

Interactive Theorem Proving (ITP) Course Part XIV

Thomas Tuerk (tuerk@kth.se)



Academic Year 2016/17, Period 4

version fa13ae6 of Mon May 29 12:37:43 2017

Part XIV

Advanced Definition Principles



Relations



- a relation is a function from some arguments to bool
- the following example types are all types of relations:
 - ▶ : 'a -> 'a -> bool
 - ▶ : 'a -> 'b -> bool
 - ▶ : 'a -> 'b -> 'c -> 'd -> bool
 - ▶ : ('a # 'b # 'c) -> bool
 - ▶ : bool
 - ▶ : 'a -> bool
- relations are closely related to sets
 - ▶ $R\ a\ b\ c \iff (a, b, c) \in \{(a, b, c) \mid R\ a\ b\ c\}$
 - ▶ $(a, b, c) \in S \iff (\lambda a\ b\ c. (a, b, c) \in S)\ a\ b\ c$

Relations II



- relations are often defined by a set of **rules**

Definition of Reflexive-Transitive Closure

The transitive reflexive closure of a relation $R : 'a \rightarrow 'a \rightarrow \text{bool}$ can be defined as the least relation $\text{RTC } R$ that satisfies the following rules:

$$\frac{R\ x\ y}{\text{RTC } R\ x\ y} \quad \frac{}{\text{RTC } R\ x\ x} \quad \frac{\text{RTC } R\ x\ y \quad \text{RTC } R\ y\ z}{\text{RTC } R\ x\ z}$$

- if the rules are monoton, a least and a greatest fix point exists (Knaster-Tarski theorem)
- least fixpoints give rise to **inductive relations**
- greatest fixpoints give rise to **coinductive relations**



- (Co)IndDefLib provides infrastructure for defining (co)inductive relations
- given a set of rules `Hol_(co)reln` defines (co)inductive relations
- 3 theorems are returned and stored in current theory
 - ▶ a rules theorem — it states that the defined constant satisfies the rules
 - ▶ a cases theorem — this is an equational form of the rules showing that the defined relation is indeed a fixpoint
 - ▶ a (co)induction theorem
- additionally a strong (co)induction theorem is stored in current theory



```
> val (RTC_REL_rules, RTC_REL_ind, RTC_REL_cases) = Hol_reln '
  (!x y. R x y ==> RTC_REL R x y) /\
  (!x. RTC_REL R x x) /\
  (!x y z. RTC_REL R x y /\ RTC_REL R x z ==> RTC_REL R x z)';

val RTC_REL_rules = |- !R.
  (!x y. R x y ==> RTC_REL R x y) /\ (!x. RTC_REL R x x) /\
  (!x y z. RTC_REL R x y /\ RTC_REL R x z ==> RTC_REL R x z)

val RTC_REL_cases = |- !R a0 a1.
  RTC_REL R a0 a1 <=>
  (R a0 a1 \/ (a1 = a0) \/ ?y. RTC_REL R a0 y /\ RTC_REL R a0 a1)
```

254 / 271

255 / 271

Example: Transitive Reflexive Closure II



```
val RTC_REL_ind = |- !R RTC_REL'.
  ((!x y. R x y ==> RTC_REL' x y) /\ (!x. RTC_REL' x x) /\
   (!x y z. RTC_REL' x y /\ RTC_REL' x z ==> RTC_REL' x z)) ==>
  (!a0 a1. RTC_REL R a0 a1 ==> RTC_REL' a0 a1)

> val RTC_REL_strongind = DB.fetch "-" "RTC_REL_strongind"

val RTC_REL_strongind = |- !R RTC_REL'.
  (!x y. R x y ==> RTC_REL' x y) /\ (!x. RTC_REL' x x) /\
  (!x y z.
    RTC_REL R x y /\ RTC_REL' x y /\ RTC_REL R x z /\
    RTC_REL' x z ==>
    RTC_REL' x z) ==>
  (!a0 a1. RTC_REL R a0 a1 ==> RTC_REL' a0 a1)
```

256 / 271

Example: EVEN



```
> val (EVEN_REL_rules, EVEN_REL_ind, EVEN_REL_cases) = Hol_reln
  '(EVEN_REL 0) /\ (!n. EVEN_REL n ==> (EVEN_REL (n + 2)))';

val EVEN_REL_cases =
  |- !a0. EVEN_REL a0 <=> (a0 = 0) \/ ?n. (a0 = n + 2) /\ EVEN_REL n

val EVEN_REL_rules =
  |- EVEN_REL 0 /\ !n. EVEN_REL n ==> EVEN_REL (n + 2)

val EVEN_REL_ind = |- !EVEN_REL'.
  (!a0.
    EVEN_REL' a0 ==>
    (a0 = 0) \/ ?n. (a0 = n + 2) /\ EVEN_REL' n) ==>
  (!a0. EVEN_REL' a0 ==> EVEN_REL a0)
```

- notice that in this example there is exactly one fixpoint
- therefore for these rule, the induction and coinductive relation coincide

257 / 271

Example: Dummy Relations



```
> val (DF_rules, DF_ind, DF_cases) = Hol_reln
  '(!n. DF (n+1) ==> (DF n))'

> val (DT_rules, DT_coind, DT_cases) = Hol_coreln
  '(!n. DT (n+1) ==> (DT n))'

val DT_coind =
  |- !DT'. (!a0. DT' a0 ==> DT' (a0 + 1)) ==> !a0. DT' a0 ==> DT a0

val DF_ind =
  |- !DF'. (!n. DF' (n + 1) ==> DF' n) ==> !a0. DF a0 ==> DF' a0

val DT_cases = |- !a0. DT a0 <=> DT (a0 + 1):
val DF_cases = |- !a0. DF a0 <=> DF (a0 + 1):
```

- notice that for both DT and DF we used essentially a non-terminating recursion
- DT is always true, i.e. $\vdash !n. DT\ n$
- DF is always false, i.e. $\vdash !n. \sim(DF\ n)$

258 / 271

Quotient Types Example



- let's assume we have an implementation of finite sets of numbers as binary trees with
 - ▶ type `binset`
 - ▶ binary tree invariant `WF_BINSET : binset -> bool`
 - ▶ constant `empty_binset`
 - ▶ add and member functions `add : num -> binset -> binset`,
`mem : binset -> num -> bool`
- we can define a partial equivalence relation by
`binset_equiv b1 b2 := (WF_BINSET b1 /\ WF_BINSET b2 /\ (!n. mem b1 n <=> mem b2 n))`
- this allows defining a quotient type of sets of numbers
- functions `empty_binset`, `add` and `mem` as well as lemmata about them can be lifted automatically

260 / 271

Quotient Types



- `quotientLib` allows to define types as quotients of existing types with respect to **partial equivalence relation**
- each equivalence class becomes a value of the new type
- partiality allows ignoring certain types
- `quotientLib` allows to lift definitions and lemmata as well
- details are technical and won't be presented here

259 / 271

Quotient Types Summary



- quotient types are sometimes very useful
 - ▶ e.g. rational numbers are defined as a quotient type
- there is powerful infrastructure for them
- many tasks are automated
- however, the details are technical and won't be discussed here

261 / 271



- pattern matching ubiquitous in functional programming
- pattern matching is a powerful technique
- it helps to write concise, readable definitions
- very handy and frequently used for interactive theorem proving in higher-order logic (HOL)
- however, it is **not directly supported** by HOL's logic
- representations in HOL
 - ▶ sets of equations as produced by Define
 - ▶ decision trees (printed as case-expressions)

Case Expressions

- sometimes one does not want to use this compilation by TFL
 - ▶ one wants to use pattern-matches somewhere nested in a term
 - ▶ one might not want to introduce a new constant
 - ▶ one might want to avoid using TFL for technical reasons
- in such situations, case-expressions can be used
- their syntax is similar to the syntax used by SML

Example

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

val ZIP_def = |- !ys xs. ZIP xs ys =
  case (xs,ys) of
    ([],[]) => []
  | ([],v4::v5) => ARB
  | (x::xs',[]) => ARB
  | (x::xs',y::ys') => (x,y)::ZIP xs' ys'
```

262 / 271



- we have already used top-level pattern matches with the TFL package
- Define is able to handle them
 - ▶ all the semantic complexity is taken care of
 - ▶ no special syntax or functions remain
 - ▶ no special rewrite rules, reasoning tools needed afterwards
- Define produces a set of equations
- this is the recommended way of using pattern matching in HOL

Example

```
> 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) /\
                        (ZIP [] [] = [])
```

263 / 271

Case Expressions II

- the datatype package define case-constants for each datatype
- the parser contains a pattern compilation algorithm
- case-expressions are by the parser compiled to decision trees using case-constants
- pretty printer prints these decision trees as case-expressions again

Example

```
val ZIP_def = |- !ys xs. ZIP xs ys =
  pair_CASE (xs,ys)
    (\v v1.
      list_CASE v (list_CASE v1 [] (\v4 v5. ARB))
        (\x xs'. list_CASE v1 ARB (\y ys'. (x,y)::ZIP xs' ys'))):
```

264 / 271

265 / 271

Case Expression Issues



- using case expressions feels very natural to functional programmers
- case-expressions allow concise, well-readable definitions
- however, there are also many drawbacks
- there is large, complicated code in the parser and pretty printer
 - ▶ this is outside the kernel
 - ▶ parsing a pretty-printed term can result in a non α -equivalent one
 - ▶ there are bugs in this code (see e.g. Issue #416 reported 8 May 2017)
- the results are hard to predict
 - ▶ heuristics involved in creating decision tree
 - ▶ results sometimes hard to predict
 - ▶ however, it is beneficial that proofs follow this internal, volatile structure

266 / 271

Case Expression Example I



Partial Proof Script

```
val _ = prove ('!l1 l2.
  (LENGTH l1 = LENGTH l2) ==>
  ((ZIP l1 l2 = []) <=> ((l1 = []) /\ (l2 = [])))',
  ONCE_REWRITE_TAC [ZIP_def]
```

Current Goal

```
!l1 l2.
  (LENGTH l1 = LENGTH l2) ==>
  (((case (l1,l2) of
    ([],[]) => []
  | ([],v4::v5) => ARB
  | (x::xs',[]) => ARB
  | (x::xs',y::ys') => (x,y)::ZIP xs' ys') =
  []) <=> (l1 = []) /\ (l2 = []))
```

268 / 271

Case Expression Issues II



- technical issues
 - ▶ it is tricky to reason about decision trees
 - ▶ rewrite rules about case-constants needs to be fetched from TypeBase
 - ★ alternative `srw_ss` often does more than wanted
 - ▶ partially evaluated decision-trees are not pretty printed nicely any more
- underspecified functions
 - ▶ decision trees are exhaustive
 - ▶ they list underspecified cases explicitly with value ARB
 - ▶ this can be lengthy
 - ▶ Define in contrast hides underspecified cases

267 / 271

Case Expression Example IIa – partial evaluation



Partial Proof Script

```
val _ = prove ('!l1 l2.
  (LENGTH l1 = LENGTH l2) ==>
  ((ZIP l1 l2 = []) <=> ((l1 = []) /\ (l2 = [])))',
  ONCE_REWRITE_TAC [ZIP_def] >>
  REWRITE_TAC[pairTheory.pair_case_def] >> BETA_TAC
```

Current Goal

```
!l1 l2.
  (LENGTH l1 = LENGTH l2) ==>
  (((case l1 of
    [] => (case l2 of [] => [] | v4::v5 => ARB)
  | x::xs' => case l2 of [] => ARB | y::ys' => (x,y)::ZIP xs' ys') =
  []) <=> (l1 = []) /\ (l2 = []))
```

269 / 271



Partial Proof Script

```
val _ = prove ('!l1 l2.
  (LENGTH l1 = LENGTH l2) ==>
  ((ZIP l1 l2 = []) <=> ((l1 = []) /\ (l2 = [])))',
ONCE_REWRITE_TAC [ZIP_def] >>
Cases_on 'l1' >| [
  REWRITE_TAC[listTheory.list_case_def]
```

Current Goal

```
!l2.
  (LENGTH [] = LENGTH l2) ==>
  (((case ([],l2) of
    ([],[]) => []
  | ([],v4::v5) => ARB
  | (x::xs',[]) => ARB
  | (x::xs',y::ys') => (x,y)::ZIP xs' ys') =
  []) <=> (l2 = []))
```

270 / 271

Case Expression Summary

- case expressions are natural to functional programmers
- they allow concise, readable definitions
- however, fancy parser and pretty-printer needed
 - ▶ trustworthiness issues
 - ▶ sanity check lemmata advisable
- reasoning about case expressions can be tricky and lengthy
- proofs about case expression often hard to maintain
- therefore, use top-level pattern matching via Define if easily possible

271 / 271