

Interactive Theorem Proving (ITP) Course

Parts X - XII

Thomas Tuerk (tuerk@kth.se)



Academic Year 2016/17, Period 4

version cf844ad of Mon May 15 09:02:18 2017

Part X

Basic Definitions



Definitional Extensions



- 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 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

Axiomatic Extensions



- **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 HOL axioms are very rarely needed
- using definitions is often considered more elegant
- it is hard to keep track of axioms
- use 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
 - ▶ HoISatLib — proved by MiniSat
 - ▶ HoISmtLib — 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 lemmata allow you finishing the proof
 - ▶ skip lengthy but boring cases and focus on critical parts first
 - ▶ experiment with exact form of invariants
 - ▶ ...
- cheats should be removed reasonable quickly
- HOL warns about cheats and skipped proofs

127 / 195

128 / 195

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
 - ▶ conformance testing
 - ▶ code review
 - ▶ comments, good names, clear coding style
 - ▶ ...
- this is highly complex and needs a lot of effort in general



Specifications



- HOL 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

129 / 195

130 / 195

Definitions



- 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 non-recursive definitions

Non Recursive Definitions

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

131 / 195

Underspecified Functions



- 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, is 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)
```

133 / 195

## Restrictions for Definitions



- 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

132 / 195

## Primitive Type Definitions



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

134 / 195

## 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`: `|- ?l. ALL_DISTINCT l`
- `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`

135 / 195

## Primitive Definition Principles Summary



- 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 (Terminating Functional Programs) package
  - ▶ IndDef (Inductive Definition) package
  - ▶ quotientLib Quotient Types Library
  - ▶ ...

137 / 195

## Primitive Type Definitions - Example 2



- `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`

136 / 195

## Functional Programming



- 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 HOL developments the feeling of a functional program
- there is really a close connection between functional programming and definitions in HOL
  - ▶ functional programming design principles apply
  - ▶ `EVAL` is a great way to test quickly, whether your definitions are working as intended

138 / 195

## 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
```



139 / 195

## Datatype Package - Example I

### Tree Datatype in SML

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

### Tree Datatype in HOL

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

### Tree Datatype in HOL — Depreciated Syntax

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



141 / 195

## Datatype Package

- 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
  - nchotomy and structural induction theorems
  - rewrites for case-split, size and record update functions
  - ...



140 / 195

## Datatype Package - Example I - Derived Theorems 1

### 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
```



142 / 195

## Datatype Package - Example I - Derived Theorems 2



### 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)))
```

### bbtree\_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')
```

143 / 195

## Datatype Package - Example II - Derived Theorems



### 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)
```

145 / 195

## Datatype Package - Example II



### Enumeration type in SML

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

### Enumeration type in HOL

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

144 / 195

## Datatype Package - Example III



### Record type in SML

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

### Record type in HOL

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

146 / 195

## Datatype Package - Example III - Derived Theorems



### 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|>
```

147 / 195

## Datatype Package - No support for Co-Algebraic Types



- there is no support for co-algebraic 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))
```

149 / 195

## 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 HOL

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

### Filesystem Datatype - Mutual Recursion in HOL

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

148 / 195

## Datatype Package - Discussion



- Datatype package allows to define many useful datatypes
- however, there are many limitations
  - ▶ some types cannot be defined in HOL, 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 lemmata are derived
- most important ones are added to TypeBase
  - ▶ tools like Induct\_on, Cases\_on use them
  - ▶ there is support for pattern matching

150 / 195



- 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 HOL

151 / 195

## Well-Founded Recursion

- a well-founded relation  $R$  can be used to define recursive functions
- this recursion principle is called WFREC in HOL
- idea of WFREC
  - ▶ if arguments get smaller according to  $R$ , perform recursive call
  - ▶ otherwise abort and return ARB
- WFREC always defines a function
- if all recursive calls indeed decrease according to  $R$ , the original recursive equations can be derived from the WFREC representation
- TFL uses this internally
- however, this is well-hidden from the user



153 / 195

## Well-Founded Relations



- a relation  $R : 'a \rightarrow 'a \rightarrow \text{bool}$  is called **well-founded**, iff there are no infinite descending chains
 
$$\text{wellfounded } R = \sim?f. !n. R (f (SUC n)) (f n)$$
- Example:  $\$< : \text{num} \rightarrow \text{num} \rightarrow \text{bool}$  is well-founded
- if arguments of recursive calls are smaller according to well-founded relation, the recursion terminates
- this is the essence of termination proofs

152 / 195

## Define - Initial Examples



### 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
```

154 / 195



## Define discussion



- Define feels like a function definition in HOL
- 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 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

## 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)
```

155 / 195



## 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
```

156 / 195

## Induction Theorems



- 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
```

157 / 195

158 / 195

## Define failing



- Define might fail for various reasons to define a function
  - ▶ such a function cannot be defined in HOL
  - ▶ 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 :-)
- **termination** is an important concept for Define
- it is easy to misunderstand termination in the context of HOL
- we need to understand what is meant by termination

## Termination in HOL



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

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.

159 / 195

160 / 195

## Termination in HOL II



- 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 HOL III



- one can define "non-terminating" functions in HOL
- however, one cannot do so (easily) with Define

### Definition of WHILE in HOL

$\vdash !P\ g\ x. \text{WHILE } P\ g\ x = \text{if } P\ x \text{ then } \text{WHILE } P\ g\ (g\ x) \text{ else } x$

### Execution Order

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

$(\text{myk} : \text{num} \rightarrow \text{num})\ (n:\text{num}) = (\text{let } x = \text{myk}\ (n+1) \text{ in } 0)$

### Unsound Definitions

A function  $f : \text{num} \rightarrow \text{num}$  with the following property cannot be defined in HOL unless HOL has an inconsistency:

$\vdash !n. f\ n = ((f\ n) + 1)$

Such a function would allow to prove  $0 = 1$ .

161 / 195

162 / 195



- TFL uses various heuristics to find a well-founded relation
- however, these heuristics may not be strong enough
- in such cases the user can provide a well-founded relation manually
- the most common well-founded relations are **measures**
- measures map values to natural numbers and use the less relation  
 $\text{!- !(f:'a -> num) x y. measure f x y <=> (f x < f y)}$
- all measures are well-founded:  $\text{!- !f. WF (measure f)}$
- moreover, existing well-founded relations can be combined
  - ▶ lexicographic order LEX
  - ▶ list lexicographic order LLEX
  - ▶ ...

## Manual Termination Proof Example 1

```
> val qsort_defn = Hol_defn "qsort" `
 (qsort ord [] = []) /\
 (qsort ord (x::rst) =
 (qsort ord (FILTER ($~ o ord x) rst)) ++
 [x] ++
 (qsort ord (FILTER (ord x) rst)))`

val qsort_defn = HOL function definition (recursive)

Equation(s) :
[...] |- qsort ord [] = []
[...] |- qsort ord (x::rst) =
 qsort ord (FILTER ($~ o ord x) rst) ++ [x] ++
 qsort ord (FILTER (ord x) rst)

Induction : ...

Termination conditions :
0. !rst x ord. R (ord, FILTER (ord x) rst) (ord, x::rst)
1. !rst x ord. R (ord, FILTER (($~ o ord) x) rst) (ord, x::rst)
2. WF R
```

163 / 195



- if Define fails to find a termination proof, Hol\_defn can be used
- Hol\_defn defers termination proofs
- it derives termination conditions and sets up the function definitions
- all results are packaged as a value of type defn
- after calling Hol\_defn the defined function(s) can be used
- however, the intended definition theorem has not been derived yet
- to derive it, one needs to
  - ▶ provide a well-founded relation
  - ▶ show that termination conditions respect that relation
- Defn.tprove and Defn.tgoal are intended for this
- proofs usually start by providing relation via tactic WF\_REL\_TAC

164 / 195

## Manual Termination Proof Example 2

```
> Defn.tgoal qsort_defn

Initial goal:

?R.
 WF R /\
 (!rst x ord. R (ord, FILTER (ord x) rst) (ord, x::rst)) /\
 (!rst x ord. R (ord, FILTER ($~ o ord x) rst) (ord, x::rst))

> e (WF_REL_TAC 'measure (\(., l). LENGTH l)')

1 subgoal :

(!rst x ord. LENGTH (FILTER (ord x) rst) < LENGTH (x::rst)) /\
(!rst x ord. LENGTH (FILTER (\x'. ~ord x x') rst) < LENGTH (x::rst))

> ...
```

165 / 195

166 / 195

## Manual Termination Proof Example 3



```
> val (qsort_def, qsort_ind) =
 Defn.tprove (qsort_defn,
 WF_REL_TAC 'measure (\(., l). LENGTH l)' >> ...)

val qsort_def =
|- (qsort ord [] = []) /\
 (qsort ord (x::rst) =
 qsort ord (FILTER ($~ o ord x) rst) ++ [x] ++
 qsort ord (FILTER (ord x) rst))

val qsort_ind =
|- !P. (!ord. P ord []) /\
 (!ord x rst.
 P ord (FILTER (ord x) rst) /\
 P ord (FILTER ($~ o ord x) rst) ==>
 P ord (x::rst)) ==>
 !v v1. P v v1
```

## Part XI

### Good Definitions



167 / 195

## Importance of Good Definitions



- using *good* definitions is very important
  - ▶ good definitions are vital for **clarity**
  - ▶ **proofs** depend a lot on the form of definitions
- unluckily, it is hard to state what a good definition is
- even harder to come up with good definitions
- let's look at it a bit closer anyhow

169 / 195

## Importance of Good Definitions — Clarity I



- HOL guarantees that theorems do indeed hold
- However, does the theorem mean what you think it does?
- you can separate your development in
  - ▶ main theorems you care for
  - ▶ auxiliary stuff used to derive your main theorems
- it is essential to understand your main theorems

170 / 195

## Importance of Good Definitions — Clarity II



### Guarded by HOL

- proofs checked
- internal, technical definitions
- technical lemmata
- proof tools

### Manual review needed for

- meaning of main theorems
- meaning of definitions used by main theorems
- meaning of types used by main theorems

## Importance of Good Definitions — Clarity III



- it is essential to understand your main theorems
  - ▶ you need to understand all the definitions directly used
  - ▶ you need to understand the indirectly used ones as well
  - ▶ you need to convince others that you express the intended statement
  - ▶ therefore, it is vital to **use very simple, clear definitions**
- defining concepts is often the main development task
- checking resulting model against real artifact is vital
  - ▶ testing via e.g. EVAL
  - ▶ formal sanity
  - ▶ conformance testing
- wrong models are main source of error when using HOL
- proofs, auxiliary lemmata and auxiliary definitions
  - ▶ can be as technical and complicated as you like
  - ▶ correctness is guaranteed by HOL
  - ▶ reviewers don't need to care

171 / 195

## Importance of Good Definitions — Proofs



- good definitions can shorten proofs significantly
- they improve maintainability
- they can improve automation drastically
- unluckily for proofs definitions often need to be technical
- this contradicts clarity aims

173 / 195

## How to come up with good definitions



- unluckily, it is hard to state what a good definition is
- it is even harder to come up with them
  - ▶ there are often many competing interests
  - ▶ a lot of experience and detailed tool knowledge is needed
  - ▶ much depends on personal style and taste
- general advice: use more than one definition
  - ▶ in HOL you can derive equivalent definitions as theorems
  - ▶ define a concept as clearly and easily as possible
  - ▶ derive equivalent definitions for various purposes
    - ★ one very close to your favourite textbook
    - ★ one nice for certain types of proofs
    - ★ another one good for evaluation
    - ★ ...
- lessons from functional programming apply

174 / 195

## Good Definitions in Functional Programming



### Objectives

- clarity (readability, maintainability)
- performance (runtime speed, memory usage, ...)

### General Advice

- use the powerful type-system
- use many small function definitions
- encode invariants in types and function signatures

175 / 195

## Good Definitions – no number encodings



- many programmers familiar with C encode everything as a number
- enumeration types are very cheap in SML and HOL
- use them instead

### Example Enumeration Types

In C the result of an order comparison is an integer with 3 equivalence classes: 0, negative and positive integers. In SML and HOL, it is better to use a variant type.

```
val _ = Datatype 'ordering = LESS | EQUAL | GREATER';

val compare_def = Define '
 (compare LESS lt eq gt = lt)
/\ (compare EQUAL lt eq gt = eq)
/\ (compare GREATER lt eq gt = gt)';

val list_compare_def = Define '
 (list_compare cmp [] [] = EQUAL) /\ (list_compare cmp [] l2 = LESS)
/\ (list_compare cmp l1 [] = GREATER)
/\ (list_compare cmp (x::l1) (y::l2) = compare (cmp (x:'a) y)
 (* x<y *) LESS
 (* x=y *) (list_compare cmp l1 l2)
 (* x>y *) GREATER)';
```

176 / 195

## Good Definitions — Isomorphic Types



- the type-checker is your friend
  - ▶ it helps you find errors
  - ▶ code becomes more robust
  - ▶ using good types is a great way of writing self-documenting code
- therefore, use many types
- even use types isomorphic to existing ones

### Virtual and Physical Memory Addresses

Virtual and physical addresses might in a development both be numbers. It is still nice to use separate types to avoid mixing them up.

```
val _ = Datatype 'vaddr = VAddr num';
val _ = Datatype 'paddr = PAddr num';

val virt_to_phys_addr_def = Define '
 virt_to_phys_addr (VAddr a) = PAddr(translation of a);
```

177 / 195

## Good Definitions — Record Types I



- often people use tuples where records would be more appropriate
- using large tuples quickly becomes awkward
  - ▶ it is easy to mix up order of tuple entries
    - ★ often types coincide, so type-checker does not help
  - ▶ no good error messages for tuples
    - ★ hard to decipher type mismatch messages for long product types
    - ★ hard to figure out which entry is missing at which position
    - ★ non-local error messages
    - ★ variable in last entry can hide missing entries
- records sometimes require slightly more proof effort
- however, records have many benefits

178 / 195

## Good Definitions — Record Types II



- using records
  - ▶ introduces field names
  - ▶ provides automatically defined accessor and update functions
  - ▶ leads to better type-checking error messages
- records improve readability
  - ▶ accessors and update functions lead to shorter code
  - ▶ field names act as documentation
- records improve maintainability
  - ▶ improved error messages
  - ▶ much easier to add extra fields

179 / 195

## Good Definitions — Encoding Invariants II



### Network Connections (Example by Yaron Minsky from Jane Street) II

The following definition of `connection_info` makes the invariants explicit:

```
type connected = { last_ping : (time * int) option,
 session_id : string };
type disconnected = { when_disconnected : time };
type connecting = { when_initiated : time };

datatype connection_state =
 Connected of connected
| Disconnected of disconnected
| Connecting of connecting;

type connection_info = {
 state : connection_state,
 server : inet_address
}
```

181 / 195

## Good Definitions — Encoding Invariants



- try to encode as many invariants as possible in the types
- this allows the type-checker to ensure them for you
- you don't have to check them manually any more
- your code becomes more robust and clearer

### Network Connections (Example by Yaron Minsky from Jane Street)

Consider the following datatype for network connections. It has many implicit invariants.

```
datatype connection_state = Connected | Disconnected | Connecting;

type connection_info = {
 state : connection_state,
 server : inet_address,
 last_ping_time : time option,
 last_ping_id : int option,
 session_id : string option,
 when_initiated : time option,
 when_disconnected : time option
}
```

180 / 195

## Good Definitions in HOL



### Objectives

- clarity (readability)
- good for proofs
- performance (good for automation, easily evaluable, ...)

### General Advice

- same advice as for functional programming applies
- use even smaller definitions
  - ▶ introduce auxiliary definitions for important function parts
  - ▶ use extra definitions for important constants
  - ▶ ...
- tiny definitions
  - ▶ allow keeping proof state small by unfolding only needed ones
  - ▶ allow many small lemmata
  - ▶ improve maintainability

182 / 195

## Good Definitions in HOL II



### Technical Issues

- write definition such that they work well with HOL's tools
- this requires you to know HOL well
- a lot of experience is required
- general advice
  - ▶ avoid explicit case-expressions
  - ▶ prefer curried functions

### Example

```
val ZIP_GOOD_def = Define '(ZIP (x::xs) (y::ys) = (x,y)::(ZIP xs ys)) /\
 (ZIP _ _ = [])'

val ZIP_BAD1_def = Define 'ZIP xs ys = case (xs, ys) of
 (x::xs, y::ys) => (x,y)::(ZIP xs ys)
 | (_, _) => []'

val ZIP_BAD2_def = Define '(ZIP (x::xs, y::ys) = (x,y)::(ZIP (xs, ys))) /\
 (ZIP _ = [])'
```

183 / 195

## Good Definitions in HOL III



### Multiple Equivalent Definitions

- satisfy competing requirements by having multiple equivalent definitions
- derive them as theorems
- initial definition should be as clear as possible
  - ▶ clarity allows simpler reviews
  - ▶ simplicity reduces the likelihood of errors

### Example - ALL\_DISTINCT

```
|- (ALL_DISTINCT [] <=> T) /\
 (!h t. ALL_DISTINCT (h::t) <=> ~MEM h t /\ ALL_DISTINCT t)

|-. !l. ALL_DISTINCT l <=>
 (!x. MEM x l ==> (FILTER ($= x) l = [x]))

|-. !ls. ALL_DISTINCT ls <=> (CARD (set ls) = LENGTH ls):
```

184 / 195

## Formal Sanity



### Formal Sanity

- to ensure correctness test your definitions via e.g. EVAL
- in HOL testing means symbolic evaluation, i.e. proving lemmata
- **formally proving sanity check lemmata** is very beneficial
  - ▶ they should express core properties of your definition
  - ▶ thereby they check your intuition against your actual definitions
  - ▶ these lemmata are often useful for following proofs
  - ▶ using them improves robustness and maintainability of your development
- I highly recommend using formal sanity checks

185 / 195

## Formal Sanity Example I



```
> val ALL_DISTINCT = Define '
 (ALL_DISTINCT [] = T) /\
 (ALL_DISTINCT (h::t) = ~MEM h t /\ ALL_DISTINCT t)';
```

### Example Sanity Check Lemmata

```
|- ALL_DISTINCT []
|-. !x xs. ALL_DISTINCT (x::xs) <=> ~MEM x xs /\ ALL_DISTINCT xs
|-. !x. ALL_DISTINCT [x]
|-. !x xs. ~(ALL_DISTINCT (x::x::xs))
|-. !l. ALL_DISTINCT (REVERSE l) <=> ALL_DISTINCT l
|-. !x l. ALL_DISTINCT (SNOC x l) <=> ~MEM x l /\ ALL_DISTINCT l
|-. !l1 l2. ALL_DISTINCT (l1 ++ l2) <=>
 ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2
```

186 / 195



## Formal Sanity Example II 1

```
> val ZIP_def = Define ‘
 (ZIP [] ys = []) /\ (ZIP xs [] = []) /\
 (ZIP (x::xs) (y::ys) = (x, y)::(ZIP xs ys))’
```

```
val ZIP_def =
|- (!ys. ZIP [] ys = []) /\ (!v3 v2. ZIP (v2::v3) [] = []) /\
 (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```

- above definition of ZIP looks straightforward
- small changes cause heuristics to produce different theorems
- use formal sanity lemmata to compensate

```
> val ZIP_def = Define ‘
 (ZIP xs [] = []) /\ (ZIP [] ys = []) /\
 (ZIP (x::xs) (y::ys) = (x, y)::(ZIP xs ys))’
```

```
val ZIP_def =
|- (!xs. ZIP xs [] = []) /\ (!v3 v2. ZIP [] (v2::v3) = []) /\
 (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```



## Formal Sanity Example II 2

```
val ZIP_def =
|- (!ys. ZIP [] ys = []) /\ (!v3 v2. ZIP (v2::v3) [] = []) /\
 (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```

### Example Formal Sanity Lemmata

```
|- (!xs. ZIP xs [] = []) /\ (!ys. ZIP [] ys = []) /\
 (!y ys x xs. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
|- !xs ys. LENGTH (ZIP xs ys) = MIN (LENGTH xs) (LENGTH ys)
|- !x y xs ys. MEM (x, y) (ZIP xs ys) ==> (MEM x xs /\ MEM y ys)
|- !xs1 xs2 ys1 ys2. LENGTH xs1 = LENGTH ys1 ==>
 (ZIP (xs1++xs2) (ys1++ys2) = (ZIP xs1 ys1 ++ ZIP xs2 ys2))
...
```

- in your proofs use sanity lemmata, not original definition
- this makes your development robust against
  - ▶ small changes to the definition required later
  - ▶ changes to Define and its heuristics
  - ▶ bugs in function definition package

187 / 195



188 / 195

## Part XII

## Deep and Shallow Embeddings



## Deep and Shallow Embeddings



- often one models some kind of formal language
- important design decision: use **deep** or **shallow** embedding
- in a nutshell:
  - ▶ shallow embeddings just model semantics
  - ▶ deep embeddings model syntax as well
- a shallow embedding directly uses the HOL logic
- a deep embedding
  - ▶ defines a datatype for the syntax of the language
  - ▶ provides a function to map this syntax to a semantic

190 / 195

## Example: Embedding of Propositional Logic I



- propositional logic is a subset of HOL
- a shallow embedding is therefore trivial

```
val sh_true_def = Define 'sh_true = T';
val sh_var_def = Define 'sh_var (v:bool) = v';
val sh_not_def = Define 'sh_not b = ~b';
val sh_and_def = Define 'sh_and b1 b2 = (b1 /\ b2)';
val sh_or_def = Define 'sh_or b1 b2 = (b1 \/ b2)';
val sh_implies_def = Define 'sh_implies b1 b2 = (b1 ==> b2)';
```

191 / 195

## Shallow vs. Deep Embeddings



### Shallow

- quick and easy to build
- extensions are simple

### Deep

- can reason about syntax
- allows verified implementations
- sometimes tricky to define
  - e.g. bound variables

### Important Questions for Deciding

- Do I need to reason about syntax?
- Do I have hard to define syntax like bound variables?
- How much time do I have?

193 / 195

## Example: Embedding of Propositional Logic II



- we can also define a datatype for propositional logic
- this leads to a deep embedding

```
val _ = Datatype 'bvar = BVar num'
val _ = Datatype 'prop = d_true | d_var bvar | d_not prop
 | d_and prop prop | d_or prop prop
 | d_implies prop prop';
```

```
val _ = Datatype 'var_assignment = BAssign (bvar -> bool)';
val VAR_VALUE_def = Define 'VAR_VALUE (BAssign a) v = (a v)';
```

```
val PROP_SEM_def = Define '
 (PROP_SEM a d_true = T) /\
 (PROP_SEM a (d_var v) = VAR_VALUE a v) /\
 (PROP_SEM a (d_not p) = ~(PROP_SEM a p)) /\
 (PROP_SEM a (d_and p1 p2) = (PROP_SEM a p1 /\ PROP_SEM a p2)) /\
 (PROP_SEM a (d_or p1 p2) = (PROP_SEM a p1 \/ PROP_SEM a p2)) /\
 (PROP_SEM a (d_implies p1 p2) = (PROP_SEM a p1 ==> PROP_SEM a p2))';
```

192 / 195

## Example: Embedding of Propositional Logic III



- with deep embedding one can easily formalise syntactic properties like
  - Which variables does a propositional formula contain?
  - Is a formula in negation-normal-form (NNF)?
- with shallow embeddings
  - syntactic concepts can't be defined in HOL
  - however, they can be defined in SML
  - no proofs about them possible

```
val _ = Define '
 (IS_NNF (d_not d_true) = T) /\ (IS_NNF (d_not (d_var v)) = T) /\
 (IS_NNF (d_not _) = F) /\

 (IS_NNF d_true = T) /\ (IS_NNF (d_var v) = T) /\
 (IS_NNF (d_and p1 p2) = (IS_NNF p1 /\ IS_NNF p2)) /\
 (IS_NNF (d_or p1 p2) = (IS_NNF p1 /\ IS_NNF p2)) /\
 (IS_NNF (d_implies p1 p2) = (IS_NNF p1 /\ IS_NNF p2))';
```

194 / 195

## Verified vs. Verifying Program



### Verified Programs

- are formalised in HOL
- their properties have been proven once and for all
- all runs have proven properties
- are usually less sophisticated, since they need verification
- is what one wants ideally
- often require deep embedding

### Verifying Programs

- are written in meta-language
- they produce a separate proof for each run
- only certain that current run has properties
- allow more flexibility, e. g. fancy heuristics
- good pragmatic solution
- shallow embedding fine