Interactive Theorem Proving (ITP) Course Part XIII

Thomas Tuerk (tuerk@kth.se)



Academic Year 2016/17, Period 4

version acf88f7 of Mon May 22 13:40:56 2017

Rewriting in HOL

▶ ...

- ${\scriptstyle \bullet} \,$ 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

Part XIII

Rewriting



• we have seen primitive inference rules for equality before

Semantic Foundations

$$\begin{array}{l}
 \Gamma \vdash s = t \\
 \Delta \vdash u = v \\
 types fit \\
 \overline{\Gamma \cup \Delta \vdash s(u) = t(v)} \end{array} \quad COMB \quad \qquad \begin{array}{l}
 \Gamma \vdash s = t \\
 \frac{x \text{ not free in } \Gamma}{\Gamma \vdash \lambda x. \ s = \lambda x. \ t} \end{array} ABS \\
 \begin{array}{l}
 \Gamma \vdash s = t \\
 \frac{\Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u} \\
 \overline{\Gamma \cup \Delta \vdash s = u} \end{array} TRANS \quad \qquad \begin{array}{l}
 \overline{\vdash t = t} \end{array} REFL
 \end{array}$$

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

KTH

KTH

Conversions



200 / 240

KTH

• in HOL, equality reasoning is implemented by conversions

- a conversion is a SML function of type term \rightarrow 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

Conversionals

- ► 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



t_org

0

[]:'a list

Examples Term Matching

LENGTH ((x:'a)::xs)



 ${\tilde{ \bullet}}$ 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) $\stackrel{\alpha}{\equiv}$ 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

b /\ T (P (x:'a) ==> Q) /\ T $b \rightarrow P x => Q$ b /\ b P x /\ P x $b \rightarrow P x$ b /\ b P x /\ P y fail !x:num. P x /\ Q x !y:num. P' y /\ Q' y $P \rightarrow P', Q \rightarrow Q'$!x:num. P x /\ Q x !y. (2 = y) / Q' y $P \rightarrow$ (\$= 2), $Q \rightarrow Q'$!x:num. P x /\ Q x !y. (y = 2) / Q' yfail

substs

'a \rightarrow 'b

empty substitution

'a \rightarrow num, x \rightarrow 1, xs \rightarrow [2;3]

• it is often very annoying that the last match fails

t_goal

0

LENGTH [1;2;3]

[]:'b list

- o 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.

204 / 240

KTH S

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



205 / 240

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



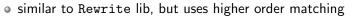
208 / 240



- 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
 - \star split conjunctions
 - \star remove outermost universal quantification
 - \star 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

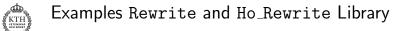
- REWRITE_CONV
- REWRITE_RULE
- REWRITE_TAC
- ASM_REWRITE_TAC
- ONCE_REWRITE_TAC
- PURE_REWRITE_TAC
- PURE_ONCE_REWRITE_TAC
- . . .

Ho_Rewrite Library



- 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)



Summary Rewrite and Ho_Rewrite Library



Term Rewriting Systems



213 / 240

KTH

- 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

Term Rewriting Systems — Termination



212 / 240

Termination — Subterm examples

• a proper subterm is always simpler

- ▶ !1. APPEND [] 1 = 1
- ▶ !n. n + 0 = n
- ▶ !1. REVERSE (REVERSE 1) = 1
- \blacktriangleright !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

Theory

- $\, \circ \,$ choose well-founded order $\, \prec \,$
- $\,$ o 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 — use simpler terms





217 / 240

KTH

- 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
 - ► |- !f 1. MAP f (REVERSE 1) = REVERSE (MAP f 1)
 - ▶ |- !11 12 13. 11 ++ (12 ++ 13) = 11 ++ 12 ++ 13

216 / 240

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)

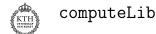
Rewrites working together

- 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





221 / 240

KTH

- ${\ensuremath{\, \bullet }}$ 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

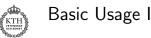
- computeLib uses compsets to store its rewrites
- a compset stores
 - rewrite rules
 - extra conversions
- the extra conversions are guarded by a term pattern for efficiency
- users can define their own compsets
- however, computeLib maintains one special compset called the_compset
- the_compset is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



220 / 240

- 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
 - ▶ ...

Basic Usage II

• 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
- most important simpset is std_ss

224 / 240

KTH

Basic Simplifier Examples



225 / 240

- 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

Common simpsets



Common simpset-fragments



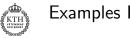
- pure_ss empty simpset
- bool_ss basic simpset
- std_ss standard simpset
- $arith_ss$ arithmetic simpset
- list_ss list simpset
- real_ss real simpset

- 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
- . . .

228 / 240

Build-In Conversions and Decision Procedures

procedure you already know from DECIDE

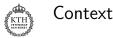


```
• in contrast to Rewrite lib the simplifier can run arbitrary conversions
                                                                             > SIMP_CONV std_ss [] ``(\x. x + 2) 5``
                                                                              val it = |-(x, x + 2) = 7
• most useful is probably \beta reduction
• std_ss has support for basic arithmetic and numerals
                                                                             > SIMP_CONV std_ss [] ''!x. Q x /\ (x = 7) ==> P x''
                                                                              val it = |- (!x. Q x /\ (x = 7) ==> P x) <=> (Q 7 ==> P 7)''
• it also has simple, syntactic conversions for instantiating quantifiers
    ▶ !x. ... /\ (x = c) /\ ... ==> ...
                                                                             > SIMP_CONV std_ss [] ''?x. Q x /\ (x = 7) /\ P x''
                                                                              val it = |- (?x. Q x /\ (x = 7) /\ P x) <=> (Q 7 /\ P 7)''
    ▶ !x. ... \/ ~(x = c) \/ ...
    ▶ ?x. ... /\ (x = c) /\ ...
                                                                             > SIMP_CONV std_ss [] ''x > 7 ==> x > 5''
                                                                              Exception- UNCHANGED raised
• besides very useful conversions, there are decision procedures as well
• the most frequently used one is probably the arithmetic decision
                                                                             > 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 [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





KTH



232 / 240

Context Examples

- > 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)

- 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)

Conditional Rewriting I

- 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



236 / 240

KTH

Conditional Rewriting Example



- 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

- consider the conditional rewrite theorem
 !1 n. LENGTH 1 <= n ==> (DROP n 1 = [])
- 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
 - \blacktriangleright 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
- ${\scriptstyle \bullet}$ 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



237 / 240

- good conditional rewrites |- c ==> (l = r) should mention only variables in c that appear in l
- $\bullet\,$ if c contains extra variables x1 $\ldots\,$ 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:
 - |- !1. DROP (LENGTH 1) 1 = []
- ${\ensuremath{\,\circ\,}}$ in proofs, one needs to be careful though to preserve exactly this form
 - $\,\triangleright\,$ one should not (partly) evaluate LENGTH 1 or modify 1 somehow
- with the conditional rewrite rule DROP_LENGTH_TOO_LONG one does not need to be as careful
 - |- !1 n. LENGTH 1 <= n ==> (DROP n 1 = [])
 - the simplifier can use simplify the precondition using information about LENGTH and even arithmetic decision procedures