Interactive Theorem Proving in HOL4

Course 04: Conversions

Dr Chun TIAN
`chun.tian@anu.edu.au`

22 August 2024

Australian National University

Acknowledgement of Country

We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present.

More information about Acknowledgement of Country can be found here and here

Conversions

- A *conversion* is an SML function of the type "`term -> thm`" (aka `conv`) taking a term t and returning an equational theorem of the form $\vdash t = t'$.
- Each conversion represent infinite number of equations.
- Conversions are mainly used for implementing rewriting tools.
- Multiple conversions can be combined to form more powerful or customised conversions.

Example: β-conversion

$$(\lambda x.\, t_1)t_2 \quad \mapsto \quad \vdash (\lambda x.\, t_1)t_2 = t_1[x \mapsto t_2]$$

```
> BETA_CONV;
val it = fn: term -> thm
> BETA_CONV ``(\y. (\z. 1 + (y + z)) 3) 2``;
val it = |- (\y. (\z. 1 + (y + z)) 3) 2 = (\z. 1 + (2 + z)) 3: thm
```

Conversion Combining Operators

A term u is said to *reduce* to a term v by a conversion c if there exists a finite sequence of terms $t_1, t_2, \ldots, t_n$ such that:

1. $u = t_1$ and $v = t_n$;
2. $c\ t_i$ evaluates to the theorem $\vdash t_i = t_{i+1}$ for $1 \leqslant i < n$;
3. The evaluation of $c\ t_n$ fails (by throwing SML exception `UNCHANGED`).

Basic conversion combining operators

- `op THENC : conv -> conv -> conv`
  If $c_1 t_1 \mapsto \Gamma_1 \vdash t_1 = t_2$ and $c_2 t_2 \mapsto \Gamma_2 \vdash t_2 = t_3$, then $(c_1\ \mathtt{THENC}\ c_2) \mapsto \Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3$.
- `op ORELSEC : conv -> conv -> conv`
  $(c_1\ \mathtt{ORELSEC}\ c_2)\ t \mapsto c_1\ t$ if that evaluation succeeds, and $\mapsto c_2\ t$ otherwise.
- `REPEATC : conv -> conv` makes a conversion repeatedly applied until it fails.

```
fun REPEATC c t = ((c THENC (REPEATC c)) ORELSEC ALL_CONV) t
```

Conversions on Subterms

- `DEPTH_CONV : conv -> conv` makes a conversion apply to every subterm of the input term.
- `TOP_DEPTH_CONV : conv -> conv` does more than `DEPTH_CONV` by recursively applying the input conversion on intermediate outputs:

```
> val t0 = ``(((\x. (\y. (\z. x + y + z))) 1) 2) 3``;
> REPEATC BETA_CONV t0;
Exception- UNCHANGED raised
> DEPTH_CONV BETA_CONV t0;
val it = |- (\x y z. x + y + z) 1 2 3 = 1 + 2 + 3: thm
> val t = ``(\f. \x. f x) (\n. n + 1)``;
> DEPTH_CONV BETA_CONV t;
val it = |- (\f x. f x) (\n. n + 1) = (\x. (\n. n + 1) x): thm
> TOP_DEPTH_CONV BETA_CONV t;
val it = |- (\f x. f x) (\n. n + 1) = (\x. x + 1): thm
```

Precise Control Over Subterms (1)

- `ONCE_DEPTH_CONV : conv -> conv` applies a conversion *once* to the first subterm (and only the first subterm) on which it succeeds in a top-down traversal.
- `SUB_CONV : conv -> conv` applies a conversion to the immediate subterms of a term.
- `RATOR_CONV : conv -> conv` converts the operator (the t of $t\,t'$) of an application.
- `RAND_CONV : conv -> conv` converts the operand (the t' of $t\,t'$) of an application.
- `ABS_CONV : conv -> conv` converts the body of an abstraction.

```
> val t = ``(\x. x + 1) m + (\x. x + 2) n``;
> dest_comb t;
val it = (``$+ ((\x. x + 1) m)``, ``(\x. x + 2) n``): term * term
> RAND_CONV BETA_CONV t;
val it = |- (\x. x + 1) m + (\x. x + 2) n = (\x. x + 1) m + (n + 2): thm
> RATOR_CONV (RAND_CONV BETA_CONV) t;
val it = |- (\x. x + 1) m + (\x. x + 2) n = m + 1 + (\x. x + 2) n: thm
```

Precise Control Over Subterms (2)

`GEN_REWRITE_CONV : (conv -> conv) -> rewrites -> thm list -> conv`
rewrites a term, selecting terms according to a user-specified strategy.

```
> open arithmeticTheory;
> val t = ``(1 + 2) + 3 = (3 + 1) + 2``;
> ADD_SYM;
val it = |- !m n. m + n = n + m: thm
> GEN_REWRITE_CONV (RATOR_CONV o ONCE_DEPTH_CONV) empty_rewrites [ADD_SYM] t;
val it = |- 1 + 2 + 3 = 3 + 1 + 2 <=> 3 + (1 + 2) = 3 + 1 + 2: thm
> GEN_REWRITE_CONV (RAND_CONV o ONCE_DEPTH_CONV) empty_rewrites [ADD_SYM] t;
val it = |- 1 + 2 + 3 = 3 + 1 + 2 <=> 1 + 2 + 3 = 2 + (3 + 1): thm
```

NOTE: For simplier rewriting purposes there are easy conversions like `REWRITE_TAC`, etc.

Tools for Writing Compound Conversions

Besides THENC, ORELSEC and REPEATC

- NO_CONV always fails.
- FIRST_CONV return $c\ t$ for the first successful conversion c in a list of conversions.
- ALL_CONV always success (ALL_CONV t evaluates to $\vdash t = t$, i.e. UNCHANGED).
- EVERY_CONV : conv list -> conv applies a list of conversion in sequence.
- TRY_CONV :conv -> conv tries to apply a conversion and (if failed) returns UNCHANGED.

```
fun FIRST_CONV [] tm = NO_CONV tm
  | FIRST_CONV (c :: rst) tm =
      c tm handle HOL_ERR _ => FIRST_CONV rst tm

fun TRY_CONV c = c ORELSEC ALL_CONV
```

Quantifier Movements Conversions

```
FORALL_AND_CONV        |- (!x. P /\ Q) = (!x.P) /\ (!x.Q)
AND_FORALL_CONV        |- (!x.P) /\ (!x.Q) = (!x. P /\ Q)
LEFT_AND_FORALL_CONV   |- (!x.P) /\ Q = (!x'. P[x'/x] /\ Q)
RIGHT_AND_FORALL_CONV  |- P /\ (!x.Q) = (!x'. P /\ Q[x'/x])

EXISTS_OR_CONV         |- (?x. P \/ Q) = (?x.P) \/ (?x.Q)
OR_EXISTS_CONV         |- (?x.P) \/ (?x.Q) = (?x. P \/ Q)
LEFT_OR_EXISTS_CONV    |- (?x.P) \/ Q = (?x'. P[x'/x] \/ Q)
RIGHT_OR_EXISTS_CONV   |- P \/ (?x.Q) = (?x'. P \/ Q[x'/x])

LEFT_OR_FORALL_CONV    |- (!x.P) \/ Q = !x'. P[x'/x] \/ Q
RIGHT_OR_FORALL_CONV   |- P \/ (!x.Q) = !x'. P \/ Q[x'/x]

LEFT_AND_EXISTS_CONV   |- (?x.P) /\ Q = ?x'. P[x'/x] /\ Q
RIGHT_AND_EXISTS_CONV  |- P /\ (?x.Q) = ?x'. P /\ Q[x'/x]
```

Conversions for Normal Forms

Several conversions for translating to Boolean normal forms are provided by `normalForms`:

```
> load "normalForms"; open normalForms;
> NNF_CONV ``(!x. P x) ==> ((?y. Q y) = ?z. P z /\ Q z)``;
val it =
   |- (!x. P x) ==> ((?y. Q y) <=> ?z. P z /\ Q z) <=>
      ((?y. Q y) \/ !z. ~P z \/ ~Q z) /\ ((!y. ~Q y) \/ ?z. P z /\ Q z) \/
      ?x. ~P x: thm
> CNF_CONV ``(!x. P x ==> ?y z. Q y \/ ~?z. P z \/ Q z)``;
val it =
   |- (!x. P x ==> ?y z. Q y \/ ~?z. P z \/ Q z) <=>
      ?y. !x z. (~P z \/ Q (y x) \/ ~P x) /\ (~Q z \/ Q (y x) \/ ~P x): thm
> DNF_CONV ``(!x. P x ==> ?y z. Q y \/ ~?z. P z \/ Q z)``;
val it =
   |- (!x. P x ==> ?y z. Q y \/ ~?z. P z \/ Q z) <=>
      !x z. ?y. ~P x \/ Q y \/ ~P (z y) /\ ~Q (z y): thm
```

Basic Rewriting Tools

```
REWRITE_CONV           : thm list -> conv
PURE_REWRITE_CONV      : thm list -> conv
ONCE_REWRITE_CONV      : thm list -> conv
PURE_ONCE_REWRITE_CONV : thm list -> conv
GEN_REWRITE_CONV       : (conv -> conv) -> rewrites -> thm list -> conv
```

- All versions deal with subterms (based on `TOP_DEPTH_CONV`);
- The "pure" version does not use any built-in (mostly Boolean related) rewriting rules;
- The "once" version rewrites only once and quits (based on `ONCE_DEPTH_CONV`).

```
> ONCE_REWRITE_CONV [ADD_SYM] ``1 + 2 + 3``;
val it = |- 1 + 2 + 3 = 3 + (1 + 2): thm
> REWRITE_CONV [Once ADD_SYM] ``1 + 2 + 3``;
val it = |- 1 + 2 + 3 = 3 + (1 + 2): thm
```

Conversions and Rules

`CONV_RULE : conv -> thm -> thm` converts a conversion to a corresponding *rule*, which translates one theorem into another theorem.

```
fun CONV_RULE c th = EQ_MP (c (concl th)) th
```

Built-in rewriting rules

```
REWRITE_RULE           : thm list -> thm -> thm
PURE_REWRITE_RULE      : thm list -> thm -> thm
ONCE_REWRITE_RULE      : thm list -> thm -> thm
PURE_ONCE_REWRITE_RULE : thm list -> thm -> thm
GEN_REWRITE_RULE       : (conv -> conv) -> rewrites -> thm list -> thm -> thm
```

NOTE: `CONV_RULE` is necessary for using Quantifier Movements Conversions like `FORALL_AND_CONV` as rules.

Conversions as Decision Procedures

A conversion for Boolean term is a decision procedure if it converts it to `T` or `F`.

```
> load "numLib"; open numLib;
> ARITH_CONV ``m < SUC m``;
val it = |- m < SUC m <=> T: thm
> ARITH_CONV ``?m n. m < n``;
val it = |- (?m n. m < n) <=> T: thm
> ARITH_CONV
  ``((p + 3) <= n) ==> (!m. (if (m EXP 2 = 0) then (n - 1) else (n - 2)) > p)``;
val it =
   |- p + 3 <= n ==> (!m. (if m ** 2 = 0 then n - 1 else n - 2) > p) <=> T:
   thm
```

NOTE: `ARITH_CONV` is a partial decision procedure for Presburger natural arithmetic. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the relations $<, \leqslant, =, \geqslant, >$ and the logical connectives $\neg, \wedge, \vee, \Rightarrow, =$ (if-and-only-if), $\forall$ and $\exists$.

Next Course: Goal Directed Proof

Now we are ready to learn writing formal proofs using *tactics* and *tacticals*.

```
Theorem xxx :
    !n. (n = 0) ==> (n * n = n)
Proof
    RW_TAC arith_ss [] (* or: rw [] *)
QED
```