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/)
- Arve Gengelbach (https://people.kth.se/~arveg/)
- Karl Palmskog (https://setoid.com)

Part XII

Good Definitions

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
- hard to state what a good definition is
- even harder to come up with good definitions

Importance of Good Definitions — Clarity I

- HOL4 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

Importance of Good Definitions — Clarity II

Guaranteed by HOL4

- proofs checked
- internal, technical definitions
- technical lemmas
- 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 artefact is vital
  - testing via e. g. `EVAL`
  - formal sanity
  - conformance testing
- wrong models are main source of error when using HOL4
- proofs, auxiliary lemmas and auxiliary definitions
  - can be as technical and complicated as you like
  - correctness is guaranteed by HOL4
  - reviewers don't need to care

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

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 HOL4 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

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

Good Definitions – no number encodings

- many programmers familiar with C encode everything as a number
- enumeration types are very cheap in SML and HOL4
- 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 HOL4, 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) ‘;
```

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 )‘;
```

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

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

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
}
```

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 disconneted
| Connecting of connecting;

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

Good Definitions in HOL4

Objectives

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

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 lemmas
  - improve maintainability

Good Definitions in HOL4 II

Technical Issues

- write definitions such that they work well with HOL4's tools
- this requires you to know HOL4 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 _ = [])`
```

Good Definitions in HOL4 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):
```

Formal Sanity

Formal Sanity

- to ensure correctness test your definitions via e. g. `EVAL`
- in HOL4 testing means symbolic evaluation, i. e. proving lemmas
- **formally proving sanity check lemmas** is very beneficial
  - they should express core properties of your definition
  - thereby they check your intuition against your actual definitions
  - these lemmas are often useful for following proofs
  - using them improves robustness and maintainability of your development
- we highly recommend using formal sanity checks

Formal Sanity Example I

```
> val ALL_DISTINCT = Define `
    (ALL_DISTINCT [] = T) /\
    (ALL_DISTINCT (h::t) = ~MEM h t /\ ALL_DISTINCT t)`;
```

Example Sanity Check Lemmas

```
|- 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
```

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 lemmas 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 ys0
```

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 Lemmas

```
|- (!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 lemmas, 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

Part XIII

Deep and Shallow Embeddings

Deep and Shallow Embeddings

- **Embedding**: modelling a language (guest) within another (host)
- Reuses syntax, semantics, and/or implementation from host language
- Avoids implementing a standalone compiler/interpreter
- important design decision: **deep** vs. **shallow** embedding

Deep

- AST represented by a data type in host
- Separate evaluation function provides semantics
- e. g. , HOL logic is deeply embedded in SML (`term`)

Shallow

- Language constructs mapped directly to their semantics
- Embeds guest semantics into host semantics
- e.g. HOL4 tactic language shallowly embedded in SML

Example: Embedding of Propositional Logic I

- propositional logic is a subset of the HOL logic
- a shallow embedding in HOL 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)`;
```

- Note: a shallow embedding in HOL is still a deep embedding in SML

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))‘
```

Shallow vs. Deep Embeddings in HOL4

Shallow

- uses the HOL logic directly
- quick to build if host syntax is similar
- leverages binding mechanisms and substitution
- easy extension: new language constructs

Deep

- can reason about syntax
- allows verified implementations
- easy extension: new semantics
- 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?

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))`
```

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

Summary Deep vs. Shallow Embeddings

- deep embeddings require more work
- they however allow reasoning about syntax
  - induction and case-splits possible
  - a semantic subset can be carved out syntactically
- syntax sometimes hard to define for deep embeddings
- combinations of deep and shallow embeddings common
  - certain parts are deeply embedded
  - others are embedded shallowly

Well-Founded Relations

- a relation `R : 'a -> 'a -> bool` is called **well-founded**, iff there are no infinite descending chains

  `wellfounded R = ~?f. !n. R (f (SUC n)) (f n)`

- Example: `$< : num -> num -> 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

Well-Founded Recursion

- a well-founded relation `R` can be used to define recursive functions
- this recursion principle is called `WFREC` in HOL4
- 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

Manual Termination Proofs I

- 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
  `|- !(f:'a -> num) x y. measure f x y <=> (f x < f y)`
- all measures are well-founded: `|- !f. WF (measure f)`
- moreover, existing well-founded relations can be combined
  - lexicographic order `LEX`
  - list lexicographic order `LLEX`
  - ...

Manual Termination Proofs II

- 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`

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 = HOL4 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
```

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))
```

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))

> ...
```

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
```