The HOL `res_quan` Library

W. Wong

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

March 1993

Contents

Chapter 1

The res_quan Library

The res_quan library provides some basic facilities for working with restricted quantifications. It consists of a single theory res_quan.th, which contains a number of theorems about the properties of some restricted quantifiers, and a set of ML functions for dealing with these quantifiers. It also contains some conditional rewriting tools which can be loaded as a separate library part.

The description in this chapter begins with a brief introduction to the syntax for restricted quantification. This is followed by an overview of the ML functions available in the library and a description of the theory res_quan.th. A complete reference manual for all ML functions appears in Chapter 2. The last chapter lists all theorems in the res_quan.th.

1.1 Syntax for restricted quantification

Since Version 2.0, HOL provides parser and pretty printer support for restricted quantification. This notation allows terms of the form

$$\mathsf{Q}\, x :: P.\, t[x],$$

where Q is a quantifier and if $x : \alpha$ then P can be any term of type $\alpha \rightarrow bool$; this denotes the quantification of x over those values satisfying P. The qualifier :: can be used with \ and any binder, including user defined ones. The appropriate meanings are predefined for \ and the built-in binders !, ? and @. This syntax automatically translates as follows:

$\backslash v :: P.tm$	<---->	RES_ABSTRACT P (\$v.tm$)
$!v :: P.tm$	<---->	RES_FORALL P (\$v.tm$)
$?v :: P.tm$	<---->	RES_EXISTS P (\$v.tm$)
$@v :: P.tm$	<---->	RES_SELECT P (\$v.tm$)

The constants RES_ABSTRACT, RES_FORALL, RES_EXISTS and RES_SELECT are defined in the theory bool to provide semantics for these restricted quantifiers as follows:

```
RES_ABSTRACT P tm  =  \x:*. (P x => tm x | ARB:**)

RES_FORALL   P tm  =  !x:*. P x ==> tm x

RES_EXISTS   P tm  =  ?x:*. P x /\ tm x

RES_SELECT   P tm  =  @x:*. P x /\ tm x
```

where the constant ARB is defined in the theory `bool` by:

```
ARB  =  @x:*. T
```

User-defined binders can also have restricted forms, which are set up with the function:

```
associate_restriction : (string # string) -> *
```

If B is the name of a binder and `RES_`B is the name of a suitable constant (which must be explicitly defined), then executing:

`associate_restriction(‘`B`‘, ‘RES_`B`‘)`

will cause the parser and pretty-printer to support:

$B\ v{::}P.\ tm$ `<---->` `RES_`$B\ P\ (\backslash v.\ tm)$

Note that associations between user defined binders and their restrictions are not stored in theory files, so they have to be set up for each HOL session (e.g. with a `hol-init.ml` initialization file).

The flag `print_restrict` has default `true`, but if set to `false` will disable the pretty printing. This is useful for seeing what the semantics of particular restricted abstractions are. Here is an example session:

1

```
#"!x y::P. x<y";;
"!x y :: P. x < y" : term

#set_flag(‘print_restrict‘, false);;
true : bool

#"!x y::P. x<y";;
"RES_FORALL P(\x. RES_FORALL P(\y. x < y))" : term

#"?(x,y) p::(\(m,n).m<n). p=(x,y)";;
"RES_EXISTS
 (\(m,n). m < n)
 (\(x,y). RES_EXISTS(\(m,n). m < n)(\p. p = x,y))"
: term

#"\x y z::P.[0;x;y;z]";;
"RES_ABSTRACT P(\x. RES_ABSTRACT P(\y. RES_ABSTRACT P(\z. [0;x;y;z])))"
: term
```

The syntax for restricted quantification provides a method of simulating subtypes and dependent types; the qualifying predicate P can be an arbitrary term containing parameters. For example: $!w::\mathsf{Word}(n).\ t[w]$, for a suitable constant Word, simulates a quantification over the 'type' of n-bit words.[1]

1.2 The theory `res_quan.th`

This theory contains a small number of theorems about the restricted universal quantifier and restricted existential quantifier. The following four theorems state the distributivity property of these quantifiers across conjunction and disjunction.

```
RESQ_FORALL_CONJ_DIST
 |- !P Q R.
    (!(i:*) :: P. (Q i /\ R i)) = (!i :: P. Q i) /\ (!i :: P. R i)

RESQ_FORALL_DISJ_DIST
 |- !P Q R.
    (!(i:*) :: \i. P i \/ Q i. R i) = (!i :: P. R i) /\ (!i :: Q. R i)

RESQ_EXISTS_DISJ_DIST
 |- !P Q R.
    (?(i:*) :: P. (Q i \/ R i)) = (?i :: P. Q i) \/ (?i :: P. R i)

RESQ_DISJ_EXISTS_DIST
 |- !P Q R.
    (?(i:*) :: \i. P i \/ Q i. R i) = (?i :: P. R i) \/ (?i :: Q. R i)
```

The theorems `RESQ_FORALL_REORDER` and `RESQ_EXISTS_REORDER` state the reordering property of these quantifiers.

```
RESQ_FORALL_REORDER
 |- !(P:*->bool) (Q:**->bool) (R:*->**->bool).
     (!i :: P. !j :: Q. R i j) = (!j :: Q. !i :: P. R i j)

RESQ_EXISTS_REORDER
 |- !(P:*->bool) (Q:**->bool) (R:*->**->bool).
     (?i :: P. ?j :: Q. R i j) = (?j :: Q. ?i :: P. R i j)
```

The theorem `RESQ_FORALL_FORALL` states the reordering property of the restricted universal quantifier and the ordinary universal quantifier.

[1] This approach is used in the library `word` to model bit vectors.

```
RESQ_FORALL_FORALL
 |- !(P:*->bool) (R:*->**->bool) x.
     (!x. !i :: P. R i x) = (!i :: P. !x. R i x)
```

1.3 ML functions

The ML functions available when this library is loaded can be divided into six groups: conditional rewriting tools, syntax functions, derived rules, conversions, tactics, and constant definitions. They will be described in separate subsections.

1.3.1 Conditional rewriting tools

The conditional rewriting tools are not specific for restricted quantifiers. They are available as a separate part of the library which can be loaded into HOL without loading other functions in this library. This is done by the command

```
load_library `res_quan:cond_rewrite`;;
```

The conditional rewriting tools consists of a simple tactic which is for use in goal-directed proof and a simple conversion which is usually used in forward proof.

1.3.1.1 Conditional theorems

Both the conditional rewriting tactic and conversion require a theorem to do the rewriting. This theorem should be an implication whose consequence is an equation, i.e., it should be of the following form:

$$A \vdash \forall x_1 \ldots x_n.\, P_1 \supset \ldots P_m \supset (Q[x_1, \ldots, x_n] = R[x_1, \ldots, x_n]) \tag{1.1}$$

where $x_1, \ldots, x_n$ are the only variables that occur free in the left-hand side of the conclusion of the theorem but do not occur free in the assumptions. Futhermore, none of the antecedents $P_1, \ldots, P_n$ should be conjunctions. The idea of conditional rewriting is that the antecedents of this input theorem are treated as conditions which have to be satisfied before the equation $Q[x_1, \ldots, x_n] = R[x_1, \ldots, x_n]$ can be used to rewrite a term.

The ML function `COND_REWR_CANON` transforms a theorem into the canonical form in 1.1. The antecedents of the input theorem to `COND_REWR_CANON` may contain conjunctions and quantification. For example, suppose that `th` is the theorem

$$A \vdash \forall x.\, P_1\, x \supset \forall y\, z.(P_2\, y \wedge P_3\, z) \supset (\forall t.Q[x, y, z, t] = R[x, y, z, t]) \tag{1.2}$$

then `COND_REWR_CANON th` returns the theorem

$$A \vdash \forall x\, y\, z\, t.\, P_1\, x \supset P_2\, y \supset P_3\, z \supset (Q[x, y, z, t] = R[x, y, z, t])$$

That is all universal quantifications are moved to the outer most level and conjunctions in the antecedents are converted to implication.

1.3.1.2 Conditional rewriting tactic

The basic conditional rewriting tactic is

```
COND_REWRITE1_TAC : thm_tactic
```

Suppose `th` is the theorem in 1.2, the effects of applying the tactic `COND_REWRITE1_TAC` th to the goal (asm, gl) is that

- all instances of Q in the goal gl are replaced by corresponding instances of R, and
- the instances of the antecedents P_i which do not appear in the assumption asm become new subgoals.

This tactic is implemented using a lower level tactic `COND_REWR_TAC`. The theorem th supplied to `COND_REWRITE1_TAC` is processed by `COND_REWR_CANON` first. The resulting theorem is passed to the low level conditional rewriting tactic `COND_REWR_TAC` together with a search function `search_top_down`. This function determines how to find the instantiations. By calling `COND_REWR_TAC` with different search function, other conditional rewriting strategy can be implemented. The details of the tactics and search functions can be found in the reference entries in Chapter 2. Note that the `1` in the name of the tactic indicates that it takes only a single theorem as its argument.

1.3.1.3 Conditional rewriting conversion

The basic conditional rewriting conversion is

```
COND_REWRITE1_CONV : (thm list -> thm -> conv)
```

which performs conversion in a way similar to the conditional rewriting tactics. The difference is that the instances of the antecedents are added to the list of assumptions of the resulting theorem. The extra argument to this conversion is a list of theorems which are used to eliminate instances of the antecedents from the assumptions.

1.3.2 Syntax functions

There are term constructors, term destructors and term testers for the four built-in restricted quantifiers. There are also iterative constructors and destructors for the restricted universal and existential quantifiers. Their names and types are:

```
mk_resq_forall = - : ((term # term # term) -> term)
mk_resq_exists = - : ((term # term # term) -> term)
mk_resq_select = - : ((term # term # term) -> term)
mk_resq_abstract = - : ((term # term # term) -> term)
list_mk_resq_forall = - : (((term # term) list # term) -> term)
list_mk_resq_exists = - : (((term # term) list # term) -> term)

dest_resq_forall = - : (term -> (term # term # term))
dest_resq_exists = - : (term -> (term # term # term))
dest_resq_select = - : (term -> (term # term # term))
dest_resq_abstract = - : (term -> (term # term # term))
strip_resq_forall = - : (term -> ((term # term) list # term))
strip_resq_exists = - : (term -> ((term # term) list # term))

is_resq_forall = - : (term -> bool)
is_resq_exists = - : (term -> bool)
is_resq_select = - : (term -> bool)
is_resq_abstract = - : (term -> bool)
```

1.3.3 Derived rules

The introduction and elimination rules for the restricted universal quantifier are RESQ_SPEC and RESQ_GEN which are in analogy to the rules for the universal quantifier. The specification of these rules are:

$$\frac{\Gamma \vdash \forall x :: P.t[x]}{\Gamma, P\,x' \vdash t[x'/x]} \quad \texttt{RESQ\_SPEC "x'"}$$

$$\frac{\Gamma, P\,x \vdash t[x]}{\Gamma \vdash \forall x :: P.t[x]} \quad \texttt{RESQ\_GEN "x" "P"}$$

There is an extra rule RESQ_HALF_SPEC which transform a restricted universal quantification into its underlying semantic representation, namely an implication.

$$\frac{\Gamma \vdash \forall x :: P.t[x]}{\Gamma \vdash \forall x.P\,x \supset t[x]} \quad \texttt{RESQ\_HALF\_SPEC}$$

There are iterative versions of the introduction and elimination rules:

```
RESQ_SPECL = - : (term list -> thm -> thm)
RESQ_SPEC_ALL = - : (thm -> thm)

RESQ_GENL = - : (term list -> thm -> thm)
RESQ_GEN_ALL = - : (thm -> thm)
```

Since instantiation of a theorem is a very common operation, for convenience, the following ML functions are provided to instantiate a theorem with a mixture of ordinary and restricted universal quantifiers:

```
GQSPEC = - : tm -> thm -> thm
GQSPECL : term list -> thm -> thm
GQSPEC_ALL : thm -> thm
```

The rule for eliminating restricted existential quantification is RESQ_HALF_EXISTS whose specification is:

$$\frac{\Gamma \vdash \exists x :: P.t[x]}{\Gamma \vdash \exists x.P\,x \wedge t[x]} \quad \texttt{RESQ\_HALF\_EXISTS}$$

This function only transforms the restricted existential quantifier to an ordinary existential quantifier.

The function RESQ_MATCH_MP eliminates a restricted universal quantifier using an instance of the condition. Its specification is:

$$\frac{\Gamma_1 \vdash \forall x :: P.t[x] \qquad \Gamma_2 \vdash P\,x'}{\Gamma_1 \cup \Gamma_2 \vdash t[x'/x]} \quad \texttt{RESQ\_MATCH\_MP}$$

1.3.4 Conversions

There are a number of conversions for manipulating restricted universal quantification. The conversion RESQ_FORALL_CONV converts a restricted universal quantification to its underlying semantic representation, namely an implication. For example, evaluating the ML expression RESQ_FORALL_CONV "!x :: P. t[x]" returns the following theorem:

$$\vdash \forall x :: P.t[x] = \forall x.Px \supset t[x]$$

The ML function IMP_RESQ_FORALL_CONV performs the reverse conversion. The ML function LIST_RESQ_FORALL_CONV is an iterative version of RESQ_FORALL_CONV which converts a term having multiple restricted universal quantifiers at the outer level.

The conversions RESQ_FORALL_AND_CONV and AND_RESQ_FORALL_CONV move the restricted universal quantification in and out of a conjunction, respectively. The conversion RESQ_FORALL_SWAP_CONV changes the order of two restricted universal quantifications. For instance, evaluating the following ML expression

```
RESQ_FORALL_SWAP_CONV "!i :: P. !j :: Q. R"
```

returns the theorem:

$$\vdash (\forall i :: P.\forall j :: Q.R) = (\forall j :: Q.\forall i :: P.R)$$

providing that i does not occur free in Q and j does not occur free in P.

The conversion RESQ_EXISTS_CONV transforms a restricted existential quantification to its underlying semantic representation. For instance, RESQ_EXISTS_CONV "?x::P. t" returns the theorem

$$\vdash \exists x :: P.t = \exists x.Px \wedge t[x]$$

A rewriting conversion RESQ_REWRITE1_CONV uses a restricted universal quantified equation to rewrite a term. For instance, if th is a theorem of the following form:

$$\vdash \forall x :: P.u[x] = v[x]$$

and tm is a term containing some instances of u, then RESQ_REWRITE1_CONV ths th tm will return the theorem

$$\Gamma \vdash tm = tm'$$

where tm' is obtained by replacing all instances of u by corresponding instances of v and Γ contains instances of P which cannot be eliminated by the theorems in the list ths. This conversion is implemented using the conditional rewriting conversion COND_REWRITE1_CONV.

1.3.5 Tactics

The simple tactics RESQ_GEN_TAC and RESQ_EXISTS_TAC are provided for stripping of a restricted universal or existential quantifier, respectively. They reduce a restricted quantified goal to a goal in the underlying semantic representation. They are in analogy to GEN_TAC and EXISTS_TAC.

The resolution tactics and tactical listed below are in analogy to RES_TAC, IMP_RES_TAC, RES_THEN and IMP_RES_THEN.

```
RESQ_RES_THEN : (thm_tactic -> tactic)
RESQ_IMP_RES_THEN : thm_tactical
RESQ_RES_TAC : tactic
RESQ_IMP_RES_TAC : thm_tactic
```

The theorem-tactic RESQ_IMP_RES_TAC uses a restricted universally quantified theorem as if it is an implication to perform resolution. Similarly, the tactic RESQ_RES_TAC uses a restricted universally quantified assumption as if it is an implication to perform resolution against other assumptions.

The theorem-tactic RESQ_REWRITE1_TAC uses a restricted universally quantified theorem to perform conditional rewriting. For instance, if th is the following theorem

$$\vdash \forall x :: P.u[x] = v[x]$$

then applying the tactic RESQ_REWRITE1_TAC th to a goal gl will reduce it to one or more subgoals gl0, ..., gln. The main subgoal gl0 is obtained by replacing instances of u in gl with corresponding instances of v. The new subgoals are the instances of P which do not occur in the assumption of gl.

1.3.6 Constant definitions

This library provides support for defining constants whose arguments can be restricted quantified variables. For example, one can defined a constant C by the following equation:

$$\forall x_1 :: P_1 . \ldots \forall x_n :: P_n .$$
$$\quad \mathsf{C}\, y\, x_1 \ldots x_n\, z = t[y, x_1, \ldots, x_n, z]$$

The constant C may be an ordinary constant, or it may have either 'infix' or 'binder' status. The ML functions for defining restricted quantified constants are:

```
new_resq_definition        : (string # term) -> thm
new_infix_resq_definition  : (string # term) -> thm
new_binder_resq_definition : (string # term) -> thm
```

Suppose `tm` is the term shown above, evaluating the ML expression

```
new_resq_definition(‘C_DEF‘,tm)
```

will store the definition under the name `C_DEF` in the current theory. The definition is returned as the value of the expression.

Chapter 2

ML Functions in the res_quan Library

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

COND_REWRITE1_CONV

```
COND_REWRITE1_CONV : thm list -> thm -> conv
```

Synopsis

A simple conditional rewriting conversion.

Description

COND_REWRITE1_CONV is a front end of the conditional rewriting conversion COND_REWR_CONV. The input theorem should be in the following form

```
A |- !x11 ... . P1 ==> ... !xm1 ... . Pm ==> (!x ... . Q = R)
```

where each antecedent Pi itself may be a conjunction or disjunction. This theorem is transformed to a standard form expected by COND_REWR_CONV which carries out the actual rewriting. The transformation is performed by COND_REWR_CANON. The search function passed to COND_REWR_CONV is search_top_down. The effect of applying the conversion COND_REWRITE1_CONV ths th to a term tm is to derive a theorem

```
A' |- tm = tm[R'/Q']
```

where the right hand side of the equation is obtained by rewriting the input term tm with an instance of the conclusion of the input theorem. The theorems in the list ths are used to discharge the assumptions generated from the antecedents of the input theorem.

Failure

COND_REWRITE1_CONV ths th fails if th cannot be transformed into the required form by COND_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

Example

The following example illustrates a straightforward use of COND_REWRITE1_CONV. We use the built-in theorem LESS_MOD as the input theorem.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#COND_REWRITE1_CONV [] LESS_MOD "2 MOD 3";;
2 < 3 |- 2 MOD 3 = 2

#let less_2_3 = REWRITE_RULE[LESS_MONO_EQ;LESS_0]
#(REDEPTH_CONV num_CONV "2 < 3");;
less_2_3 = |- 2 < 3

#COND_REWRITE1_CONV [less_2_3] LESS_MOD "2 MOD 3";;
|- 2 MOD 3 = 2
```

In the first example, an empty theorem list is supplied to COND_REWRITE1_CONV so the resulting theorem has an assumption `2 < 3`. In the second example, a list containing a theorem `|- 2 < 3` is supplied, the resulting theorem has no assumptions.

See also

COND_REWR_TAC, COND_REWRITE1_TAC, COND_REWR_CONV, COND_REWR_CANON, search_top_down.

COND_REWRITE1_TAC

```
COND_REWRITE1_TAC : thm_tactic
```

Synopsis

A simple conditional rewriting tactic.

Description

COND_REWRITE1_TAC is a front end of the conditional rewriting tactic COND_REWR_TAC. The input theorem should be in the following form

```
A |- !x11 ... . P1 ==> ... !xm1 ... . Pm ==> (!x ... . Q = R)
```

where each antecedent `Pi` itself may be a conjunction or disjunction. This theorem is transformed to a standard form expected by COND_REWR_TAC which carries out the actual

rewriting. The transformation is performed by COND_REWR_CANON. The search function passed to COND_REWR_TAC is search_top_down. The effect of applying this tactic is to substitute into the goal instances of the right hand side of the conclusion of the input theorem Ri' for the corresponding instances of the left hand side. The search is top-down left-to-right. All matches found by the search function are substituted. New subgoals corresponding to the instances of the antecedents which do not appear in the assumption of the original goal are created. See manual page of COND_REWR_TAC for details of how the instantiation and substitution are done.

Failure

COND_REWRITE1_TAC th fails if th cannot be transformed into the required form by the function COND_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

Example

The following example illustrates a straightforward use of COND_REWRITE1_TAC. We use the built-in theorem LESS_MOD as the input theorem.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)
```

We set up a goal

```
#g"2 MOD 3 = 2";;
"2 MOD 3 = 2"

() : void
```

and then apply the tactic

```
#e(COND_REWRITE1_TAC LESS_MOD);;
OK..
2 subgoals
"2 = 2"
    [ "2 < 3" ]

"2 < 3"

() : void
```

See also

COND_REWR_TAC, COND_REWRITE1_CONV, COND_REWR_CONV, COND_REWR_CANON, search_top_down.

COND_REWR_CANON

```
COND_REWR_CANON : thm -> thm
```

Synopsis
Transform a theorem into a form accepted by COND_REWR_TAC.

Description
COND_REWR_CANON transforms a theorem into a form accepted by COND_REWR_TAC. The input theorem should be an implication of the following form

```
!x1 ... xn. P1[xi] ==> ... ==> !y1 ... ym. Pr[xi,yi] ==>
  (!z1 ... zk. u[xi,yi,zi] = v[xi,yi,zi])
```

where each antecedent Pi itself may be a conjunction or disjunction. The output theorem will have all universal quantifications moved to the outer most level with possible renaming to prevent variable capture, and have all antecedents which are a conjunction transformed to implications. The output theorem will be in the following form

```
!x1 ... xn y1 ... ym z1 ... zk.
 P11[xi] ==> ... ==> P1p[xi] ==> ... ==>
  Pr1[xi,yi] ==> ... ==> Prq[x1,yi] ==> (u[xi,yi,zi] = v[xi,yi,zi])
```

Failure
This function fails if the input theorem is not in the correct form.

Example
COND_REWR_CANON transforms the built-in theorem CANCL_SUB into the form for conditional rewriting:

```
#COND_REWR_CANON CANCEL_SUB;;
Theorem CANCEL_SUB autoloading from theory `arithmetic` ...
CANCEL_SUB = |- !p n m. p <= n /\ p <= m ==> ((n - p = m - p) = (n = m))

|- !p n m. p <= n ==> p <= m ==> ((n - p = m - p) = (n = m))
```

See also
COND_REWRITE1_TAC, COND_REWR_TAC, COND_REWRITE1_CONV, COND_REWR_CONV, search_top_down.

COND_REWR_CONV

```
COND_REWR_CONV : ((term -> term ->
 ((term # term) list # (type # type) list) list) -> thm -> conv)
```

Synopsis

A lower level conversion implementing simple conditional rewriting.

Description

COND_REWR_CONV is one of the basic building blocks for the implementation of the simple conditional rewriting conversions in the HOL system. In particular, the conditional term replacement or rewriting done by all the conditional rewriting conversions in this library is ultimately done by applications of COND_REWR_CONV. The description given here for COND_REWR_CONV may therefore be taken as a specification of the atomic action of replacing equals by equals in a term under certain conditions that are used in all these higher level conditional rewriting conversions.

The first argument to COND_REWR_CONV is expected to be a function which returns a list of matches. Each of these matches is in the form of the value returned by the built-in function match. It is used to search the input term for instances which may be rewritten.

The second argument to COND_REWR_CONV is expected to be an implicative theorem in the following form:

```
A |- !x1 ... xn. P1 ==> ... Pm ==> (Q[x1,...,xn] = R[x1,...,xn])
```

where x1, ..., xn are all the variables that occur free in the left hand side of the conclusion of the theorem but do not occur free in the assumptions.

The last argument to COND_REWR_CONV is the term to be rewritten.

If fn is a function and th is an implicative theorem of the kind shown above, then COND_REWR_CONV fn th will be a conversion. When applying to a term tm, it will return a theorem

```
P1', ..., Pm' |- tm = tm[R'/Q']
```

if evaluating fn Q[x1,...,xn] tm returns a non-empty list of matches. The assumptions of the resulting theorem are instances of the antecedents of the input theorem th. The right hand side of the equation is obtained by rewriting the input term tm with instances of the conclusion of the input theorem.

Failure

COND_REWR_CONV fn th fails if th is not an implication of the form described above. If th is such an equation, but the function fn returns a null list of matches, or the function fn returns a non-empty list of matches, but the term or type instantiation fails.

Example

The following example illustrates a straightforward use of COND_REWR_CONV. We use the built-in theorem LESS_MOD as the input theorem, and the function search_top_down as the search function.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#search_top_down;;
- : (term -> term -> ((term # term) list # (type # type) list) list)

#COND_REWR_CONV search_top_down LESS_MOD "2 MOD 3";;
2 < 3 |- 2 MOD 3 = 2
```

See also

COND_REWR_TAC, COND_REWRITE1_TAC, COND_REWRITE1_CONV, COND_REWR_CANON, search_top_down.

COND_REWR_TAC

```
COND_REWR_TAC :
 (term -> term -> ((term * term) list * (type * type) list) list) ->
 thm_tactic
```

Synopsis

A lower level tactic used to implement simple conditional rewriting tactic.

Description

COND_REWR_TAC is one of the basic building blocks for the implementation of conditional rewriting in the HOL system. In particular, the conditional term replacement or rewriting done by all the built-in conditional rewriting tactics is ultimately done by applications of COND_REWR_TAC. The description given here for COND_REWR_TAC may therefore be

taken as a specification of the atomic action of replacing equals by equals in the goal under certain conditions that aare used in all these higher level conditional rewriting tactics.

The first argument to COND_REWR_TAC is expected to be a function which returns a list of matches. Each of these matches is in the form of the value returned by the built-in function match. It is used to search the goal for instances which may be rewritten.

The second argument to COND_REWR_TAC is expected to be an implicative theorem in the following form:

```
A |- !x1 ... xn. P1 ==> ... Pm ==> (Q[x1,...,xn] = R[x1,...,xn])
```

where x1, ..., xn are all the variables that occur free in the left-hand side of the conclusion of the theorem but do not occur free in the assumptions.

If fn is a function and th is an implicative theorem of the kind shown above, then COND_REWR_TAC fn th will be a tactic which returns a list of subgoals if evaluating

```
fn Q[x1,...,xn] gl
```

returns a non-empty list of matches when applied to a goal (asm,gl).

Let ml be the match list returned by evaluating fn Q[x1,...,xn] gl. Each element in this list is in the form of

```
([(e1,x1);...;(ep,xp)], [(ty1,vty1);...;(tyq,vtyq)])
```

which specifies the term and type instantiations of the input theorem th. Either the term pair list or the type pair list may be empty. In the case that both lists are empty, an exact match is found, i.e., no instantiation is required. If ml is an empty list, no match has been found and the tactic will fail.

For each match in ml, COND_REWR_TAC will perform the following: 1) instantiate the input theorem th to get

```
th' = A |- P1' ==> ... ==> Pm' ==> (Q' = R')
```

where the primed subterms are instances of the corresponding unprimed subterms obtained by applying INST_TYPE with [(ty1,vty1);...;(tyq,vtyq)] and then INST with [(e1,x1);...;(ep,xp)]; 2) search the assumption list asm for occurrences of any antecedents P1', ..., Pm'; 3) if all antecedents appear in asm, the goal gl is reduced to gl' by substituting R' for each free occurrence of Q', otherwise, in addition to the substitution, all antecedents which do not appear in asm are added to it and new subgoals corresponding to these antecedents are created. For example, if Pk', ..., Pm' do not

appear in `asm`, the following subgoals are returned:

```
asm ?- Pk'  ...  asm ?- Pm'   {asm,Pk',...,Pm'} ?- gl'
```

If `COND_REWR_TAC` is given a theorem `th`:

```
A |- !x1 ... xn y1 ... yk. P1 ==> ... ==> Pm ==> (Q = R)
```

where the variables `y1`, ..., `ym` do not occur free in the left-hand side of the conclusion `Q` but they do occur free in the antecedents, then, when carrying out Step 2 described above, `COND_REWR_TAC` will attempt to find instantiations for these variables from the assumption `asm`. For example, if `x1` and `y1` occur free in `P1`, and a match is found in which `e1` is an instantiation of `x1`, then `P1'` will become `P1[e1/x1, y1]`. If a term `P1'' = P1[e1,e1'/x1,y1]` appears in `asm`, `th'` is instantiated with `(e1', y1)` to get

```
th'' = A |- P1'' ==> ... ==> Pm'' ==> (Q' = R'')
```

then `R''` is substituted into `gl` for all free occurrences of `Q'`. If no consistent instantiation is found, then `P1'` which contains the uninstantiated variable `y1` will become one of the new subgoals. In such a case, the user has no control over the choice of the variable `yi`.

Failure

`COND_REWR_TAC fn th` fails if `th` is not an implication of the form described above. If `th` is such an equation, but the function `fn` returns a null list of matches, or the function `fn` returns a non-empty list of matches, but the term or type instantiation fails.

Example

The following example illustrates a straightforward use of `COND_REWR_TAC`. We use the built-in theorem `LESS_MOD` as the input theorem, and the function `search_top_down` as

the search function.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory ‘arithmetic‘ ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#search_top_down;;
- : (term -> term -> ((term # term) list # (type # type) list) list)
```

We set up a goal

```
#g"2 MOD 3 = 2";;
"2 MOD 3 = 2"

() : void
```

and then apply the tactic

```
#e(COND_REWR_TAC search_top_down LESS_MOD);;
OK..
2 subgoals
"2 = 2"
    [ "2 < 3" ]

"2 < 3"

 () : void
```

See also

COND_REWRITE1_TAC, COND_REWRITE1_CONV, COND_REWR_CONV, COND_REWR_CANON, search_top_down.

dest_resq_abstract

```
dest_resq_abstract : (term -> (term # term # term))
```

Synopsis

Breaks apart a restricted abstract term into the quantified variable, predicate and body.

Description

dest_resq_abstract is a term destructor for restricted abstraction:

```
dest_resq_abstract "\var::P. t"
```

returns ("var","P","t").

Failure

Fails with dest_resq_abstract if the term is not a restricted abstraction.

See also

mk_resq_abstract, is_resq_abstract, strip_resq_abstract.

dest_resq_exists

```
dest_resq_exists : (term -> (term # term # term))
```

Synopsis

Breaks apart a restricted existentially quantified term into the quantified variable, predicate and body.

Description

dest_resq_exists is a term destructor for restricted existential quantification:

```
dest_resq_exists "?var::P. t"
```

returns ("var","P","t").

Failure

Fails with dest_resq_exists if the term is not a restricted existential quantification.

See also

mk_resq_exists, is_resq_exists, strip_resq_exists.

dest_resq_forall

```
dest_resq_forall : (term -> (term # term # term))
```

Synopsis
Breaks apart a restricted universally quantified term into the quantified variable, predicate and body.

Description
`dest_resq_forall` is a term destructor for restricted universal quantification:

```
dest_resq_forall "!var::P. t"
```

returns `("var","P","t")`.

Failure
Fails with `dest_resq_forall` if the term is not a restricted universal quantification.

See also
`mk_resq_forall, is_resq_forall, strip_resq_forall.`

dest_resq_select

```
dest_resq_select : (term -> (term # term # term))
```

Synopsis
Breaks apart a restricted choice quantified term into the quantified variable, predicate and body.

Description
`dest_resq_select` is a term destructor for restricted choice quantification:

```
dest_resq_select "@var::P. t"
```

returns `("var","P","t")`.

Failure
Fails with `dest_resq_select` if the term is not a restricted choice quantification.

See also
`mk_resq_select, is_resq_select, strip_resq_select.`

GQSPECL

```
GQSPECL : (term list -> thm -> thm)
```

Synopsis
Specializes zero or more variables in the conclusion of a universally quantified theorem.

Description
When applied to a term list `[u1;...;un]` and a theorem whose conclusion has zero or more ordinary or restricted universal quantifications, the inference rule `GQSPECL` returns a theorem which is the result of specializing the quantified variables. The substitutions are made sequentially left-to-right in the same way as for `GQSPEC`, with the same sort of alpha-conversions applied to the body of the conclusion. The two kinds of universal quantification can be mixed.

```
    A |- !x1::P1. ... !xk. ... !xn::Pn. t
-------------------------------------------------  GQSPECL "[u1;...;un]"
 A,P1 u1,...,Pn un |- t[u1/x1]...[uk/xk]...[un/xn]
```

It is permissible for the term-list to be empty, in which case the application of `GQSPECL` has no effect.

Failure
Fails if one of the specialization of the quantified variable in the original theorem fails.

See also
`GQSPEC, GQSPEC_ALL, SPECL, GENL, RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL, RESQ_GEN_TAC, RESQ_SPEC, RESQ_SPECL, RESQ_SPEC_ALL.`

GQSPEC_ALL

```
GQSPEC_ALL : (thm -> thm)
```

Synopsis
Specializes the conclusion of a theorem with its own quantified variables.

Description
When applied to a theorem whose conclusion has zero or more ordinary or restricted universal quantifications, the inference rule `GQSPEC_ALL` returns a theorem which is the result of specializing the quantified variables with its own variables. If this will cause

name clashes, a variant of the variable is used instead. Normally `xi'` is just `xi`, in which case `GQSPEC_ALL` simply removes all universal quantifiers.

```
      A |- !x1::P1. ...!xk. ... !xn::Pn. t
  ------------------------------------------------------  GQSPEC_ALL
   A,P1 x1,...,Pn xn |- t[x1'/x1]...[xk'/xk]...[xn'/xn]
```

Failure
Never fails.

See also
`GQSPEC`, `GQSPECL`, `SPEC`, `SPECL`, `SPEC_ALL`, `RESQ_GEN`, `RESQ_GENL`, `RESQ_GEN_ALL`, `RESQ_GEN_TAC`, `RESQ_SPEC`, `RESQ_SPECL`, `RESQ_SPEC_ALL`.

IMP_RESQ_FORALL_CONV

```
IMP_RESQ_FORALL_CONV : conv
```

Synopsis
Converts an implication to a restricted universal quantification.

Description
When applied to a term of the form `!x.P x ==> Q`, the conversion `IMP_RESQ_FORALL_CONV` returns the theorem:

```
|- (!x. P x ==> Q) = !x::P. Q
```

Failure
Fails if applied to a term not of the form `!x.P x ==> Q`.

See also
`RESQ_FORALL_CONV`, `LIST_RESQ_FORALL_CONV`.

is_resq_abstract

```
is_resq_abstract : (term -> bool)
```

Synopsis
Tests a term to see if it is a restricted abstraction.

Description
`is_resq_abstract "\var::P. t"` returns `true`. If the term is not a restricted abstraction the result is `false`.

Failure
Never fails.

See also
`mk_resq_abstract, dest_resq_abstract.`

is_resq_exists

```
is_resq_exists : (term -> bool)
```

Synopsis
Tests a term to see if it is a restricted existential quantification.

Description
`is_resq_exists "?var::P. t"` returns `true`. If the term is not a restricted existential quantification the result is `false`.

Failure
Never fails.

See also
`mk_resq_exists, dest_resq_exists.`

is_resq_forall

```
is_resq_forall : (term -> bool)
```

Synopsis
Tests a term to see if it is a restricted universal quantification.

Description

`is_resq_forall "!var::P. t"` returns `true`. If the term is not a restricted universal quantification the result is `false`.

Failure

Never fails.

See also

`mk_resq_forall, dest_resq_forall.`

is_resq_select

```
is_resq_select : (term -> bool)
```

Synopsis

Tests a term to see if it is a restricted choice quantification.

Description

`is_resq_select "@var::P. t"` returns `true`. If the term is not a restricted choice quantification the result is `false`.

Failure

Never fails.

See also

`mk_resq_select, dest_resq_select.`

list_mk_resq_exists

```
list_mk_resq_exists : ((term # term) list # term) -> term)
```

Synopsis

Iteratively constructs a restricted existential quantification.

Description

```
   list_mk_resq_exists([("x1","P1");...;("xn","Pn")],"t")
```

returns `"?x1::P1. ... ?xn::Pn. t"`.

Failure
Fails with `list_mk_resq_exists` if the first terms `xi` in the pairs are not a variable or if the second terms `Pi` in the pairs and `t` are not of type `":bool"` if the list is non-empty. If the list is empty the type of `t` can be anything.

See also
`strip_resq_exists, mk_resq_exists.`

list_mk_resq_forall

```
list_mk_resq_forall : ((term # term) list # term) -> term)
```

Synopsis
Iteratively constructs a restricted universal quantification.

Description

```
   list_mk_resq_forall([("x1","P1");...;("xn","Pn")],"t")
```

returns `"!x1::P1. ... !xn::Pn. t"`.

Failure
Fails with `list_mk_resq_forall` if the first terms `xi` in the pairs are not a variable or if the second terms `Pi` in the pairs and `t` are not of type `":bool"` if the list is non-empty. If the list is empty the type of `t` can be anything.

See also
`strip_resq_forall, mk_resq_forall.`

LIST_RESQ_FORALL_CONV

```
LIST_RESQ_FORALL_CONV : conv
```

Synopsis
Converts restricted universal quantifications iteratively to implications.

Description
When applied to a term whose outer level is a series of restricted universal quantifications, the conversion `LIST_RESQ_FORALL_CONV` returns the theorem:

```
|- !x1::P1. ... !xn::Pn. Q = (!x1...xn. P1 x1 ==> ... ==> Pn xn ==> Q)
```

Failure
Never fails.

See also
`IMP_RESQ_FORALL_CONV`, `RESQ_FORALL_CONV`.

mk_resq_abstract

```
mk_resq_abstract : ((term # term # term) -> term)
```

Synopsis
Term constructor for restricted abstraction.

Description
`mk_resq_abstract("var","P","t")` returns `"\var :: P . t"`.

Failure
Fails with `mk_resq_abstract` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also
`dest_resq_abstract`, `is_resq_abstract`, `list_mk_resq_abstract`.

mk_resq_exists

```
mk_resq_exists : ((term # term # term) -> term)
```

Synopsis
Term constructor for restricted existential quantification.

Description
`mk_resq_exists("var","P","t")` returns `"?var :: P . t"`.

Failure
Fails with `mk_resq_exists` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also
`dest_resq_exists, is_resq_exists, list_mk_resq_exists.`

mk_resq_forall

```
mk_resq_forall : ((term # term # term) -> term)
```

Synopsis
Term constructor for restricted universal quantification.

Description
`mk_resq_forall("var","P","t")` returns `"!var :: P . t"`.

Failure
Fails with `mk_resq_forall` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also
`dest_resq_forall, is_resq_forall, list_mk_resq_forall.`

mk_resq_select

```
mk_resq_select : ((term # term # term) -> term)
```

Synopsis
Term constructor for restricted choice quantification.

Description
`mk_resq_select("var","P","t")` returns `"@var :: P . t"`.

Failure

Fails with `mk_resq_select` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`dest_resq_select`, `is_resq_select`, `list_mk_resq_select`.

new_binder_resq_definition

```
new_binder_resq_definition : ((string # term) -> thm)
```

Synopsis

Declare a new binder and install a definitional axiom in the current theory.

Description

The function `new_binder_resq_definition` provides a facility for definitional extensions to the current theory. The new constant defined using this function may take arguments which are restricted quantified. The function `new_binder_resq_definition` takes a pair argument consisting of the name under which the resulting definition will be saved in the current theory segment, and a term giving the desired definition. The value returned by `new_binder_resq_definition` is a theorem which states the definition requested by the user.

Let `x_1,...,x_n` be distinct variables. Evaluating

```
new_binder_resq_definition (‘name‘,
 "!x_i::P_i. ... !x_j::P_j. B x_1 ... x_n = t")
```

where `B` is not already a constant, `i` is greater or equal to 1 and `i <= j <= n`, declares `B` to be a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem with the form

```
|- !x_i::P_i. ... !x_j::P_j. !x_k .... B x_1 ... x_n = t
```

where the variables `x_k` are the free variables occurring on the left hand side of the definition and are not restricted quantified. This theorem is saved in the current theory under (the name) `name`.

The constant `B` defined by this function will have the binder status only after the definition has been processed. It is therefore necessary to use the constant in normal prefix position when making the definition.

If the restricting predicates P_l contains free occurrence of variable(s) of the left hand side, the constant B will stand for a family of functions.

Failure

new_binder_resq_definition fails if called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification of the given name in the current theory segment; if ‘B‘ is already a constant in the current theory or is not an allowed name for a constant; if t contains free variables that do not occur in the left hand side, or if any variable occurs more than once in x_1, ..., x_n. Finally, failure occurs if there is a type variable in x_1, ..., x_n or t that does not occur in the type of B.

See also

new_infix_resq_definition, new_resq_definition, new_definition, new_specification.

new_infix_resq_definition

```
new_infix_resq_definition : ((string # term) -> thm)
```

Synopsis

Declare a new infix constant and install a definitional axiom in the current theory.

Description

The function new_infix_resq_definition provides a facility for definitional extensions to the current theory. The new constant defined using this function may take arguments which are restricted quantified. The function new_infix_resq_definition takes a pair argument consisting of the name under which the resulting definition will be saved in the current theory segment, and a term giving the desired definition. The value returned by new_infix_resq_definition is a theorem which states the definition requested by the user.

Let x_1,...,x_n be distinct variables. Evaluating

```
new_infix_resq_definition (‘name‘,
 "!x_i::P_i. ... !x_j::P_j. IX x_1 ... x_n = t")
```

where IX is not already a constant, i is greater or equal to 1 and i <= j <= n, declares IX to be a new constant in the current theory with this definition as its specification. This

constant specification is returned as a theorem with the form

```
|- !x_i::P_i. ... !x_j::P_j. !x_k .... IX x_1 ... x_n = t
```

where the variables `x_k` are the free variables occurring on the left hand side of the definition and are not restricted quantified. This theorem is saved in the current theory under (the name) `name`.

The constant `IX` defined by this function will have the infix status only after the definition has been processed. It is therefore necessary to use the constant in normal prefix position when making the definition.

If the restricting predicates `P_1` contains free occurrence of variable(s) of the left hand side, the constant `IX` will stand for a family of functions.

Failure

`new_infix_resq_definition` fails if called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification of the given name in the current theory segment; if `‘IX‘` is already a constant in the current theory or is not an allowed name for a constant; if `t` contains free variables that do not occur in the left hand side, or if any variable occurs more than once in `x_1, ..., x_n`. Finally, failure occurs if there is a type variable in `x_1`, ..., `x_n` or `t` that does not occur in the type of `IX`.

Example

A function for indexing list element starting from 1 can be defined as follows:

```
#let IXEL1_DEF = new_infix_resq_definition (‘IXEL1_DEF‘,
# "!n:: (\k. 0 < k). IXEL1 n (l:* list) = EL (n -1) l");;
IXEL1_DEF = |- !n :: \k. 0 < k. !l. IXEL1 n l = EL(n - 1)l
```

One can then use `IXEL1` as an infix and do the following proof:

```
#g"2 IXEL1 [1;2;3] = 2";;
"2 IXEL1 [1;2;3] = 2"

#e(RESQ_REWRITE1_TAC IXEL1_DEF THENL[
#   CONV_TAC(ONCE_DEPTH_CONV num_CONV) THEN MATCH_ACCEPT_TAC LESS_0;
#   CONV_TAC((LHS_CONV o LHS_CONV)(REDEPTH_CONV num_CONV))
#   THEN REWRITE_TAC[SUB_MONO_EQ;SUB_0;EL;HD;TL]]);;
OK..
goal proved
|- 2 IXEL1 [1;2;3] = 2

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

See also

`new_binder_resq_definition, new_resq_definition, new_definition, new_specification.`

new_resq_definition

```
new_resq_definition : ((string # term) -> thm)
```

Synopsis

Declare a new constant and install a definitional axiom in the current theory.

Description

The function `new_resq_definition` provides a facility for definitional extensions to the current theory. The new constant defined using this function may take arguments which are restricted quantified. The function `new_resq_definition` takes a pair argument consisting of the name under which the resulting definition will be saved in the

current theory segment, and a term giving the desired definition. The value returned by `new_resq_definition` is a theorem which states the definition requested by the user.

Let `x_1,...,x_n` be distinct variables. Evaluating

```
new_resq_definition (‘name‘,
 "!x_i::P_i. ... !x_j::P_j. C x_1 ... x_n = t")
```

where `C` is not already a constant, `i` is greater or equal to 1 and `i <= j <= n`, declares `C` to be a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem with the form

```
|- !x_i::P_i. ... !x_j::P_j. !x_k .... C x_1 ... x_n = t
```

where the variables `x_k` are the free variables occurring on the left hand side of the definition and are not restricted quantified. This theorem is saved in the current theory under (the name) `name`.

If the restricting predicates `P_1` contains free occurrence of variable(s) of the left hand side, the constant `C` will stand for a family of functions.

Failure

`new_resq_definition` fails if called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification of the given name in the current theory segment; if `‘C‘` is already a constant in the current theory or is not an allowed name for a constant; if `t` contains free variables that do not occur in the left hand side, or if any variable occurs more than once in `x_1, ..., x_n`. Finally, failure occurs if there is a type variable in `x_1`, ..., `x_n` or `t` that does not occur in the type of `C`.

Example

A function for indexing list elements starting from 1 can be defined as follows:

```
#new_resq_definition (‘EL1_DEF‘,
# "!n:: (\k. 0 < k). EL1 n (l:* list) = EL (n - 1) l");;
|- !n :: \k. 0 < k. !l. EL1 n l = EL(n - 1)l
```

The following example shows how a family of constants may be defined if the restricting predicate involves free variable on the left hand side of the definition.

```
#new_resq_definition (‘ELL_DEF‘,
# "!n:: (\k. k < (LENGTH l)). ELL n (l:* list) = EL n  l");;
|- !l. !n :: \k. k < (LENGTH l). !l'. ELL l n l' = EL n l'
```

See also

`new_resq_binder_definition`, `new_resq_infix_definition`, `new_definition`, `new_specification`.

RESQ_EXISTS_CONV

```
RESQ_EXISTS_CONV : conv
```

Synopsis
Converts a restricted existential quantification to a conjunction.

Description
When applied to a term of the form `?x::P. Q[x]`, the conversion `RESQ_EXISTS_CONV` returns the theorem:

```
|- ?x::P. Q[x] = (?x. P x /\ Q[x])
```

which is the underlying semantic representation of the restricted existential quantification.

Failure
Fails if applied to a term not of the form `?x::P. Q`.

See also
`RESQ_FORALL_CONV`, `RESQ_EXISTS_TAC`.

RESQ_EXISTS_TAC

```
RESQ_EXISTS_TAC : term -> tactic
```

Synopsis
Strips the outermost restricted existential quantifier from the conclusion of a goal.

Description
When applied to a goal `A ?- ?x::P. t`, the tactic `RESQ_EXISTS_TAC` reduces it to a new subgoal `A ?- P x' /\ t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
    A ?- ?x::P. t
  ======================  RESQ_EXISTS_TAC
   A ?- P x' /\ t[x'/x]
```

Failure
Fails unless the goal's conclusion is a restricted extistential quantification.

See also
RESQ_HALF_EXISTS.

RESQ_FORALL_AND_CONV

```
RESQ_FORALL_AND_CONV : conv
```

Synopsis
Splits a restricted universal quantification across a conjunction.

Description
When applied to a term of the form `!x::P. Q /\ R`, the conversion RESQ_FORALL_AND_CONV returns the theorem:

```
|- (!x::P. Q /\ R)  = ((!x::P. Q) /\ (!x::P. R))
```

Failure
Fails if applied to a term not of the form `!x::P. Q /\ R`.

See also
AND_RESQ_FORALL_CONV.

RESQ_FORALL_CONV

```
RESQ_FORALL_CONV : conv
```

Synopsis
Converts a restricted universal quantification to an implication.

Description
When applied to a term of the form `!x::P. Q`, the conversion RESQ_FORALL_CONV returns the theorem:

```
|- !x::P. Q = (!x. P x ==> Q)
```

which is the underlying semantic representation of the restricted universal quantification.

Failure
Fails if applied to a term not of the form `!x::P. Q`.

See also
`IMP_RESQ_FORALL_CONV, LIST_RESQ_FORALL_CONV.`

RESQ_FORALL_SWAP_CONV

```
RESQ_FORALL_SWAP_CONV : conv
```

Synopsis
Changes the order of two restricted universal quantifications.

Description
When applied to a term of the form `!x::P. !y::Q. R`, the conversion `RESQ_FORALL_SWAP_CONV` returns the theorem:

```
|- (!x::P. !y::Q. R) =  !y::Q. !x::P. R
```

providing that `x` does not occur free in `Q` and `y` does not occur free in `P`.

Failure
Fails if applied to a term not of the correct form.

See also
`RESQ_FORALL_CONV.`

RESQ_GEN

```
RESQ_GEN : ((term # term) -> thm -> thm)
```

Synopsis
Generalizes the conclusion of a theorem to a restricted universal quantification.

Description
When applied to a pair of terms `x`, `P` and a theorem `A |- t`, the inference rule `RESQ_GEN` returns the theorem `A |- !x::P. t`, provided that `P` is a predicate taking an argument

of the same type as `x` and that `x` is a variable not free in any of the assumptions except `P x` if it occurs. There is no compulsion that `x` should be free in `t` or `P x` should be in the assumptions.

```
    A |- t
--------------- RESQ_GEN ("x","P") [where x is not free in A except P x]
 A |- !x::P. t
```

Failure

Fails if `x` is not a variable, or if it is free in any of the assumptions other than `P x`.

See also

`RESQ_GENL`, `RESQ_GEN_ALL`, `RESQ_GEN_TAC`, `RESQ_SPEC`, `RESQ_SPECL`, `RESQ_SPEC_ALL`.

RESQ_GENL

```
RESQ_GENL : ((term # term) list -> thm -> thm)
```

Synopsis

Generalizes zero or more variables to restricted universal quantification in the conclusion of a theorem.

Description

When applied to a term-pair list `[(x1,P1);...;(xn,Pn)]` and a theorem `A |- t`, the inference rule `RESQ_GENL` returns the theorem `A |- !x1::P1. ... !xn::Pn. t`, provided none of the variables `xi` are free in any of the assumptions except in the corresponding `Pi`. It is not necessary that any or all of the `xi` should be free in `t`.

```
      A |- t
------------------------------  RESQ_GENL "[(x1,P1);...;(xn,Pn)]"
 A |- !x1::P1. ... !xn::Pn. t   [where no xi is free in A except in Pi]
```

Failure

Fails unless all the terms `xi` in the list are variables, none of which are free in the assumption list except in `Pi`.

See also

`RESQ_GEN`, `RESQ_GEN_ALL`, `RESQ_GEN_TAC`, `RESQ_SPEC`, `RESQ_SPECL`, `RESQ_SPEC_ALL`.

RESQ_GEN_ALL

```
RESQ_GEN_ALL : (thm -> thm)
```

Synopsis
Generalizes the conclusion of a theorem over its own assumptions.

Description
When applied to a theorem `A |- t`, the inference rule `RESQ_GEN_ALL` returns the theorem `A' |- !x1::P1. ...!xn::Pn. t`, where the `Pi xi` are in the assumptions.

```
       A |- t
  ------------------------------------------------  RESQ_GEN_ALL
  A - (P1 x1,...,Pn xn) |- !x1::P1. ... !xn::Pn. t
```

Failure
Never fails.

See also
`RESQ_GEN`, `RESQ_GENL`, `GEN_ALL`, `RESQ_SPEC`, `RESQ_SPECL`, `RESQ_SPEC_ALL`.

RESQ_GEN_TAC

```
RESQ_GEN_TAC : tactic
```

Synopsis
Strips the outermost restricted universal quantifier from the conclusion of a goal.

Description
When applied to a goal `A ?- !x::P. t`, the tactic `RESQ_GEN_TAC` reduces it to a new goal `A,P x' ?- t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
    A ?- !x::P. t
  ==================  RESQ_GEN_TAC
   A,P x' ?- t[x'/x]
```

Failure
Fails unless the goal's conclusion is a restricted universal quantification.

Uses
The tactic `REPEAT RESQ_GEN_TAC` strips away a series of restricted universal quantifiers, and is commonly used before tactics relying on the underlying term structure.

See also
`RESQ_HALF_GEN_TAC, RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL, RESQ_SPEC, RESQ_SPECL, RESQ_SPEC_ALL, GGEN_TAC, STRIP_TAC, GEN_TAC, X_GEN_TAC.`

RESQ_HALF_EXISTS

```
RESQ_HALF_EXISTS : (thm -> thm)
```

Synopsis
Strip a restricted existential quantification from the conclusion of a theorem.

Description
When applied to a theorem `A |- ?x::P. t`, `RESQ_HALF_EXISTS` returns the theorem

```
A |- ?x. P x /\ t
```

i.e., it transforms the restricted existential quantification to its underlying semantic representation.

```
   A |- ?x::P. t
--------------------  RESQ_HALF_EXISTS
 A |- ?x. P x ==> t
```

Failure
Fails if the theorem's conclusion is not a restricted existential quantification.

See also
`RESQ_EXISTS_TAC, EXISTS.`

RESQ_HALF_GEN_TAC

```
RESQ_HALF_GEN_TAC : tactic
```

Synopsis
Strips the outermost restricted universal quantifier from the conclusion of a goal.

Description
When applied to a goal `A ?- !x::P. t`, `RESQ_GEN_TAC` reduces it to `A ?- !x. P x ==> t` which is the underlying semantic representation of the restricted universal quantification.

```
    A ?- !x::P. t
====================  RESQ_HALF_GEN_TAC
 A ?- !x. P x ==> t
```

Failure
Fails unless the goal's conclusion is a restricted universal quantification.

Uses
The tactic `REPEAT RESQ_GEN_TAC` strips away a series of restricted universal quantifiers, and is commonly used before tactics relying on the underlying term structure.

See also
`RESQ_GEN_TAC`, `RESQ_GEN`, `RESQ_GENL`, `RESQ_GEN_ALL`, `RESQ_SPEC`, `RESQ_SPECL`, `RESQ_SPEC_ALL`, `GGEN_TAC`, `STRIP_TAC`, `GEN_TAC`, `X_GEN_TAC`.

RESQ_HALF_SPEC

```
RESQ_HALF_SPEC : (thm -> thm)
```

Synopsis
Strip a restricted universal quantification in the conclusion of a theorem.

Description
When applied to a theorem `A |- !x::P. t`, the derived inference rule `RESQ_HALF_SPEC` returns the theorem `A |- !x. P x ==> t`, i.e., it transforms the restricted universal quantification to its underlying semantic representation.

```
    A |- !x::P. t
--------------------  RESQ_HALF_SPEC
 A |- !x. P x ==> t
```

Failure
Fails if the theorem's conclusion is not a restricted universal quantification.

See also
RESQ_SPEC, RESQ_SPECL, RESQ_SPEC_ALL, RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL.

RESQ_IMP_RES_TAC

RESQ_IMP_RES_TAC : thm_tactic

Synopsis
Repeatedly resolves a restricted universally quantified theorem with the assumptions of a goal.

Description
The function RESQ_IMP_RES_TAC performs repeatedly resolution using a restricted quantified theorem. It takes a restricted quantified theorem and transforms it into an implication. This resulting theorem is used in the resolution.

Given a theorem th, the theorem-tactic RESQ_IMP_RES_TAC applies RESQ_IMP_RES_THEN repeatedly to resolve the theorem with the assumptions.

Failure
Never fails

See also
RESQ_IMP_RES_THEN, RESQ_RES_THEN, RESQ_RES_TAC, IMP_RES_THEN, IMP_RES_TAC, MATCH_MP, RES_CANON, RES_TAC, RES_THEN.

RESQ_IMP_RES_THEN

RESQ_IMP_RES_THEN : thm_tactical

Synopsis
Resolves a restricted universally quantified theorem with the assumptions of a goal.

Description
The function RESQ_IMP_RES_THEN is the basic building block for resolution using a restricted quantified theorem. It takes a restricted quantified theorem and transforms it into an implication. This resulting theorem is used in the resolution.

Given a theorem-tactic `ttac` and a theorem `th`, the theorem-tactical `RESQ_IMP_RES_THEN` transforms the theorem into an implication `th'`. It then passes `th'` together with `ttac` to `IMP_RES_THEN` to carry out the resolution.

Failure

Evaluating `RESQ_IMP_RES_THEN ttac th` fails if the supplied theorem `th` is not restricted universally quantified, or if the call to `IMP_RES_THEN` fails.

See also

`RESQ_IMP_RES_TAC`, `RESQ_RES_THEN`, `RESQ_RES_TAC`, `IMP_RES_THEN`, `IMP_RES_TAC`, `MATCH_MP`, `RES_CANON`, `RES_TAC`, `RES_THEN`.

RESQ_MATCH_MP

```
RESQ_MATCH_MP : (thm -> thm -> thm)
```

Synopsis

Eliminating a restricted universal quantification with automatic matching.

Description

When applied to theorems `A1 |- !x::P. Q[x]` and `A2 |- P x'`, the derived inference rule `RESQ_MATCH_MP` matches `x'` to `x` by instantiating free or universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- Q[x'/x]`. Polymorphic types are also instantiated if necessary.

```
 A1 |- !x::P.Q[x]   A2 |- P x'
-----------------------------------  RESQ_MATCH_MP
       A1 u A2 |- Q[x'/x]
```

Failure

Fails unless the first theorem is a (possibly repeatedly) restricted universal quantification whose quantified variable can be instantiated to match the conclusion of the second theorem, without instantiating any variables which are free in `A1`, the first theorem's assumption list.

See also

`MATCH_MP`, `RESQ_HALF_SPEC`.

RESQ_RES_TAC

```
RESQ_RES_TAC : tactic
```

Synopsis
Enriches assumptions by repeatedly resolving restricted universal quantifications in them against the others.

Description
RESQ_RES_TAC uses those assumptions which are restricted universal quantifications in resolution in a way similar to RES_TAC. It calls RESQ_RES_THEN repeatedly until there is no more resolution can be done. The conclusions of all the new results are returned as additional assumptions of the subgoal(s). The effect of RESQ_RES_TAC on a goal is to enrich the assumption set with some of its collective consequences.

Failure
RESQ_RES_TAC cannot fail and so should not be unconditionally REPEATed.

See also
RESQ_IMP_RES_TAC, RESQ_IMP_RES_THEN, RESQ_RES_THEN, IMP_RES_TAC, IMP_RES_THEN, RES_CANON, RES_THEN, RES_TAC.

RESQ_RES_THEN

```
RESQ_RES_THEN : thm_tactic -> tactic
```

Synopsis
Resolves all restricted universally quantified assumptions against other assumptions of a goal.

Description
Like the function RESQ_IMP_RES_THEN, the function RESQ_RES_THEN performs a single step resolution. The difference is that the restricted universal quantification used in the resolution is taken from the assumptions.

Given a theorem-tactic ttac, applying the tactic RESQ_RES_THEN ttac to a goal (asml,gl) has the effect of:

```
MAP_EVERY (mapfilter ttac [... ; (ai,aj |- vi) ; ...]) (amsl ?- g)
```

where the theorems ai,aj |- vi are all the consequences that can be drawn by a (single) matching modus-ponens inference from the assumptions amsl and the implications derived from the restricted universal quantifications in the assumptions.

Failure

Evaluating RESQ_RES_TAC ttac th fails if there are no restricted universal quantifications in the assumptions, or if the theorem-tactic ttac applied to all the consequences fails.

See also

RESQ_IMP_RES_TAC, RESQ_IMP_RES_THEN, RESQ_RES_TAC, IMP_RES_THEN, IMP_RES_TAC, MATCH_MP, RES_CANON, RES_TAC, RES_THEN.

RESQ_REWRITE1_CONV

```
RESQ_REWRITE1_CONV : thm list -> thm -> conv
```

Synopsis

Rewriting conversion using a restricted universally quantified theorem.

Description

RESQ_REWRITE1_CONV is a rewriting conversion similar to COND_REWRITE1_CONV. The only difference is the rewriting theorem it takes. This should be an equation with restricted universal quantification at the outer level. It is converted to a theorem in the form accepted by the conditional rewriting conversion.

Suppose that th is the following theorem

```
A |- !x::P. Q[x] = R[x])
```

evaluating RESQ_REWRITE1_CONV thms th "t[x']" will return a theorem

```
A, P x' |- t[x'] = t'[x']
```

where t' is the result of substituting instances of R[x'/x] for corresponding instances of Q[x'/x] in the original term t[x]. All instances of P x' which do not appear in the original assumption asml are added to the assumption. The theorems in the list thms are used to eliminate the instances P x' if it is possible.

Failure

RESQ_REWRITE1_CONV fails if th cannot be transformed into the required form by the function RESQ_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

See also

RESQ_REWRITE1_TAC, RESQ_REWR_CANON, COND_REWR_TAC, COND_REWRITE1_CONV, COND_REWR_CONV, COND_REWR_CANON, search_top_down.

RESQ_REWRITE1_TAC

RESQ_REWRITE1_TAC : thm_tactic

Synopsis

Rewriting with a restricted universally quantified theorem.

Description

RESQ_REWRITE1_TAC takes an equational theorem which is restricted universally quantified at the outer level. It calls RESQ_REWR_CANON to convert the theorem to the form accepted by COND_REWR_TAC and passes the resulting theorem to this tactic which carries out conditional rewriting.

Suppose that th is the following theorem

```
A |- !x::P. Q[x] = R[x])
```

Applying the tactic RESQ_REWRITE1_TAC th to a goal (asml,gl) will return a main subgoal (asml',gl') where gl' is obtained by substituting instances of R[x'/x] for corresponding instances of Q[x'/x] in the original goal gl. All instances of P x' which do not appear in the original assumption asml are added to it to form asml', and they also become new subgoals (asml,P x').

Failure

RESQ_REWRITE1_TAC th fails if th cannot be transformed into the required form by the function RESQ_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

See also

RESQ_REWRITE1_CONV, RESQ_REWR_CANON, COND_REWR_TAC, COND_REWRITE1_CONV, COND_REWR_CONV, COND_REWR_CANON, search_top_down.

RESQ_REWR_CANON

```
RESQ_REWR_CANON : thm -> thm
```

Synopsis
Transform a theorem into a form accepted for rewriting.

Description
RESQ_REWR_CANON transforms a theorem into a form accepted by COND_REWR_TAC. The input theorem should be headed by a series of restricted universal quantifications in the following form

```
!x1::P1. ... !xn::Pn. u[xi] = v[xi])
```

Other variables occurring in u and v may be universally quantified. The output theorem will have all ordinary universal quantifications moved to the outer most level with possible renaming to prevent variable capture, and have all restricted universal quantifications converted to implications. The output theorem will be in the form accepted by COND_REWR_TAC.

Failure
This function fails is the input theorem is not in the correct form.

See also
RESQ_REWRITE1_TAC, RESQ_REWRITE1_CONV, COND_REWR_CANON, COND_REWR_TAC, COND_REWR_CONV,.

RESQ_SPEC

```
RESQ_SPEC : (term -> thm -> thm)
```

Synopsis
Specializes the conclusion of a restricted universally quantified theorem.

Description
When applied to a term u and a theorem A |- !x::P. t, RESQ_SPEC returns the theorem A, P u |- t[u/x]. If necessary, variables will be renamed prior to the specialization

to ensure that `u` is free for `x` in `t`, that is, no variables free in `u` become bound after substitution.

```
    A |- !x::P. t
------------------  RESQ_SPEC "u"
 A, P u |- t[u/x]
```

Failure

Fails if the theorem's conclusion is not restricted universally quantified, or if type instantiation fails.

Example

The following example shows how `RESQ_SPEC` renames bound variables if necessary, prior to substitution: a straightforward substitution would result in the clearly invalid theorem `(\y. 0 < y)y |- y = y`.

```
#let th = RESQ_GEN "x:num" "\y.0<y" (REFL "x:num");;
th = |- !x :: \y. 0 < y. x = x

#RESQ_SPEC "y:num" th;;
(\y'. 0 < y')y |- y = y
```

See also

`RESQ_SPECL, RESQ_SPEC_ALL, RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL.`

RESQ_SPECL

```
RESQ_SPECL : (term list -> thm -> thm)
```

Synopsis

Specializes zero or more variables in the conclusion of a restricted universally quantified theorem.

Description

When applied to a term list `[u1;...;un]` and a theorem `A |- !x1::P1. ... !xn::Pn. t`, the inference rule `RESQ_SPECL` returns the theorem

```
A,P1 u1,...,Pn un |- t[u1/x1]...[un/xn]
```

where the substitutions are made sequentially left-to-right in the same way as for `RESQ_SPEC`,

with the same sort of alpha-conversions applied to t if necessary to ensure that no variables which are free in ui become bound after substitution.

```
         A |- !x1::P1. ... !xn::Pn. t
------------------------------------------  RESQ_SPECL "[u1;...;un]"
  A,P1 u1, ..., Pn un |- t[u1/x1]...[un/xn]
```

It is permissible for the term-list to be empty, in which case the application of RESQ_SPECL has no effect.

Failure
Fails if one of the specialization of the restricted universally quantified variable in the original theorem fails.

See also
RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL, RESQ_GEN_TAC, RESQ_SPEC, RESQ_SPEC_ALL.

RESQ_SPEC_ALL

```
RESQ_SPEC_ALL : (thm -> thm)
```

Synopsis
Specializes the conclusion of a theorem with its own restricted quantified variables.

Description
When applied to a theorem A |- !x1::P1. ...!xn::Pn. t, the inference rule RESQ_SPEC_ALL returns the theorem A,P1 x1',...,Pn xn' |- t[x1'/x1]...[xn'/xn] where the xi' are distinct variants of the corresponding xi, chosen to avoid clashes with any variables free in the assumption list and with the names of constants. Normally xi' is just xi, in which case RESQ_SPEC_ALL simply removes all restricted universal quantifiers.

```
     A |- !x1::P1. ... !xn::Pn. t
------------------------------------------  RESQ_SPEC_ALL
 A,P1 x1,...,Pn xn |- t[x1'/x1]...[xn'/xn]
```

Failure
Never fails.

See also
RESQ_GEN, RESQ_GENL, RESQ_GEN_ALL, RESQ_GEN_TAC, RESQ_SPEC, RESQ_SPECL.

search_top_down

```
search_top_down
 : (term -> term -> ((term # term) list # (type # type) list) list)
```

Synopsis

Search a term in a top-down fashion to find matches to another term.

Description

`search_top_down tm1 tm2` returns a list of instantiations which make the whole or part of `tm2` match `tm1`. The first term should not have a quantifier at the outer most level. `search_top_down` first attempts to match the whole second term to `tm1`. If this fails, it recursively descend into the subterms of `tm2` to find all matches.

The length of the returned list indicates the number of matches found. An empty list means no match can be found between `tm1` and `tm2` or any subterms of `tm2`. The instantiations returned in the list are in the same format as for the function `match`. Each instantiation is a pair of lists: the first is a list of term pairs and the second is a list of type pairs. Either of these lists may be empty. The situation in which both lists are empty indicates that there is an exact match between the two terms, i.e., no instantiation is required to make the entire `tm2` or a part of `tm2` the same as `tm1`.

Failure

Never fails.

Example

```
#search_top_down "x = y:*" "3 = 5";;
[([("5", "y"); ("3", "x")], [(":num", ":*")])]
: ((term # term) list # (type # type) list) list

#search_top_down "x = y:*" "x =y:*";;
[([], [])] : ((term # term) list # (type # type) list) list

#search_top_down "x = y:*" "0 < p ==> (x <= p = y <= p)";;
[([("y <= p", "y"); ("x <= p", "x")], [(":bool", ":*")])]
: ((term # term) list # (type # type) list) list
```

The first example above shows the entire `tm2` matching `tm1`. The second example shows the two terms match exactly. No instantiation is required. The last example shows that a subterm of `tm2` can be instantiated to match `tm1`.

See also
match, COND_REWR_TAC, CONV_REWRITE_TAC, COND_REWR_CONV, CONV_REWRITE_CONV.

strip_resq_exists

```
strip_resq_exists : (term -> ((term # term) list # term))
```

Synopsis
Iteratively breaks apart a restricted existentially quantified term.

Description
strip_resq_exists is an iterative term destructor for restricted existential quantifications. It iteratively breaks apart a restricted existentially quantified term into a list of pairs which are the restricted quantified variables and predicates and the body.

```
strip_resq_exists "?x1::P1. ... ?xn::Pn. t"
```

returns [("x1","P1");...;("xn","Pn")],"t").

Failure
Never fails.

See also
list_mk_resq_exists, is_resq_exists, dest_resq_exists.

strip_resq_forall

```
strip_resq_forall : (term -> ((term # term) list # term))
```

Synopsis
Iteratively breaks apart a restricted universally quantified term.

Description
strip_resq_forall is an iterative term destructor for restricted universal quantifications. It iteratively breaks apart a restricted universally quantified term into a list of pairs which are the restricted quantified variables and predicates and the body.

```
strip_resq_forall "!x1::P1. ... !xn::Pn. t"
```

returns [("x1","P1");...;("xn","Pn")],"t").

Failure
Never fails.

See also
`list_mk_resq_forall`, `is_resq_forall`, `dest_resq_forall`.

Chapter 3

Pre-proved Theorems

The sections that follow list all theorems in the theories res_quan and restr_binder. The theorems listed in this chapter will be available by name at the top-level when the theories in which they are declared are open-ed.

```
RESQ_DISJ_EXISTS_DIST  (res_quan)
   |- !P Q R. (?i::(\i. P i \/ Q i). R i) = (?i::P. R i) \/ ?i::Q. R i

RESQ_EXISTS_DISJ_DIST  (res_quan)
   |- !P Q R. (?i::P. Q i \/ R i) = (?i::P. Q i) \/ ?i::P. R i

RESQ_EXISTS_REORDER  (res_quan)
   |- !P Q R. (?(i::P) (j::Q). R i j) = ?(j::Q) (i::P). R i j

RESQ_EXISTS_UNIQUE  (res_quan)
   |- !P j. (?i::$= j. P i) = P j

RESQ_FORALL_CONJ_DIST  (res_quan)
   |- !P Q R. (!i::P. Q i /\ R i) = (!i::P. Q i) /\ !i::P. R i

RESQ_FORALL_DISJ_DIST  (res_quan)
   |- !P Q R. (!i::(\i. P i \/ Q i). R i) = (!i::P. R i) /\ !i::Q. R i

RESQ_FORALL_FORALL  (res_quan)
   |- !P R x. (!x (i::P). R i x) = !(i::P) x. R i x

RESQ_FORALL_REORDER  (res_quan)
   |- !P Q R. (!(i::P) (j::Q). R i j) = !(j::Q) (i::P). R i j

RESQ_FORALL_UNIQUE  (res_quan)
   |- !P j. (!i::$= j. P i) = P j

RES_ABSTRACT  (res_quan)
   |- !P B. RES_ABSTRACT P B = (\x. (if P x then B x else ARB))

RES_EXISTS  (res_quan)
   |- !P B. RES_EXISTS P B = ?x. P x /\ B x
```

```
RES_FORALL  (res_quan)
   |- !P B. RES_FORALL P B = !x. P x ==> B x

RES_SELECT  (res_quan)
   |- !P B. RES_SELECT P B = @x. P x /\ B x
```

Index