathtt{K}\ x\ y = x$	[K_THM]
$\vdash \forall f\ x\ y.\ \mathtt{flip}\ f\ x\ y = f\ y\ x$	[C_THM]

Basic Types in HOL

Core Types

- `one` and `itself`;
- `option`;
- pairs (`pairTheory`);
- sum (`sumTheory`);

Numbers, sequences and collections

- `num` (natural numbers);
- `list`;
- `set` (predicate-based sets);
- ...

The `unit` and `itself` Types

The `unit` type (in `oneTheory`)

The theory `one` defines the type `unit` (aka `one`) which contains only one element "()".

$\vdash$ `∀(`v` :unit). `v` = ()` [one]

The `one` type is usually used for instantiating type variables in other complex types, to disable certain features in the type (For example, the type of graphs with labelled edges may have a dedicated type variable for the labels.)

The `itself` type (part of `boolTheory`)

For every type α, α `itself` is a type containing just one value.

$\vdash$ `∀(`i` :`α` itself). `i` = (:`α`)` [ITSELF_UNIQUE]

The `itself` type is usually used as a type parameter of functions (if the whole purpose of the parameter is to provide the type information), e.g. `univ(:'a)` is the set of all values of type α.

The option Type

The theory `option` can be used to define a new type, which contains one more element than the old type. The constructors of this type are

```
NONE : 'a option
SOME : 'a -> 'a option
```

`SOME x` and `NONE` are distinct terms:

$\vdash \forall (x : \alpha).\ (\texttt{NONE} : \alpha\ \texttt{option}) \neq \texttt{SOME}\ x$ [NOT_NONE_SOME]

The function `THE` maps terms of option type back to the original type (`THE NONE` is unspecified):

$\vdash \forall (x : \alpha).\ \texttt{THE}\ (\texttt{SOME}\ x) = x$ [THE_DEF]

Standard ML has also an `Option` structure

```
datatype 'a option = NONE | SOME of 'a
val valOf : 'a option -> 'a
```

Pairs (pairTheory)

Values of the Cartesian product type "$:\sigma_1$ # σ_2" are ordered pairs whose first component has type σ_1 and whose second component has type σ_2. Pairs are constructed with an infixed comma symbol:

```
$, : 'a -> 'b -> 'a # 'b
```

"Pairs" of more than two values are supported (The comma symbol associates to the *right*):

```
> ``a = (1,(2,3))``;
val it = ``a = (1,2,3)``: term
> ``b = ((1,2),3)``;
val it = ``b = ((1,2),3)``: term
```

In the above terms, the type of a is `num # num # num` (= `num # (num # num)`), while the type of b is `(num # num) # num`.

In Standard ML, the comma operator is not associative at all (thus only support two values).

Pairs (pairTheory), continued

Accessing values in pairs

$\vdash \forall (x :\alpha)\ (y :\beta).\ \texttt{FST}\ (x,y) = x$ [FST]

$\vdash \forall (x :\alpha)\ (y :\beta).\ \texttt{SND}\ (x,y) = y$ [SND]

Important pair theorems

$\vdash (\texttt{FST}\ x, \texttt{SND}\ x) = x$ [PAIR]

$\vdash (x,y) = (a,b) \iff x = a \land y = b$ [PAIR_EQ]

$\vdash (\forall p.\ P\ p) \iff \forall p_{-1}\ p_{-2}.\ P\ (p_{-1},p_{-2})$ [FORALL_PROD]

$\vdash (\exists p.\ P\ p) \iff \exists p_{-1}\ p_{-2}.\ P\ (p_{-1},p_{-2})$ [EXISTS_PROD]

Tactics

`Cases_on ‘t‘`, where `t` is a pair, will rewrite all occurrences of `t` with the pair `(q,r)`.

Natural numbers (num)

The natural numbers are developed in a series of theories: num, prim_rec, arithmetic, ...

numTheory

The constants 0 and SUC (the successor function) are defined and Peano's axioms pre-proved:

$\vdash \mathtt{SUC}\ n \neq 0$ [numTheory.NOT_SUC]

$\vdash \mathtt{SUC}\ m = \mathtt{SUC}\ n \Rightarrow m = n$ [numTheory.INV_SUC]

$\vdash P\ 0 \land (\forall n.\ P\ n \Rightarrow P\ (\mathtt{SUC}\ n)) \Rightarrow \forall n.\ P\ n$ [numTheory.INDUCTION]

prim_recTheory

The ordering of natural numbers ($m < n$) is defined here. There's also the PRE constant.

$\vdash m < n \iff \exists P.\ (\forall n.\ P\ (\mathtt{SUC}\ n) \Rightarrow P\ n) \land P\ m \land \neg P\ n$
[prim_recTheory.LESS_DEF]

$\vdash \mathtt{PRE}\ m = \textbf{if}\ m = 0\ \textbf{then}\ 0\ \textbf{else}\ \varepsilon n.\ m = \mathtt{SUC}\ n$ [prim_recTheory.PRE_DEF]

Natural numbers (`num`), continued

`arithmeticTheory`

Other natural numbers are constructed by `0` and `SUC`, e.g.

$\vdash$ `1 = SUC 0` [arithmeticTheory.ONE]
$\vdash$ `2 = SUC 1` [arithmeticTheory.TWO]

More such theorems can be generated by `numLib.num_CONV`:

```
> numLib.num_CONV ``5 :num``;
val it = |- 5 = SUC 4: thm
```

`numeralTheory`

In HOL4, the literal numeral term, say 5, is actually an abbreviation of another term `NUMERAL(BIT1(BIT2 ZERO))`, which evaluates to 5. The uses of `ZERO`, `BIT1` and `BIT2` are similar with binary representations of integers in computer.

Arithmetics

Some operators on num

$\vdash (\forall n.\ 0 + n = n) \wedge \forall m\ n.\ \mathtt{SUC}\ m + n = \mathtt{SUC}\ (m + n)$ [ADD]

$\vdash (\forall m.\ 0 - m = 0) \wedge$
$\forall m\ n.\ \mathtt{SUC}\ m - n = \textbf{if}\ m < n\ \textbf{then}\ 0\ \textbf{else}\ \mathtt{SUC}\ (m - n)$ [SUB]

$\vdash (\forall n.\ 0 \times n = 0) \wedge \forall m\ n.\ \mathtt{SUC}\ m \times n = m \times n + n$ [MULT]

$\vdash (\forall m.\ m\ \mathtt{**}\ 0 = 1) \wedge \forall m\ n.\ m\ \mathtt{**}\ \mathtt{SUC}\ n = m \times m\ \mathtt{**}\ n$ [EXP]

$\vdash m \le n \iff m < n \vee m = n$ [LESS_OR_EQ]

$\vdash \mathtt{MAX}\ m\ n = \textbf{if}\ m < n\ \textbf{then}\ n\ \textbf{else}\ m$ [MAX_DEF]

$\vdash \mathtt{MIN}\ m\ n = \textbf{if}\ m < n\ \textbf{then}\ m\ \textbf{else}\ n$ [MIN_DEF]

$\vdash r < n \Rightarrow \forall q.\ (q \times n + r)\ \mathtt{DIV}\ n = q$ [DIV_MULT]

Searching for arithmetic theorems

```
> DB.match ["num", "prim_rec", "arithmetic"] ``a + b < a + c:num``;
val it =
   [(("arithmetic", "LT_ADD_LCANCEL"),
     (|- !m n p. p + m < p + n <=> m < n, Thm))]: public_data list
```

Lists

Lists (of type `:'a list`) are finite sequence of elements from the same type α. The theory of lists are mainly developed in `listTheory` and `rich_listTheory`. The primitive constructors are `NIL` (`[]`) and `CONS` (`::`):

```
NIL  : 'a list
CONS : 'a -> 'a list -> 'a list
```

Literal lists in HOL are abbreviations (pretty printing) of calls of `NIL` and `CONS` on list elements:

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

Operations on Lists

There is a *huge* number of operations and predicates on lists. Users should constantly check the entries `list` and `rich_list` of *HOL Reference Page*.

A selected list of constants for lists

ALL_DISTINCT, APPEND, DROP, EL, EVERY, EXISTS, FOLDL, FOLDR, FRONT, GENLIST, HD, LAST, LENGTH, MAP, MEM, REVERSE, SNOC, TAKE, TL, ZIP, isPREFIX.

$\vdash \texttt{HD}\ (h\texttt{::}t) = h$ [HD]

$\vdash \texttt{TL}\ (h\texttt{::}t) = t$ [TL]

$\vdash (\forall l.\ \texttt{EL}\ 0\ l = \texttt{HD}\ l) \wedge \forall l\ n.\ \texttt{EL}\ (\texttt{SUC}\ n)\ l = \texttt{EL}\ n\ (\texttt{TL}\ l)$ [EL]

Inductions on Lists

$\vdash P\ [\,] \wedge (\forall t.\ P\ t \Rightarrow \forall h.\ P\ (h\texttt{::}t)) \Rightarrow \forall l.\ P\ l$ [list_INDUCT]

$\vdash P\ [\,] \wedge (\forall l.\ P\ l \Rightarrow \forall x.\ P\ (\texttt{SNOC}\ x\ l)) \Rightarrow \forall l.\ P\ l$ [SNOC_INDUCT]

Predicate-based sets

A set P of elements of type α is essentially a function of type $\alpha \to$ **bool**: $x \in P$ iff $P(x)$ is true:

$\vdash \forall (P : \alpha \to \texttt{bool})\ (x : \alpha).\ x \in P \iff P\ x$ [SPECIFICATION]

Most set operations have the usual textbook definitions:

$\vdash s = t \iff \forall x.\ x \in s \iff x \in t$ [EXTENSION]
$\vdash s \subseteq t \iff \forall x.\ x \in s \Rightarrow x \in t$ [SUBSET_DEF]
$\vdash s \cap t = \{x \mid x \in s \land x \in t\}$ [INTER_DEF]
$\vdash s \cup t = \{x \mid x \in s \lor x \in t\}$ [UNION_DEF]
$\vdash s\ \texttt{DIFF}\ t = \{x \mid x \in s \land x \notin t\}$ [DIFF_DEF]
$\vdash \texttt{POW}\ set = \{s \mid s \subseteq set\}$ [POW_DEF]
$\vdash \emptyset = (\lambda x.\ \texttt{F})$ [EMPTY_DEF]
$\vdash x\ \texttt{INSERT}\ s = \{y \mid y = x \lor y \in s\}$ [INSERT_DEF]

Literal sets with explicit members are constructed by empty set and the insert operation.