

This document is available under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license:
<http://creativecommons.org/licenses/by-sa/4.0/>

This document is based on material from the “Interactive Theorem Proving Course” by Thomas Tuerk
(<https://www.thomas-tuerk.de>):
<https://github.com/thtuerk/ITP-course>

This document includes additions by:

- ▶ Pablo Buiras (<https://people.kth.se/~buiras/>)
- ▶ Karl Palmkog (<https://setoid.com>)

Interactive Theorem Proving and Program Verification

Lecture 4

Pablo Buiras and Karl Palmskog



Academic Year 2019/20, Period 3–4

Based on slides by Thomas Tuerk

Part X

Basic Definitions



- there are **conservative definition principles** for types and constants
- conservative means that all theorems that can be proved in extended theory can also be proved in the original one
- however, such extensions make the theory more comfortable
- definitions introduce **no new inconsistencies**
- the HOL community has a very strong tradition of a purely definitional approach

- **axioms** are a different approach
- they allow postulating arbitrary properties, i. e. extending the logic with arbitrary theorems
- this approach might introduce new inconsistencies
- in HOL4, axioms are rarely needed
- using definitions is considered more elegant and proper (“honest toil”)
- it is hard to keep track of axioms
- use custom axioms only if you really know what you are doing

- **oracles** are families of axioms
- however, they are used differently than axioms
- they are used to enable usage of external tools and knowledge
- you might want to use an external automated prover
- this external tool acts as an oracle
 - ▶ it provides answers
 - ▶ it does not explain or justify these answers
- you don't know whether this external tool might be buggy
- all theorems proved via it are tagged with a special oracle-tag
- tags are propagated
- this allows keeping track of everything depending on the correctness of this tool

- Common oracle-tags
 - ▶ `DISK_THM` — theorem was written to disk and read again
 - ▶ `HolSatLib` — proved by MiniSat
 - ▶ `HolSmtLib` — proved by external SMT solver
 - ▶ `fast_proof` — proof was skipped to compile a theory rapidly
 - ▶ `cheat` — we cheated :-)
- `cheating` via, e. g. , the `cheat` tactic means skipping proofs
- it can be helpful during proof development
 - ▶ test whether some lemma allows you to finish the proof
 - ▶ skip lengthy but boring cases and focus on critical parts first
 - ▶ experiment with exact form of invariants
 - ▶ ...
- cheats should be removed at a reasonable pace
- HOL4 warns about cheats and skipped proofs

Pitfalls of Definitional Approach



- definitions can't introduce new inconsistencies
- they force you to state all assumed properties at one location
- however, you still need to be careful:
 - ▶ Is your definition really expressing what you had in mind?
 - ▶ Does your formalisation correspond to the real world artefact?
 - ▶ How can you convince others that this is the case?
- we will discuss methods to deal with this later in this course:
 - ▶ formal sanity
 - ▶ reduction theorems
 - ▶ conformance testing
 - ▶ code review
 - ▶ comments, good names, clear coding style
 - ▶ ...
- this is complex and needs a lot of effort in general

- HOL4 allows to introduce new constants with certain properties, provided the existence of such constants has been shown

Specification of EVEN and ODD

```
> EVEN_ODD_EXISTS
```

```
val it = |- ?even odd. even 0 /\ ~odd 0 /\ (!n. even (SUC n) <=> odd n) /\  
          (!n. odd (SUC n) <=> even n)
```

```
> val EO_SPEC = new_specification ("EO_SPEC", ["EVEN", "ODD"], EVEN_ODD_EXISTS);  
val EO_SPEC = |- EVEN 0 /\ ~ODD 0 /\ (!n. EVEN (SUC n) <=> ODD n) /\  
              (!n. ODD (SUC n) <=> EVEN n)
```

- `new_specification` is a convenience wrapper
 - ▶ it uses existential quantification instead of Hilbert's choice
 - ▶ deals with pair syntax
 - ▶ stores resulting definitions in theory
- `new_specification` captures the underlying principle nicely

- special case: new constant defined by equality

Specification with Equality

```
> double_EXISTS
val it =
|- ?double. (!n. double n = (n + n))

> val double_def = new_specification ("double_def", ["double"], double_EXISTS);
val double_def =
  |- !n. double n = n + n
```

- there is a specialised methods for such simple definitions

Non Recursive Definitions

```
> val DOUBLE_DEF = new_definition ("DOUBLE_DEF", ‘‘DOUBLE n = n + n‘‘)
val DOUBLE_DEF =
  |- !n. DOUBLE n = n + n
```

- all variables occurring on right-hand-side (RHS) need to be arguments
 - ▶ e.g. `new_definition (... , 'F n = n + m')` fails
 - ▶ `m` is free on RHS
- all type variables occurring on RHS need to occur on LHS
 - ▶ e.g. `new_definition ("IS_FIN_TY",
 'IS_FIN_TY = FINITE (UNIV : 'a set)')` fails
 - ▶ `IS_FIN_TY` would lead to inconsistency
 - ▶ `|- FINITE (UNIV : bool set)`
 - ▶ `|- ~FINITE (UNIV : num set)`
 - ▶ `T <=> FINITE (UNIV:bool set) <=>
IS_FIN_TY <=>
FINITE (UNIV:num set) <=> F`
 - ▶ therefore, such definitions can't be allowed

- function specification do not need to define the function precisely
- multiple different functions satisfying one spec are possible
- functions resulting from such specs are called **underspecified**
- underspecified functions are still total, one just lacks knowledge
- one common application: modelling **partial functions**
 - ▶ functions like e. g. **HD** and **TL** are total
 - ▶ they are defined for empty lists
 - ▶ however, it is not specified, which value they have for empty lists
 - ▶ only known: **HD [] = HD []** and **TL [] = TL []**

```
val MY_HD_EXISTS = prove (“?hd. !x xs. (hd (x::xs) = x)”, ...);  
val MY_HD_SPEC =  
  new_specification (“MY_HD_SPEC”, [“MY_HD”], MY_HD_EXISTS)
```

- HOL4 allows introducing non-empty subtypes of existing types
- a predicate $P : \text{ty} \rightarrow \text{bool}$ describes a subset of an existing type ty
- ty may contain type variables
- only **non-empty** types are allowed
- therefore a non-emptiness proof ex-thm of form $?e. P e$ is needed
- `new_type_definition (op-name, ex-thm)` then introduces a new type `op-name` specified by P

Primitive Type Definitions - Example 1



- lets try to define a type `dlist` of lists containing no duplicates
- predicate `ALL_DISTINCT : 'a list -> bool` is used to define it
- easy to prove theorem `dlist_exists: |- ?1. ALL_DISTINCT 1`
- `val dlist_TY_DEF = new_type_definitions("dlist",
dlist_exists)` defines a new type `'a dlist` and returns a theorem

```
|- ?(rep : 'a dlist -> 'a list).  
    TYPE_DEFINITION ALL_DISTINCT rep
```

- `rep` is a function taking a `'a dlist` to the list representing it
 - ▶ `rep` is injective
 - ▶ a list satisfies `ALL_DISTINCT` iff there is a corresponding `dlist`

- `define_new_type_bijections` can be used to define bijections between old and new type

```
> define_new_type_bijections {name="dlist_tybij", ABS="abs_dlist",  
    REP="rep_dlist", tyax=dlist_TY_DEF}
```

```
val it =  
  |- (!a. abs_dlist (rep_dlist a) = a) /\  
      (!r. ALL_DISTINCT r <=> (rep_dlist (abs_dlist r) = r))
```

- other useful theorems can be automatically proved by
 - ▶ `prove_abs_fn_one_one`
 - ▶ `prove_abs_fn_onto`
 - ▶ `prove_rep_fn_one_one`
 - ▶ `prove_rep_fn_onto`

- primitive definition principles are easily explained
- they lead to conservative extensions
- however, they are cumbersome to use
- LCF approach allows implementing more convenient definition tools
 - ▶ **Datatype** package
 - ▶ **TFL** (Total Functional Language) package
 - ▶ **IndDef** (Inductive Definition) package
 - ▶ **quotientLib** Quotient Types Library
 - ▶ ...

- the **Datatype** package allows to define datatypes conveniently
- the **TFL** package allows to define (mutually recursive) functions
- the **Eval** conversion allows evaluating those definitions
- this gives many HOL4 developments the feeling of a functional program
- there is really a close connection between functional programming and definitions in HOL4
 - ▶ functional programming design principles apply
 - ▶ **Eval** is a great way to test quickly, whether your definitions are working as intended
- more details on these connections later in the context of the CakeML language and compiler

Functional Programming Example



```
> Datatype 'mylist = E | L 'a mylist'  
val it = (): unit
```

```
> Define '(mylen E = 0) /\ (mylen (L x xs) = SUC (mylen xs))'  
Definition has been stored under "mylen_def"
```

```
val it =  
  |- (mylen E = 0) /\ !x xs. mylen (L x xs) = SUC (mylen xs):  
  thm
```

```
> EVAL ''mylen (L 2 (L 3 (L 1 E)))''  
val it =  
  |- mylen (L 2 (L 3 (L 1 E))) = 3:  
  thm
```

- the `Datatype` package allows to define SML style datatypes easily
- there is support for
 - ▶ algebraic datatypes
 - ▶ record types
 - ▶ mutually recursive types
 - ▶ ...
- many constants are automatically introduced
 - ▶ constructors
 - ▶ case-split constant
 - ▶ size function
 - ▶ field-update and accessor functions for records
 - ▶ ...
- many theorems are derived and stored in current theory
 - ▶ injectivity and distinctness of constructors
 - ▶ dichotomy and structural induction theorems
 - ▶ rewrites for case-split, size and record update functions
 - ▶ ...

Tree Datatype in SML

```
datatype ('a,'b) btree =   Leaf of 'a
                          | Node of ('a,'b) btree * 'b * ('a,'b) btree
```

Tree Datatype in HOL4

```
Datatype 'btree =   Leaf 'a
                   | Node btree 'b btree'
```

Tree Datatype in HOL4 — Deprecated Syntax

```
Hol_datatype 'btree =   Leaf of 'a
                       | Node of btree => 'b => btree'
```

btree_distinct

```
|- !a2 a1 a0 a. Leaf a <> Node a0 a1 a2
```

btree_11

```
|- (!a a'. (Leaf a = Leaf a') <=> (a = a')) /\  
  (!a0 a1 a2 a0' a1' a2'.  
   (Node a0 a1 a2 = Node a0' a1' a2') <=>  
   (a0 = a0') /\ (a1 = a1') /\ (a2 = a2'))
```

btree_nchotomy

```
|- !bb. (?a. bb = Leaf a) \\/ (?b b1 b0. bb = Node b b1 b0)
```

btree_induction

```
|- !P. (!a. P (Leaf a)) /\  
  (!b b0. P b /\ P b0 ==> !b1. P (Node b b1 b0)) ==>  
  !b. P b
```

btree_size_def

```
|- (!f f1 a. btree_size f f1 (Leaf a) = 1 + f a) /\
  (!f f1 a0 a1 a2.
    btree_size f f1 (Node a0 a1 a2) =
      1 + (btree_size f f1 a0 + (f1 a1 + btree_size f f1 a2)))
```

btree_case_def

```
|- (!a f f1. btree_CASE (Leaf a) f f1 = f a) /\
  (!a0 a1 a2 f f1. btree_CASE (Node a0 a1 a2) f f1 = f1 a0 a1 a2)
```

btree_case_cong

```
|- !M M' f f1.
  (M = M') /\ (!a. (M' = Leaf a) ==> (f a = f' a)) /\
  (!a0 a1 a2.
    (M' = Node a0 a1 a2) ==> (f1 a0 a1 a2 = f1' a0 a1 a2)) ==>
  (btree_CASE M f f1 = btree_CASE M' f' f1')
```

Enumeration type in SML

```
datatype my_enum = E1 | E2 | E3
```

Enumeration type in HOL4

```
Datatype 'my_enum = E1 | E2 | E3'
```

my_enum_nchotomy

```
|- !P. P E1 /\ P E2 /\ P E3 ==> !a. P a
```

my_enum_distinct

```
|- E1 <> E2 /\ E1 <> E3 /\ E2 <> E3
```

my_enum2num_thm

```
|- (my_enum2num E1 = 0) /\ (my_enum2num E2 = 1) /\ (my_enum2num E3 = 2)
```

my_enum2num_num2my_enum

```
|- !r. r < 3 <=> (my_enum2num (num2my_enum r) = r)
```


Record type in SML

```
type rgb = { r : int, g : int, b : int }
```

Record type in HOL4

```
Datatype 'rgb = <| r : num; g : num; b : num |>'
```

rgb_component_equality

```
|- !r1 r2. (r1 = r2) <=>  
    (r1.r = r2.r) /\ (r1.g = r2.g) /\ (r1.b = r2.b)
```

rgb_nchotomy

```
|- !rr. ?n n0 n1. rr = rgb n n0 n1
```

rgb_r_fupd

```
|- !f n n0 n1. rgb n n0 n1 with r updated_by f = rgb (f n) n0 n1
```

rgb_updates_eq_literal

```
|- !r n1 n0 n.  
    r with <|r := n1; g := n0; b := n|> = <|r := n1; g := n0; b := n|>
```

Datatype Package - Example IV

- nested record types are not allowed
- however, mutual recursive types can mitigate this restriction

Filesystem Datatype in SML

```
datatype file = Text of string
              | Dir of {owner : string ,
                       files : (string * file) list}
```

Not Supported Nested Record Type Example in HOL4

```
Datatype 'file = Text string
              | Dir <| owner : string ;
                  files : (string # file) list |>'
```

Filesystem Datatype - Mutual Recursion in HOL4

```
Datatype 'file = Text string
              | Dir directory
;
directory = <| owner : string ;
            files : (string # file) list |>'
```

- there is no support for co-algebraic (“infinite”) types
- the Datatype package could be extended to do so
- other systems like Isabelle/HOL provide high-level methods for defining such types

Co-algebraic Type Example in SML — Lazy Lists

```
datatype 'a lazylist = Nil
                  | Cons of ('a * (unit -> 'a lazylist))
```

- Datatype package allows to define many useful datatypes
- however, there are many limitations
 - ▶ some types cannot be defined in HOL4, e. g. , empty types
 - ▶ some types are not supported, e. g. co-algebraic types
 - ▶ there are bugs (currently, e. g. , some trouble with certain mutually recursive definitions)
- biggest restrictions in practice (in my opinion and my line of work)
 - ▶ no support for co-algebraic datatypes
 - ▶ no nested record datatypes
- depending on datatype, different sets of useful lemmas are derived
- most important ones are added to `TypeBase`
 - ▶ tools like `Induct_on`, `Cases_on` use them
 - ▶ there is support for pattern matching

Total Functional Language (TFL) package



- TFL package implements support for terminating functional definitions
- `Define` defines functions from high-level descriptions
- there is support for pattern matching
- look and feel is like function definitions in SML
- based on **well-founded recursion** principle
- `Define` is the most common way for definitions in HOL4

Simple Definitions

```
> val DOUBLE_def = Define 'DOUBLE n = n + n'
val DOUBLE_def =
  |- !n. DOUBLE n = n + n:
  thm

> val MY_LENGTH_def = Define '(MY_LENGTH [] = 0) /\
  (MY_LENGTH (x::xs) = SUC (MY_LENGTH xs))'
val MY_LENGTH_def =
  |- (MY_LENGTH [] = 0) /\ !x xs. MY_LENGTH (x::xs) = SUC (MY_LENGTH xs):
  thm

> val MY_APPEND_def = Define '(MY_APPEND [] ys = ys) /\
  (MY_APPEND (x::xs) ys = x :: (MY_APPEND xs ys))'
val MY_APPEND_def =
  |- (!ys. MY_APPEND [] ys = ys) /\
  (!x xs ys. MY_APPEND (x::xs) ys = x::MY_APPEND xs ys):
  thm
```

- `Define` feels like a function definition in HOL4
- it can be used to define “terminating” recursive functions
- `Define` is implemented by a large, non-trivial piece of SML code
- it uses many heuristics
- outcome of `Define` is sometimes hard to predict
- the input descriptions are only hints
 - ▶ the produced function and the definitional theorem might be different
 - ▶ in simple examples, quantifiers added
 - ▶ pattern compilation takes place
 - ▶ earlier “conjuncts” have precedence

Define - More Examples



```
> val MY_HD_def = Define 'MY_HD (x :: xs) = x'
val MY_HD_def = |- !x xs. MY_HD (x::xs) = x : thm

> val IS_SORTED_def = Define '
  (IS_SORTED (x1 :: x2 :: xs) = ((x1 < x2) /\ (IS_SORTED (x2::xs)))) /\
  (IS_SORTED _ = T)'
val IS_SORTED_def =
  |- (!xs x2 x1. IS_SORTED (x1::x2::xs) <=> x1 < x2 /\ IS_SORTED (x2::xs)) /\
    (IS_SORTED [] <=> T) /\ (!v. IS_SORTED [v] <=> T)

> val EVEN_def = Define '(EVEN 0 = T) /\ (ODD 0 = F) /\
  (EVEN (SUC n) = ODD n) /\ (ODD (SUC n) = EVEN n)'
val EVEN_def =
  |- (EVEN 0 <=> T) /\ (ODD 0 <=> F) /\ (!n. EVEN (SUC n) <=> ODD n) /\
    (!n. ODD (SUC n) <=> EVEN n) : thm

> val ZIP_def = Define '(ZIP (x::xs) (y::ys) = (x,y)::(ZIP xs ys)) /\
  (ZIP _ _ = [])'
val ZIP_def =
  |- (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys) /\
    (!v1. ZIP [] v1 = []) /\ (!v4 v3. ZIP (v3::v4) [] = []) : thm
```

Primitive Definitions

- `Define` introduces (if needed) the function using `WFREC`
- intended definition derived as a theorem
- the theorems are stored in current theory
- usually, one never needs to look at it

Examples

```

val IS_SORTED_primitive_def =
|- IS_SORTED =
  WFREC (@R. WF R /\ !x1 xs x2. R (x2::xs) (x1::x2::xs))
    (\IS_SORTED a.
      case a of
        [] => I T
      | [x1] => I T
      | x1::x2::xs => I (x1 < x2 /\ IS_SORTED (x2::xs)))

|- !R M. WF R ==> !x. WFREC R M x = M (RESTRICT (WFREC R M) R x) x
|- !f R x. RESTRICT f R x = (\y. if R y x then f y else ARB)
  
```

- `Define` automatically defines induction theorems
- these theorems are stored in current theory with suffix `ind`
- use `DB.fetch "-" "something_ind"` to retrieve them
- these induction theorems are useful to reason about corresponding recursive functions

Example

```
val IS_SORTED_ind = |- !P.  
  ((!x1 x2 xs. P (x2::xs) ==> P (x1::x2::xs)) /\  
   P [] /\  
   (!v. P [v])) ==>  
  !v. P v
```

- there are many induction theorems in HOL4
- Example: complete induction principle

$$\vdash \neg P. (\neg n. (\neg m. m < n \implies P m) \implies P n) \implies \neg n. P n$$

- ▶ besides datatype definitions, recursive relation definitions also give rise to induction theorems
- ▶ many are manually defined, e. g. , proved from other induction theorems

- Examples

$$\vdash \neg P. P [] \wedge (\neg l. P l \implies \neg x. P (\text{SNOC } x l)) \implies \neg l. P l$$
$$\vdash \neg P. P \text{ FEMPTY} \wedge$$
$$(\neg f. P f \implies \neg x y. x \text{ NOTIN } \text{FDOM } f \implies P (f |+ (x,y))) \implies \neg f. P f$$
$$\vdash \neg P. P \{\} \wedge$$
$$(\neg s. \text{FINITE } s \wedge P s \implies \neg e. e \text{ NOTIN } s \implies P (e \text{ INSERT } s)) \implies$$
$$\neg s. \text{FINITE } s \implies P s$$
$$\vdash \neg R P. (\neg x y. R x y \implies P x y) \wedge (\neg x y z. P x y \wedge P y z \implies P x z) \implies$$
$$\neg u v. R^+ u v \implies P u v$$

- **Define** might fail for various reasons to define a function
 - ▶ such a function cannot be defined in HOL4
 - ▶ such a function can be defined, but not via the methods used by TFL
 - ▶ TFL can define such a function, but its heuristics are too weak and user guidance is required
 - ▶ there is a bug in HOL4
- **termination** is an important concept for **Define**
- it is easy to misunderstand termination in the context of HOL4
- we need to understand what is meant by termination

- in SML it is natural to talk about termination of functions
- in the HOL4 logic there is no concept of execution
- thus, there is no concept of termination in HOL4

3 characterisations of a function $f : \text{num} \rightarrow \text{num}$

- ▶ $\vdash \!n. f\ n = 0$
- ▶ $\vdash (f\ 0 = 0) \wedge \!n. (f\ (\text{SUC}\ n) = f\ n)$
- ▶ $\vdash (f\ 0 = 0) \wedge \!n. (f\ n = f\ (\text{SUC}\ n))$

Is f terminating? All 3 theorems are equivalent.

- it is useful to think in terms of termination
- the TFL package implements heuristics to define functions that would terminate in SML
- the TFL package uses well-founded recursion
- the required well-founded relation corresponds to a termination proof
- therefore, it is very natural to think of `Define` searching a termination proof
- important: this is the idea behind this function definition package, not a property of HOL

HOL is not limited to "terminating" functions

Termination in HOL4 III

- one can define “non-terminating” functions in HOL4
- however, one cannot do so (easily) with `Define`

Definition of WHILE in HOL4

```
|- !P g x. WHILE P g x = if P x then WHILE P g (g x) else x
```

Execution Order

There is no “execution order”. One can easily define a complicated constant function:

```
(myk : num -> num) (n:num) = (let x = myk (n+1) in 0)
```

Unsound Definitions

A function `f : num -> num` with the following property cannot be defined in HOL4 unless HOL4 has an inconsistency:

```
!n. f n = ((f n) + 1)
```

Such a function would allow proving `0 = 1`.

Alternative Syntax for Definitions and Theorems



- HOL4 syntax for definitions and theorems is verbose and requires explicit quoting
- Kananaskis-13 introduced a (shallow) less verbose layer on top

Classic Definition Syntax

```
val num2boolList_def = Define `
  (num2boolList 0 = []) /\
  (num2boolList 1 = []) /\
  (num2boolList n = (EVEN n) :: num2boolList (n DIV 2))`;
```

Alternative Definition Syntax

```
Definition num2boolList_def:
  (num2boolList 0 = []) /\
  (num2boolList 1 = []) /\
  (num2boolList n = (EVEN n) :: num2boolList (n DIV 2))
End
```

Classic Theorem Syntax

```
val num2boolList_REWRS = store_thm ("num2boolList_REWRS",
  '(num2boolList 0 = []) /\ (num2boolList 1 = []) /\
  (!n. 2 <= n ==>
    ((num2boolList n = (EVEN n) :: num2boolList (n DIV 2))))',
  REPEAT STRIP_TAC >| [
    METIS_TAC[num2boolList_def],
    METIS_TAC[num2boolList_def],
    'n <> 0 /\ n <> 1' by DECIDE_TAC >>
    METIS_TAC[num2boolList_def]
  ]);
```

Alternative Theorem Syntax

Theorem num2boolList_REWRS:

```
(num2boolList 0 = []) /\ (num2boolList 1 = []) /\  
(!n. 2 <= n ==>  
  ((num2boolList n = (EVEN n) :: num2boolList (n DIV 2))))
```

Proof

```
REPEAT STRIP_TAC >| [  
  METIS_TAC[num2boolList_def],  
  METIS_TAC[num2boolList_def],  
  'n <> 0 /\ n <> 1' by DECIDE_TAC >>  
  METIS_TAC[num2boolList_def]
```

]

QED