Interactive Theorem Proving (ITP) Course
Parts XIII, XIV

Thomas Tuerk (tuerk@kth.se)

Academic Year 2016/17, Period 4

version e129362 of Mon May 22 09:50:13 2017

Part XIII

Rewriting

Rewriting in HOL

- simplification via rewriting was already a strength of Edinburgh LCF
- it was further improved for Cambridge LCF
- HOL inherited this powerful rewriter
- equational reasoning is still the main workhorse
- there are many different equational reasoning tools in HOL
  - `Rewrite` library
    inherited from Cambridge LCF
    you have seen it in the form of `REWRITE_TAC`
  - `computeLib` — fast evaluation
    build for speed, optimised for ground terms
    seen in the form of `EVAL`
  - `simpLib` — Simplification
    sophisticated rewrite engine, HOL's main workhorse
    not discussed in this lecture, yet
  - ...

Semantic Foundations

- we have seen primitive inference rules for equality before

$$\frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v \qquad \textit{types fit}}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \mathrm{COMB} \qquad \frac{\Gamma \vdash s = t \qquad x \textit{ not free in } \Gamma}{\Gamma \vdash \lambda x.\ s = \lambda x.\ t}\ \mathrm{ABS}$$

$$\frac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \mathrm{TRANS} \qquad \frac{}{\vdash t = t}\ \mathrm{REFL}$$

- these rules allow us to replace any subterm with an equal one
- this is the core of rewriting

Conversions

- in HOL, equality reasoning is implemented by **conversions**
- a conversion is a SML function of type `term -> thm`
- given a term `t`, a conversion
  - produce a theorem of the form `|- t = t'`
  - raise an `UNCHANGED` exception
  - fail, i. e. raise an `HOL_ERR` exception

Example

```
> BETA_CONV ``(\x. SUC x) y``
val it = |- (\x. SUC x) y = SUC y

> BETA_CONV ``SUC y``
Exception-HOL_ERR ... raised

> REPEATC BETA_CONV ``SUC y``
Exception- UNCHANGED raised
```


Conversionals

- similar to tactics and tacticals there are **conversionals** for conversions
- conversionals allow building conversions from simpler ones
- there are many of them
  - `THENC`
  - `ORELSEC`
  - `REPEATC`
  - `TRY_CONV`
  - `RAND_CONV`
  - `RATOR_CONV`
  - `ABS_CONV`
  - ...

Depth Conversionals

- for rewriting depth-conversionals are important
- a depth-conversional applies a conversion to all subterms
- there are many different ones
  - `ONCE_DEPTH_CONV c` — top down, applies c once at highest possible positions in distinct subterms
  - `TOP_SWEEP_CONV c` — top down, like `ONCE_DEPTH_CONV`, but continues processing rewritten terms
  - `TOP_DEPTH_CONV c` — top down, like `TOP_SWEEP_CONV`, but try top-level again after change
  - `DEPTH_CONV c` — bottom up, recurse over subterms, then apply c repeatedly at top-level
  - `REDEPTH_CONV c` — bottom up, like `DEPTH_CONV`, but revisits subterms

REWR_CONV

- it remains to rewrite terms at top-level
- this is achieved by `REWR_CONV`
- given a theorem and a term `t` and a theorem `|- t1 = t2`, `REWR_CONV t thm`
  - searches an instantiation of term and type variables such that `t1` becomes α-equivalent to `t`
  - fails, if no instantiation is found
  - otherwise, instantiate the theorem and get `|- t1' = t2'`
  - return theorem `|- t = t2'`

Example

term `LENGTH [1;2;3]`, theorem `|- LENGTH ((x:'a)::xs) = SUC (LENGTH xs)`
found type instantiation: `` [``:'a`` |-> ``:num``] ``
found term instantiation: `` [``x:num`` |-> ``1``; ``xs`` |-> ``[2;3]``] ``
returned theorem: `|- LENGTH [1;2;3] = SUC (LENGTH [2;3])`

- the tricky part is finding the instantiation
- this problem is called the (term) **matching** problem

Term Matching

- given term t_org and a term t_goal try to find
  - type substitution ty_s
  - term substitution tm_s
- such that subst tm_s (inst ty_s t_org) $\overset{\alpha}{\equiv}$ t_goal
- this can be easily implemented by a recursive search

t_org	t_goal	**action**
t1_org t2_org	t1_goal t2_goal	recurse
t1_org t2_org	otherwise	fail
\x. t_org x	\y. t_goal y	match types of x, y and recurse
\x. t_org x	otherwise	fail
const	same const	match types
const	otherwise	fail
var	anything	try to bind var, take care of existing bindings

Examples Term Matching

t_org	t_goal	**substs**
LENGTH ((x:'a)::xs)	LENGTH [1;2;3]	'a $\to$ num, x $\to$ 1, xs $\to$ [2;3]
[]:'a list	[]:'b list	'a $\to$ 'b
0	0	empty substitution
b /\ T	(P (x:'a) ==> Q) /\ T	b $\to$ P x ==> Q
b /\ b	P x /\ P x	b $\to$ P x
b /\ b	P x /\ P y	fail
!x:num. P x /\ Q x	!y:num. P' y /\ Q' y	P $\to$ P', Q $\to$ Q'
!x:num. P x /\ Q x	!y. (2 = y) /\ Q' y	P $\to$ (\$= 2), Q $\to$ Q'
!x:num. P x /\ Q x	!y. (y = 2) /\ Q' y	fail

- it is often very annoying that the last match fails
- it prevents us for example rewriting !y. (2 = y) /\ Q y to (!y. (2=y)) /\ (!y. Q y)
- Can we do better? Yes, with higher order (term) matching.

Higher Order Term Matching

- term matching searches for substitutions such that t_org becomes α-equivalent to t_goal
- **higher order term matching** searches for substitutions such that t_org becomes t_subst such that the $\beta\eta$-normalform of t_subst is α-equivalent equivalent to $\beta\eta$-normalform of t_goal, i. e.
  higher order term matching is aware of the semantics of λ

β-reduction	$(\lambda x.\ f)\ y = f[y/x]$
η-conversion	$(\lambda x.\ f\ x) = f$ where x is not free in f

- the HOL implementation expects t_org to be a **higher-order pattern**
  - t_org is β-reduced
  - if X is a variable that should be instantiated, then all arguments should be distinct variables
- for other forms of t_org, HOL's implementation might fail
- higher order matching is used by HO_REWR_CONV

Examples Higher Order Term Matching

t_org	t_goal	**substs**
!x:num. P x /\ Q x	!y. (y = 2) /\ Q' y	P $\to$ (\y. y = 2), Q $\to$ Q'
!x. P x /\ Q x	!x. P x /\ Q x /\ Z x	Q $\to$ \x. Q x /\ Z x
!x. P x /\ Q	!x. P x /\ Q x	fails
!x. P (x, x)	!x. Q x	fails
!x. P (x, x)	!x. FST (x,x) = SND (x,x)	P $\to$ \xx. FST xx = SND xx

Don't worry, it might look complicated, but in practice it is easy to get a feeling for higher order matching.

Rewrite Library

- the rewrite library combines REWR_CONV with depth conversions
- there are many different conversions, rules and tactics
- at they core, they all work very similarly
  - given a list of theorems, a set of rewrite theorems is derived
    - split conjunctions
    - remove outermost universal quantification
    - introduce equations by adding = T (or = F) if needed
  - REWR_CONV is applied to all the resulting rewrite theorems
  - a depth-conversion is used with resulting conversion
- for performance reasons an efficient indexing structure is used
- by default implicit rewrites are added

Rewrite Library II

- REWRITE_CONV
- REWRITE_RULE
- REWRITE_TAC
- ASM_REWRITE_TAC
- ONCE_REWRITE_TAC
- PURE_REWRITE_TAC
- PURE_ONCE_REWRITE_TAC
- ...

Ho_Rewrite Library

- similar to Rewrite lib, but uses higher order matching
- internally uses HO_REWR_CONV
- similar conversions, rules and tactics as Rewrite lib
  - Ho_Rewrite.REWRITE_CONV
  - Ho_Rewrite.REWRITE_RULE
  - Ho_Rewrite.REWRITE_TAC
  - Ho_Rewrite.ASM_REWRITE_TAC
  - Ho_Rewrite.ONCE_REWRITE_TAC
  - Ho_Rewrite.PURE_REWRITE_TAC
  - Ho_Rewrite.PURE_ONCE_REWRITE_TAC
  - ...

Examples Rewrite and Ho_Rewrite Library

```
> REWRITE_CONV [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (SUC 0)

> ONCE_REWRITE_CONV [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (LENGTH [2])

> ONCE_REWRITE_CONV [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (LENGTH [2])

> REWRITE_CONV [] ``A /\ A /\ ~A``
Exception- UNCHANGED raised

> PURE_REWRITE_CONV [NOT_AND] ``A /\ A /\ ~A``
val it = |- A /\ A /\ ~A <=> A /\ F

> REWRITE_CONV [NOT_AND] ``A /\ A /\ ~A``
val it = |- A /\ A /\ ~A <=> F

> REWRITE_CONV [FORALL_AND_THM] ``!x. P x /\ Q x /\ R x``
Exception- UNCHANGED raised

> Ho_Rewrite.REWRITE_CONV [FORALL_AND_THM] ``!x. P x /\ Q x /\ R x``
val it = |- !x. P x /\ Q x /\ R x <=> (!x. P x) /\ (!x. Q x) /\ (!x. R x)
```

Summary `Rewrite` and `Ho_Rewrite` Library

- the `Rewrite` and `Ho_Rewrite` library provide powerful infrastructure for term rewriting
- thanks to clever implementations they are reasonably efficient
- basics are easily explained
- however, efficient usage needs some experience

Term Rewriting Systems

- to use rewriting efficiently, one needs to understand about term rewriting systems
- this is a large topic
- one can easily give whole course just about term rewriting systems
- however, in practise you quickly get a feeling
- important points in practise
  - ensure termination of your rewrites
  - make sure they work nicely together

Term Rewriting Systems — Termination

Theory

- choose well-founded order $\prec$
- for each rewrite theorem `|- t1 = t2` ensure `t2` $\prec$ `t1`

Practice

- informally define for yourself what **simpler** means
- ensure each rewrite makes terms simpler
- good heuristics
  - subterms are simpler than whole term
  - use an order on functions

Termination — Subterm examples

- a proper subterm is always simpler
  - `!l. APPEND [] l = l`
  - `!n. n + 0 = n`
  - `!l. REVERSE (REVERSE l) = l`
  - `!t1 t2. if T then t1 else t2 <=> t1`
  - `!n. n * 0 = 0`
- the right hand side should not use extra vars, throwing parts away is usually simpler
  - `!x xs. (SNOC x xs = []) = T`
  - `!x xs. LENGTH (x::xs) = SUC (LENGTH xs)`
  - `!n x xs. DROP (SUC n) (x::xs) = DROP n xs`

Termination — use simpler terms

- it is useful to consider some functions simple and other complicated
- replace complicated ones with simple ones
- never do it in the opposite direction
- clear examples
  - `|- !m n. MEM m (COUNT_LIST n) <=> (m < n)`
  - `|- !ls n. (DROP n ls = []) <=> (n >= LENGTH ls)`
- unclear examples
  - `|- !L. REVERSE L = REV L []`

Termination — Normalforms

- some equations can be used in both directions
- one should decide on one direction
- this implicitly defined a **normalform** one wants terms to be in
- examples
  - `|- !f l. MAP f (REVERSE l) = REVERSE (MAP f l)`
  - `|- !l1 l2 l3. l1 ++ (l2 ++ l3) = l1 ++ l2 ++ l3`

Termination — Problematic rewrite rules

- some equations immediately lead to non-termination, e. g.
  - `|- !m n. m + n = n + m`
  - `|- !m. m = m + 0`
- slightly more subtle are rules like
  - `|- !n. fact n = if (n = 0) then 1 else n * fact(n-1)`
- often combination of multiple rules leads to non-termination this is especially problematic when adding to predefined set of rewrites
  - `|- !m n p. m + (n + p) = (m + n) + p` and `|- !m n p. (m + n) + p = m + (n + p)`

Rewrites working together

- rewrite rules should not complete with each other
- if a term `ta` can be rewritten to `ta1` and `ta2` applying different rewrite rules, then the `ta1` and `ta2` should be further rewritten to a common `tb`
- this can often be achieved by adding extra rewrite rules

Example

Assume we have the rewrite rules `|- DOUBLE n = n + n` and `|- EVEN (DOUBLE n) = T`.
With these the term `EVEN (DOUBLE 2)` can be rewritten to

- `T` or
- `EVEN (2 + 2)`.

To avoid a hard to predict result, `EVEN (2+2)` should be rewritten to `T`.
Adding an extra rewrite rule `|- EVEN (n + n) = T` achieves this.

Rewrites working together II

- to design rewrite systems that work well, normalforms are vital
- a term is in **normalform**, if it cannot be rewritten any further
- one should have a clear idea what the normalform of common terms looks like
- all rules should work together to establish this normalform
- the right-hand-side of each rule should be in normalform
- the left-hand-side should not be simplifiable by any other rule
- the order in which rules are applied should not influence the final result

computeLib

- computeLib is the library behind EVAL
- it is a rewriting library designed for evaluating ground terms (i. e. terms without variables) efficiently
- it uses a call-by-value strategy similar to SML's
- it uses first order term matching
- it performs β reduction in addition to rewrites

compset

- computeLib uses compsets to store its rewrites
- a compset stores
  - rewrite rules
  - extra conversions
- the extra conversions are guarded by a term pattern for efficiency
- users can define their own compsets
- however, computeLib maintains one special compset called the_compset
- the_compset is used by EVAL

EVAL

- EVAL uses the_compset
- tools like the Datatype of TFL automatically extend the_compset
- this way, EVAL knows about (nearly) all types and functions
- one can extended the_compset manually as well
- rewrites exported by Define are good for ground terms but may lead to non-termination for non-ground terms
- zDefine prevents TFL from automatically extending the_compset

`simpLib`

- `simpLib` is a sophisticated rewrite engine
- it is HOL's main workhorse
- it provides
  - higher order rewriting
  - usage of context information
  - conditional rewriting
  - arbitrary conversions
  - support for decision procedures
  - simple heuristics to avoid non-termination
  - fancier preprocessing of rewrite theorems
  - ...
- it is very powerful, but compared to `Rewrite` lib sometimes slow

Basic Usage I

- `simpLib` uses **simpsets**
- simpsets are special datatypes storing
  - rewrite rules
  - conversions
  - decision procedures
  - congruence rules
  - ...
- in addition there are simpset-fragments
- simpset-fragments contain similar information as simpsets
- fragments can be added to and removed from simpsets
- most important simpset is `std_ss`

Basic Usage II

- a call to the simplifier takes as arguments
  - a simpset
  - a list of rewrite theorems
- common high-level entry points are
  - `SIMP_CONV ss thmL` — conversion
  - `SIMP_RULE ss thmL` — rule
  - `SIMP_TAC ss thmL` — tactic without considering assumptions
  - `ASM_SIMP_TAC ss thmL` — tactic using assumptions to simplify goal
  - `FULL_SIMP_TAC ss thmL` — tactic simplifying assumptions with each other and goal with assumptions
  - `REV_FULL_SIMP_TAC ss thmL` — similar to `FULL_SIMP_TAC` but with reversed order of assumptions
- there are many derived tools not discussed here

Basic Simplifier Examples

```
> SIMP_CONV bool_ss [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (SUC 0)

> SIMP_CONV std_ss [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = 2

> SIMP_CONV list_ss [] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = 2
```

Common simpsets

- pure_ss — empty simpset
- bool_ss — basic simpset
- std_ss — standard simpset
- arith_ss — arithmetic simpset
- list_ss — list simpset
- real_ss — real simpset

Common simpset-fragments

- many theories and libraries provide their own simpset-fragments
- PRED_SET_ss — simplify sets
- STRING_ss — simplify strings
- QI_ss — extra quantifier instantiations
- gen_beta_ss — β reduction for pairs
- ETA_ss — η conversion
- EQUIV_EXTRACT_ss — extract common part of equivalence
- CONJ_ss — use conjunctions for context
- ...

Build-In Conversions and Decision Procedures

- in contrast to Rewrite lib the simplifier can run arbitrary conversions
- most useful is probably β reduction
- std_ss has support for basic arithmetic and numerals
- it also has simple, syntactic conversions for instantiating quantifiers
  - `!x. ... /\ (x = c) /\ ... ==> ...`
  - `!x. ... \/ ~(x = c) \/ ...`
  - `?x. ... /\ (x = c) /\ ...`
- besides very useful conversions, there are decision procedures as well
- the most frequently used one is probably the arithmetic decision procedure you already know from DECIDE

Examples I

```
> SIMP_CONV std_ss [] ``(\x. x + 2) 5``
val it = |- (\x. x + 2) 5 = 7

> SIMP_CONV std_ss [] ``!x. Q x /\ (x = 7) ==> P x``
val it = |- (!x. Q x /\ (x = 7) ==> P x) <=> (Q 7 ==> P 7)``

> SIMP_CONV std_ss [] ``?x. Q x /\ (x = 7) /\ P x``
val it = |- (?x. Q x /\ (x = 7) /\ P x) <=> (Q 7 /\ P 7)``

> SIMP_CONV std_ss [] ``x > 7 ==> x > 5``
Exception- UNCHANGED raised

> SIMP_CONV arith_ss [] ``x > 7 ==> x > 5``
val it = |- (x > 7 ==> x > 5) <=> T
```

Higher Order Rewriting

- the simplifier supports higher order rewriting
- this is often very handy
- for example it allows moving quantifiers around easily

Examples

```
> SIMP_CONV std_ss [FORALL_AND_THM] ``!x. P x /\ Q /\ R x``
val it = |- (!x. P x /\ Q /\ R x) <=>
            (!x. P x) /\ Q /\ (!x. R x)

> SIMP_CONV std_ss [GSYM RIGHT_EXISTS_AND_THM, GSYM LEFT_FORALL_IMP_THM]
  ``!y. (P y /\ (?x. y = SUC x)) ==> Q y``
val it = |- (!y. P y /\ (?x. y = SUC x) ==> Q y) <=>
            !x. P (SUC x) ==> Q (SUC x)
```


Context

- a great feature of the simplifier is that it can use context information
- by default simple context information is used like
  - the precondition of an implication
  - the condition of `if-then-else`
- one can configure which context to use via congruence rules
  - by using `CONJ_ss` one can easily use context of conjunctions
  - warning: using `CONJ_ss` can be slow
  - using other contexts is outside the scope of this lecture
- using context often simplifies proofs drastically
  - using `Rewrite` lib, often a goal needs to be split and a precondition moved to the assumptions
  - then `ASM_REWRITE_TAC` can be used
  - with `SIMP_TAC` there is no need to split the goal

Context Examples

```
> SIMP_CONV std_ss [] ``((l = []) ==> P l) /\ Q l``
val it = |- ((l = []) ==> P l) /\ Q l <=>
            ((l = []) ==> P []) /\ Q l

> SIMP_CONV arith_ss [] ``if (c /\ x < 5) then (P c /\ x < 6) else Q c``
val it = |- (if c /\ x < 5 then P c /\ x < 6 else Q c) <=>
            if c /\ x < 5 then P T else Q c:

> SIMP_CONV std_ss [] ``P x /\ (Q x /\ P x ==> Z x)``
Exception- UNCHANGED raised

> SIMP_CONV (std_ss++boolSimps.CONJ_ss) [] ``P x /\ (Q x /\ P x ==> Z x)``
val it = |- P x /\ (Q x /\ P x ==> Z x) <=> P x /\ (Q x ==> Z x)
```


Conditional Rewriting I

- perhaps the most powerful feature of the simplifier is that it supports conditional rewriting
- this means it allows **conditional** rewrite theorem of the form
  `|- cond ==> (t1 = t2)`
- if the simplifier finds a term `t1'` it can rewrite via `t1 = t2` to `t2'`, it tries to discharge the assumption `cond'`
- for this, it calls itself recursively on `cond'`
  - all the decision procedures and all context information is used
  - conditional rewriting can be used
  - to prevent divergence, there is a limit on recursion depth
- if `cond' = T` can be shown, `t1'` is rewritten to `t2'`
- otherwise `t1'` is not modified

Conditional Rewriting II

- conditional rewriting is a very powerful technique
- decision procedures and sophisticated rewrites can be used to discharge preconditions without cluttering proof state
- it provides a powerful search for theorems that apply
- however, if used naively, it can be slow
- moreover, to work well, rewrite theorems need to of a special form

Conditional Rewriting Example

- consider the conditional rewrite theorem
  `!l n. LENGTH l <= n ==> (DROP n l = [])`
- let's assume we want to prove
  `(DROP 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7]`
- we can without conditional rewriting
  - show `|- LENGTH [1;2;3;4] <= 7`
  - use this to discharge the precondition of the rewrite theorem
  - use the resulting theorem to rewrite the goal
- with conditional rewriting, this is all automated

```
> SIMP_CONV list_ss [DROP_LENGTH_TOO_LONG]
    ``(DROP 7 [1;2;3;4]) ++ [5;6;7]``
val it = |- DROP 7 [1; 2; 3; 4] ++ [5; 6; 7] = [5; 6; 7]
```

- conditional rewriting often shortens proofs considerably

Conditional Rewriting Pitfalls I

- if the pattern is too general, the simplifier becomes very slow
- consider the following, trivial but hopefully useful example

Looping example

```
> val my_thm = prove (``~P ==> (P = F)``, PROVE_TAC[])
> time (SIMP_CONV std_ss [my_thm]) ``P1 /\ P2 /\ P3 /\ ... /\ P10``
runtime: 0.84000s,    gctime: 0.02400s,     systime: 0.02400s.
Exception- UNCHANGED raised

> time (SIMP_CONV std_ss []) ``P1 /\ P2 /\ P3 /\ ... /\ P10``
runtime: 0.00000s,    gctime: 0.00000s,     systime: 0.00000s.
Exception- UNCHANGED raised
```

- notice that the rewrite is applied at plenty of places (quadratic in number of conjuncts)
- notice that each backchaining triggers many more backchainings
- each has to be aborted to prevent diverging
- as a result, the simplifier becomes very slow
- incidentally, the conditional rewrite is useless

Conditional Rewriting Pitfalls II

- good conditional rewrites `|- c ==> (l = r)` should mention only variables in `c` that appear in `l`
- if `c` contains extra variables `x1 ... xn`, the conditional rewrite engine has to search instantiations for them
- this mean that conditional rewriting is trying discharge the precondition `?x1 ... xn. c`
- the simplifier is usually not able to find such instances

Transitivity

```
> val P_def = Define `P x y = x < y`;
> val my_thm = prove (``!x y z. P x y ==> P y z ==> P x z``, ...)
> SIMP_CONV arith_ss [my_thm] ``P 2 3 /\ P 3 4 ==> P 2 4``
Exception- UNCHANGED raised

(* However transitivity of < build in via decision procedure *)
> SIMP_CONV arith_ss [P_def] ``P 2 3 /\ P 3 4 ==> P 2 4``
val it = |- P 2 3 /\ P 3 4 ==> P 2 4 <=> T:
```

Conditional vs. Unconditional Rewrite Rules

- conditional rewrite rules are often much more powerful
- however, `Rewrite` lib does not support them
- for this reason there are often two versions of rewrite theorems

`drop` example

- `DROP_LENGTH_NIL` is a useful rewrite rule:
  `|- !l. DROP (LENGTH l) l = []`
- in proofs, one needs to be careful though to preserve exactly this form
  - one should not (partly) evaluate `LENGTH l` or modify `l` somehow
- with the conditional rewrite rule `DROP_LENGTH_TOO_LONG` one does not need to be as careful
  `|- !l n. LENGTH l <= n ==> (DROP n l = [])`
  - the simplifier can use simplify the precondition using information about `LENGTH` and even arithmetic decision procedures

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 <=> (a, b, c) IN {(a, b, c) | R a b c}`
  - `(a, b, c) IN S <=> (\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 -> 'a -> bool` can be defined as the least relation `RTC R` that satisfies the following rules:

$$\frac{\texttt{R x y}}{\texttt{RTC R x y}} \qquad \frac{}{\texttt{RTC R x x}} \qquad \frac{\texttt{RTC R x y} \quad \texttt{RTC R y z}}{\texttt{RTC R x z}}$$

- 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)inductive Relations in HOL

- (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

Example: Transitive Reflexive Closure

```
> 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)
```


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)
```


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

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-derminating recursion
- `DT` is always true, i. e. `|- !n. DT n`
- `DF` is always false, i. e. `|- !n. ~(DF n)`

???

???