The HOL reduce Library

J. R. Harrison

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

June 1991

Contents

Chapter 1

The reduce Library

This manual describes the use of the `reduce` library, as well as discussing its design. The library is intended to ease the burden of performing tedious and intellectually trivial tasks of arithmetic such as proving:

```
|- 2 EXP 6 = 64
```

Anyone trying to prove the above by hand will testify to its extreme tediousness. However, using the `reduce` library, the evaluation of:

```
REDUCE_CONV "2 EXP 6"
```

will perform the above automatically, and probably in far fewer primitive inferences than a human would take. The library will also take care of certain boolean expressions. This is mainly for the sake of completeness, since the same effect can be achieved by careful use of rewriting.

1.1 The representation of numbers

The approach to representing natural number constants in HOL is to provide a conversion `num_CONV` which generates the definition of any nonzero numeral in terms of its predecessor, for example:

```
#num_CONV "1";;
|- 1 = SUC 0

#num_CONV "256";;
|- 256 = SUC 255
```

This conversion uses `mk_thm`, so could be regarded as unsatisfactory; however it is arguably no worse than expanding a constant defined through the normal constant definition mechanism.

The `reduce` library uses only the above conversion, together with certain definitions and preproved theorems concerning the various arithmetic and boolean operators, to derive, strictly by inference, reduction theorems for certain expressions.

1.2 Sample algorithm

The following is an example of an algorithm for reducing addition expressions. In fact, for reasons of efficiency, the library's `ADD_CONV` conversion is implemented in a slightly more sophisticated way, but this example gives the general flavour of how the various arithmetic conversions are defined. Suppose we want to apply the conversion to `"m + n"` where `m` and `n` are both numerals. Then

- If the first numeral is zero, we need only specialize the first conjunct of the inbuilt definition `ADD`:

  ```
  SPEC "n" (CONJUNCT1 ADD)
  ```

- If the first numeral is not zero, then we call the conversion recursively to give:

  ```
  |- p + n = s
  ```

  where `p` is the predecessor of `m`, and `s` the corresponding sum. Now we can apply the following inferences:

  ```
  |- p + n = s                 % [1] From recursive call             %
  |- (SUC p) + n = SUC(p + n)  % [2] Instance of 2nd conjunct of ADD %
  |- (SUC p) + n = SUC s       % [3] Substituting [1] in [2]         %
  |- SUC p = m                 % [4] SYM (num_CONV m)                %
  |- SUC s = t                 % [5] SYM (num_CONV (s+1))            %
  |- m + n = t                 % [6] Substituting [4] and [5] in [6] %
  ```

  This gives the desired result. The above can easily be converted into a simple recursive procedure.

1.3 Using the library

The straightforward way to use the library is simply to do:

```
load_library `reduce`;;
```

This will work only if the HOL being used has been installed correctly; if this is not the case the user should refer to the *TUTORIAL* or the *DESCRIPTION*.

The library provides the following three ML bindings, which should be all that is needed for most purposes:

- `REDUCE_CONV` is a conversion which, when given an expression, will return a theorem expressing its equivalence with a reduced version, which in many cases will be simply a single numeral or boolean literal. For example:

```
#REDUCE_CONV "(50 < 51) => (2 * 25) | (60 - 17)";;
|- (50 < 51 => 2 * 25 | 60 - 17) = 50

#REDUCE_CONV "(3 * x) + (1 + 2)";;
|- (3 * x) + (1 + 2) = (3 * x) + 3

#REDUCE_CONV "(1 < 2) /\ (2 <= 2)";;
|- 1 < 2 /\ 2 <= 2 = T
```

- `REDUCE_RULE` is a rule which applies the reductions corresponding to `REDUCE_CONV` to subterms of the theorem's conclusion

- `REDUCE_TAC` performs the same reductions on a goal.

For more sophisticated use, there are conversions specific to each operator, for example `ADD_CONV` and `OR_CONV`. For more details of these, refer to the next chapter. The arithmetic conversions and boolean conversions may be loaded separately by loading (with `loadt` for instance) `arithconv.ml` and `boolconv.ml` respectively, in the main library directory.

Chapter 2

ML Functions in the reduce Library

This chapter provides documentation on all the ML functions that are made available in HOL when the `reduce` library is loaded. This documentation is also available online via the `help` facility.

ADD_CONV

```
ADD_CONV : conv
```

Synopsis

Calculates by inference the sum of two numerals.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `ADD_CONV "m + n"` returns the theorem:

```
   |- m + n = s
```

where `s` is the numeral that denotes the sum of the natural numbers denoted by `m` and `n`.

Failure

`ADD_CONV tm` fails unless `tm` is of the form `"m + n"`, where `m` and `n` are numerals.

Example

```
#ADD_CONV "75 + 25";;
|- 75 + 25 = 100
```

AND_CONV

```
AND_CONV : conv
```

Synopsis
Simplifies certain boolean conjunction expressions.

Description
If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `AND_CONV tm` returns the corresponding theorem. Note that in the last case the conjuncts need only be alpha-equivalent rather than strictly identical.

```
AND_CONV "T /\ t" = |- T /\ t = t
AND_CONV "t /\ T" = |- t /\ T = t
AND_CONV "F /\ t" = |- F /\ t = F
AND_CONV "t /\ F" = |- t /\ F = F
AND_CONV "t /\ t" = |- t /\ t = t
```

Failure
`AND_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#AND_CONV "(x = T) /\ F";;
|- (x = T) /\ F = F

#AND_CONV "T /\ (x = T)";;
|- T /\ (x = T) = (x = T)

#AND_CONV "(?x. x=T) /\ (?y. y=T)";;
|- (?x. x = T) /\ (?y. y = T) = (?x. x = T)
```

BEQ_CONV

```
BEQ_CONV : conv
```

Synopsis
Simplifies certain expressions involving boolean equality.

Description
If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `BEQ_CONV tm` returns the corresponding theorem. Note that in the last case the

left-hand and right-hand sides need only be alpha-equivalent rather than strictly identical.

```
BEQ_CONV "T = t" = |- T = t = t
BEQ_CONV "t = T" = |- t = T = t
BEQ_CONV "F = t" = |- F = t = ~t
BEQ_CONV "t = F" = |- t = F = ~t
BEQ_CONV "t = t" = |- t = t = T
```

Failure

BEQ_CONV tm fails unless tm has one of the forms indicated above.

Example

```
#BEQ_CONV "T = T";;
|- (T = T) = T

#BEQ_CONV "F = T";;
|- (F = T) = F

#BEQ_CONV "(!x:*#**. x = (FST x,SND x)) = (!y:*#**. y = (FST y,SND y))";;
|- ((!x. x = FST x,SND x) = (!y. y = FST y,SND y)) = T
```

COND_CONV

```
COND_CONV : conv
```

Synopsis

Simplifies certain conditional expressions.

Description

If tm corresponds to one of the forms given below, where b has type bool and t1 and t2 have the same type, then COND_CONV tm returns the corresponding theorem. Note that in the last case the arms need only be alpha-equivalent rather than strictly identical.

```
COND_CONV "F => t1 | t2" = |- (T => t1 | t2) = t2
COND_CONV "T => t1 | t2" = |- (T => t1 | t2) = t1
COND_CONV "b => t | t    = |- (b => t | t) = t
```

Failure

COND_CONV tm fails unless tm has one of the forms indicated above.

Example

```
#COND_CONV "F => F | T";;
|- (F => F | T) = T

#COND_CONV "T => F | T";;
|- (T => F | T) = F

#COND_CONV "b => (\x. SUC x) | (\p. SUC p)";;
|- (b => (\x. SUC x) | (\p. SUC p)) = (\x. SUC x)
```

DIV_CONV

```
DIV_CONV : conv
```

Synopsis

Calculates by inference the result of dividing, with truncation, one numeral by another.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `DIV_CONV "m DIV n"` returns the theorem:

```
|- m DIV n = s
```

where `s` is the numeral that denotes the result of dividing the natural number denoted by `m` by the natural number denoted by `n`, with truncation.

Failure

`DIV_CONV tm` fails unless `tm` is of the form `"m DIV n"`, where `m` and `n` are numerals, or if `n` denotes zero.

Example

```
#DIV_CONV "0 DIV 0";;
evaluation failed     DIV_CONV

#DIV_CONV "0 DIV 12";;
|- 0 DIV 12 = 0

#DIV_CONV "2 DIV 0";;
evaluation failed     DIV_CONV

#DIV_CONV "144 DIV 12";;
|- 144 DIV 12 = 12

#DIV_CONV "7 DIV 2";;
|- 7 DIV 2 = 3
```

EXP_CONV

```
EXP_CONV : conv
```

Synopsis

Calculates by inference the result of raising one numeral to the power of another.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `EXP_CONV "m EXP n"` returns the theorem:

```
   |- m EXP n = s
```

where `s` is the numeral that denotes the result of raising the natural number denoted by `m` to the power of the natural number denoted by `n`.

Failure

`EXP_CONV tm` fails unless `tm` is of the form `"m EXP n"`, where `m` and `n` are numerals.

Example

```
#EXP_CONV "0 EXP 0";;
|- 0 EXP 0 = 1

#EXP_CONV "15 EXP 0";;
|- 15 EXP 0 = 1

#EXP_CONV "12 EXP 1";;
|- 12 EXP 1 = 12

#EXP_CONV "2 EXP 6";;
|- 2 EXP 6 = 64
```

GE_CONV

```
GE_CONV : conv
```

Synopsis

Proves result of less-than-or-equal-to ordering on two numerals.

Description

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `GE_CONV "m >= n"` returns the theorem:

```
|- (m >= n) = T
```

if the natural number denoted by `m` is greater than or equal to that denoted by `n`, or

```
|- (m >= n) = F
```

otherwise.

Failure

`GE_CONV tm` fails unless `tm` is of the form `"m >= n"`, where `m` and `n` are numerals.

Example

```
#GE_CONV "15 >= 14";;
|- 15 >= 14 = T

#GE_CONV "100 >= 100";;
|- 100 >= 100 = T

#GE_CONV "0 >= 107";;
|- 0 >= 107 = F
```

GT_CONV

```
GT_CONV : conv
```

Synopsis

Proves result of greater-than ordering on two numerals.

Description

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `GT_CONV "m > n"` returns the theorem:

```
|- (m > n) = T
```

if the natural number denoted by `m` is greater than that denoted by `n`, or

```
|- (m > n) = F
```

otherwise.

Failure

`GT_CONV tm` fails unless `tm` is of the form `"m > n"`, where `m` and `n` are numerals.

Example

```
#GT_CONV "100 > 10";;
|- 100 > 10 = T

#GT_CONV "15 > 15";;
|- 15 > 15 = F

#GT_CONV "11 > 27";;
|- 11 > 27 = F
```

IMP_CONV

```
IMP_CONV : conv
```

Synopsis

Simplifies certain implicational expressions.

Description

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `IMP_CONV tm` returns the corresponding theorem. Note that in the last case the antecedent and consequent need only be alpha-equivalent rather than strictly identical.

```
IMP_CONV "T ==> t" = |- T ==> t = t
IMP_CONV "t ==> T" = |- t ==> T = T
IMP_CONV "F ==> t" = |- F ==> t = T
IMP_CONV "t ==> F" = |- t ==> F = ~t
IMP_CONV "t ==> t" = |- t ==> t = T
```

Failure

`IMP_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#IMP_CONV "T ==> F";;
|- T ==> F = F

#IMP_CONV "F ==> x";;
|- F ==> x = T

#IMP_CONV "(!z:(num)list. z = z) ==> (!x:(num)list. x = x)";;
|- (!z. z = z) ==> (!x. x = x) = T
```

LE_CONV

```
LE_CONV : conv
```

Synopsis

Proves result of less-than-or-equal-to ordering on two numerals.

Description

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `LE_CONV "m <= n"` returns the theorem:

```
|- (m <= n) = T
```

if the natural number denoted by `m` is less than or equal to that denoted by `n`, or

```
|- (m <= n) = F
```

otherwise.

Failure

`LE_CONV tm` fails unless `tm` is of the form `"m <= n"`, where `m` and `n` are numerals.

Example

```
#LE_CONV "12 <= 198";;
|- 12 <= 198 = T

#LE_CONV "46 <= 46";;
|- 46 <= 46 = T

#LE_CONV "13 <= 12";;
|- 13 <= 12 = F
```

LT_CONV

```
LT_CONV : conv
```

Synopsis

Proves result of less-than ordering on two numerals.

Description

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `LT_CONV "m < n"` returns the theorem:

```
|- (m < n) = T
```

if the natural number denoted by `m` is less than that denoted by `n`, or

```
|- (m < n) = F
```

otherwise.

Failure

`LT_CONV tm` fails unless `tm` is of the form `"m < n"`, where `m` and `n` are numerals.

Example

```
#LT_CONV "0 < 12";;
|- 0 < 12 = T

#LT_CONV "13 < 13";;
|- 13 < 13 = F

#LT_CONV "25 < 12";;
|- 25 < 12 = F
```

MOD_CONV

```
MOD_CONV : conv
```

Synopsis

Calculates by inference the remainder after dividing one numeral by another.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `MOD_CONV "m MOD n"` returns the theorem:

```
   |- m MOD n = s
```

where `s` is the numeral that denotes the remainder after dividing, with truncation, the natural number denoted by `m` by the natural number denoted by `n`.

Failure

`MOD_CONV tm` fails unless `tm` is of the form `"m MOD n"`, where `m` and `n` are numerals, or if `n` denotes zero.

Example

```
#MOD_CONV "0 MOD 0";;
evaluation failed     MOD_CONV

#MOD_CONV "0 MOD 12";;
|- 0 MOD 12 = 0

#MOD_CONV "2 MOD 0";;
evaluation failed     MOD_CONV

#MOD_CONV "144 MOD 12";;
|- 144 MOD 12 = 0

#MOD_CONV "7 MOD 2";;
|- 7 MOD 2 = 1
```

MUL_CONV

```
MUL_CONV : conv
```

Synopsis

Calculates by inference the product of two numerals.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `MUL_CONV "m * n"` returns the theorem:

```
|- m * n = s
```

where `s` is the numeral that denotes the product of the natural numbers denoted by `m` and `n`.

Failure

`MUL_CONV tm` fails unless `tm` is of the form `"m * n"`, where `m` and `n` are numerals.

Example

```
#MUL_CONV "0 * 12";;
|- 0 * 12 = 0

#MUL_CONV "1 * 1";;
|- 1 * 1 = 1

#MUL_CONV "6 * 11";;
|- 6 * 11 = 66
```

NEQ_CONV

```
NEQ_CONV : conv
```

Synopsis

Proves equality or inequality of two numerals.

Description

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `NEQ_CONV "m = n"` returns the theorem:

```
|- (m = n) = T
```

if `m` and `n` are identical, or

```
|- (m = n) = F
```

if `m` and `n` are distinct.

Failure

`NEQ_CONV tm` fails unless `tm` is of the form `"m = n"`, where `m` and `n` are numerals.

Example

```
#NEQ_CONV "12 = 12";;
|- (12 = 12) = T

#NEQ_CONV "14 = 25";;
|- (14 = 25) = F
```

NOT_CONV

`NOT_CONV : conv`

Synopsis

Simplifies certain boolean negation expressions.

Description

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `NOT_CONV tm` returns the corresponding theorem.

```
   NOT_CONV "~F"  = |-  ~F = T
   NOT_CONV "~T"  = |-  ~T = F
   NOT_CONV "~~t" = |- ~~t = t
```

Failure

`NOT_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#NOT_CONV "~~~~T";;
|- ~~~~T = ~~T

#NOT_CONV "~~T";;
|- ~~T = T

#NOT_CONV "~T";;
|- ~T = F
```

OR_CONV

`OR_CONV : conv`

Synopsis

Simplifies certain boolean disjunction expressions.

Description

If tm corresponds to one of the forms given below, where t is an arbitrary term of type bool, then OR_CONV tm returns the corresponding theorem. Note that in the last case the disjuncts need only be alpha-equivalent rather than strictly identical.

```
OR_CONV "T \/ t" = |- T \/ t = T
OR_CONV "t \/ T" = |- t \/ T = T
OR_CONV "F \/ t" = |- F \/ t = t
OR_CONV "t \/ F" = |- t \/ F = t
OR_CONV "t \/ t" = |- t \/ t = t
```

Failure

OR_CONV tm fails unless tm has one of the forms indicated above.

Example

```
#OR_CONV "F \/ T";;
|- F \/ T = T

#OR_CONV "X \/ F";;
|- X \/ F = X

#OR_CONV "(!n. n + 1 = SUC n) \/ (!m. m + 1 = SUC m)";;
|- (!n. n + 1 = SUC n) \/ (!m. m + 1 = SUC m) = (!n. n + 1 = SUC n)
```

PRE_CONV

```
PRE_CONV : conv
```

Synopsis

Calculates by inference the predecessor of a numeral.

Description

If n is a numeral (e.g. 0, 1, 2, 3,...), then PRE_CONV "PRE n" returns the theorem:

```
|- PRE n = s
```

where s is the numeral that denotes the predecessor of the natural number denoted by n.

Failure

`PRE_CONV tm` fails unless `tm` is of the form `"PRE n"`, where `n` is a numeral.

Example

```
#PRE_CONV "PRE 0";;
|- PRE 0 = 0

#PRE_CONV "PRE 1";;
|- PRE 1 = 0

#PRE_CONV "PRE 22";;
|- PRE 22 = 21
```

REDUCE_CONV

```
REDUCE_CONV : conv
```

Synopsis

Performs arithmetic or boolean reduction at all levels possible.

Description

The conversion `REDUCE_CONV` attempts to apply, in bottom-up order to all suitable redexes, one of the following conversions from the `reduce` library (only one can succeed):

```
ADD_CONV  AND_CONV  BEQ_CONV  COND_CONV
DIV_CONV  EXP_CONV   GE_CONV    GT_CONV
IMP_CONV   LE_CONV   LT_CONV   MOD_CONV
MUL_CONV  NEQ_CONV  NOT_CONV    OR_CONV
PRE_CONV  SBC_CONV  SUC_CONV
```

In particular, it will prove the appropriate reduction for an arbitrarily complicated expression constructed from numerals and the boolean constants `T` and `F`.

Failure

Never fails, but may give a reflexive equation.

Example

```
#REDUCE_CONV "(2=3) = F";;
|- ((2 = 3) = F) = T

#REDUCE_CONV "(100 < 200) => (2 EXP (8 DIV 2)) | (3 EXP ((26 EXP 0) * 3))";;
|- (100 < 200 => 2 EXP (8 DIV 2) | 3 EXP ((26 EXP 0) * 3)) = 16

#REDUCE_CONV "(15 = 16) \/ (15 < 16)";;
|- (15 = 16) \/ 15 < 16 = T

#REDUCE_CONV "0 + x";;
|- 0 + x = 0 + x
```

See also

RED_CONV, REDUCE_RULE, REDUCE_TAC

REDUCE_RULE

```
REDUCE_RULE : (thm -> thm)
```

Synopsis

Performs arithmetic or boolean reduction on a theorem at all levels possible.

Description

REDUCE_RULE attempts to transform a theorem by applying REDUCE_CONV.

Failure

Never fails, but may just return the original theorem.

Example

```
#REDUCE_RULE (ASSUME "x = (100 + (60 - 17))");;
. |- x = 143

#REDUCE_RULE (REFL "100 + 12 DIV 6");;
|- T
```

See also

RED_CONV, REDUCE_CONV, REDUCE_TAC

REDUCE_TAC

```
REDUCE_TAC : tactic
```

Synopsis

Performs arithmetic or boolean reduction on a goal at all levels possible.

Description

`REDUCE_TAC` attempts to transform a goal by applying `REDUCE_CONV`. It will prove any true goal which is constructed from numerals and the boolean constants `T` and `F`.

Failure

Never fails, but may not advance the goal.

Example

The following example takes a couple of minutes' CPU time:

```
#g "((1 EXP 3) + (12 EXP 3) = 1729) /\ ((9 EXP 3) + (10 EXP 3) = 1729)";;
"((1 EXP 3) + (12 EXP 3) = 1729) /\ ((9 EXP 3) + (10 EXP 3) = 1729)"

() : void

#e REDUCE_TAC;;
OK..
goal proved
|- ((1 EXP 3) + (12 EXP 3) = 1729) /\ ((9 EXP 3) + (10 EXP 3) = 1729)

Previous subproof:
goal proved
() : void
```

See also

`RED_CONV`, `REDUCE_CONV`, `REDUCE_RULE`

RED_CONV

```
RED_CONV : conv
```

Synopsis
Performs arithmetic or boolean reduction at top level if possible.

Description
The conversion `RED_CONV` attempts to apply, at the top level only, one of the following conversions from the `reduce` library (only one can succeed):

```
ADD_CONV  AND_CONV  BEQ_CONV  COND_CONV
DIV_CONV  EXP_CONV   GE_CONV    GT_CONV
IMP_CONV   LE_CONV   LT_CONV   MOD_CONV
MUL_CONV  NEQ_CONV  NOT_CONV    OR_CONV
PRE_CONV  SBC_CONV  SUC_CONV
```

Failure
Fails if none of the above conversions are applicable at top level.

Example

```
#RED_CONV "(2=3) = F";;
|- ((2 = 3) = F) = ~(2 = 3)

#RED_CONV "15 DIV 13";;
|- 15 DIV 13 = 1

#RED_CONV "100 + 100";;
|- 100 + 100 = 200

#RED_CONV "0 + x";;
evaluation failed     RED_CONV
```

See also
`REDUCE_CONV`, `REDUCE_RULE`, `REDUCE_TAC`

SBC_CONV

```
SBC_CONV : conv
```

Synopsis
Calculates by inference the difference of two numerals.

Description

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `SBC_CONV "m - n"` returns the theorem:

```
|- m - n = s
```

where `s` is the numeral that denotes the difference of the natural numbers denoted by `m` and `n`.

Failure

`SBC_CONV tm` fails unless `tm` is of the form `"m - n"`, where `m` and `n` are numerals.

Example

```
#SBC_CONV "25 - 30";;
|- 25 - 30 = 0

#SBC_CONV "200 - 200";;
|- 200 - 200 = 0

#SBC_CONV "60 - 17";;
|- 60 - 17 = 43
```

SUC_CONV

```
SUC_CONV : conv
```

Synopsis

Calculates by inference the successor of a numeral.

Description

If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `SUC_CONV "SUC n"` returns the theorem:

```
|- SUC n = s
```

where `s` is the numeral that denotes the successor of the natural number denoted by `n`.

Failure

`SUC_CONV tm` fails unless `tm` is of the form `"SUC n"`, where `n` is a numeral.

Example

```
#SUC_CONV "SUC 33";;
|- SUC 33 = 34
```

Index