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/)
- Karl Palmskog (https://setoid.com)

Interactive Theorem Proving and Program Verification
Lecture 6

Pablo Buiras and Karl Palmskog

Academic Year 2019/20, Period 3–4

Based on slides by Thomas Tuerk

Part XIII

Rewriting

Rewriting in HOL4

- simplification via rewriting was already a strength of Edinburgh LCF
- it was further improved for Cambridge LCF
- HOL4 inherited this powerful rewriter
- equational reasoning is still the main workhorse
- there are many different equational reasoning tools in HOL4
  - `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, HOL4's main workhorse
    not discussed in this lecture, yet
  - ...

Semantic Foundations

- we have seen primitive inference rules for equality before

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

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

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

Conversions

- in HOL4, 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
```

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 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 ρ
  - term substitution σ
- such that $\text{subst}\,\sigma\,(\text{inst}\,\rho\,\texttt{t\_org}) \equiv_\alpha \texttt{t\_goal}$
- 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 → num, x → 1, xs → [2;3]
`[]:'a list`	`[]:'b list`	'a → 'b
`0`	`0`	empty substitution
`b /\ T`	`(P (x:'a) ==> Q) /\ T`	b → P x ==> Q
`b /\ b`	`P x /\ P x`	b → P x
`b /\ b`	`P x /\ P y`	fail
`!x:num. P x /\ Q x`	`!y:num. P' y /\ Q' y`	P → P', Q → Q'
`!x:num. P x /\ Q x`	`!y. (2 = y) /\ Q' y`	P → ($= 2), Q → Q'
`!x:num. P x /\ Q x`	`!y. (y = 2) /\ Q' y`	fail

- it is often very annoying that the last match in the list above fails
- it prevents us from 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 a substitution $\langle \sigma, \rho \rangle$ such that subst σ (inst ρ `t_org`) is α-equivalent to `t_goal`
- **higher order term matching** searches for a substitution $\langle \sigma, \rho \rangle$ such that subst σ (inst ρ `t_org`) and `t_goal` have α-equivalent $\beta\eta$-normal forms, i. e.
  if `t_subst` $=$ subst σ (inst ρ `t_org`), then
  `t_subst` $\downarrow_{\beta\eta} v_1 \wedge$ `t_goal` $\downarrow_{\beta\eta} v_2 \Rightarrow v_1 \equiv_\alpha v_2$

  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

Higher Order Term Matching II

- the HOL4 implementation expects `t_org` to be a **higher-order pattern**
  - `t_org` is in β-normal form
  - if `X a` is to be instantiated, then all occurrences of the bound variables in `a` have to appear in a subterm matching `a`
- for other forms of `t_org`, HOL4'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` → `(\y. y = 2)`, `Q` → `Q'`
`!x. P x /\ Q x`	`!x. P x /\ Q x /\ Z x`	`Q` → `\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` → `\xx. FST xx = SND xx`

Rewrite Library

- the rewrite library combines `REWR_CONV` with depth conversions
- there are many different conversions, rules and tactics
- at their 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])

> 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
- unluckily, it cannot be covered here in detail for time constraints
- 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 = []) = F`
  - `!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 example
  - `|- !L. REVERSE L = REV L []`

Termination — Normal forms

- some equations can be used in both directions
- one should decide on one direction
- this implicitly defines a **normal form** 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 sets 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 compete with each other
- **Confluence:** if a term `ta` can be rewritten to `ta1` and `ta2` applying different rewrite rules, then it should be possible to further rewrite `ta1` and `ta2` 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, normal forms are vital
- a term is in **normal form** if it cannot be rewritten any further
- one should have a clear idea what the normal form of common terms looks like
- all rules should work together to establish this normal form
- 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` or `TFL` libraries 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 HOL4'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
- common usage: basic simpset combined with one or more simpset-fragments, e. g.
  - `list_ss ++ pairSimps.gen_beta_ss`
  - `std_ss ++ QI_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
```

FULL_SIMP_TAC Example

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
```

REV_FULL_SIMP_TAC Example

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
```

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
- `LIFT_COND_ss` — lifting if-then-else
- …

Build-In Conversions and Decision Procedures

- in contrast to `Rewrite` lib the simplifier can run arbitrary conversions
- most common and useful conversion 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
  - e. g. by using `CONJ_ss` one can easily use context of conjunctions
  - warning: using `CONJ_ss` can be slow
- 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 theorems 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 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 Example II

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]``,
SIMP_TAC list_ss [])
```

Notice that `DROP_LENGTH_TOO_LONG` is part of `list_ss`.

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 Pitfalls I

- if the pattern is too general, the simplifier becomes very slow
- consider the following, trivial but hopefully educational 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 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 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 simplify the precondition using information about `LENGTH` and even arithmetic decision procedures

Special Rewrite Forms

- some theorems given in the list of rewrites to the simplifier are used for special purposes
- there are marking 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 ...
```

Stateful Simpset

- 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
```

Discussion on Stateful Simpset

- 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
- We advise to not use `srw_ss` at the beginning
- once you get a good intuition of how the simplifier works, make your own choice

Adding Own Conversions

- 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-fragement

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``]}
```

Summary Simplifier

- the simplifier is HOL4's main workhorse for automation
- 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.