The HOL pair Library

Jim Grundy

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

November 1992

start at page 1

Chapter 1

Statement of Rights

Jim Grundy, hereafter referred to as ‘the Author’, retains the copyright and all other legal rights to the software contained in the pair library, hereafter referred to as ‘the Software’. The Software is made available free of charge on an ‘as is’ basis. No guarantee, either express or implied, of maintenance, reliability, merchantability or suitability for any purpose is made by the Author.

The user is granted the right to make personal or internal use of the Software provided that both:

1. The Software is not used for commercial gain.
2. The user shall not hold the Author liable for any consequences arising from use of the Software.

The user is granted the right to further distribute the Software provided that both:

1. The Software and this statement of rights is not modified.
2. The Software does not form part or the whole of a system distributed for commercial gain.

The user is granted the right to modify the Software for personal or internal use provided that all of the following conditions are observed:

1. The user does not distribute the modified software.
2. The modified software is not used for commercial gain.
3. The Author retains all rights to the modified software.

Anyone seeking a licence to use this software for commercial purposes is invited to contact the Author.

Chapter 2

The pair Library

This manual describes the use of the pair library. The pair library has been provided to reduce the difficulty of reasoning about pairs (and tuples), particularly paired quantifications and abstractions. The pair library contains a version of every standard HOL function for manipulating abstractions and quantifications. The table below sets out all the standard HOL functions for which the pair library provides paired equivalents:

Function	Paired Version	Function	Paired Version
ABS	PABS	LEFT_IMP_FORALL_CONV	LEFT_IMP_PFORALL_CONV
ABS_CONV	PABS_CONV	LEFT_OR_EXISTS_CONV	LEFT_OR_PEXISTS_CONV
aconv	paconv	LEFT_OR_FORALL_CONV	LEFT_OR_PFORALL_CONV
ALPHA	PALPHA	LIST_BETA_CONV	LIST_PBETA_CONV
ALPHA_CONV	PALPHA_CONV	list_mk_abs	list_mk_pabs
AND_EXISTS_CONV	AND_PEXISTS_CONV	LIST_MK_EXISTS	LIST_MK_PEXISTS
AND_FORALL_CONV	AND_PFORALL_CONV	list_mk_exists	list_mk_pexists
BETA_CONV	PBETA_CONV	list_mk_forall	list_mk_pforall
BETA_RULE	PBETA_RULE	MATCH_MP	PMATCH_MP
BETA_TAC	PBETA_TAC	MATCH_MP_TAC	PMATCH_MP_TAC
bndvar	bndpair	MK_ABS	MK_PABS
body	pbody	mk_abs	mk_pabs
CHOOSE	PCHOOSE	MK_EXISTS	MK_PEXISTS
CHOOSE_TAC	PCHOOSE_TAC	mk_exists	mk_pexists
CHOOSE_THEN	PCHOOSE_THEN	mk_forall	mk_pforall
dest_abs	dest_pabs	mk_select	mk_pselect
dest_exists	dest_pexists	NOT_EXISTS_CONV	NOT_PEXISTS_CONV
dest_forall	dest_pforall	NOT_FORALL_CONV	NOT_PFORALL_CONV
dest_select	dest_pselect	OR_EXISTS_CONV	OR_PEXISTS_CONV
ETA_CONV	PETA_CONV	OR_FORALL_CONV	OR_PFORALL_CONV
EXISTENCE	PEXISTENCE	PART_MATCH	PART_PMATCH
EXISTS	PEXISTS	RIGHT_AND_EXISTS_CONV	RIGHT_AND_PEXISTS_CONV
EXISTS_AND_CONV	PEXISTS_AND_CONV	RIGHT_AND_FORALL_CONV	RIGHT_AND_PFORALL_CONV
EXISTS_EQ	PEXISTS_EQ	RIGHT_BETA	RIGHT_PBETA
EXISTS_IMP	PEXISTS_IMP	RIGHT_IMP_EXISTS_CONV	RIGHT_IMP_PEXISTS_CONV
EXISTS_IMP_CONV	PEXISTS_IMP_CONV	RIGHT_IMP_FORALL_CONV	RIGHT_IMP_PFORALL_CONV
EXISTS_NOT_CONV	PEXISTS_NOT_CONV	RIGHT_LIST_BETA	RIGHT_LIST_PBETA
EXISTS_OR_CONV	PEXISTS_OR_CONV	RIGHT_OR_EXISTS_CONV	RIGHT_OR_PEXISTS_CONV
EXISTS_TAC	PEXISTS_TAC	RIGHT_OR_FORALL_CONV	RIGHT_OR_PFORALL_CONV
EXISTS_UNIQUE_CONV	PEXISTS_UNIQUE_CONV	SELECT_CONV	PSELECT_CONV
EXT	PEXT	SELECT_ELIM	PSELECT_ELIM
FILTER_GEN_TAC	FILTER_PGEN_TAC	SELECT_EQ	PSELECT_EQ
FILTER_STRIP_TAC	FILTER_PSTRIP_TAC	SELECT_INTRO	PSELECT_INTRO
FILTER_STRIP_THEN	FILTER_PSTRIP_THEN	SELECT_RULE	PSELECT_RULE
FORALL_AND_CONV	PFORALL_AND_CONV	SKOLEM_CONV	PSKOLEM_CONV
FORALL_EQ	PFORALL_EQ	SPEC	PSPEC
FORALL_IMP_CONV	PFORALL_IMP_CONV	SPECL	PSPECL
FORALL_NOT_CONV	PFORALL_NOT_CONV	SPEC_ALL	PSPEC_ALL
FORALL_OR_CONV	PFORALL_OR_CONV	SPEC_TAC	PSPEC_TAC
free_in	occs_in	SPEC_VAR	PSPEC_PAIR
GEN	PGEN	strip_abs	strip_pabs
GEN_ALPHA_CONV	GEN_PALPHA_CONV	STRIP_ASSUME_TAC	PSTRIP_ASSUME_TAC
GEN_TAC	PGEN_TAC	strip_exists	strip_pexists
GENL	PGENL	strip_forall	strip_pforall
genvar	genlike	STRIP_GOAL_THEN	PSTRIP_GOAL_THEN
GSPEC	GPSPEC	STRIP_TAC	PSTRIP_TAC
HALF_MK_ABS	HALF_MK_PABS	STRIP_THM_THEN	PSTRIP_THM_THEN
is_abs	is_pabs	STRUCT_CASES_TAC	PSTRUCT_CASES_TAC
is_exists	is_pexists	SUB_CONV	PSUB_CONV
is_forall	is_pforall	SWAP_EXISTS_CONV	SWAP_PEXISTS_CONV
is_select	is_pselect	variant	pvariant
is_var	is_pvar	X_CHOOSE_TAC	P_PCHOOSE_TAC
ISPEC	IPSPEC	X_CHOOSE_THEN	P_PCHOOSE_THEN
ISPECL	IPSPECL	X_FUN_EQ_CONV	P_FUN_EQ_CONV
LEFT_AND_EXISTS_CONV	LEFT_AND_PEXISTS_CONV	X_GEN_TAC	P_PGEN_TAC
LEFT_AND_FORALL_CONV	LEFT_AND_PFORALL_CONV	X_SKOLEM_CONV	P_PSKOLEM_CONV
LEFT_IMP_EXISTS_CONV	LEFT_IMP_PEXISTS_CONV		

The pair library also contains many functions for which there are no analogous non-paired functions.

2.1 Getting Started

Before you can use any of the functions described in this manual, you must load the pair library. To load the pair library, issue the following command:

```
load_library `pair`;;
```

The pair library contains no theories, so it is always possible to load it.

2.2 The pair Library Philosophy

Two main design decisions should be noted about the pair library. These decisions run counter to the usual HOL philosophy that each inference rule should perform a single simple inference, and should do so only under a particular restricted set of circumstances. The philosophy of the pair library is that each inference rule should do whatever is necessary to eliminate the distinctions between reasoning about paired and unpaired abstractions and quantifications.

The first design decision is that all the functions for dealing with paired quantifications and abstractions have a very general notion of what a pair is. For the purposes of such functions, a pair may be an arbitrary *paired structure*. A paired structure is either a term, or a pair of terms which may themselves be paired structures. For example, the following are all considered to be paired structures:

```
a    (a,b)    (a,b,c)    ((a1,a2),(b1,b2))    ((a1,a2),(b2,b2),(c1,c2))
```

Note that it is always possible to use the paired version of an inference rule in place of the standard version.

The other design decision is that the a pair (or subpair) bound by a paired abstraction should be treated as much like a single variable as possible. This means that paired and nonpaired abstractions can be considered α-equivalent. For example:

```
#PALPHA "\(x,y). (f (x,y))" "\xy. (f xy)";;
|- (\(x,y). f(x,y)) = (\xy. f xy)
```

The effect of this decision can be seen in evidence in β-conversion and other inference rules:

```
#PBETA_CONV "(\(x,y). (f x y (x,y))) ab";;
|- (\(x,y). f x y(x,y))xy = f(FST ab)(SND ab)ab
```

2.3 Bugs and Future Changes

At the time of release there were no known bugs in the system. However, this is more likely to be a result of poor testing than of good coding. If you do find a bug please report it to me, preferably along with a short example that exhibits the bug and the version number of the pair library that you are using. The constant `pair_version` contains the version number of the pair library. I will provide bug fixes as soon as possible. I can be contacted at:

Jim Grundy		
Defence Science & Technology Organisation	phone:	+61 8 259 6162
Building 171 Laboratories Area	fax:	+61 8 259 5980
PO Box 1500	telex:	AA82799
Salisbury SA 5108	email:	Jim.Grundy@dsto.defence.gov.au
AUSTRALIA		jim@grundy-j.apana.org.au

I would also welcome any suggestions for improving to the library, including optimisations and suggestions for new functions.

Chapter 3

ML Functions in the pair Library

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

AND_PEXISTS_CONV

```
AND_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification outwards through a conjunction.

Description
When applied to a term of the form `(?p. t) /\ (?p. u)`, where no variables in `p` are free in either `t` or `u`, `AND_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) /\ (?p. u) = (?p. t /\ u)
```

Failure
`AND_PEXISTS_CONV` fails if it is applied to a term not of the form `(?p. t) /\ (?p. u)`, or if it is applied to a term `(?p. t) /\ (?p. u)` in which variables from `p` are free in either `t` or `u`.

See also
AND_EXISTS_CONV, PEXISTS_AND_CONV, LEFT_AND_PEXISTS_CONV, RIGHT_AND_PEXISTS_CONV.

AND_PFORALL_CONV

```
AND_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification outwards through a conjunction.

Description
When applied to a term of the form `(!p. t) /\ (!p. t)`, the conversion `AND_PFORALL_CONV` returns the theorem:

```
|- (!p. t) /\ (!p. u) = (!p. t /\ u)
```

Failure
Fails if applied to a term not of the form `(!p. t) /\ (!p. t)`.

See also
`AND_FORALL_CONV`, `PFORALL_AND_CONV`, `LEFT_AND_PFORALL_CONV`, `RIGHT_AND_PFORALL_CONV`.

bndpair

```
bndpair : (term -> term)
```

Synopsis
Returns the bound pair of a paired abstraction.

Description
`bndpair "\pair. t"` returns `"pair"`.

Failure
Fails unless the term is a paired abstraction.

See also
`bndvar`, `pbody`, `dest_pabs`.

CURRY_CONV

```
CURRY_CONV : conv
```

Synopsis
Currys an application of a paired abstraction.

Example

```
#CURRY_CONV "(\(x,y). x + y) (1,2)";;
|- (\(x,y). x + y)(1,2) = (\x y. x + y)1 2

#CURRY_CONV "(\(x,y). x + y) z";;
|- (\(x,y). x + y)z = (\x y. x + y)(FST z)(SND z)
```

Failure
`CURRY_CONV tm` fails if `tm` is not an application of a paired abstraction.

See also
UNCURRY_CONV.

CURRY_EXISTS_CONV

```
CURRY_EXISTS_CONV : conv
```

Synopsis
Currys paired existential quantifications into consecutive existential quantifications.

Example

```
#CURRY_EXISTS_CONV "?(x,y). x + y = y + x";;
|- (?(x,y). x + y = y + x) = (?x y. x + y = y + x)

#CURRY_EXISTS_CONV "?((w,x),(y,z)). w+x+y+z = z+y+x+w";;
|- (?((w,x),y,z). w + (x + (y + z)) = z + (y + (x + w))) =
   (?(w,x) (y,z). w + (x + (y + z)) = z + (y + (x + w)))
```

Failure
`CURRY_EXISTS_CONV tm` fails if `tm` is not a paired existential quantification.

See also
CURRY_CONV, UNCURRY_CONV, UNCURRY_EXISTS_CONV, CURRY_FORALL_CONV, UNCURRY_FORALL_CONV.

CURRY_FORALL_CONV

```
CURRY_FORALL_CONV : conv
```

Synopsis
Currys paired universal quantifications into consecutive universal quantifications.

Example

```
#CURRY_FORALL_CONV "!(x,y). x + y = y + x";;
|- (!(x,y). x + y = y + x) = (!x y. x + y = y + x)

#CURRY_FORALL_CONV "!((w,x),(y,z)). w+x+y+z = z+y+x+w";;
|- (!((w,x),y,z). w + (x + (y + z)) = z + (y + (x + w))) =
   (!(w,x) (y,z). w + (x + (y + z)) = z + (y + (x + w)))
```

Failure
`CURRY_FORALL_CONV tm` fails if `tm` is not a paired universal quantification.

See also
`CURRY_CONV`, `UNCURRY_CONV`, `UNCURRY_FORALL_CONV`, `CURRY_EXISTS_CONV`, `UNCURRY_EXISTS_CONV`.

dest_pabs

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

Synopsis
Breaks apart a paired abstraction into abstracted pair and body.

Description
`dest_pabs` is a term destructor for paired abstractions: `dest_abs "\pair. t"` returns `("pair","t")`.

Failure
Fails with `dest_pabs` if term is not a paired abstraction.

See also
`dest_abs`, `mk_pabs`, `is_pabs`, `strip_pabs`.

dest_pexists

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

Synopsis
Breaks apart paired existential quantifiers into the bound pair and the body.

Description
`dest_pexists` is a term destructor for paired existential quantification. The application of `dest_pexists` to `"?pair. t"` returns `("pair","t")`.

Failure
Fails with `dest_pexists` if term is not a paired existential quantification.

See also
`dest_exists`, `mk_pexists`, `is_pexists`, `strip_pexists`.

dest_pforall

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

Synopsis
Breaks apart paired universal quantifiers into the bound pair and the body.

Description
`dest_pforall` is a term destructor for paired universal quantification. The application of `dest_pforall` to `"!pair. t"` returns `("pair","t")`.

Failure
Fails with `dest_pforall` if term is not a paired universal quantification.

See also
`dest_forall`, `mk_pforall`, `is_pforall`, `strip_pforall`.

dest_prod

```
dest_prod : (type -> (type # type))
```

Synopsis
Breaks apart a product type into its component types.

Description
`dest_prod` is a type destructor for products: `dest_pair ":t1#t2"` returns `(":t1",":t2")`.

Failure
Fails with `dest_prod` if the argument is not a product type.

See also
`is_prod`, `mk_prod`.

dest_pselect

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

Synopsis
Breaks apart a paired choice-term into the selected pair and the body.

Description
`dest_pselect` is a term destructor for paired choice terms. The application of `dest_select` to `"@pair. t"` returns `("pair","t")`.

Failure
Fails with `dest_pselect` if term is not a paired choice-term.

See also
`dest_select`, `mk_pselect`, `is_pselect`.

FILTER_PGEN_TAC

```
FILTER_PGEN_TAC : (term -> tactic)
```

Synopsis
Strips off a paired universal quantifier, but fails for a given quantified pair.

Description
When applied to a term `q` and a goal `A ?- !p. t`, the tactic `FILTER_PGEN_TAC` fails if the quantified pair `p` is the same as `p`, but otherwise advances the goal in the same way as `PGEN_TAC`, i.e. returns the goal `A ?- t[p'/p]` where `p'` is a variant of `p` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `p'` is just `p`.

```
    A ?- !p. t
  ==============  FILTER_PGEN_TAC "q"
   A ?- t[p'/p]
```

Failure
Fails if the goal's conclusion is not a paired universal quantifier or the quantified pair is equal to the given term.

See also
`FILTER_GEN_TAC`, `PGEN`, `PGEN_TAC`, `PGENL`, `PGEN_ALL`, `PSPEC`, `PSPECL`, `PSPEC_ALL`, `PSPEC_TAC`, `PSTRIP_TAC`.

FILTER_PSTRIP_TAC

```
FILTER_PSTRIP_TAC : (term -> tactic)
```

Synopsis
Conditionally strips apart a goal by eliminating the outermost connective.

Description
Stripping apart a goal in a more careful way than is done by `PSTRIP_TAC` may be necessary when dealing with quantified terms and implications. `FILTER_PSTRIP_TAC` behaves like `PSTRIP_TAC`, but it does not strip apart a goal if it contains a given term.

If `u` is a term, then `FILTER_PSTRIP_TAC u` is a tactic that removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`, provided the term being stripped does not contain `u`. `FILTER_PSTRIP_TAC` will strip paired universal quantifications. A negation `~t` is treated as the implication `t ==> F`. `FILTER_PSTRIP_TAC` also breaks apart conjunctions without applying any filtering.

If `t` is a universally quantified term, `FILTER_PSTRIP_TAC u` strips off the quantifier:

```
    A ?- !p. v
================  FILTER_PSTRIP_TAC "u"       [where p is not u]
   A ?- v[p'/p]
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the assumptions `A`. If `t` is a conjunction, no filtering is done and `FILTER_PSTRIP_TAC` simply splits the conjunction:

```
    A ?- v /\ w
=================  FILTER_PSTRIP_TAC "u"
  A ?- v   A ?- w
```

If `t` is an implication and the antecedent does not contain a free instance of `u`, then `FILTER_PSTRIP_TAC u` moves the antecedent into the assumptions and recursively splits the antecedent according to the following rules (see `PSTRIP_ASSUME_TAC`):

```
  A ?- v1 /\ ... /\ vn ==> v              A ?- v1 \/ ... \/ vn ==> v
============================            =================================
    A u {v1,...,vn} ?- v                 A u {v1} ?- v ... A u {vn} ?- v

   A ?- (?p. w) ==> v
====================
  A u {w[p'/p]} ?- v
```

where `p'` is a variant of the pair `p`.

Failure

`FILTER_PSTRIP_TAC u (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains `u` in the sense described above (conjunction excluded).

Uses

`FILTER_PSTRIP_TAC` is used when stripping outer connectives from a goal in a more delicate way than `PSTRIP_TAC`. A typical application is to keep stripping by using the tactic `REPEAT (FILTER_PSTRIP_TAC u)` until one hits the term `u` at which stripping is to stop.

See also

`PGEN_TAC`, `PSTRIP_GOAL_THEN`, `FILTER_PSTRIP_THEN`, `PSTRIP_TAC`, `FILTER_STRIP_TAC`.

FILTER_PSTRIP_THEN

```
FILTER_PSTRIP_THEN : (thm_tactic -> term -> tactic)
```

Synopsis
Conditionally strips a goal, handing an antecedent to the theorem-tactic.

Description
Given a theorem-tactic `ttac`, a term `u` and a goal `(A,t)`, `FILTER_STRIP_THEN ttac u` removes one outer connective (`!`, `==>`, or `~`) from `t`, if the term being stripped does not contain a free instance of `u`. Note that `FILTER_PSTRIP_THEN` will strip paired universal quantifiers. A negation `~t` is treated as the implication `t ==> F`. The theorem-tactic `ttac` is applied only when stripping an implication, by using the antecedent stripped off. `FILTER_PSTRIP_THEN` also breaks conjunctions.

`FILTER_PSTRIP_THEN` behaves like `PSTRIP_GOAL_THEN`, if the term being stripped does not contain a free instance of `u`. In particular, `FILTER_PSTRIP_THEN PSTRIP_ASSUME_TAC` behaves like `FILTER_PSTRIP_TAC`.

Failure
`FILTER_PSTRIP_THEN ttac u (A,t)` fails if `t` is not a paired universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains the term `u` (conjunction excluded); or if the application of `ttac` fails, after stripping the goal.

Uses
`FILTER_PSTRIP_THEN` is used to manipulate intermediate results using theorem-tactics, after stripping outer connectives from a goal in a more delicate way than `PSTRIP_GOAL_THEN`.

See also
`PGEN_TAC, PSTRIP_GOAL_THEN, FILTER_STRIP_THEN, PSTRIP_TAC, FILTER_PSTRIP_TAC.`

genlike

```
genlike : (term -> term)
```

Synopsis
Returns a pair structure of variables whose names have not been previously used.

Description
When given a pair structure, `genlike` returns a paired structure of variables whose names have not been used for variables or constants in the HOL session so far. The structure of the term returned will be identical to the structure of the argument.

Failure
Never fails.

Example
The following example illustrates the behaviour of `genlike`:

```
#genlike "((1,2),(x:*,x:*))";;
"(GEN%VAR%487,GEN%VAR%488),GEN%VAR%489,GEN%VAR%490" : term
```

Uses
Unique variables are useful in writing derived rules, for specializing terms without having to worry about such things as free variable capture. It is often important in such rules to keep the same structure. If not, `genvar` will be adequate. If the names are to be visible to a typical user, the function `pvariant` can provide rather more meaningful names.

See also
`genvar, GPSPEC, pvariant.`

GEN_PALPHA_CONV

```
GEN_PALPHA_CONV : (term -> conv)
```

Synopsis
Renames the bound pair of a paired abstraction, quantified term, or other binder.

Description
The conversion `GEN_PALPHA_CONV` provides alpha conversion for lambda abstractions of the form `"\p.t"`, quantified terms of the forms `"!p.t"`, `"?p.t"` or `"?!p.t"`, and epsilon terms of the form `"@p.t"`. In general, if `B` is a binder constant, then `GEN_ALPHA_CONV` implements alpha conversion for applications of the form `"B p.t"`. The function `is_binder` determines what is regarded as a binder in this context.

The renaming of pairs is as described for `PALPHA_CONV`.

Failure
`GEN_PALPHA_CONV q tm` fails if `q` is not a variable, or if `tm` does not have one of the forms `"\p.t"` or `"B p.t"`, where `B` is a binder (that is, `is_binder ‘B‘` returns `true`). `GEN_ALPHA_CONV q tm` also fails if `tm` does have one of these forms, but types of the variables `p` and `q` differ.

See also
GEN_ALPHA_CONV, PALPHA, PALPHA_CONV, is_binder.

GPSPEC

```
GPSPEC : (thm -> thm)
```

Synopsis
Specializes the conclusion of a theorem with unique pairs.

Description
When applied to a theorem `A |- !p1...pn. t`, where the number of universally quantified variables may be zero, `GPSPEC` returns `A |- t[g1/p1]...[gn/pn]`, where the `gi` is paired structures of the same structure as `pi` and made up of distinct variables , chosen by `genvar`.

```
      A |- !p1...pn. t
   ------------------------  GPSPEC
    A |- t[g1/p1]...[gn/pn]
```

Failure
Never fails.

Uses
`GPSPEC` is useful in writing derived inference rules which need to specialize theorems while avoiding using any variables that may be present elsewhere.

See also
GSPEC, PGEN, PGENL, genvar, PGEN_ALL, PGEN_TAC, PSPEC, PSPECL, PSPEC_ALL, PSPEC_TAC, PSPEC_PAIR.

HALF_MK_PABS

```
HALF_MK_PABS : (thm -> thm)
```

Synopsis
Converts a function definition to lambda-form.

Description

When applied to a theorem `A |- !p. t1 p = t2`, whose conclusion is a universally quantified equation, `HALF_MK_PABS` returns the theorem `A |- t1 = (\p. t2)`.

```
    A |- !p. t1 p = t2
   --------------------  HALF_MK_PABS            [where p is not free in t1]
    A |- t1 = (\p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation whose left-hand side is a function applied to the quantified pair, or if any of the the variables in the quantified pair is free in that function.

See also

`HALF_MK_ABS`, `PETA_CONV`, `MK_PABS`, `MK_PEXISTS`.

IPSPEC

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

Synopsis

Specializes a theorem, with type instantiation if necessary.

Description

This rule specializes a paired quantification as does `PSPEC`; it differs from it in also instantiating the type if needed:

```
     A |- !p:ty.tm
  -----------------------  IPSPEC "q:ty'"
      A |- tm[q/p]
```

(where `q` is free for `p` in `tm`, and `ty'` is an instance of `ty`).

Failure

`IPSPEC` fails if the input theorem is not universally quantified, if the type of the given term is not an instance of the type of the quantified variable, or if the type variable is free in the assumptions.

See also

`ISPEC`, `INST_TY_TERM`, `INST_TYPE`, `IPSPECL`, `PSPEC`, `match`.

IPSPECL

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

Synopsis
Specializes a theorem zero or more times, with type instantiation if necessary.

Description
`IPSPECL` is an iterative version of `IPSPEC`

```
        A |- !p1...pn.tm
   ----------------------------  IPSPECL ["q1",...,"qn"]
    A |- t[q1,...qn/p1,...,pn]
```

(where `qi` is free for `pi` in `tm`).

Failure
`IPSPECL` fails if the list of terms is longer than the number of quantified variables in the term, if the type instantiation fails, or if the type variable being instantiated is free in the assumptions.

See also
`ISPECL`, `INST_TYPE`, `INST_TY_TERM`, `IPSPEC`, `MATCH`, `SPEC`, `PSPECL`.

is_pabs

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

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

Description
`is_pabs "\pair. t"` returns `true`. If the term is not a paired abstraction the result is `false`.

Failure
Never fails.

See also
`is_abs`, `mk_pabs`, `dest_pabs`.

is_pexists

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

Synopsis
Tests a term to see if it as a paired existential quantification.

Description
`is_pexists "?pair. t"` returns `true`. If the term is not a paired existential quantification the result is `false`.

Failure
Never fails.

See also
`is_exists`, `mk_pexists`, `dest_pexists`.

is_pforall

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

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

Description
`is_pforall "!pair. t"` returns `true`. If the term is not a a paired universal quantification the result is `false`.

Failure
Never fails.

See also
`is_forall`, `mk_pforall`, `dest_pforall`.

is_prod

`is_prod : (type -> bool)`

Synopsis
Tests a type to see if it is a product type.

Description
`is_prod ":t1#t2"` returns `true`.

Failure
Never fails.

See also
`dest_prod`, `mk_prod`.

is_pselect

`is_pselect : (term -> bool)`

Synopsis
Tests a term to see if it is a paired choice-term.

Description
`is_select "@pair. t"` returns `true`. If the term is not a paired choice-term the result is `false`.

Failure
Never fails.

See also
`is_select`, `mk_pselect`, `dest_pselect`.

is_pvar

`is_pvar : (term -> bool)`

Synopsis
Tests a term to see if it is a paired structure of variables.

Description
`is_pvar "pvar"` returns `true` iff `pvar` is a paired structure of variables. For example, `((a:*,b:*),(d:*,e:*))` is a paired structure of variables, `(1,2)` is not.

Failure
Never fails.

See also
`is_var`.

LEFT_AND_PEXISTS_CONV

```
LEFT_AND_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the left conjunct outwards through a conjunction.

Description
When applied to a term of the form `(?p. t) /\ u`, the conversion `LEFT_AND_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) /\ u = (?p'. t[p'/p] /\ u)
```

where `p'` is a primed variant of the pair `p` that does not contains variables free in the input term.

Failure
Fails if applied to a term not of the form `(?p. t) /\ u`.

See also
`LEFT_AND_EXISTS_CONV`, `AND_PEXISTS_CONV`, `PEXISTS_AND_CONV`, `RIGHT_AND_PEXISTS_CONV`.

LEFT_AND_PFORALL_CONV

```
LEFT_AND_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the left conjunct outwards through a conjunction.

Description
When applied to a term of the form `(!p. t) /\ u`, the conversion `LEFT_AND_PFORALL_CONV` returns the theorem:

```
|- (!p. t) /\ u = (!p'. t[p'/p] /\ u)
```

where `p'` is a primed variant of `p` that does not appear free in the input term.

Failure
Fails if applied to a term not of the form `(!p. t) /\ u`.

See also
`LEFT_AND_FORALL_CONV`, `AND_PFORALL_CONV`, `PFORALL_AND_CONV`, `RIGHT_AND_PFORALL_CONV`.

LEFT_IMP_PEXISTS_CONV

```
LEFT_IMP_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the antecedent outwards through an implication.

Description
When applied to a term of the form `(?p. t) ==> u`, the conversion `LEFT_IMP_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) ==> u = (!p'. t[p'/p] ==> u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure
Fails if applied to a term not of the form `(?p. t) ==> u`.

See also
`LEFT_IMP_EXISTS_CONV`, `PFORALL_IMP_CONV`, `RIGHT_IMP_PFORALL_CONV`.

LEFT_IMP_PFORALL_CONV

```
LEFT_IMP_PFORALL_CONV : conv
```

Synopsis

Moves a paired universal quantification of the antecedent outwards through an implication.

Description

When applied to a term of the form `(!p. t) ==> u`, the conversion `LEFT_IMP_PFORALL_CONV` returns the theorem:

```
|- (!p. t) ==> u = (?p'. t[p'/p] ==> u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `(!p. t) ==> u`.

See also

`LEFT_IMP_FORALL_CONV`, `PEXISTS_IMP_CONV`, `RIGHT_IMP_PFORALL_CONV`.

LEFT_LIST_PBETA

```
LEFT_LIST_PBETA : (thm -> thm)
```

Synopsis

Iteratively beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

Description

When applied to an equational theorem, `LEFT_LIST_PBETA` applies paired beta-reduction over a top-level chain of beta-redexes to the left-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
    A |- (\p1...pn. t) q1 ... qn = s
   ----------------------------------  LEFT_LIST_BETA
       A |- t[q1/p1]...[qn/pn] = s
```

Failure
Fails unless the theorem is equational, with its left-hand side being a top-level paired beta-redex.

See also
RIGHT_LIST_BETA, PBETA_CONV, PBETA_RULE, PBETA_TAC, LIST_PBETA_CONV, LEFT_PBETA, RIGHT_PBETA, RIGHT_LIST_PBETA.

LEFT_OR_PEXISTS_CONV

```
LEFT_OR_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the left disjunct outwards through a disjunction.

Description
When applied to a term of the form `(?p. t) \/ u`, the conversion `LEFT_OR_PEXISTS_CONV` returns the theorem:

```
   |- (?p. t) \/ u = (?p'. t[p'/p] \/ u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure
Fails if applied to a term not of the form `(?p. t) \/ u`.

See also
LEFT_OR_EXISTS_CONV, PEXISTS_OR_CONV, OR_PEXISTS_CONV, RIGHT_OR_PEXISTS_CONV.

LEFT_OR_PFORALL_CONV

```
LEFT_OR_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the left disjunct outwards through a disjunction.

Description
When applied to a term of the form `(!p. t) \/ u`, the conversion `LEFT_OR_FORALL_CONV` returns the theorem:

```
|- (!p. t) \/ u = (!p'. t[p'/p] \/ u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure
Fails if applied to a term not of the form `(!p. t) \/ u`.

See also
`LEFT_OR_FORALL_CONV`, `OR_PFORALL_CONV`, `PFORALL_OR_CONV`, `RIGHT_OR_PFORALL_CONV`.

LEFT_PBETA

```
LEFT_PBETA : (thm -> thm)
```

Synopsis
Beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

Description
When applied to an equational theorem, `LEFT_PBETA` applies paired beta-reduction at top level to the left-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
    A |- (\x. t1) t2 = s
   ----------------------  LEFT_PBETA
     A |- t1[t2/x] = s
```

Failure
Fails unless the theorem is equational, with its left-hand side being a top-level paired beta-redex.

See also
`RIGHT_BETA`, `PBETA_CONV`, `PBETA_RULE`, `PBETA_TAC`, `RIGHT_PBETA`, `RIGHT_LIST_PBETA`, `LEFT_LIST_PBETA`.

list_mk_pabs

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

Synopsis
Iteratively constructs paired abstractions.

Description
`list_mk_pabs(["p1";...;"pn"],"t")` returns `"\p1 ... pn. t"`.

Failure
Fails with `list_mk_pabs` if the terms in the list are not paired structures of variables.

Comments
The system shows the type as `goal -> term`.

See also
`list_mk_abs`, `strip_pabs`, `mk_pabs`.

list_mk_pexists

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

Synopsis
Iteratively constructs paired existential quantifications.

Description
`list_mk_pexists(["p1";...;"pn"],"t")` returns `"?p1 ... pn. t"`.

Failure
Fails with `list_mk_pexists` if the terms in the list are not paired structures of variables or if `t` is not of type `":bool"` and the list of terms is nonempty. If the list of terms is empty the type of `t` can be anything.

Comments
The system shows the type as (goal -> term).

See also
list_mk_exists, strip_pexists, mk_pexists.

LIST_MK_PEXISTS

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

Synopsis
Multiply existentially quantifies both sides of an equation using the given pairs.

Description
When applied to a list of terms [p1;...;pn], where the pi are all paired structures of variables, and a theorem A |- t1 = t2, the inference rule LIST_MK_PEXISTS existentially quantifies both sides of the equation using the pairs given, none of the variables in the pairs should be free in the assumption list.

```
              A |- t1 = t2
--------------------------------------  LIST_MK_PEXISTS ["x1";...;"xn"]
 A |- (?x1...xn. t1) = (?x1...xn. t2)
```

Failure
Fails if any term in the list is not a paired structure of variables, or if any variable is free in the assumption list, or if the theorem is not equational.

See also
LIST_MK_EXISTS, PEXISTS_EQ, MK_PEXISTS.

list_mk_pforall

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

Synopsis
Iteratively constructs a paired universal quantification.

Description

`list_mk_pforall(["p1";...;"pn"],"t")` returns `"!p1 ... pn. t"`.

Failure

Fails with `list_mk_pforall` if the terms in the list are not paired structures of variables or if `t` is not of type `":bool"` and the list of terms is nonempty. If the list of terms is empty the type of `t` can be anything.

Comments

The system shows the type as `(goal -> term)`.

See also

`list_mk_forall`, `strip_pforall`, `mk_pforall`.

LIST_MK_PFORALL

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

Synopsis

Multiply universally quantifies both sides of an equation using the given pairs.

Description

When applied to a list of terms `[p1;...;pn]`, where the `pi` are all paired structures of variables, and a theorem `A |- t1 = t2`, the inference rule `LIST_MK_PFORALL` universally quantifies both sides of the equation using the pairs given, none of the variables in the pairs should be free in the assumption list.

```
              A |- t1 = t2
--------------------------------------  LIST_MK_PFORALL ["x1";...;"xn"]
 A |- (!x1...xn. t1) = (!x1...xn. t2)
```

Failure

Fails if any term in the list is not a paired structure of variables, or if any variable is free in the assumption list, or if the theorem is not equational.

See also

`LIST_MK_EXISTS`, `PFORALL_EQ`, `MK_PFORALL`.

LIST_PBETA_CONV

```
LIST_PBETA_CONV : conv
```

Synopsis
Performs an iterated paired beta-conversion.

Description
The conversion `LIST_PBETA_CONV` maps terms of the form

```
"(\p1 p2 ... pn. t) q1 q2 ... qn"
```

to the theorems of the form

```
|- (\p1 p2 ... pn. t) q1 q2 ... qn = t[q1/p1][q2/p2] ... [qn/pn]
```

where `t[qi/pi]` denotes the result of substituting `qi` for all free occurrences of `pi` in `t`, after renaming sufficient bound variables to avoid variable capture.

Failure
`LIST_PBETA_CONV tm` fails if `tm` does not have the form `"(\p1 ... pn. t) q1 ... qn"` for `n` greater than 0.

Example

```
#LIST_PBETA_CONV "(\(a,b) (c,d) . a + b + c + d) (1,2) (3,4)";;
|- (\(a,b) (c,d). a + (b + (c + d)))(1,2)(3,4) = 1 + (2 + (3 + 4))
```

See also
`LIST_BETA_CONV`, `PBETA_CONV`, `BETA_RULE`, `BETA_TAC`, `RIGHT_PBETA`, `RIGHT_LIST_PBETA`, `LEFT_PBETA`, `LEFT_LIST_PBETA`.

mk_pabs

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

Synopsis
Constructs a paired abstraction.

Description

`mk_pabs "pair","t"` returns the abstraction `"\pair. t"`.

Failure

Fails with `mk_pabs` if first term is not a pair structure of variables.

See also

`mk_abs, dest_pabs, is_pabs, list_mk_pabs.`

MK_PABS

```
MK_PABS : (thm -> thm)
```

Synopsis

Abstracts both sides of an equation.

Description

When applied to a theorem `A |- !p. t1 = t2`, whose conclusion is a paired universally quantified equation, `MK_PABS` returns the theorem `A |- (\p. t1) = (\p. t2)`.

```
        A |- !p. t1 = t2
   --------------------------  MK_PABS
    A |- (\p. t1) = (\p. t2)
```

Failure

Fails unless the theorem is a (singly) paired universally quantified equation.

See also

`MK_ABS, PABS, HALF_MK_PABS, MK_PEXISTS.`

MK_PAIR

```
MK_PAIR : (thm -> thm -> thm)
```

Synopsis

Proves equality of pairs constructed from equal components.

Description

When applied to theorems `A1 |- a = x` and `A2 |- b = y`, the inference rule `MK_PAIR` returns the theorem `A1 u A2 |- (a,b) = (x,y)`.

```
    A1 |- a = x   A2 |- b = y
   ---------------------------  MK_PAIR
    A1 u A2 |- (a,b) = (x,y)
```

Failure

Fails unless both theorems are equational.

See also

mk_pexists

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

Synopsis

Constructs a paired existential quantification.

Description

`mk_pexists("pair","t")` returns `"?pair. t"`.

Failure

Fails with `mk_exists` if first term is not a paired structure of variables or if `t` is not of type `":bool"`.

See also

`mk_exists`, `dest_pexists`, `is_pexists`, `list_mk_pexists`.

MK_PEXISTS

```
MK_PEXISTS : (thm -> thm)
```

Synopsis

Existentially quantifies both sides of a universally quantified equational theorem.

Description

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PEXISTS` returns the theorem `A |- (?x. t1) = (?x. t2)`.

```
       A |- !p. t1 = t2
   --------------------------  MK_PEXISTS
    A |- (?p. t1) = (?p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PEXISTS_EQ, PGEN, LIST_MK_PEXISTS, MK_PABS.`

mk_pforall

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

Synopsis

Constructs a paired universal quantification.

Description

`mk_pforall("pair","t")` returns `"!pair. t"`.

Failure

Fails with `mk_pforall` if first term is not a a paired structure of variables or if `t` is not of type `":bool"`.

See also

`mk_forall, dest_pforall, is_pforall, list_mk_pforall.`

MK_PFORALL

```
MK_PFORALL : (thm -> thm)
```

Synopsis

Universally quantifies both sides of a universally quantified equational theorem.

Description

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PFORALL` returns the theorem `A |- (!x. t1) = (!x. t2)`.

```
       A |- !p. t1 = t2
   --------------------------  MK_PFORALL
    A |- (!p. t1) = (!p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PFORALL_EQ`, `LIST_MK_PFORALL`, `MK_PABS`.

mk_prod

```
mk_prod : ((type # type) -> type)
```

Synopsis

Constructs a product type from two constituent types.

Description

`mk_prod(":t1",":t2")` returns `":t1#t2"`.

Failure

Never fails.

See also

`is_prod`, `dest_prod`.

mk_pselect

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

Synopsis

Constructs a paired choice-term.

Description

`mk_pselect("pair","t")` returns `"@pair. t"`.

Failure

Fails with `mk_select` if first term is not a paired structure of variables or if `t` is not of type `":bool"`.

See also

`mk_select, dest_pselect, is_pselect.`

MK_PSELECT

```
MK_PSELECT : (thm -> thm)
```

Synopsis

Quantifies both sides of a universally quantified equational theorem with the choice quantifier.

Description

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PSELECT` returns the theorem `A |- (@x. t1) = (@x. t2)`.

```
       A |- !p. t1 = t2
   --------------------------  MK_PSELECT
    A |- (@p. t1) = (@p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PSELECT_EQ, MK_PABS.`

NOT_PEXISTS_CONV

```
NOT_PEXISTS_CONV : conv
```

Synopsis
Moves negation inwards through a paired existential quantification.

Description
When applied to a term of the form `~(?p. t)`, the conversion `NOT_PEXISTS_CONV` returns the theorem:

```
|- ~(?p. t) = (!p. ~t)
```

Failure
Fails if applied to a term not of the form `~(?p. t)`.

See also
`NOT_EXISTS_CONV`, `PEXISTS_NOT_CONV`, `PFORALL_NOT_CONV`, `NOT_PFORALL_CONV`.

NOT_PFORALL_CONV

```
NOT_PFORALL_CONV : conv
```

Synopsis
Moves negation inwards through a paired universal quantification.

Description
When applied to a term of the form `~(!p. t)`, the conversion `NOT_PFORALL_CONV` returns the theorem:

```
|- ~(!p. t) = (?p. ~t)
```

It is irrelevant whether any variables in `p` occur free in `t`.

Failure
Fails if applied to a term not of the form `~(!p. t)`.

See also
`NOT_FORALL_CONV`, `PEXISTS_NOT_CONV`, `PFORALL_NOT_CONV`, `NOT_PEXISTS_CONV`.

occs_in

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

Synopsis
Occurrence check for bound variables.

Description
When applied to two terms `p` and `t`, where `p` is a paired structure of variables, the function `occs_in` returns `true` if and of the constituent variables of `p` occurs free in `t`, and `false` otherwise.

Failure
Fails of `p` is not a paired structure of variables.

See also
`free_in, frees, freesl, thm_frees.`

OR_PEXISTS_CONV

```
OR_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification outwards through a disjunction.

Description
When applied to a term of the form `(?p. t) \/ (?p. u)`, the conversion `OR_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) \/ (?p. u) = (?p. t \/ u)
```

Failure
Fails if applied to a term not of the form `(?p. t) \/ (?p. u)`.

See also
`OR_EXISTS_CONV, PEXISTS_OR_CONV, LEFT_OR_PEXISTS_CONV, RIGHT_OR_PEXISTS_CONV.`

OR_PFORALL_CONV

```
OR_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification outwards through a disjunction.

Description
When applied to a term of the form `(!p. t) \/ (!p. u)`, where no variables from `p` are free in either `t` nor `u`, `OR_PFORALL_CONV` returns the theorem:

```
|- (!p. t) \/ (!p. u) = (!p. t \/ u)
```

Failure
`OR_PFORALL_CONV` fails if it is applied to a term not of the form `(!p. t) \/ (!p. u)`, or if it is applied to a term `(!p. t) \/ (!p. u)` in which the variables from `p` are free in either `t` or `u`.

See also
`OR_FORALL_CONV`, `PFORALL_OR_CONV`, `LEFT_OR_PFORALL_CONV`, `RIGHT_OR_PFORALL_CONV`.

PABS

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

Synopsis
Paired abstraction of both sides of an equation.

Description

```
       A |- t1 = t2
------------------------  ABS "p"          [Where p is not free in A]
 A |- (\p.t1) = (\p.t2)
```

Failure
If the theorem is not an equation, or if any variable in the paired structure of variables `p` occurs free in the assumptions `A`.
EXAMPLE

```
#PABS "(x:*,y:**)" (REFL "(x:*,y:**)");;
|- (\(x,y). (x,y)) = (\(x,y). (x,y))
```

See also
`ABS`, `PABS_CONV`, `PETA_CONV`, `PEXT`, `MK_PABS`.

PABS_CONV

```
PABS_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to the body of a paired abstraction.

Description
If `c` is a conversion that maps a term `"t"` to the theorem `|- t = t'`, then the conversion `PABS_CONV c` maps abstractions of the form `"\p.t"` to theorems of the form:

```
|- (\p.t) = (\p.t')
```

That is, `ABS_CONV c "\p.t"` applies `p` to the body of the paired abstraction `"\p.t"`.

Failure
`PABS_CONV c tm` fails if `tm` is not a paired abstraction or if `tm` has the form `"\p.t"` but the conversion `c` fails when applied to the term `t`. The function returned by `ABS_CONV p` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
#PABS_CONV SYM_CONV "\(x,y). (1,2) = (x,y)";;
|- (\(x,y). 1,2 = x,y) = (\(x,y). x,y = 1,2)
```

See also
ABS_CONV, PSUB_CONV.

paconv

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

Synopsis
Tests for alpha-equivalence of terms.

Description
When applied to a pair of terms `t1` and `t2`, `paconv` returns true if the terms are alpha-equivalent.

Failure
Never fails.

Comments
`paconv` is implemented as `curry (can (uncurry PALPHA))`.

See also
`PALPHA, aconv.`

PAIR_CONV

```
PAIR_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to all the components of a pair structure.

Description
For any conversion `c`, the function returned by `PAIR_CONV c` is a conversion that applies `c` to all the components of a pair. If the term `t` is not a pair, them `PAIR_CONV c t` applies `c` to `t`. If the term `t` is the pair `(t1,t2)` then `PAIR c t` recursively applies `PAIR_CONV c` to `t1` and `t2`.

Failure
The conversion returned by `PAIR_CONV c` will fail for the pair structure `t` if the conversion `c` would fail for any of the components of `t`.

See also
`RAND_CONV, RATOR_CONV.`

PALPHA

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

Synopsis

Proves equality of paired alpha-equivalent terms.

Description

When applied to a pair of terms `t1` and `t1'` which are alpha-equivalent, `ALPHA` returns the theorem `|- t1 = t1'`.

```
   ------------  PALPHA "t1" "t1'"
    |- t1 = t1'
```

The difference between `PALPHA` and `ALPHA` is that `PALPHA` is prepared to consider pair structures of different structure to be alpha-equivalent. In its most trivial case this means that `PALPHA` can consider a variable and a pair to alpha-equivalent.

Failure

Fails unless the terms provided are alpha-equivalent.

Example

```
#PALPHA "\(x:*,y:*). (x,y)" "\xy:*#*.xy";;
|- (\(x,y). (x,y)) = (\xy. xy)
```

Comments

The system shows the type of `PALPHA` as `term -> conv`.

Alpha-converting a paired abstraction to a nonpaired abstraction can introduce instances of the terms `"FST"` and `"SND"`. A paired abstraction and a nonpaired abstraction will be considered equivalent by `PALPHA` if the nonpaired abstraction contains all those instances of `"FST"` and `"SND"` present in the paired abstraction, plus the minimum additional instances of `"FST"` and `"SND"`. For example:

```
#PALPHA
   "\(x:*,y:**). (f x y (x,y)):***"
   "\xy:*#**. (f (FST xy) (SND xy) xy):***";;
|- (\(x,y). f x y(x,y)) = (\xy. f(FST xy)(SND xy)xy)

#PALPHA
   "\(x:*,y:**). (f x y (x,y)):***"
   "\xy:*#**. (f (FST xy) (SND xy) (FST xy, SND xy)):***";;
evaluation failed     PALPHA
```

See also

`ALPHA`, `aconv`, `PALPHA_CONV`, `GEN_PALPHA_CONV`.

PALPHA_CONV

`PALPHA_CONV : (term -> conv)`

Synopsis

Renames the bound variables of a paired lambda-abstraction.

Description

If `"q"` is a variable of type `ty` and `"\p.t"` is a paired abstraction in which the bound pair `p` also has type `ty`, then `ALPHA_CONV "q" "\p.t"` returns the theorem:

```
|- (\p.t) = (\q'. t[q'/p])
```

where the pair `q':ty` is a primed variant of `q` chosen so that none of its components are free in `"\p.t"`. The pairs `p` and `q` need not have the same structure, but they must be of the same type.

Example

`PALPHA_CONV` renames the variables in a bound pair:

```
#PALPHA_CONV
  "((w:*,x:*),(y:*,z:*))"
  "\((a:*,b:*),(c:*,d:*)). (f a b c d):*";;
|- (\((a,b),c,d). f a b c d) = (\((w,x),y,z). f w x y z)
```

The new bound pair and the old bound pair need not have the same structure.

```
#PALPHA_CONV
  "((wx:*#*),(y:*,z:*))"
  "\((a:*,b:*),(c:*,d:*)). (f a b c d):*";;
|- (\((a,b),c,d). f a b c d) = (\(wx,y,z). f(FST wx)(SND wx)y z)
```

`PALPHA_CONV` recognises subpairs of a pair as variables and preserves structure accordingly.

```
#PALPHA_CONV
  "((wx:*#*),(y:*,z:*))"
  "\((a:*,b:*),(c:*,d:*)). (f (a,b) c d):*";;
|- (\((a,b),c,d). f(a,b)c d) = (\(wx,y,z). f wx y z)
```

Comments

`PALPHA_CONV` will only ever add the terms `"FST"` and `"SND"`. (i.e. it will never remove them). This means that while `"\(x,y). x + y"` can be converted to `"\xy. (FST xy) + (SND xy)"`, it can not be converted back again.

Failure

`PALPHA_CONV "q" "tm"` fails if `q` is not a variable, if `tm` is not an abstraction, or if `q` is a variable and `tm` is the lambda abstraction `\p.t` but the types of `p` and `q` differ.

See also

`ALPHA_CONV`, `PALPHA`, `GEN_PALPHA_CONV`.

PART_PMATCH

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

Synopsis

Instantiates a theorem by matching part of it to a term.

Description

When applied to a 'selector' function of type `term -> term`, a theorem and a term:

```
PART_MATCH fn (A |- !p1...pn. t) tm
```

the function `PART_PMATCH` applies `fn` to `t'` (the result of specializing universally quantified pairs in the conclusion of the theorem), and attempts to match the resulting term to the argument term `tm`. If it succeeds, the appropriately instantiated version of the theorem is returned.

Failure

Fails if the selector function `fn` fails when applied to the instantiated theorem, or if the match fails with the term it has provided.

See also

`PART_MATCH`.

PBETA_CONV

```
PBETA_CONV : conv
```

Synopsis

Performs a general beta-conversion.

Description

The conversion PBETA_CONV maps a paired beta-redex "(\p.t)q" to the theorem

```
|- (\p.t)q = t[q/p]
```

where u[q/p] denotes the result of substituting q for all free occurrences of p in t, after renaming sufficient bound variables to avoid variable capture. Unlike PAIRED_BETA_CONV, PBETA_CONV does not require that the structure of the argument match the structure of the pair bound by the abstraction. However, if the structure of the argument does match the structure of the pair bound by the abstraction, then PAIRED_BETA_CONV will do the job much faster.

Failure

PBETA_CONV tm fails if tm is not a paired beta-redex.

Example

PBETA_CONV will reduce applications with arbitrary structure.

```
#PBETA_CONV "((\((a:*,b:*),(c:*,d:*)). f a b c d) ((w,x),(y,z))):*";;
|- (\((a,b),c,d). f a b c d)((w,x),y,z) = f w x y z
```

PBETA_CONV does not require the structure of the argument and the bound pair to match.

```
#PBETA_CONV "((\((a:*,b:*),(c:*,d:*)). f a b c d) ((w,x),yz)):*";;
|- (\((a,b),c,d). f a b c d)((w,x),yz) = f w x(FST yz)(SND yz)
```

PBETA_CONV regards component pairs of the bound pair as variables in their own right and preserves structure accordingly:

```
#PBETA_CONV "((\((a:*,b:*),(c:*,d:*)). f (a,b) (c,d)) (wx,(y,z))):*";;
|- (\((a,b),c,d). f(a,b)(c,d))(wx,y,z) = f wx(y,z)
```

See also

BETA_CONV, PAIRED_BETA_CONV, PBETA_RULE, PBETA_TAC, LIST_PBETA_CONV, RIGHT_PBETA, RIGHT_LIST_PBETA, LEFT_PBETA, LEFT_LIST_PBETA.

PBETA_RULE

```
PBETA_RULE : (thm -> thm)
```

Synopsis
Beta-reduces all the paired beta-redexes in the conclusion of a theorem.

Description
When applied to a theorem `A |- t`, the inference rule `PBETA_RULE` beta-reduces all beta-redexes, at any depth, in the conclusion `t`. Variables are renamed where necessary to avoid free variable capture.

```
    A |- ....((\p. s1) s2)....
   ----------------------------  BETA_RULE
      A |- ....(s1[s2/p])....
```

Failure
Never fails, but will have no effect if there are no paired beta-redexes.

See also
`BETA_RULE`, `PBETA_CONV`, `PBETA_TAC`, `RIGHT_PBETA`, `LEFT_PBETA`.

PBETA_TAC

```
PBETA_TAC : tactic
```

Synopsis
Beta-reduces all the paired beta-redexes in the conclusion of a goal.

Description
When applied to a goal `A ?- t`, the tactic `PBETA_TAC` produces a new goal which results from beta-reducing all paired beta-redexes, at any depth, in `t`. Variables are renamed where necessary to avoid free variable capture.

```
    A ?- ...((\p. s1) s2)...
   ==========================  PBETA_TAC
     A ?- ...(s1[s2/p])...
```

Failure
Never fails, but will have no effect if there are no paired beta-redexes.

See also
`BETA_TAC`, `PBETA_CONV`, `PBETA_RULE`.

pbody

```
pbody : (term -> term)
```

Synopsis
Returns the body of a paired abstraction.

Description
`pbody "\pair. t"` returns `"t"`.

Failure
Fails unless the term is a paired abstraction.

See also
`body`, `bndpair`, `dest_pabs`.

PCHOOSE

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

Synopsis
Eliminates paired existential quantification using deduction from a particular witness.

Description
When applied to a term-theorem pair `(q,A1 |- ?p. s)` and a second theorem of the form `A2 u {s[q/p]} |- t`, the inference rule `PCHOOSE` produces the theorem `A1 u A2 |- t`.

```
    A1 |- ?p. s   A2 u {s[q/p]} |- t
   ------------------------------------  PCHOOSE ("q",(A1 |- ?q. s))
               A1 u A2 |- t
```

Where no variable in the paired variable structure `q` is free in `A1`, `A2` or `t`.

Failure
Fails unless the terms and theorems correspond as indicated above; in particular `q` must have the same type as the pair existentially quantified over, and must not contain any variable free in `A1`, `A2` or `t`.

See also

CHOOSE, PCHOOSE_TAC, PEXISTS, PEXISTS_TAC, PSELECT_ELIM.

PCHOOSE_TAC

```
PCHOOSE_TAC : thm_tactic
```

Synopsis

Adds the body of a paired existentially quantified theorem to the assumptions of a goal.

Description

When applied to a theorem `A' |- ?p. t` and a goal, `CHOOSE_TAC` adds `t[p'/p]` to the assumptions of the goal, where `p'` is a variant of the pair `p` which has no components free in the assumption list; normally `p'` is just `p`.

```
        A ?- u
   ====================  CHOOSE_TAC (A' |- ?q. t)
    A u {t[p'/p]} ?- u
```

Unless `A'` is a subset of `A`, this is not a valid tactic.

Failure

Fails unless the given theorem is a paired existential quantification.

See also

CHOOSE_TAC, PCHOOSE_THEN, P_PCHOOSE_TAC.

PCHOOSE_THEN

```
PCHOOSE_THEN : thm_tactical
```

Synopsis

Applies a tactic generated from the body of paired existentially quantified theorem.

Description

When applied to a theorem-tactic `ttac`, a paired existentially quantified theorem:

```
A' |- ?p. t
```

and a goal, `CHOOSE_THEN` applies the tactic `ttac (t[p'/p] |- t[p'/p])` to the goal, where `p'` is a variant of the pair `p` chosen to have no components free in the assumption list of the goal. Thus if:

```
   A ?- s1
  =========  ttac (t[q'/q] |- t[q'/q])
   B ?- s2
```

then

```
   A ?- s1
  ==========  CHOOSE_THEN ttac (A' |- ?q. t)
   B ?- s2
```

This is invalid unless `A'` is a subset of `A`.

Failure

Fails unless the given theorem is a paired existential quantification, or if the resulting tactic fails when applied to the goal.

See also

`CHOOSE_THEN, PCHOOSE_TAC, P_PCHOOSE_THEN.`

PETA_CONV

```
PETA_CONV : conv
```

Synopsis

Performs a top-level paired eta-conversion.

Description

`PETA_CONV` maps an eta-redex `"\p. t p"`, where none of variables in the paired structure of variables `p` occurs free in `t`, to the theorem `|- (\p. t p) = t`.

Failure

Fails if the input term is not a paired eta-redex.

PEXISTENCE

```
PEXISTENCE : (thm -> thm)
```

Synopsis
Deduces paired existence from paired unique existence.

Description
When applied to a theorem with a paired unique-existentially quantified conclusion, `EXISTENCE` returns the same theorem with normal paired existential quantification over the same pair.

```
    A |- ?!p. t
   -------------  PEXISTENCE
    A |- ?p. t
```

Failure
Fails unless the conclusion of the theorem is a paired unique-existential quantification.

See also
`EXISTENCE`, `PEXISTS_UNIQUE_CONV`.

PEXISTS

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

Synopsis
Introduces paired existential quantification given a particular witness.

Description
When applied to a pair of terms and a theorem, where the first term a paired existentially quantified pattern indicating the desired form of the result, and the second a

witness whose substitution for the quantified pair gives a term which is the same as the conclusion of the theorem, PEXISTS gives the desired theorem.

```
 A |- t[q/p]
-------------  EXISTS ("?p. t","q")
 A |- ?p. t
```

Failure

Fails unless the substituted pattern is the same as the conclusion of the theorem.

Example

The following examples illustrate the various uses of PEXISTS:

```
#PEXISTS ("?x. x + 2 = x + 2", "1") (REFL "1 + 2");;
|- ?x. x + 2 = x + 2

#PEXISTS ("?y. 1 + y = 1 + y", "2") (REFL "1 + 2");;
|- ?y. 1 + y = 1 + y

#PEXISTS ("?(x,y). x + y = x + y", "(1,2)") (REFL "1 + 2");;
|- ?(x,y). x + y = x + y

#PEXISTS ("?(a:*,b:*). (a,b) = (a,b)", "ab:*#*") (REFL "ab:*#*");;
|- ?(a,b). a,b = a,b
```

See also

EXISTS, PCHOOSE, PEXISTS_TAC.

PEXISTS_AND_CONV

```
PEXISTS_AND_CONV : conv
```

Synopsis

Moves a paired existential quantification inwards through a conjunction.

Description

When applied to a term of the form `?p. t /\ u`, where variables in `p` are not free in both `t` and `u`, `PEXISTS_AND_CONV` returns a theorem of one of three forms, depending on occurrences of variables from p in t and u. If p contains variables free in t but none in u, then the theorem:

```
|- (?p. t /\ u) = (?p. t) /\ u
```

is returned. If p contains variables free in u but none in t, then the result is:

```
|- (?p. t /\ u) = t /\ (?x. u)
```

And if p does not contain any variable free in either t nor u, then the result is:

```
|- (?p. t /\ u) = (?x. t) /\ (?x. u)
```

Failure

PEXISTS_AND_CONV fails if it is applied to a term not of the form ?p. t /\ u, or if it is applied to a term ?p. t /\ u in which variables in p are free in both t and u.

See also

EXISTS_AND_CONV, AND_PEXISTS_CONV, LEFT_AND_PEXISTS_CONV, RIGHT_AND_PEXISTS_CONV.

PEXISTS_CONV

```
PEXISTS_CONV : conv
```

Synopsis

Eliminates paired existential quantifier by introducing a paired choice-term.

Description

The conversion PEXISTS_CONV expects a boolean term of the form (?p. t[p]), where p may be a paired structure or variables, and converts it to the form (t [@p. t[p]]).

```
---------------------------------  PEXISTS_CONV "(?p. t[p])"
 (|- (?p. t[p]) = (t [@p. t[p]])
```

Failure

Fails if applied to a term that is not a paired existential quantification.

See also

PSELECT_RULE, PSELECT_CONV, PEXISTS_RULE, PSELECT_INTRO, PSELECT_ELIM.

PEXISTS_EQ

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

Synopsis
Existentially quantifies both sides of an equational theorem.

Description
When applied to a paired structure of variables `p` and a theorem whose conclusion is equational:

```
A |- t1 = t2
```

the inference rule `PEXISTS_EQ` returns the theorem:

```
A |- (?p. t1) = (?p. t2)
```

provided the none of the variables in `p` is not free in any of the assumptions.

```
          A |- t1 = t2
   --------------------------  PEXISTS_EQ "p"      [where p is not free in A]
    A |- (?p. t1) = (?p. t2)
```

Failure
Fails unless the theorem is equational with both sides having type `bool`, or if the term is not a paired structure of variables, or if any variable in the pair to be quantified over is free in any of the assumptions.

See also
`EXISTS_EQ`, `PEXISTS_IMP`, `PFORALL_EQ`, `MK_PEXISTS`, `PSELECT_EQ`.

PEXISTS_IMP

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

Synopsis
Existentially quantifies both the antecedent and consequent of an implication.

Description

When applied to a paired structure of variables `p` and a theorem `A |- t1 ==> t2`, the inference rule `PEXISTS_IMP` returns the theorem `A |- (?p. t1) ==> (?p. t2)`, provided no variable in `p` is free in the assumptions.

```
      A |- t1 ==> t2
--------------------------  EXISTS_IMP "x"   [where x is not free in A]
 A |- (?x.t1) ==> (?x.t2)
```

Failure

Fails if the theorem is not implicative, or if the term is not a paired structure of variables, of if any variable in the pair is free in the assumption list.

See also

`EXISTS_IMP`, `PEXISTS_EQ`.

PEXISTS_IMP_CONV

```
PEXISTS_IMP_CONV : conv
```

Synopsis

Moves a paired existential quantification inwards through an implication.

Description

When applied to a term of the form `?p. t ==> u`, where variables from `p` are not free in both `t` and `u`, `PEXISTS_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of variable from `p` in `t` and `u`. If variables from `p` are free in `t` but none are in `u`, then the theorem:

```
|- (?p. t ==> u) = (!p. t) ==> u
```

is returned. If variables from `p` are free in `u` but none are in `t`, then the result is:

```
|- (?p. t ==> u) = t ==> (?p. u)
```

And if no variable from `p` is free in either `t` nor `u`, then the result is:

```
|- (?p. t ==> u) = (!p. t) ==> (?p. u)
```

Failure

`PEXISTS_IMP_CONV` fails if it is applied to a term not of the form `?p. t ==> u`, or if it is applied to a term `?p. t ==> u` in which the variables from `p` are free in both `t` and `u`.

See also
EXISTS_IMP_CONV, LEFT_IMP_PFORALL_CONV, RIGHT_IMP_PEXISTS_CONV.

PEXISTS_NOT_CONV

```
PEXISTS_NOT_CONV : conv
```

Synopsis
Moves a paired existential quantification inwards through a negation.

Description
When applied to a term of the form `?p. ~t`, the conversion `PEXISTS_NOT_CONV` returns the theorem:

```
   |- (?p. ~t) = ~(!p. t)
```

Failure
Fails if applied to a term not of the form `?p. ~t`.

See also
EXISTS_NOT_CONV, PFORALL_NOT_CONV, NOT_PEXISTS_CONV, NOT_PFORALL_CONV.

PEXISTS_OR_CONV

```
PEXISTS_OR_CONV : conv
```

Synopsis
Moves a paired existential quantification inwards through a disjunction.

Description
When applied to a term of the form `?p. t \/ u`, the conversion `PEXISTS_OR_CONV` returns the theorem:

```
   |- (?p. t \/ u) = (?p. t) \/ (?p. u)
```

Failure
Fails if applied to a term not of the form `?p. t \/ u`.

See also
EXISTS_OR_CONV, OR_PEXISTS_CONV, LEFT_OR_PEXISTS_CONV, RIGHT_OR_PEXISTS_CONV.

PEXISTS_RULE

```
PEXISTS_RULE : (thm -> thm)
```

Synopsis
Introduces a paired existential quantification in place of a paired choice.

Description
The inference rule `PEXISTS_RULE` expects a theorem asserting that `(@p. t)` denotes a pair for which `t` holds. The equivalent assertion that there exists a `p` for which `t` holds is returned.

```
    A |- t[(@p. t)/p]
   ------------------  PEXISTS_RULE
       A |- ?p. t
```

Failure
Fails if applied to a theorem the conclusion of which is not of the form `(t[(@p.t)/p])`.

See also
PEXISTS_CONV, PSELECT_RULE, PSELECT_CONV, PSELECT_INTRO, PSELECT_ELIM.

PEXISTS_TAC

```
PEXISTS_TAC : (term -> tactic)
```

Synopsis
Reduces paired existentially quantified goal to one involving a specific witness.

Description

When applied to a term `q` and a goal `?p. t`, the tactic `PEXISTS_TAC` reduces the goal to `t[q/p]`.

```
 A ?- ?p. t
=============  EXISTS_TAC "q"
 A ?- t[q/p]
```

Failure

Fails unless the goal's conclusion is a paired existential quantification and the term supplied has the same type as the quantified pair in the goal.

Example

The goal:

```
?- ?(x,y). (x,y)=(1,2)
```

can be solved by:

```
PEXISTS_TAC "(1,2)" THEN REFL_TAC
```

See also

`EXISTS_TAC`, `PEXISTS`.

PEXISTS_UNIQUE_CONV

```
PEXISTS_UNIQUE_CONV : conv
```

Synopsis

Expands with the definition of paired unique existence.

Description

Given a term of the form `"?!p. t[p]"`, the conversion `PEXISTS_UNIQUE_CONV` proves that this assertion is equivalent to the conjunction of two statements, namely that there exists at least one pair `p` such that `t[p]`, and that there is at most one value `p` for which `t[p]` holds. The theorem returned is:

```
|- (?!p. t[p]) = (?p. t[p]) /\ (!p p'. t[p] /\ t[p'] ==> (p = p'))
```

where `p'` is a primed variant of the pair `p` none of the components of which appear free in the input term. Note that the quantified pair `p` need not in fact appear free in the body of the input term. For example, `PEXISTS_UNIQUE_CONV "?!(x,y). T"` returns the theorem:

```
|- (?!(x,y). T) =
   (?(x,y). T) /\ (!(x,y) (x',y'). T /\ T ==> ((x,y) = (x',y')))
```

Failure

`PEXISTS_UNIQUE_CONV tm` fails if `tm` does not have the form `"?!p.t"`.

See also

`EXISTS_UNIQUE_CONV`, `PEXISTENCE`.

PEXT

```
PEXT : (thm -> thm)
```

Synopsis

Derives equality of functions from extensional equivalence.

Description

When applied to a theorem `A |- !p. t1 p = t2 p`, the inference rule `PEXT` returns the theorem `A |- t1 = t2`.

```
    A |- !p. t1 p = t2 p
   ----------------------  PEXT          [where p is not free in t1 or t2]
        A |- t1 = t2
```

Failure

Fails if the theorem does not have the form indicated above, or if any of the component variables in the paired variable structure `p` is free either of the functions `t1` or `t2`.

Example

```
#PEXT (ASSUME "!(x,y). ((f:(*#*)->*) (x,y)) = (g (x,y))");;
. |- f = g
```

See also

`EXT`, `AP_THM`, `PETA_CONV`, `FUN_EQ_CONV`, `P_FUN_EQ_CONV`.

PFORALL_AND_CONV

```
PFORALL_AND_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through a conjunction.

Description

When applied to a term of the form `!p. t /\ u`, the conversion `PFORALL_AND_CONV` returns the theorem:

```
   |- (!p. t /\ u) = (!p. t) /\ (!p. u)
```

Failure

Fails if applied to a term not of the form `!p. t /\ u`.

See also

`FORALL_AND_CONV`, `AND_PFORALL_CONV`, `LEFT_AND_PFORALL_CONV`, `RIGHT_AND_PFORALL_CONV`.

PFORALL_EQ

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

Synopsis

Universally quantifies both sides of an equational theorem.

Description

When applied to a paired structure of variables `p` and a theorem

```
A |- t1 = t2
```

whose conclusion is an equation between boolean terms:

```
PFORALL_EQ
```

returns the theorem:

```
A |- (!p. t1) = (!p. t2)
```

unless any of the variables in `p` is free in any of the assumptions.

```
       A |- t1 = t2
-------------------------  PFORALL_EQ "p"      [where p is not free in A]
 A |- (!p. t1) = (!p. t2)
```

Failure

Fails if the theorem is not an equation between boolean terms, or if the supplied term is not a paired structure of variables, or if any of the variables in the supplied pair is free in any of the assumptions.

See also

FORALL_EQ, PEXISTS_EQ, PSELECT_EQ.

PFORALL_IMP_CONV

```
PFORALL_IMP_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through an implication.

Description

When applied to a term of the form `!p. t ==> u`, where variables from `p` are not free in both `t` and `u`, `PFORALL_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of the variables from `p` in `t` and `u`. If variables from `p` are free in `t` but none are in `u`, then the theorem:

```
|- (!p. t ==> u) = (?p. t) ==> u
```

is returned. If variables from `p` are free in `u` but none are in `t`, then the result is:

```
|- (!p. t ==> u) = t ==> (!p. u)
```

And if no variable from `p` is free in either `t` nor `u`, then the result is:

```
|- (!p. t ==> u) = (?p. t) ==> (!p. u)
```

Failure

`PFORALL_IMP_CONV` fails if it is applied to a term not of the form `!p. t ==> u`, or if it is applied to a term `!p. t ==> u` in which variables from `p` are free in both `t` and `u`.

See also

`FORALL_IMP_CONV`, `LEFT_IMP_PEXISTS_CONV`, `RIGHT_IMP_PFORALL_CONV`.

PFORALL_NOT_CONV

```
PFORALL_NOT_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through a negation.

Description

When applied to a term of the form `!p. ~t`, the conversion `PFORALL_NOT_CONV` returns the theorem:

```
|- (!p. ~t) = ~(?p. t)
```

Failure

Fails if applied to a term not of the form `!p. ~t`.

See also

`FORALL_NOT_CONV`, `PEXISTS_NOT_CONV`, `NOT_PEXISTS_CONV`, `NOT_PFORALL_CONV`.

PFORALL_OR_CONV

```
PFORALL_OR_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through a disjunction.

Description

When applied to a term of the form `!p. t \/ u`, where no variable in `p` is free in both `t` and `u`, `PFORALL_OR_CONV` returns a theorem of one of three forms, depending on occurrences of the variables from `p` in `t` and `u`. If variables from `p` are free in `t` but not in `u`, then the theorem:

```
|- (!p. t \/ u) = (!p. t) \/ u
```

is returned. If variables from `p` are free in `u` but none are free in `t`, then the result is:

```
|- (!p. t \/ u) = t \/ (!t. u)
```

And if no variable from `p` is free in either `t` nor `u`, then the result is:

```
|- (!p. t \/ u) = (!p. t) \/ (!p. u)
```

Failure

`PFORALL_OR_CONV` fails if it is applied to a term not of the form `!p. t \/ u`, or if it is applied to a term `!p. t \/ u` in which variables from `p` are free in both `t` and `u`.

See also

`FORALL_OR_CONV`, `OR_PFORALL_CONV`, `LEFT_OR_PFORALL_CONV`, `RIGHT_OR_PFORALL_CONV`.

PGEN

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

Synopsis

Generalizes the conclusion of a theorem.

Description

When applied to a paired structure of variables `p` and a theorem `A |- t`, the inference rule `PGEN` returns the theorem `A |- !p. t`, provided that no variable in `p` occurs free in the assumptions `A`. There is no compulsion that the variables of `p` should be free in `t`.

```
      A |- t
   ------------  PGEN "p"              [where p does not occur free in A]
    A |- !p. t
```

Failure

Fails if `p` is not a paired structure of variables, of if any variable in `p` is free in the assumptions.

See also

`GEN`, `PGENL`, `PGEN_ALL`, `PGEN_TAC`, `PSPEC`, `PSPECL`, `PSPEC_ALL`, `PSPEC_TAC`.

PGENL

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

Synopsis

Generalizes zero or more pairs in the conclusion of a theorem.

Description

When applied to a list of paired variable structures `[p1;...;pn]` and a theorem `A |- t`, the inference rule `PGENL` returns the theorem `A |- !p1...pn. t`, provided none of the constituent variables from any of the pairs `pi` occur free in the assumptions.

```
         A |- t
   ------------------  PGENL "[p1;...;pn]"       [where no pi is free in A]
    A |- !p1...pn. t
```

Failure

Fails unless all the terms in the list are paired structures of variables, none of the variables from which are free in the assumption list.

See also

`GENL`, `PGEN`, `PGEN_ALL`, `PGEN_TAC`, `PSPEC`, `PSPECL`, `PSPEC_ALL`, `PSPEC_TAC`.

PGEN_TAC

```
PGEN_TAC : tactic
```

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

Description
When applied to a goal `A ?- !p. t`, the tactic `PGEN_TAC` reduces it to `A ?- t[p'/p]` where `p'` is a variant of the paired variable structure `p` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `p'` is just `p`.

```
     A ?- !p. t
   ==============  PGEN_TAC
    A ?- t[p'/p]
```

Failure
Fails unless the goal's conclusion is a paired universally quantification.

See also
`GEN_TAC`, `FILTER_PGEN_TAC`, `PGEN`, `PGENL`, `PGEN_ALL`, `PSPEC`, `PSPECL`, `PSPEC_ALL`, `PSPEC_TAC`, `PSTRIP_TAC`, `P_PGEN_TAC`.

PMATCH_MP

```
PMATCH_MP : (thm -> thm -> thm)
```

Synopsis
Modus Ponens inference rule with automatic matching.

Description
When applied to theorems `A1 |- !p1...pn. t1 ==> t2` and `A2 |- t1'`, the inference rule `PMATCH_MP` matches `t1` to `t1'` by instantiating free or paired universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- !pa..pk. t2'`, where `t2'` is a correspondingly instantiated version of `t2`. Polymorphic types are also instantiated if necessary.

Variables free in the consequent but not the antecedent of the first argument theorem will be replaced by variants if this is necessary to maintain the full generality of the theorem, and any pairs which were universally quantified over in the first argument theorem will be universally quantified over in the result, and in the same order.

```
    A1 |- !p1..pn. t1 ==> t2   A2 |- t1'
   --------------------------------------  MATCH_MP
          A1 u A2 |- !pa..pk. t2'
```

Failure

Fails unless the first theorem is a (possibly repeatedly paired universally quantified) implication whose antecedent 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`.

PMATCH_MP_TAC

```
PMATCH_MP_TAC : thm_tactic
```

Synopsis

Reduces the goal using a supplied implication, with matching.

Description

When applied to a theorem of the form

```
   A' |- !p1...pn. s ==> !q1...qm. t
```

`PMATCH_MP_TAC` produces a tactic that reduces a goal whose conclusion `t'` is a substitution and/or type instance of `t` to the corresponding instance of `s`. Any variables free in `s` but not in `t` will be existentially quantified in the resulting subgoal:

```
     A ?- !u1...ui. t'
  ======================  PMATCH_MP_TAC (A' |- !p1...pn. s ==> !q1...qm. t)
     A ?- ?w1...wp. s'
```

where `w1`, ..., `wp` are (type instances of) those pairs among `p1`, ..., `pn` having variables that do not occur free in `t`. Note that this is not a valid tactic unless `A'` is a subset of `A`.

Failure

Fails unless the theorem is an (optionally paired universally quantified) implication whose consequent can be instantiated to match the goal. The generalized pairs `u1`, ..., `ui` must occur in `s'` in order for the conclusion `t` of the supplied theorem to match `t'`.

See also

`MATCH_MP_TAC`.

PSELECT_CONV

```
PSELECT_CONV : conv
```

Synopsis

Eliminates a paired epsilon term by introducing a existential quantifier.

Description

The conversion `PSELECT_CONV` expects a boolean term of the form `"t[@p.t[p]/p]"`, which asserts that the epsilon term `@p.t[p]` denotes a pair, `p` say, for which `t[p]` holds. This assertion is equivalent to saying that there exists such a pair, and `PSELECT_CONV` applied to a term of this form returns the theorem `|- t[@p.t[p]/p] = ?p. t[p]`.

Failure

Fails if applied to a term that is not of the form `"p[@p.t[p]/p]"`.

See also

`SELECT_CONV`, `PSELECT_ELIM`, `PSELECT_INTRO`, `PSELECT_RULE`.

PSELECT_ELIM

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

Synopsis

Eliminates a paired epsilon term, using deduction from a particular instance.

Description

`PSELECT_ELIM` expects two arguments, a theorem `th1`, and a pair `(p,th2):(term # thm)`. The conclusion of `th1` must have the form `P($@ P)`, which asserts that the epsilon term

`$@ P` denotes some value at which `P` holds. The paired variable structure `p` appears only in the assumption `P p` of the theorem `th2`. The conclusion of the resulting theorem matches that of `th2`, and the hypotheses include the union of all hypotheses of the premises excepting `P p`.

```
    A1 |- P($@ P)     A2 u {P p} |- t
   -------------------------------------  PSELECT_ELIM th1 (p ,th2)
              A1 u A2 |- t
```

where `p` is not free in `A2`. If `p` appears in the conclusion of `th2`, the epsilon term will NOT be eliminated, and the conclusion will be `t[$@ P/p]`.

Failure

Fails if the first theorem is not of the form `A1 |- P($@ P)`, or if any of the variables from the variable structure `p` occur free in any other assumption of `th2`.

See also

`SELECT_ELIM`, `PCHOOSE`, `SELECT_AX`, `PSELECT_CONV`, `PSELECT_INTRO`, `PSELECT_RULE`.

PSELECT_EQ

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

Synopsis

Applies epsilon abstraction to both terms of an equation.

Description

When applied to a paired structure of variables `p` and a theorem whose conclusion is equational:

```
    A |- t1 = t2
```

the inference rule `PSELECT_EQ` returns the theorem:

```
    A |- (@p. t1) = (@p. t2)
```

provided no variable in `p` is free in the assumptions.

```
         A |- t1 = t2
   --------------------------  SELECT_EQ "p"      [where p is not free in A]
    A |- (@p. t1) = (@p. t2)
```

Failure

Fails if the conclusion of the theorem is not an equation, of if `p` is not a paired structure of variables, or if any variable in p is free in A.

See also
SELECT_EQ, PFORALL_EQ, PEXISTS_EQ.

PSELECT_INTRO

```
PSELECT_INTRO : (thm -> thm)
```

Synopsis
Introduces an epsilon term.

Description
PSELECT_INTRO takes a theorem with an applicative conclusion, say P x, and returns a theorem with the epsilon term $@ P in place of the original operand x.

```
    A |- P x
  --------------  PSELECT_INTRO
   A |- P($@ P)
```

The returned theorem asserts that $@ P denotes some value at which P holds.

Failure
Fails if the conclusion of the theorem is not an application.

Comments
This function is exactly the same as SELECT_INTRO, it is duplicated in the pair library for completeness.

See also
SELECT_INTRO, PEXISTS, SELECT_AX, PSELECT_CONV, PSELECT_ELIM, PSELECT_RULE.

PSELECT_RULE

```
PSELECT_RULE : (thm -> thm)
```

Synopsis
Introduces a paired epsilon term in place of a paired existential quantifier.

Description

The inference rule PSELECT_RULE expects a theorem asserting the existence of a pair p such that t holds. The equivalent assertion that the epsilon term @p.t denotes a pair p for which t holds is returned as a theorem.

```
       A |- ?p. t
  ------------------  PSELECT_RULE
   A |- t[(@p.t)/p]
```

Failure

Fails if applied to a theorem the conclusion of which is not a paired existential quantifier.

See also

SELECT_RULE, PCHOOSE, SELECT_AX, PSELECT_CONV, PEXISTS_CONV, PSELECT_ELIM, PSELECT_INTRO.

PSKOLEM_CONV

```
PSKOLEM_CONV : conv
```

Synopsis

Proves the existence of a pair of Skolem functions.

Description

When applied to an argument of the form !p1...pn. ?q. tm, the conversion PSKOLEM_CONV returns the theorem:

```
|- (!p1...pn. ?q. tm) = (?q'. !p1...pn. tm[q' p1 ... pn/yq)
```

where q' is a primed variant of the pair q not free in the input term.

Failure

PSKOLEM_CONV tm fails if tm is not a term of the form !p1...pn. ?q. tm.

Example

Both `q` and any `pi` may be a paired structure of variables:

```
#PSKOLEM_CONV
  "!(x11:*,x12:*) (x21:*,x22:*). ?(y1:*,y2:*). tm x11 x12 x21 x21 y1 y2";;
|- (!(x11,x12) (x21,x22). ?(y1,y2). tm x11 x12 x21 x21 y1 y2) =
   (?(y1,y2).
     !(x11,x12) (x21,x22).
      tm x11 x12 x21 x21(y1(x11,x12)(x21,x22))(y2(x11,x12)(x21,x22)))
```

See also

SKOLEM_CONV, P_PSKOLEM_CONV.

PSPEC

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

Synopsis

Specializes the conclusion of a theorem.

Description

When applied to a term `q` and a theorem `A |- !p. t`, then `PSPEC` returns the theorem `A |- t[q/p]`. If necessary, variables will be renamed prior to the specialization to ensure that `q` is free for `p` in `t`, that is, no variables free in `q` become bound after substitution.

```
   A |- !p. t
--------------  PSPEC "q"
  A |- t[q/p]
```

Failure

Fails if the theorem's conclusion is not a paired universal quantification, or if `p` and `q` have different types.

Example

`PSPEC` specialised paired quantifications.

```
#PSPEC "(1,2)" (ASSUME "!(x,y). (x + y) = (y + x)");;
. |- 1 + 2 = 2 + 1
```

`PSPEC` treats paired structures of variables as variables and preserves structure accordingly.

```
#PSPEC "x:*#*" (ASSUME "!(x:*,y:*). (x,y) = (x,y)");;
. |- x = x
```

See also

SPEC, IPSPEC, PSPECL, PSPEC_ALL, PSPEC_VAR, PGEN, PGENL, PGEN_ALL.

PSPECL

PSPECL : (term list -> thm -> thm)

Synopsis

Specializes zero or more pairs in the conclusion of a theorem.

Description

When applied to a term list `[q1;...;qn]` and a theorem `A |- !p1...pn. t`, the inference rule `SPECL` returns the theorem `A |- t[q1/p1]...[qn/pn]`, where the substitutions are made sequentially left-to-right in the same way as for `PSPEC`.

```
      A |- !p1...pn. t
--------------------------  SPECL "[q1;...;qn]"
   A |- t[q1/p1]...[qn/pn]
```

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

Failure

Fails unless each of the terms is of the same type as that of the appropriate quantified variable in the original theorem. Fails if the list of terms is longer than the number of quantified pairs in the theorem.

See also

SPECL, PGEN, PGENL, PGEN_ALL, PGEN_TAC, PSPEC, PSPEC_ALL, PSPEC_TAC.

PSPEC_ALL

PSPEC_ALL : (thm -> thm)

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

Description
When applied to a theorem `A |- !p1...pn. t`, the inference rule `PSPEC_ALL` returns the theorem `A |- t[p1'/p1]...[pn'/pn]` where the `pi'` are distinct variants of the corresponding `pi`, chosen to avoid clashes with any variables free in the assumption list and with the names of constants. Normally `pi'` is just `pi`, in which case `PSPEC_ALL` simply removes all universal quantifiers.

```
    A |- !p1...pn. t
--------------------------  PSPEC_ALL
 A |- t[p1'/x1]...[pn'/xn]
```

Failure
Never fails.

See also
`SPEC_ALL, PGEN, PGENL, PGEN_ALL, PGEN_TAC, PSPEC, PSPECL, PSPEC_TAC.`

PSPEC_PAIR

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

Synopsis
Specializes the conclusion of a theorem, returning the chosen variant.

Description
When applied to a theorem `A |- !p. t`, the inference rule `PSPEC_PAIR` returns the term `q'` and the theorem `A |- t[q'/p]`, where `q'` is a variant of `p` chosen to avoid free variable capture.

```
  A |- !p. t
--------------  PSPEC_PAIR
 A |- t[q'/q]
```

Failure
Fails unless the theorem's conclusion is a paired universal quantification.

Comments

This rule is very similar to plain PSPEC, except that it returns the variant chosen, which may be useful information under some circumstances.

See also

SPEC_VAR, PGEN, PGENL, PGEN_ALL, PGEN_TAC, PSPEC, PSPECL, PSPEC_ALL.

PSPEC_TAC

```
PSPEC_TAC : ((term # term) -> tactic)
```

Synopsis

Generalizes a goal.

Description

When applied to a pair of terms (q,p), where p is a paired structure of variables and a goal A ?- t, the tactic PSPEC_TAC generalizes the goal to A ?- !p. t[p/q], that is, all components of q are turned into the corresponding components of p.

```
        A ?- t
   =================  PSPEC_TAC ("q","p")
    A ?- !x. t[p/q]
```

Failure

Fails unless p is a paired structure of variables with the same type as q.

Example

```
g "1 + 2 = 2 + 1";;
"1 + 2 = 2 + 1"

() : void

#e (PSPEC_TAC ("(1,2)","(x:num,y:num)"));;
OK..
"!(x,y). x + y = y + x"

() : void
```

Uses

Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

See also
PGEN, PGENL, PGEN_ALL, PGEN_TAC, PSPEC, PSPECL, PSPEC_ALL, PSTRIP_TAC.

PSTRIP_ASSUME_TAC

```
PSTRIP_ASSUME_TAC : thm_tactic
```

Synopsis
Splits a theorem into a list of theorems and then adds them to the assumptions.

Description
Given a theorem `th` and a goal `(A,t)`, `PSTRIP_ASSUME_TAC th` splits `th` into a list of theorems. This is done by recursively breaking conjunctions into separate conjuncts, cases-splitting disjunctions, and eliminating paired existential quantifiers by choosing arbitrary variables. Schematically, the following rules are applied:

```
          A ?- t
   ======================  PSTRIP_ASSUME_TAC (A' |- v1 /\ ... /\ vn)
    A u {v1,...,vn} ?- t

               A ?- t
   =================================  PSTRIP_ASSUME_TAC (A' |- v1 \/ ... \/ vn)
    A u {v1} ?- t ... A u {vn} ?- t

         A ?- t
   ====================  PSTRIP_ASSUME_TAC (A' |- ?p. v)
    A u {v[p'/p]} ?- t
```

where `p'` is a variant of the pair `p`.

If the conclusion of `th` is not a conjunction, a disjunction or a paired existentially quantified term, the whole theorem `th` is added to the assumptions.

As assumptions are generated, they are examined to see if they solve the goal (either by being alpha-equivalent to the conclusion of the goal or by deriving a contradiction).

The assumptions of the theorem being split are not added to the assumptions of the goal(s), but they are recorded in the proof. This means that if `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `PSTRIP_ASSUME_TAC (A'|-v)` results in an invalid tactic.

Failure
Never fails.

Uses

PSTRIP_ASSUME_TAC is used when applying a previously proved theorem to solve a goal, or when enriching its assumptions so that resolution, rewriting with assumptions and other operations involving assumptions have more to work with.

See also

PSTRIP_THM_THEN, ,PSTRIP_ASSUME_TAC, PSTRIP_GOAL_THEN, PSTRIP_TAC.

PSTRIP_GOAL_THEN

```
PSTRIP_GOAL_THEN : (thm_tactic -> tactic)
```

Synopsis

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

Description

Given a theorem-tactic ttac and a goal (A,t), PSTRIP_GOAL_THEN removes one outermost occurrence of one of the connectives !, ==>, ~ or /\ from the conclusion of the goal t. If t is a universally quantified term, then STRIP_GOAL_THEN strips off the quantifier. Note that PSTRIP_GOAL_THEN will strip off paired universal quantifications.

```
     A ?- !p. u
  ==============  PSTRIP_GOAL_THEN ttac
    A ?- u[p'/p]
```

where p' is a primed variant that contains no variables that appear free in the assumptions A. If t is a conjunction, then PSTRIP_GOAL_THEN simply splits the conjunction into two subgoals:

```
      A ?- v /\ w
   =================  PSTRIP_GOAL_THEN ttac
    A ?- v   A ?- w
```

If t is an implication "u ==> v" and if:

```
      A ?- v
  ===============  ttac (u |- u)
     A' ?- v'
```

then:

```
      A ?- u ==> v
  ====================  PSTRIP_GOAL_THEN ttac
        A' ?- v'
```

Finally, a negation ~t is treated as the implication t ==> F.

Failure

PSTRIP_GOAL_THEN ttac (A,t) fails if t is not a paired universally quantified term, an implication, a negation or a conjunction. Failure also occurs if the application of ttac fails, after stripping the goal.

Uses

PSTRIP_GOAL_THEN is used when manipulating intermediate results (obtained by stripping outer connectives from a goal) directly, rather than as assumptions.

See also

PGEN_TAC, STRIP_GOAL_THEN, FILTER_PSTRIP_THEN, PSTRIP_TAC, FILTER_PSTRIP_TAC.

PSTRIP_TAC

```
PSTRIP_TAC : tactic
```

Synopsis

Splits a goal by eliminating one outermost connective.

Description

Given a goal (A,t), PSTRIP_TAC removes one outermost occurrence of one of the connectives !, ==>, ~ or /\ from the conclusion of the goal t. If t is a universally quantified term, then STRIP_TAC strips off the quantifier. Note that PSTRIP_TAC will strip off paired quantifications.

```
    A ?- !p. u
  ==============  PSTRIP_TAC
   A ?- u[p'/p]
```

where p' is a primed variant of the pair p that does not contain any variables that appear free in the assumptions A. If t is a conjunction, then PSTRIP_TAC simply splits the conjunction into two subgoals:

```
     A ?- v /\ w
  =================  PSTRIP_TAC
   A ?- v   A ?- w
```

If t is an implication, PSTRIP_TAC moves the antecedent into the assumptions, stripping conjunctions, disjunctions and existential quantifiers according to the following rules:

```
   A ?- v1 /\ ... /\ vn ==> v             A ?- v1 \/ ... \/ vn ==> v
  ============================        =================================
      A u {v1,...,vn} ?- v              A u {v1} ?- v ... A u {vn} ?- v

    A ?- (?p. w) ==> v
  =====================
   A u {w[p'/p]} ?- v
```

where p' is a primed variant of the pair p that does not appear free in A. Finally, a negation ~t is treated as the implication t ==> F.

Failure

PSTRIP_TAC (A,t) fails if t is not a paired universally quantified term, an implication, a negation or a conjunction.

Uses

When trying to solve a goal, often the best thing to do first is REPEAT PSTRIP_TAC to split the goal up into manageable pieces.

See also

PGEN_TAC, PSTRIP_GOAL_THEN, FILTER_PSTRIP_THEN, STRIP_TAC, FILTER_PSTRIP_TAC.

PSTRIP_THM_THEN

PSTRIP_THM_THEN : thm_tactical

Synopsis

`PSTRIP_THM_THEN` applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

Description

Given a theorem-tactic `ttac`, a theorem `th` whose conclusion is a conjunction, a disjunction or a paired existentially quantified term, and a goal `(A,t)`, `STRIP_THM_THEN ttac th` first strips apart the conclusion of `th`, next applies `ttac` to the theorem(s) resulting from the stripping and then applies the resulting tactic to the goal.

In particular, when stripping a conjunctive theorem `A'|- u /\ v`, the tactic

```
ttac(u|-u) THEN ttac(v|-v)
```

resulting from applying `ttac` to the conjuncts, is applied to the goal. When stripping a disjunctive theorem `A'|- u \/ v`, the tactics resulting from applying `ttac` to the disjuncts, are applied to split the goal into two cases. That is, if

```
 A ?- t                           A ?- t
=========  ttac (u|-u)    and    =========  ttac (v|-v)
 A ?- t1                          A ?- t2
```

then:

```
      A ?- t
================== PSTRIP_THM_THEN ttac (A'|- u \/ v)
 A ?- t1  A ?- t2
```

When stripping a paired existentially quantified theorem `A'|- ?p. u`, the tactic resulting from applying `ttac` to the body of the paired existential quantification, `ttac(u|-u)`, is applied to the goal. That is, if:

```
 A ?- t
=========  ttac (u|-u)
 A ?- t1
```

then:

```
   A ?- t
=============  PSTRIP_THM_THEN ttac (A'|- ?p. u)
   A ?- t1
```

The assumptions of the theorem being split are not added to the assumptions of the goal(s) but are recorded in the proof. If `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `PSTRIP_THM_THEN ttac th` results in an invalid tactic.

Failure

PSTRIP_THM_THEN ttac th fails if the conclusion of th is not a conjunction, a disjunction or a paired existentially quantification. Failure also occurs if the application of ttac fails, after stripping the outer connective from the conclusion of th.

Uses

PSTRIP_THM_THEN is used enrich the assumptions of a goal with a stripped version of a previously-proved theorem.

See also

STRIP_THM_THEN, , PSTRIP_ASSUME_TAC, PSTRIP_GOAL_THEN, PSTRIP_TAC.

PSTRUCT_CASES_TAC

```
PSTRUCT_CASES_TAC : thm_tactic
```

Synopsis

Performs very general structural case analysis.

Description

When it is applied to a theorem of the form:

```
th = A' |- ?p11...p1m. (x=t1) /\ (B11 /\ ... /\ B1k) \/ ... \/
             ?pn1...pnp. (x=tn) /\ (Bn1 /\ ... /\ Bnp)
```

in which there may be no paired existential quantifiers where a 'vector' of them is shown above, PSTRUCT_CASES_TAC th splits a goal A ?- s into n subgoals as follows:

```
                          A ?- s
 ===============================================================
  A u {B11,...,B1k} ?- s[t1/x] ... A u {Bn1,...,Bnp} ?- s[tn/x]
```

that is, performs a case split over the possible constructions (the ti) of a term, providing as assumptions the given constraints, having split conjoined constraints into separate assumptions. Note that unless A' is a subset of A, this is an invalid tactic.

Failure

Fails unless the theorem has the above form, namely a conjunction of (possibly multiply paired existentially quantified) terms which assert the equality of the same variable x and the given terms.

Uses

Generating a case split from the axioms specifying a structure.

See also

STRUCT_CASES_TAC.

PSUB_CONV

```
PSUB_CONV : (conv -> conv)
```

Synopsis

Applies a conversion to the top-level subterms of a term.

Description

For any conversion `c`, the function returned by `PSUB_CONV c` is a conversion that applies `c` to all the top-level subterms of a term. If the conversion `c` maps `t` to `|- t = t'`, then `SUB_CONV c` maps a paired abstraction `"\p.t"` to the theorem:

```
|- (\p.t) = (\p.t')
```

That is, `PSUB_CONV c "\p.t"` applies `c` to the body of the paired abstraction `"\p.t"`. If `c` is a conversion that maps `"t1"` to the theorem `|- t1 = t1'` and `"t2"` to the theorem `|- t2 = t2'`, then the conversion `PSUB_CONV c` maps an application `"t1 t2"` to the theorem:

```
|- (t1 t2) = (t1' t2')
```

That is, `PSUB_CONV c "t1 t2"` applies `c` to the both the operator `t1` and the operand `t2` of the application `"t1 t2"`. Finally, for any conversion `c`, the function returned by `PSUB_CONV c` acts as the identity conversion on variables and constants. That is, if `"t"` is a variable or constant, then `PSUB_CONV c "t"` returns `|- t = t`.

Failure

`PSUB_CONV c tm` fails if `tm` is a paired abstraction `"\p.t"` and the conversion `c` fails when applied to `t`, or if `tm` is an application `"t1 t2"` and the conversion `c` fails when applied to either `t1` or `t2`. The function returned by `PSUB_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

See also

SUB_CONV, PABS_CONV, RAND_CONV, RATOR_CONV.

pvariant

```
pvariant : (term list -> term -> term)
```

Synopsis
Modifies variable and constant names in a paired structure to avoid clashes.

Description
When applied to a list of (possibly paired structures of) variables to avoid clashing with, and a pair to modify, `pvariant` returns a variant of the pair. That is, it changes the names of variables and constants in the pair as intuitively as possible to make them distinct from any variables in the list, or any (non-hidden) constants. This is normally done by adding primes to the names.

The exact form of the altered names should not be relied on, except that the original variables will be unmodified unless they are in the list to avoid clashing with. Also note that if the same variable occurs more that one in the pair, then each instance of the variable will be modified in the same way.

Failure
`pvariant l p` fails if any term in the list `l` is not a paired structure of variables, or if `p` is not a paired structure of variables and constants.

Example
The following shows a case that exhibits most possible behaviours:

```
#pvariant ["b:*"; "(c:*,c':*)"] "((a:*,b:*),(c:*,b':*,T,b:*))";;
"(a,b''),c'',b',T',b''" : term
```

Uses
The function `pvariant` is extremely useful for complicated derived rules which need to rename pairs variable to avoid free variable capture while still making the role of the pair obvious to the user.

See also
`variant`, `genvar`, `hide_constant`, `genlike`.

P_FUN_EQ_CONV

```
P_FUN_EQ_CONV : (term -> conv)
```

Synopsis
Performs extensionality conversion for functions (function equality).

Description
The conversion P_FUN_EQ_CONV embodies the fact that two functions are equal precisely when they give the same results for all values to which they can be applied. For any paired variable structure "p" and equation "f = g", where p is of type ty1 and f and g are functions of type ty1->ty2, a call to P_FUN_EQ_CONV "p" "f = g" returns the theorem:

```
|- (f = g) = (!p. f p = g p)
```

Failure
P_FUN_EQ_CONV p tm fails if p is not a paired structure of variables or if tm is not an equation f = g where f and g are functions. Furthermore, if f and g are functions of type ty1->ty2, then the pair x must have type ty1; otherwise the conversion fails. Finally, failure also occurs if any of the variables in p is free in either f or g.

See also
FUN_EQ_CONV, PEXT.

P_PCHOOSE_TAC

```
P_PCHOOSE_TAC : (term -> thm_tactic)
```

Synopsis
Assumes a theorem, with existentially quantified pair replaced by a given witness.

Description
P_PCHOOSE_TAC expects a pair q and theorem with a paired existentially quantified conclusion. When applied to a goal, it adds a new assumption obtained by introducing the pair q as a witness for the pair p whose existence is asserted in the theorem.

```
          A ?- t
===================  P_CHOOSE_TAC "q" (A1 |- ?p. u)
 A u {u[q/p]} ?- t         ("y" not free anywhere)
```

Failure
Fails if the theorem's conclusion is not a paired existential quantification, or if the first argument is not a paired structure of variables. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in u

or `t`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

See also

`X_CHOOSE_TAC`, `PCHOOSE`, `PCHOOSE_THEN`, `P_PCHOOSE_THEN`.

P_PCHOOSE_THEN

```
P_PCHOOSE_THEN : (term -> thm_tactical)
```

Synopsis

Replaces existentially quantified pair with given witness, and passes it to a theorem-tactic.

Description

`P_PCHOOSE_THEN` expects a pair `q`, a tactic-generating function `f:thm->tactic`, and a theorem of the form `(A1 |- ?p. u)` as arguments. A new theorem is created by introducing the given pair `q` as a witness for the pair `p` whose existence is asserted in the original theorem, `(u[q/p] |- u[q/p])`. If the tactic-generating function `f` applied to this theorem produces results as follows when applied to a goal `(A ?- u)`:

```
    A ?- t
   =========  f ({u[q/p]} |- u[q/p])
    A ?- t1
```

then applying `(P_PCHOOSE_THEN "q" f (A1 |- ?p. u))` to the goal `(A ?- t)` produces the subgoal:

```
    A ?- t
   =========  P_PCHOOSE_THEN "q" f (A1 |- ?p. u)
    A ?- t1         ("q" not free anywhere)
```

Failure

Fails if the theorem's conclusion is not existentially quantified, or if the first argument is not a paired structure of variables. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in `u` or `t`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

See also

`X_CHOOSE_THEN`, `PCHOOSE`, `PCHOOSE_THEN`, `P_PCHOOSE_TAC`.

P_PGEN_TAC

```
P_PGEN_TAC : (term -> tactic)
```

Synopsis
Specializes a goal with the given paired structure of variables.

Description
When applied to a paired structure of variables `p'`, and a goal `A ?- !p. t`, the tactic `P_PGEN_TAC` returns the goal `A ?- t[p'/p]`.

```
     A ?- !p. t
   ==============  P_PGEN_TAC "p'"
    A ?- t[p'/x]
```

Failure
Fails unless the goal's conclusion is a paired universal quantification and the term a paired structure of variables of the appropriate type. It also fails if any of the variables of the supplied structure occurs free in either the assumptions or (initial) conclusion of the goal.

See also
`X_GEN_TAC`, `FILTER_PGEN_TAC`, `PGEN`, `PGENL`, `PGEN_ALL`, `PSPEC`, `PSPECL`, `PSPEC_ALL`, `PSPEC_TAC`.

P_PSKOLEM_CONV

```
P_PSKOLEM_CONV : (term -> conv)
```

Synopsis
Introduces a user-supplied Skolem function.

Description
`P_PSKOLEM_CONV` takes two arguments. The first is a variable `f`, which must range over functions of the appropriate type, and the second is a term of the form `!p1...pn. ?q. t`

(where `pi` and `q` may be pairs). Given these arguments, `P_PSKOLEM_CONV` returns the theorem:

```
|- (!p1...pn. ?q. t) = (?f. !p1...pn. tm[f p1 ... pn/q])
```

which expresses the fact that a skolem function `f` of the universally quantified variables `p1...pn` may be introduced in place of the the existentially quantified pair `p`.

Failure

`P_PSKOLEM_CONV f tm` fails if `f` is not a variable, or if the input term `tm` is not a term of the form `!p1...pn. ?q. t`, or if the variable `f` is free in `tm`, or if the type of `f` does not match its intended use as an `n`-place curried function from the pairs `p1...pn` to a value having the same type as `p`.

See also

`X_SKOLEM_CONV`, `PSKOLEM_CONV`.

RIGHT_AND_PEXISTS_CONV

```
RIGHT_AND_PEXISTS_CONV : conv
```

Synopsis

Moves a paired existential quantification of the right conjunct outwards through a conjunction.

Description

When applied to a term of the form `t /\ (?p. t)`, the conversion `RIGHT_AND_PEXISTS_CONV` returns the theorem:

```
|- t /\ (?p. u) = (?p'. t /\ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `t /\ (?p. u)`.

See also

`RIGHT_AND_EXISTS_CONV`, `AND_PEXISTS_CONV`, `PEXISTS_AND_CONV`,
`LEFT_AND_PEXISTS_CONV`.

RIGHT_AND_PFORALL_CONV

```
RIGHT_AND_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the right conjunct outwards through a conjunction.

Description
When applied to a term of the form `t /\ (!p. u)`, the conversion `RIGHT_AND_PFORALL_CONV` returns the theorem:

```
|- t /\ (!p. u) = (!p'. t /\ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure
Fails if applied to a term not of the form `t /\ (!p. u)`.

See also
`RIGHT_AND_FORALL_CONV`, `AND_PFORALL_CONV`, `PFORALL_AND_CONV`, `LEFT_AND_PFORALL_CONV`.

RIGHT_IMP_PEXISTS_CONV

```
RIGHT_IMP_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the consequent outwards through an implication.

Description
When applied to a term of the form `t ==> (?p. u)`, `RIGHT_IMP_PEXISTS_CONV` returns the theorem:

```
|- t ==> (?p. u) = (?p'. t ==> (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure
Fails if applied to a term not of the form `t ==> (?p. u)`.

See also
`RIGHT_IMP_EXISTS_CONV`, `PEXISTS_IMP_CONV`, `LEFT_IMP_PFORALL_CONV`.

RIGHT_IMP_PFORALL_CONV

```
RIGHT_IMP_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the consequent outwards through an implication.

Description
When applied to a term of the form `t ==> (!p. u)`, the conversion `RIGHT_IMP_FORALL_CONV` returns the theorem:

```
|- t ==> (!p. u) = (!p'. t ==> (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure
Fails if applied to a term not of the form `t ==> (!p. u)`.

See also
`RIGHT_IMP_FORALL_CONV`, `PFORALL_IMP_CONV`, `LEFT_IMP_PEXISTS_CONV`.

RIGHT_LIST_PBETA

```
RIGHT_LIST_PBETA : (thm -> thm)
```

Synopsis
Iteratively beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

Description

When applied to an equational theorem, RIGHT_LIST_PBETA applies paired beta-reduction over a top-level chain of beta-redexes to the right-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
    A |- s = (\p1...pn. t) q1 ... qn
   ----------------------------------  RIGHT_LIST_BETA
       A |- s = t[q1/p1]...[qn/pn]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level paired beta-redex.

See also

RIGHT_LIST_BETA, PBETA_CONV, PBETA_RULE, PBETA_TAC, LIST_PBETA_CONV, RIGHT_PBETA, LEFT_PBETA, LEFT_LIST_PBETA.

RIGHT_OR_PEXISTS_CONV

```
RIGHT_OR_PEXISTS_CONV : conv
```

Synopsis

Moves a paired existential quantification of the right disjunct outwards through a disjunction.

Description

When applied to a term of the form `t \/ (?p. u)`, the conversion `RIGHT_OR_PEXISTS_CONV` returns the theorem:

```
   |- t \/ (?p. u) = (?p'. t \/ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `t \/ (?p. u)`.

See also

RIGHT_OR_EXISTS_CONV, OR_PEXISTS_CONV, PEXISTS_OR_CONV, LEFT_OR_PEXISTS_CONV.

RIGHT_OR_PFORALL_CONV

`RIGHT_OR_PFORALL_CONV : conv`

Synopsis
Moves a paired universal quantification of the right disjunct outwards through a disjunction.

Description
When applied to a term of the form `t \/ (!p. u)`, the conversion `RIGHT_OR_FORALL_CONV` returns the theorem:

```
|- t \/ (!p. u) = (!p'. t \/ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure
Fails if applied to a term not of the form `t \/ (!p. u)`.

See also
`RIGHT_OR_FORALL_CONV`, `OR_PFORALL_CONV`, `PFORALL_OR_CONV`, `LEFT_OR_PFORALL_CONV`.

RIGHT_PBETA

`RIGHT_PBETA : (thm -> thm)`

Synopsis
Beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

Description
When applied to an equational theorem, `RIGHT_PBETA` applies paired beta-reduction at top level to the right-hand side (only). Variables are renamed if necessary to avoid free

variable capture.

```
    A |- s = (\p. t1) t2
   ----------------------  RIGHT_PBETA
     A |- s = t1[t2/p]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level paired beta-redex.

See also

RIGHT_BETA, PBETA_CONV, PBETA_RULE, PBETA_TAC, RIGHT_LIST_PBETA, LEFT_PBETA, LEFT_LIST_PBETA.

rip_pair

```
rip_pair : (term -> term list)
```

Synopsis

Recursively breaks a paired structure into its constituent pieces.

Example

```
#rip_pair "((1,2),(3,4))";;
["1"; "2"; "3"; "4"] : term list
```

Comments

Note that `rip_pair` is similar, but not identical, to `strip_pair` which iteratively breaks apart tuples (flat paired structures).

Failure

Never fails.

See also

strip_pair.

strip_pabs

```
strip_pabs : (term -> goal)
```

Synopsis
Iteratively breaks apart paired abstractions.

Description
`strip_pabs "\p1 ... pn. t"` returns `(["p1";...;"pn"],"t")`. Note that

```
strip_pabs(list_mk_abs(["p1";...;"pn"],"t"))
```

will not return `(["p1";...;"pn"],"t")` if `t` is a paired abstraction.

Failure
Never fails.

See also
`strip_abs`, `list_mk_pabs`, `dest_pabs`.

strip_pexists

```
strip_pexists : (term -> goal)
```

Synopsis
Iteratively breaks apart paired existential quantifications.

Description
`strip_pexists "?p1 ... pn. t"` returns `(["p1";...;"pn"],"t")`. Note that

```
strip_pexists(list_mk_pexists(["[p1";...;"pn"],"t"))
```

will not return `(["p1";...;"pn"],"t")` if `t` is a paired existential quantification.

Failure
Never fails.

See also
`strip_exists`, `list_mk_pexists`, `dest_pexists`.

strip_pforall

```
strip_pforall : (term -> goal)
```

Synopsis
Iteratively breaks apart paired universal quantifications.

Description
`strip_pforall "!p1 ... pn. t"` returns `(["p1";...;"pn"],"t")`. Note that

```
strip_pforall(list_mk_pforall(["p1";...;"pn"],"t"))
```

will not return `(["p1";...;"pn"],"t")` if `t` is a paired universal quantification.

Failure
Never fails.

See also
`strip_forall`, `list_mk_pforall`, `dest_pforall`.

SWAP_PEXISTS_CONV

```
SWAP_PEXISTS_CONV : conv
```

Synopsis
Interchanges the order of two existentially quantified pairs.

Description
When applied to a term argument of the form `?p q. t`, the conversion `SWAP_PEXISTS_CONV` returns the theorem:

```
|- (?p q. t) = (?q t. t)
```

Failure
`SWAP_PEXISTS_CONV` fails if applied to a term that is not of the form `?p q. t`.

See also
`SWAP_EXISTS_CONV`, `SWAP_PFORALL_CONV`.

SWAP_PFORALL_CONV

```
SWAP_PFORALL_CONV : conv
```

Synopsis
Interchanges the order of two universally quantified pairs.

Description
When applied to a term argument of the form `!p q. t`, the conversion `SWAP_PFORALL_CONV` returns the theorem:

```
   |- (!p q. t) = (!q t. t)
```

Failure
`SWAP_PFORALL_CONV` fails if applied to a term that is not of the form `!p q. t`.

See also
`SWAP_PEXISTS_CONV`.

UNCURRY_CONV

```
UNCURRY_CONV : conv
```

Synopsis
Uncurrys an application of an abstraction.

Example

```
#UNCURRY_CONV "(\x y. x + y) 1 2";;
|- (\x y. x + y)1 2 = (\(x,y). x + y)(1,2)
```

Failure
`UNCURRY_CONV tm` fails if `tm` is not double abstraction applied to two arguments

See also
`CURRY_CONV`.

UNCURRY_EXISTS_CONV

```
UNCURRY_EXISTS_CONV : conv
```

Synopsis
Uncurrys consecutive existential quantifications into a paired existential quantification.

Example

```
#UNCURRY_EXISTS_CONV "?x y. x + y = y + x";;
|- (?x y. x + y = y + x) = (?(x,y). x + y = y + x)

#UNCURRY_EXISTS_CONV "?(w,x) (y,z). w+x+y+z = z+y+x+w";;
|- (?(w,x) (y,z). w + (x + (y + z)) = z + (y + (x + w))) =
   (?((w,x),y,z). w + (x + (y + z)) = z + (y + (x + w)))
```

Failure
`UNCURRY_EXISTS_CONV tm` fails if `tm` is not a consecutive existential quantification.

See also
CURRY_CONV, UNCURRY_CONV, CURRY_EXISTS_CONV, CURRY_FORALL_CONV, UNCURRY_FORALL_CONV.

UNCURRY_FORALL_CONV

```
UNCURRY_FORALL_CONV : conv
```

Synopsis
Uncurrys consecutive universal quantifications into a paired universal quantification.

Example

```
#UNCURRY_FORALL_CONV "!x y. x + y = y + x";;
|- (!x y. x + y = y + x) = (!(x,y). x + y = y + x)

#UNCURRY_FORALL_CONV "!(w,x) (y,z). w+x+y+z = z+y+x+w";;
|- (!(w,x) (y,z). w + (x + (y + z)) = z + (y + (x + w))) =
   (!((w,x),y,z). w + (x + (y + z)) = z + (y + (x + w)))
```

Failure
`UNCURRY_FORALL_CONV tm` fails if `tm` is not a consecutive universal quantification.

See also
CURRY_CONV, UNCURRY_CONV, CURRY_FORALL_CONV, CURRY_EXISTS_CONV, UNCURRY_EXISTS_CONV.

UNPBETA_CONV

```
UNPBETA_CONV : (term -> conv)
```

Synopsis
Creates an application of a paired abstraction from a term.

Description
The user nominates some pair structure of variables `p` and a term `t`, and `UNPBETA_CONV` turns `t` into an abstraction on `p` applied to `p`.

```
   ------------------  UNPBETA_CONV "p" "t"
    |- t = (\p. t) p
```

Failure
Fails if `p` is not a paired structure of variables.

See also
`PBETA_CONV`, `PAIRED_BETA_CONV`.

Chapter 4

Pre-proved Theorems

The section that follows lists the theorems in the `pair` library.

4.1 Theorems

```
ABS_PAIR_THM  (pair)
  |- !x. ?q r. x = (q,r)

ABS_REP_prod  (pair)
  |- (!a. ABS_prod (REP_prod a) = a) /\
     !r. IS_PAIR r = (REP_prod (ABS_prod r) = r)

CLOSED_PAIR_EQ  (pair)
  |- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b)

COMMA_DEF  (pair)
  |- !x y. (x,y) = ABS_prod (MK_PAIR x y)

CURRY_DEF  (pair)
  |- !f x y. CURRY f x y = f (x,y)

CURRY_ONE_ONE_THM  (pair)
  |- (CURRY f = CURRY g) = (f = g)

CURRY_UNCURRY_THM  (pair)
  |- !f. CURRY (UNCURRY f) = f

EXISTS_PROD  (pair)
  |- (?p. P p) = ?p_1 p_2. P (p_1,p_2)

FORALL_PROD  (pair)
  |- (!p. P p) = !p_1 p_2. P (p_1,p_2)

FST  (pair)
  |- !x y. FST (x,y) = x
```

```
FST_DEF  (pair)
  |- !p. FST p = @x. ?y. p = (x,y)

IS_PAIR_DEF  (pair)
  |- !P. IS_PAIR P = ?x y. P = MK_PAIR x y

LET2_RAND  (pair)
  |- !P M N. P (let (x,y) = M in N x y) = (let (x,y) = M in P (N x y))

LET2_RATOR  (pair)
  |- !M N b. (let (x,y) = M in N x y) b = (let (x,y) = M in N x y b)

LEX_DEF  (pair)
  |- !R1 R2. R1 LEX R2 = (\(s,t) (u,v). R1 s u \/ (s = u) /\ R2 t v)

MK_PAIR_DEF  (pair)
  |- !x y. MK_PAIR x y = (\a b. (a = x) /\ (b = y))

PAIR  (pair)
  |- !x. (FST x,SND x) = x

pair_Axiom  (pair)
  |- !f. ?fn. !x y. fn (x,y) = f x y

pair_case_cong  (pair)
  |- !f' f M' M.
       (M = M') /\ (!x y. (M' = (x,y)) ==> (f x y = f' x y)) ==>
       (pair_case f M = pair_case f' M')

pair_case_def  (pair)
  |- pair_case = UNCURRY

pair_case_thm  (pair)
  |- pair_case f (x,y) = f x y

PAIR_EQ  (pair)
  |- ((x,y) = (a,b)) = (x = a) /\ (y = b)

pair_induction  (pair)
  |- (!p_1 p_2. P (p_1,p_2)) ==> !p. P p

PEXISTS_THM  (pair)
  |- !P. (?x y. P x y) = ?(x,y). P x y

PFORALL_THM  (pair)
  |- !P. (!x y. P x y) = !(x,y). P x y
```

```
prod_TY_DEF  (pair)
   |- ?rep. TYPE_DEFINITION IS_PAIR rep

RPROD_DEF  (pair)
   |- !R1 R2. RPROD R1 R2 = (\(s,t) (u,v). R1 s u /\ R2 t v)

SND  (pair)
   |- !x y. SND (x,y) = y

SND_DEF  (pair)
   |- !p. SND p = @y. ?x. p = (x,y)

UNCURRY_CONG  (pair)
   |- !f' f M' M.
        (M = M') /\ (!x y. (M' = (x,y)) ==> (f x y = f' x y)) ==>
        (UNCURRY f M = UNCURRY f' M')

UNCURRY_CURRY_THM  (pair)
   |- !f. UNCURRY (CURRY f) = f

UNCURRY_DEF  (pair)
   |- !f x y. UNCURRY f (x,y) = f x y

UNCURRY_ONE_ONE_THM  (pair)
   |- (UNCURRY f = UNCURRY g) = (f = g)

UNCURRY_VAR  (pair)
   |- !f v. UNCURRY f v = f (FST v) (SND v)

WF_LEX  (pair)
   |- !R Q. WF R /\ WF Q ==> WF (R LEX Q)

WF_RPROD  (pair)
   |- !R Q. WF R /\ WF Q ==> WF (RPROD R Q)
```

Index