

Interactive Theorem Proving (ITP) Course

Part XIII

Thomas Tuerk (tuerk@kth.se)



Academic Year 2016/17, Period 4

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

Part XIII

Rewriting



- 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
built 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
 - ▶ ...

- we have seen primitive inference rules for equality before

$$\frac{\begin{array}{c} \Gamma \vdash s = t \\ \Delta \vdash u = v \\ \text{types fit} \end{array}}{\Gamma \cup \Delta \vdash s(u) = t(v)} \text{ COMB}$$

$$\frac{\begin{array}{c} \Gamma \vdash s = t \\ x \text{ not free in } \Gamma \end{array}}{\Gamma \vdash \lambda x. s = \lambda x. t} \text{ ABS}$$

$$\frac{\begin{array}{c} \Gamma \vdash s = t \\ \Delta \vdash t = u \end{array}}{\Gamma \cup \Delta \vdash s = u} \text{ TRANS}$$

$$\frac{}{\vdash t = t} \text{ REFL}$$

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

- in HOL, equality reasoning is implemented by **conversions**
- a conversion is a SML function of type `term -> thm`
- given a term `t`, a conversion
 - ▶ produces a theorem of the form `|- t = t'`
 - ▶ raises an **UNCHANGED** exception or
 - ▶ fails, i. e. raises 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
```

- 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
 - ▶ ...

- 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

- it remains to rewrite terms at top-level
- this is achieved by REWR_CONV
- given a term t and a theorem $\vdash t_1 = t_2$, REWR_CONV t thm
 - ▶ searches an instantiation of term and type variables such that t_1 becomes α -equivalent to t
 - ▶ fails, if no instantiation is found
 - ▶ otherwise, instantiate the theorem and get $\vdash t_1' = t_2'$
 - ▶ return theorem $\vdash t = t_2'$

Example

term LENGTH [1;2;3], theorem $\vdash \text{LENGTH } ((x:'a)::xs) = \text{SUC } (\text{LENGTH } xs)$
 found type instantiation: $[('a' \rightarrow 'num')]$
 found term instantiation: $[('x:num' \rightarrow '1'; 'xs' \rightarrow '[2;3]')]$
 returned theorem: $\vdash \text{LENGTH } [1;2;3] = \text{SUC } (\text{LENGTH } [2;3])$

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

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

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

t_org

```
LENGTH ((x:'a)::xs)
[]:'a list
0
b /\ T
b /\ b
b /\ b
!x:num. P x /\ Q x
!x:num. P x /\ Q x
!x:num. P x /\ Q x
```

t_goal

```
LENGTH [1;2;3]
[]:'b list
0
(P (x:'a) ==> Q) /\ T
P x /\ P x
P x /\ P y
!y:num. P' y /\ Q' y
!y. (2 = y) /\ Q' y
!y. (y = 2) /\ Q' y
```

subst

```
'a → num, x → 1, xs → [2;3]
'a → 'b
empty substitution
b → P x ==> Q
b → P x
fail
P → P', Q → Q'
P → ($= 2), Q → Q'
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.

- 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

$!x:\text{num}. P\ x \wedge Q\ x$
 $!x. P\ x \wedge Q\ x$
 $!x. P\ x \wedge Q$
 $!x. P\ (x, x)$
 $!x. P\ (x, x)$

t_goal

$!y. (y = 2) \wedge Q'\ y$
 $!x. P\ x \wedge Q\ x \wedge Z\ x$
 $!x. P\ x \wedge Q\ x$
 $!x. Q\ x$
 $!x. \text{FST}\ (x, x) = \text{SND}\ (x, x)$

substs

$P \rightarrow (\lambda y. y = 2), Q \rightarrow Q'$
 $Q \rightarrow \lambda x. Q\ x \wedge Z\ x$
fails
fails
 $P \rightarrow \lambda xx. \text{FST}\ xx = \text{SND}\ xx$

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

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

- 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`
 - ▶ ...

```
> 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])
```

```
> 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)
```


- 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

- 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

Theory

- choose well-founded order $\prec$
- for each rewrite theorem $\vdash t_1 = t_2$ ensure $t_2 \prec t_1$

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

- 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 = []) = F
 - ▶ !x xs. LENGTH (x::xs) = SUC (LENGTH xs)
 - ▶ !n x xs. DROP (SUC n) (x::xs) = DROP n xs

- 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 example
 - ▶ `| - !L. REVERSE L = REV L []`

- 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
 - ▶ $\vdash !f\ l.\ \text{MAP}\ f\ (\text{REVERSE}\ l) = \text{REVERSE}\ (\text{MAP}\ f\ l)$
 - ▶ $\vdash !l1\ l2\ l3.\ l1\ ++\ (l2\ ++\ l3) = l1\ ++\ l2\ ++\ l3$

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

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 $\mid-$ $\text{DOUBLE } n = n + n$ and $\mid-$ $\text{EVEN } (\text{DOUBLE } n) = T$.

With these the term $\text{EVEN } (\text{DOUBLE } 2)$ can be rewritten to

- T or
- $\text{EVEN } (2 + 2)$.

To avoid a hard to predict result, $\text{EVEN } (2+2)$ should be rewritten to T . Adding an extra rewrite rule $\mid-$ $\text{EVEN } (n + n) = T$ achieves this.

- 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` 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

- `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 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** 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

- `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`
- common usage: basic `simpset` combined with one or more `simpset-fragments`, e. g.
 - ▶ `list_ss ++ pairSimps.gen_beta_ss`
 - ▶ `std_ss ++ QI_ss`
 - ▶ ...

- 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

```
> 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
```


Current GoalStack

P (SUC (SUC x0)) (SUC (SUC y0))

0. SUC y1 = y2
1. x1 = SUC x0
2. y1 = SUC y0
3. SUC x1 = x2

Action

FULL_SIMP_TAC std_ss []

Resulting GoalStack

P (SUC (SUC x0)) y2

0. SUC (SUC y0) = y2
1. x1 = SUC x0
2. y1 = SUC y0
3. SUC x1 = x2

Current GoalStack

P (SUC (SUC x0)) y2

0. SUC (SUC y0) = y2
1. x1 = SUC x0
2. y1 = SUC y0
3. SUC x1 = x2

Action

REV_FULL_SIMP_TAC std_ss []

Resulting GoalStack

P x2 y2

0. SUC (SUC y0) = y2
1. x1 = SUC x0
2. y1 = SUC y0
3. SUC (SUC x0) = x2

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

- 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
- ...

- 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`

```
> 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
```

- 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)
```

- 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


```
> 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++boolSims.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)
```

- perhaps the most powerful feature of the simplifier is that it supports conditional rewriting
- this means it allows **conditional** rewrite theorems of the form
 $\text{|- cond} \implies (t1 = t2)$
- 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 $\text{cond}' = T$ can be shown, $t1'$ is rewritten to $t2'$
- otherwise $t1'$ is not modified

Conditional Rewriting Example



- consider the conditional rewrite theorem

`!1 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

Proof with Rewrite

```
prove ((' (DROP 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7] ',  
  'DROP 7 [1;2;3;4] = []' by (  
    MATCH_MP_TAC DROP_LENGTH_TOO_LONG >>  
    REWRITE_TAC[LENGTH] >>  
    DECIDE_TAC  
  ) >>  
  ASM_REWRITE_TAC[APPEND])
```

Proof with Simplifier

```
prove ((' (DROP 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7] ',  
  ASM_SIMP_TAC list_ss [])
```

- 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 be of a special form

- 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

- good conditional rewrites $\mid- c \implies (l = r)$ should mention only variables in c that appear in l
- if c contains extra variables $x_1 \dots x_n$, the conditional rewrite engine has to search instantiations for them
- this means that conditional rewriting is trying to discharge the precondition $?x_1 \dots x_n. c$
- 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 Rewriting Pitfalls III



- let's look in detail why `SIMP_CONV` did not make progress above

```
> set_trace "simplifier" 2;
> SIMP_CONV arith_ss [my_thm] 'P 2 3 /\ P 3 4 ==> P 2 4'
[468000]: more context: |- !x y z. P x y ==> P y z ==> P x z
[468000]: New rewrite: |- (?y. P x y /\ P y z) ==> (P x z <=> T)
...
[584000]: more context: [.] |- P 2 3 /\ P 3 4
[584000]: New rewrite: [.] |- P 2 3 <=> T
[584000]: New rewrite: [.] |- P 3 4 <=> T
[588000]: rewriting P 2 4 with |- (?y. P x y /\ P y z) ==> (P x z <=> T)
[588000]: trying to solve: ?y. P 2 y /\ P y 4
[588000]: rewriting P 2 y with |- (?y. P x y /\ P y z) ==> (P x z <=> T)
[592000]: trying to solve: ?y'. P 2 y' /\ P y' y
...
[596000]: looping - cut
...
[608000]: looping - stack limit reached
...
[640000]: couldn't solve: ?y. P 2 y /\ P y 4
Exception- UNCHANGED raised
```


- 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

- some theorems given in the list of rewrites to the simplifier are used for special purposes
- there are marked functions that mark these theorems
 - ▶ `Once : thm -> thm` use given theorem at most once
 - ▶ `Ntimes : thm -> int -> thm` use given theorem at most the given number of times
 - ▶ `AC : thm -> thm -> thm` use given associativity and commutativity theorems for AC rewriting
 - ▶ `Cong : thm -> thm` use given theorem as a congruence rule
- these special forms are easy ways to add this information to a simpset
- it can be directly set in a simpset as well

Example Once



```
> SIMP_CONV pure_ss [Once ADD_COMM] ``a + b = c + d``  
val it = |- (a + b = c + d) <=> (b + a = c + d)  
  
> SIMP_CONV pure_ss [Ntimes ADD_COMM 2] ``a + b = c + d``  
val it = |- (a + b = c + d) <=> (a + b = c + d)  
  
> SIMP_CONV pure_ss [ADD_COMM] ``a + b = c + d``  
Exception- UNCHANGED raised  
  
> ONCE_REWRITE_CONV [ADD_COMM] ``a + b = c + d``  
val it = |- (a + b = c + d) <=> (b + a = d + c)  
  
> REWRITE_CONV [ADD_COMM] ``a + b = c + d``  
... diverges ...
```

- the simpset `srw_ss()` is maintained by the system
 - ▶ it is automatically extended by new type-definitions
 - ▶ theories can extend it via `export_rewrites`
 - ▶ libs can augment it via `augment_srw_ss`
- the stateful simpset contains many rewrites
- it is very powerful and easy to use

Example

```
> SIMP_CONV (srw_ss()) [] ‘‘case [] of [] => (2 + 4)‘‘  
val it = |- (case [] of [] => 2 + 4 | v::v1 => ARB) = 6
```

- the stateful simpset is very powerful and easy to use
- however, results are hard to predict
- proofs using it unwisely are hard to maintain
- the stateful simpset can expand too much
 - ▶ bigger, harder to read proof states
 - ▶ high level arguments become hard to see
- whether to use the stateful simpset depends on personal proof style
- I advise at the beginning to not use `srw_ss`
- once you got a good intuition on how the simplifier works, make your own choice

- it is complicated to add arbitrary decision procedures to a simpset
- however, adding simple conversions is straightforward
- a conversion is described by a `stdconvdata` record

```
type stdconvdata = {  
  name: string,      (* name for debugging *)  
  pats: term list,   (* list of patterns, when to try conv *)  
  conv: conv         (* the conversion *)  
}
```

- use `std_conv_ss` to create simpset-fragment

Example

```
val WORD_ADD_ss =  
  simpLib.std_conv_ss  
    {conv = CHANGED_CONV WORD_ADD_CANON_CONV,  
     name = "WORD_ADD_CANON_CONV",  
     pats = ['words$word_add (w:'a word) y[']]}
```

- the simplifier is HOL's main workhorse for automation
- it is very powerful
- conditional rewriting very powerful
 - ▶ here only simple examples were presented
 - ▶ experiment with it to get a feeling
- many advanced features not discussed here at all
 - ▶ using congruence rules
 - ▶ writing own decision procedures
 - ▶ rewriting with respect to arbitrary congruence relations

Warning

The simplifier is very powerful. Make sure you understand it and are in control when using it. Otherwise your proofs easily become lengthy, convoluted and hard to maintain.