Miscellaneous Tactics

Roger Bishop Jones

Date: 2011-05-16 21:40:17

Abstract

Several structures providing tactics, tacticals, etc. for theories, forward chaining, backward chaining, theory trawling et.al.


Copyright © : Roger Bishop Jones
References


1 Introduction

For context and motivation see [2].

Several structures are provided, each section below provides a signature and a structure matching the signature (though the code is not listed).

```sml
infix 4 AND OR T;
infix 4 AND OR;
open theory "basic_hol";
set pc "basic_hol";
```

To enable the use of square subset and three-bar equivalence (without the compliance tool) the following script is included:

```sml
val _ = let open ReaderWriterSupport.PrettyNames;
    in add_new_symbols [ ("sqsubseteq2", Value "\subseteq", Simple) ]
    end
handle _ => ();
val _ = let open ReaderWriterSupport.PrettyNames;
    in add_new_symbols [ ("identical2", Value "\equiv", Simple) ]
    end
handle _ => ();
```

2 Sundry Tacticals etc.

```sml
signature RbjTactics1 = sig

Description A canon is provided for use with backchaining, and an elaboration of the backchaining facilities which is intended to solve certain kinds of goal by repeated backchaining.

val pc_canon: string -> CANON -> CANON;
Description Creates a CANON which executes in a specific proof context.

val rule_canon: (THM -> THM) -> CANON;
Description Converts a rule into a CANON which yeilds a singleton list containing the result of applying the rule to the argument of the CANON.
```
### SML

| ```val ⇒ T_canon: CANON;``` |
| Description | If in `asms ⊢ conc, conc` is a universally quantified implication, then ⇒ `T_canon(asms ⊢ conc)` is `[asms ⊢ conc]`, otherwise it is `[asms ⊢ conc ⇒ T]`. |

| ```val ⇔ FC.T: (THM list −> TACTIC) −> THM −> TACTIC;``` |
| Description | For doing forward chaining using `fc ⇔ _canon`. |

| ```val all ⇒ intro_canon: CANON;``` |
| Description | This is `rule_canon all ⇒ _intro`. |
| See Also | `rule_canon, all ⇒ _intro` |

| ```val abc_canon: CANON;``` |
| Description | A CANON for stripping theorems for backward chaining (used by `abc_tac` q.v.). It removes universal quantifiers, splits conjunctions into two, undisharges implications repeatedly until these can no longer be done, then it discarges all the assumptions and closes the result. |

| ```val abc_tac: THM list −> TACTIC;``` |
| ```val asm.abc_tac: THM list −> TACTIC;``` |
| Description | A backchaining tactic which preprocesses theorems using `abc_canon` and then repeatedly backchains, terminating only if the conclusion can be reduced to `T` and discharged. The `asm_` version uses the assumptions as rules or for reducing the conclusion to `T`. |

| ```val map_eq_sym_rule : THM −> THM;``` |
| ```val list_map_eq_sym_rule : THM list −> THM list;``` |
| ```val SYM_ASMSTM: (THM list −> TACTIC) −> TACTIC;``` |
| Description | These are for turning round equations in order to use them for rewriting, when the equation is not at the top level. `map_eq_sym_rule` turns round the equations in the conclusion of the theorem, wherever they occur. `list_map_eq_sym_rule` does the same thing to every one of a list of theorems. `SYM_ASMSTM thmltac` applies `list_map_eq_sym_rule` to the list of assumptions and then passes the result to `thmltac`. |
| See Also | `eq_sym_conv, eq_sym_rule` |
These facilities are to permit rewriting with the definition of or theorems about functions which take pairs as arguments, and are defined using paired abstraction or pattern matching on pairs.

`split_pair_conv` \(\langle tm \rangle\) yields the theorem \(\vdash tm = (\text{Fst}tm, \text{Snd}tm)\).

`split_pair_rewrite_tac`, when supplied with a list of terms which have the type of ordered pairs, will expand each occurrence of a term in the list to an explicit ordered pair using `split_pair_conv`, and will then apply `pure_rewrite_tac` to the theorems.

`map_uncurry_conv` takes a term and eliminates all occurrences of `Uncurry` in it by rewriting with the definition and beta reducing the result, and then eliminates all resulting terms of the form \((\text{Fst}tm, \text{Snd}tm)\) in favour of \(tm\).

`map_uncurry_rule` applies `map_uncurry_conv` to the conclusion of a theorem. The effect is to make a definition or theorem using pair patterns work for rewriting in cases where the argument is not supplied as an explicit pair, provided that a paired abstraction was used in a universal quantification enclosing the equation. So if you want to formulate definitions and generalise them with this rule, use paired abstractions in the quantifiers.

Example

```sml
val it = \langle @ x y p @ v, w = A p x, y q p x, Fst p ^ y, Snd p q \rangle : TERM
```

---

```
val rule_asm_tac : TERM \(\rightarrow\) (THM \(\rightarrow\) THM) \(\rightarrow\) TACTIC;
val rule_nth_asm_tac : int \(\rightarrow\) (THM \(\rightarrow\) THM) \(\rightarrow\) TACTIC;

Description For transforming assumptions in situ.

Definitions

\(\text{fun rule_asm_tac term rule} = \text{DROP_AsM_T} \text{term (strip_asm_tac o rule)};\)
\(\text{fun rule_nth_asm_tac int rule} = \text{DROP_NTH_AsM_T} \text{int (strip_asm_tac o rule)};\)
```

---

```
val try : (\'a \rightarrow \'a) \(\rightarrow\) (\'a \rightarrow \'a);

Description Intended for application to rules, but more generally applicable, \(\text{try\,fa}\) is \(fa\) unless an exception is raised during its evaluation, in which case it is \(a\).

Definition

\(\text{fun try f a = f a handle _ => a;}\)
```
SML

**val R.top_anf_tac : TACTIC;**

**Description** Convert real arithmetic subexpressions of the conclusion of the current goal to normal form.

**Example**

```sml
set_goal([], \(\forall x \ y \ z : R \bullet \ z = if \ x = y \ then \ (z +_R y) *_R x else x *_R (z -_R y)\));
```

```sml
a R.top_anf_tac;
```

```sml
(* *** Goal "" *** *)
```

```sml
(* 2 *) \(\forall x \ y \ z : z = (if \ x = y \ then \ x *_R y +_R x *_R z else \sim_R x *_R y +_R x *_R z)\)
```

**Definition**

```sml
val R.top_anf_tac = conv_tac (TOP_MAP_C R.anf_conv);
```

SML

```sml
val COND_CASES_T : TERM -> THM_TACTIC -> TACTIC;
val cond_cases_tac : TERM -> TACTIC;
val less_cases_conv : CONV;
val less_cases_rule : THM -> THM;
```

**Description** A version of CASES_T for use in rewriting conditional goals. It does a case split assuming the term argument or its denial and then rewrites with that assumption before applying the thm tactical argument.

A version of cases_tac for use in rewriting conditional goals. It does a case split assuming the argument or its denial and then rewrites with the un-stripped assumption before stripping it into the assumptions.

**Example**

```sml
set_goal([], \(\forall x +_R y +_R z = if \ x = y \land y = z \ then \ x *_R (\sim_R 3) else x +_R y +_R z\));
```

```sml
a (cond_cases_tac \(\forall x = y \land y = z\));
```

```sml
(* *** Goal "" *** *)
```

```sml
(* 2 *) \(\forall x = y\)
```

```sml
(* 1 *) \(\forall y = z\)
```

```sml
(* ?|-* \(\forall z + z + z = z * 3.\)
```

**Definition**

```sml
fun COND_CASES_T x tt = CASES_T x (fn y => TRY (rewrite_tac [y]) THEN (tt y))
```

```sml
fun cond_cases_tac x = COND_CASES_T x strip_asm_tac;
```

```sml
local
fun less_suc_n_conv t =
    let val (_, [m,sn]) = strip_app t;
        val (_, [n,]) = strip_app sn;
    in
        list_N.elim [m, n] less_plus1_thm
    end;
```

```sml
in
    val less_cases_conv = (RIGHT_C plus1_conv) THEN_C less_suc_n_conv THEN_TRY_C (RIGHT_C
```

```sml
end;
```

6
3 Stripping With Failure

**Signature** `StripFail = sig`

**Description** This signature provides facilities for stripping assumptions which fail if the current goal remains unchanged. This is so that tactics which generate new assumptions, e.g., `fc_tac` can be repeated until no new assumptions are generated.

**Val** `check_asm_tac1 : THM → TACTIC;`

**Description** `check_asm_tac1` is similar to `check_asm_tac` but will fail rather than leave the goal unchanged.

`check_asm_tac1` checks the form of the theorem, `thm`, and then takes the first applicable action from the following table:

<table>
<thead>
<tr>
<th><code>thm</code></th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash t )</td>
<td>proves goal if its conclusion is <code>t</code></td>
</tr>
<tr>
<td>( \Gamma \vdash T )</td>
<td>as <code>fail_tac</code></td>
</tr>
<tr>
<td>( \Gamma \vdash F )</td>
<td>proves goal</td>
</tr>
<tr>
<td>( \Gamma \vdash \neg t )</td>
<td>proves goal if <code>t</code> in assumptions, fails if <code>\neg t</code> is in assumptions, else as <code>asm_tac</code></td>
</tr>
<tr>
<td>( \Gamma \vdash t )</td>
<td>proves goal if <code>\neg t</code> in assumptions, fails if <code>t</code> is in assumptions, else as <code>asm_tac</code></td>
</tr>
</tbody>
</table>

During the search through the assumptions in the last two cases, `check_asm_tac1` also checks to see whether any of the assumptions is equal to the conclusion of the goal, and if so proves the goal. It also checks to see if the conclusion of the theorem is already an assumption, in which case the tactic fails. When all the assumptions have been examined, if none of the above actions is applicable, the conclusion of the theorem is added to the assumption list.

**Uses** Tactic programming.

**See Also** `check_asm_tac`, `strip_asm_tac1`. 


val strip_asm_tac1 : THM -> TACTIC;

Description  strip_asm_tac1 is a tactic for stripping down or otherwise transforming a theorem before adding it into the assumptions.

The transformations it undertakes are determined primarily by the current proof context which contains a conversion for stripping assumptions, but there are in addition a small number of effects which cannot be achieved by a conversion and are built into this tactic.

First the current stripping conversion will be applied repeatedly until it no longer applies.

Then the following simplification techniques will be tried. Using sat as an abbreviation for strip_asm_tac:

\[
\begin{align*}
\text{sat} (\vdash a \land b) & \quad \rightarrow \ \text{sat} (\vdash a) \ \text{THEN sat} (\vdash b) \\
\text{sat} (\exists x \cdot a) & \quad \rightarrow \ \text{sat} (a[x'/x] \vdash a[x'/x]) \\
\text{sat} (\vdash a \lor b)(\{\Gamma\} t) & \quad \rightarrow \ \text{sat} (a \vdash a)(\{\Gamma\} t) \ ; \ \text{sat} (b \vdash b)(\{\Gamma\} t)
\end{align*}
\]

The effect is to break conjunctions into two separate theorems, to do a case split on disjunctions and to skolemise existentials.

After all of the available transformation techniques have been exhausted strip_asm_tac then passes the theorems to check_asm_tac1 (q.v.) to discharge the goal or to generate additional assumptions.

See Also  STRIP_THM_THEN, used to implement this function. check_asm_tac1, strip_tac, strip_asm_conv.

val strip_asm_tac1s : THM list -> TACTIC;

Description  strip_asm_tac1s is a tactic for stripping down or otherwise transforming a list of theorems before adding them into the assumptions.

The effect is similar to applying strip_asm_tac1 to each of the theorems, except that it will fail only if every application of strip_asm_tac1 fails, i.e. if the total effect is null.

See Also  STRIP_THM_THEN1, used to implement this function. check_asm_tac1, strip_tac, strip_asm_conv.

val AND_OR_T : TACTIC * TACTIC -> TACTIC;
val AND_OR : TACTIC * TACTIC -> TACTIC;

Description  t1 AND_OR_T t2 has the same effect as ((TRY t1) THEN t2)ORELSET1 but is faster. AND_OR is an alias for AND_OR_T.

See Also  THEN, ORELS, TRY
**SML**

| val \_THEN\_T1 : (THM \to TACTIC) \to (THM \to TACTIC); |
| val \_THEN1 : (THM \to TACTIC) \to (THM \to TACTIC); |

**Description**  
Similar in effect to \_THEN but will fail only if both the conjuncts fail.  
A theorem tactical to apply a given theorem tactic to both conjuncts of a theorem of the form \( \Gamma \vdash t_1 \land t_2 \).

\[ \_THEN1 \, \text{thmtac} \,(\Gamma \vdash t_1 \land t_2) = \text{thmtac} \,(\Gamma \vdash t_1) \land \text{OR thmtac} \,(\Gamma \vdash t_2) \]

**See Also**  
\_THEN

---

**SML**

| val STRIP\_THM\_THEN1 : THM\_TACTICAL; |

**Description**  
STRIP\_THM\_THEN1 provides a general purpose way of stripping or transforming theorems before using them in a tactic proof. STRIP\_THM\_THEN1 attempts to apply the conversion held for the function in the current proof context, which is extracted by current\_ad\_st\_conv\_. to rewrite the theorem. If that fails it attempts to apply a theorem tactical from the following list (in order):

\[ \_THEN1, \lor \_THEN, \text{SIMPLE}_{\exists} \_THEN \]

The conversion in the current proof context got by current\_ad\_st\_conv\_(q.v.) is derived by applying eqn\_cxt\_conv\_ to an equational context in the proof context extracted by get\_st\_eqn\_cxt.  
The function is partially evaluated with only the theorem tactic and theorem arguments.  
**See Also**  
STRIP\_THM\_THEN

---

**SML**

| val LIST\_AND\_OR\_T : TACTIC list \to TACTIC; |

**Description**  
SOME\_T is similar to EVERY\_T except that it fails only if all the tactics fail.  
SOME\_T tlist is a tactic that applies the head of tlist to its subgoal, and recursively applies the tail of tlist to each resulting subgoal. If any application of a tactic fails then the failure is ignored, but if no applications succeed then SOME\_T will fail.  
SOME is NOT an alias for SOME\_T (its a already a constructor). SOME \[] is equal to fail\_tac.  

**Example**

\[ \text{SOME \[\forall \_tac, \_tac, \forall \_tac\]} \]

is equivalent to

\[ \forall \_tac \land \text{OR} \_tac \land \text{OR} \forall \_tac \]

**See Also**  
EVERY\_T
val MAP_LIST_AND.OR.T : ('a -> TACTIC) -> 'a list -> TACTIC;
val MAP_LIST_AND.OR : ('a -> TACTIC) -> 'a list -> TACTIC;

**Description**  
`MAP_LIST_AND.OR.T` is the same as `MAPEVERY.T` except that it will fail only if no resulting application of a tactic succeeds.

`MAP_LIST_AND.OR.T mapf alist` maps `mapf` over `alist`, and then applies the resulting list of tactics to the goal in sequence (in the same manner as `SOME`, q.v.). `MAP_LIST_AND.OR` is an alias for `MAP_LIST_AND.OR.T`.

**See Also**  
`MAPEVERY`
**Proof Context**

This partial proof context contains only the existence prover `savedthm_cs_\_conv` which attempts to “prove” the consistency of a specification by referring to a standard location in which the consistency theorem may have previously been saved.

**See Also** `savedthm_cs_\_conv, save_cs_\_thm`

---

**Conversion**

Conversion that pushes an existential through an implication where the bound variable is free in the antecedent:

```
val \_\_ => _\_conv : CONV;
```

**Conversion**

Conversion to prove the result of applying the basic existence proving conversion to a conditional function definition using `\_\_ \_ \_conv` to push the existential for the function value through the condition and then discarding the antecedent.

```
val prove_\_\_ \_ \_conv : CONV;
```

---

### 4.2 Force New Theory

Conversion which otherwise has to appear at the top of every document which creates a `ProofPower` theory.

```
val force_new_theory : string \_ unit;
```

**Description**

This is just to save the exception handling which otherwise has to appear at the top of every document which creates a `ProofPower` theory.

It deletes the old theory (if present, from your previous build, by using `force_delete_theory`) and all its children and starts the theory afresh.

**See Also** `force_delete_theory, force_new_pc`

---

### 4.3 Proof Contexts

Conversion which otherwise has to appear at the top of every document which creates a `ProofPower` proof context.

```
val force_new_pc : string \_ unit;
```

**Description**

This is just to save the exception handling which otherwise has to appear at the top of every document which creates a `ProofPower` proof context.

It deletes the old proof context (if present, from your previous build, using `delete_pc`) and starts the proof context afresh.

**See Also** `force_new_theory, delete_pc`

---

**Conversion**

These function allows you to add theorems to a proof context. `add_pc_thms` adds them for all three purposes (stripping conclusions and assumptions and rewriting). `add_pc_thms1` omits assumption stripping.

```
val add_pc_thms : string \_ THM list \_ unit;
val add_pc_thms1 : string \_ THM list \_ unit;
```

**Description**

These function allows you to add theorems to a proof context. `add_pc_thms` adds them for all three purposes (stripping conclusions and assumptions and rewriting). `add_pc_thms1` omits assumption stripping.

**See Also** `add_rw_thms, add_sc_thms, add_st_thms`
4.4 Output Stats

```sml
val output_stats : string -> unit;
```

**Description**  Writes the current values of the profiling statistics to a file as a LaTeX table.

**See Also**  `get_stats`

```sml
end; (* of signature PreConsisProof *)
```

```sml
structure PreConsisProof : PreConsisProof = struct
```

4.5 Partial Primitive Recursive Definitions

This functionality is now to be incorporated into ProofPower and is therefore removed (a patch has been applied).

For each item of clausal definition material we hold:

1. The list of data constructor recognisers. These are the generic terms which must be matchable to the actual argument.

2. The number of free variables there should be in the use of the constructor (e.g. 2 for `Cons`, 0 for `Nil`).

3. An instance of the most general type of the function’s argument.

4. A list of dummy arguments for each “constructor”, to allow dummy conjuncts to be created.

5. The actual theorem. The theorem is an equation, whose LHS is of the form:
   - Universally quantify by one predicate per “constructor”,
   - Existentially quantify by function, `f`,
   - one conjunct per constructor, in same order as predicates.
   - Each conjunct will universally quantified in the order that the the free variables of the subterm to which `f` is first applied, that is a recognised argument by the data constructor recogniser.
   - The body of the conjunct will be the associated predicate applied to each available use of `f` and its arguments, the first being the recognised argument.

```sml
val lthy = get_current_theory_name ();
val _ = open_theory "basic_hol";
val _ = push_merge_pcs ["propositions","paired_abstractions"];
```

To make certain functions independent of proof context changes we need to create a (temporary) build proof context equivalent to the supplied “predicates”, in the fields that matter. As we have not committed the sources, we have to do this the hard way:
fun lget x = fst(hd x);
val _ = new_pc "build_predicates";
val _ = set_rw_eqn_cxt ((lget o get rw_eqn_ctx) "propositions" @
    (lget o get rw_eqn_ctx) "paired_abstractions")
    "build_predicates";
val _ = set_sc_eqn_cxt ((lget o get sc_eqn_ctx) "propositions" @
    (lget o get sc_eqn_ctx) "paired_abstractions")
    "build_predicates";
val _ = set_st_eqn_cxt ((lget o get st_eqn_ctx) "propositions" @
    (lget o get st_eqn_ctx) "paired_abstractions")
    "build_predicates";
val _ = set_rw_canons ((lget o get rw_canons) "propositions" @
    (lget o get rw_canons) "paired_abstractions")
    "build_predicates";

Flatten a paired structure:

val strip_pair : TERM -> TERM list = strip_leaves dest_pair;

Flatten a conjunction structure (strip_\& only flattens to the right):

val full_strip_\& : TERM -> TERM list = strip_leaves dest_\&;

We wish to “mark” some terms, to prevent stripping going too far. We use pf/TS as a marker.

“mark” a term:

local
  val ci = "pf/TS:BOOL -> BOOL";
in
fun mark (tm:TERM):TERM = mk_app(ci,tm)
end;

val _ = delete_pc "build_predicates";
val _ = pop_pc();
val _ = open_theory lthy;

Conversion (written by rda) that pushes an existential through an implication where the bound
variable is free in the antecedent:
Conversion (written by rda) to prove the result of applying the basic existence proving conversion to a conditional function definition using the above conversion to push the existential for the function value through the condition and then discarding the antecedent.

A new value of type refTHM called saved.cs.∃.thm is used to store consistency results.

I also have a special partial proof context with a consistency prover which knows to look for the consistency proof in this special place. This is the consistency prover:

```
fun savedthm cs.∃.conv x =
  if x = $ (concl(!saved.cs.∃.thm))
  then (∀t_intro (!saved.cs.∃.thm)) handle _ => (∗ eq ∗) refl_conv x
  else (∗ eq ∗) refl_conv x;
```
5 Unifying Forward Chaining

5.1 Specifications

```sml
signature UnifyForwardChain = sig
  Description This is the signature of facilities for forward chaining based on unification rather than matching.

val simple⇒unify_mp_rule1 : THM ⇒ THM ⇒ THM ;
  Description A unifying Modus Ponens rule for an implicative theorem.
  Rule
  \[ \Gamma_1 \vdash \forall x_1 \ldots \bullet t_1 \Rightarrow t_2 ; \Gamma_2 \vdash \forall y_1 \ldots \bullet t_1' \Rightarrow \Gamma_1 \cup \Gamma_2 \vdash \forall z_1 \ldots \bullet t_2' \Rightarrow \_unify_mp_rule1 \]
  where \( t_1' \) is unifiable with \( t_1 \). Type instantiation and substitution is permitted for the \( x_i \) in \( t_1 \), the \( y_i \) in \( t_1' \) and instantiation of the type variables in \( t_1 \) which do not occur in \( \Gamma_1 \) and those in \( t_1' \) which do not occur in \( \Gamma_2 \). \( t_2' \) is obtained from \( t_2 \) by applying to it the substitution to \( t_1 \) required for its unification. The \( z_i \) will be the variables free in \( t_2' \) which were not previously free either in \( t_2 \) or \( t_1' \). No type instantiation or substitution will occur in the assumptions of either theorem.

Pairs are not supported in the bindings.

Errors
  7044 Cannot match \(?0 \) and \(?1 \)
  7045 \(?0 \) is not of the form \( \forall x_1 \ldots \bullet u \Rightarrow v' \)
```

end; (* of structure PreConsisProof *)
A matching Modus Ponens rule for an implicative theorem, supporting paired abstraction.

\[ \Gamma_1 \vdash \forall x_1 \ldots \bullet t_1 \Rightarrow t_2; \quad \Gamma_2 \vdash \forall y_1 \ldots \bullet t_1' \]
\[ \Gamma_1 \cup \Gamma_2 \vdash \forall z_1 \ldots \bullet t_2' \Rightarrow \text{unify_mp_rule1} \]

where \( t_1' \) is unifiable with \( t_1 \). Type instantiation and substitution is permitted for the \( x_i \) in \( t_1 \), the and \( y_i \) in \( t_1' \) and instantiation of the type variables in \( t_1 \) which do not occur in \( \Gamma_1 \) and those in \( t_1' \) which do not occur in \( \Gamma_2 \). \( t_2' \) is obtained from \( t_2 \) by applying to it the substitution to \( t_1 \) required for its unification. The \( z_i \) will be the variables free in \( t_2' \) which were not previously free either in \( t_2 \) or \( t_1' \). No type instantiation or substitution will occur in the assumptions of either theorem.

Pairs are supported in the bindings.

\[ \text{Errors} \]
\[ 7044 \quad \text{Cannot match } ?0 \text{ and } ?1 \]
\[ 7045 \quad ?0 \text{ is not of the form } '\Gamma \vdash \forall x_1 \ldots \bullet u \Rightarrow v' \]

This is a rule which uses a list of possibly universally quantified implications and a list of other theorems to infer new theorems, using \( \Rightarrow \text{unify_mp_rule1} \). (ufc_rule is an alias for unify_forward_chain_rule.) ufc_rule imps ants returns the list of all theorems which may be derived by applying \( \Rightarrow \text{unify_mp_rule1} \) to a theorem from imps and one from ants. As a special case, if any theorem to be returned is determined to have \( \lnot F \) as its conclusion, the first such found will be returned as a singleton list. In order to work well in conjunction with fc_canon and ufc_tac the theorems returned by \( \Rightarrow \text{unify_mp_rule1} \) are transformed as follows:

1. Theorems of the form: \( \vdash \forall x_1 \ldots \bullet t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow \lnot t_k \Rightarrow F \) have their final implication changed to \( t_k \).
2. Theorems of the form: \( \vdash \forall x_1 \ldots \bullet t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow t_k \Rightarrow F \) have their final implication changed to \( \Rightarrow \lnot t_k \).
3. All theorems are universally quantified over all the variables which appear free in their conclusions but not in their assumptions (using all_\forall_intro).

Note that the use of \( \Rightarrow \text{unify_mp_rule1} \) gives some control over the number of results generated, since variables which appear free in imps are not considered as candidates for instantiation.

The rule does not check that the theorems in its first argument are (possible universally) quantified implications. Theorems which are not of this form will be ignored.
val UFC_T1 : 
  (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_UFC_T1 : 
  (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_UFC_T1 : 
  (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_UFC_T1 : 
  (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;

**Description**  These are tacticals which use theorems whose conclusions are implications, or from which implications can be derived, to reason forwards from the assumptions of a goal.

The description of ufctac should be consulted for the basic forward chaining algorithms used. The significance of the final argument and of the presence or absence of ASM and ALL in the name is exactly as for fc_tac and its relatives.

The tacticals allow variation of the canonicalisation function used to obtain implications from the argument theorems and of the tactic generating function used to process the theorems derived by the forward inference. The canonicalisation function to use is the first argument and the tactic generating function is the second. (Related tacticals with names ending in T rather than T1 are also available for the simpler case when wants to use the same canonicalisation function as fc_tac and just to vary the tactic generating function.)

**Examples**  If the theorem argument comprises only implications which are to be used without canonicalisation, one might use: UFC_T1 id_canon (MAP_LIST_AND_OR strip_asm_tac).

If one has an instance of t1 as an assumption and one wishes to use the bi-implication in a theorem of the form t1 \implies (t2 \iff t3) for rewriting, one might use UFC_T1 id_canon rewrite_tac.

**See Also**  ufctac, asm_ufctac, bc_tac, UFC_T.
val \texttt{UFC.T} : \\
  (THM list \rightarrow TACTIC) \rightarrow THM list \rightarrow TACTIC; \\
val \texttt{ALL_UFC.T} : \\
  (THM list \rightarrow TACTIC) \rightarrow THM list \rightarrow TACTIC; \\
val \texttt{ASM_UFC.T} : \\
  (THM list \rightarrow TACTIC) \rightarrow THM list \rightarrow TACTIC; \\
val \texttt{ALL_ASM_UFC.T} : \\
  (THM list \rightarrow TACTIC) \rightarrow THM list \rightarrow TACTIC; \\
val \texttt{ALL_ASM_UFC.T1} : \\
  (THM list \rightarrow TACTIC) \rightarrow THM list \rightarrow TACTIC; \\

\textbf{Description} \hspace{1em} \text{These are tacticals which use theorems whose conclusions are implications, or from which implications can be derived, to reason forwards from the assumptions of a goal. (The tacticals with } \texttt{UFC} \text{ are aliases for the corresponding ones with } \texttt{UNIFY_FORWARD_CHAIN}.\text{)}

The description of \texttt{ufc_tac} should be consulted for the basic forward chaining algorithms used. The significance of the final argument and of the presence or absence of \texttt{ASM} and \texttt{ALL} in the name is exactly as for \texttt{ufc_tac} and its relatives.

The tacticals allow variation of the tactic generating function used to process the theorems derived by the forward inference. The tactic generating function to be used is given as the first argument.

\textbf{Examples} \hspace{1em} \texttt{ufc_tac} \text{ is the same as: } \texttt{UFC.T strip_asm_tac1}. \\

To rewrite the goal with the results of the forward inference one could use \texttt{UFC.T rewrite_tac}.

\textbf{See Also} \hspace{1em} \texttt{ufc_tac, asm_ufc_tac, UFC.T1}. \\

18
These are tactics which use theorems whose conclusions are implications, or from which implications can be derived using the canonicalisation function \( \text{fc\_canon} \), q.v., to reason forwards from the assumptions of a goal.

The basic step is to take a theorem of the form \( \Gamma \vdash t_1 \Rightarrow t_2 \) and an assumption of the form \( t_1' \) where \( t_1' \) is unifiable with \( t_1 \) and to deduce the corresponding instance of \( t_2' \). The new theorem, \( \Delta \vdash t_2' \) say, may then be stripped into the assumptions.

In the case of \( \text{ufc\_tac} \) the implicative theorem is always derived from the list of theorems given as an argument. In the case of \( \text{asm\_ufc\_tac} \) the assumptions are also used. In all of the tactics the rule \( \text{fc\_canon} \) is used to derive an implicative canonical form from the candidate implicative theorems. Normally combination of an implicative theorem and an assumption is then tried in turn and all resulting theorems are stripped into the assumptions of the goal. However, if the chaining results contain a theorem whose conclusion is \( \begin{eqnarray*} \neg \end{eqnarray*} F \end{eqnarray*} \) then the first such found will be stripped into the assumptions, and all other theorems discarded.

If one of the implications has the form \( t_1 \Rightarrow t_2 \Rightarrow t_3 \) or \( t_1 \land t_2 \Rightarrow t_3 \) and if assumptions matching \( t_1 \) and \( t_2 \) are available, \( \text{ufc\_tac} \) or \( \text{asm\_ufc\_tac} \) will derive an intermediate implication \( t_2 \Rightarrow t_3 \) and \( \text{asm\_ufc\_tac} \) could then be used to derive \( t_3 \). The variants with \( \text{all} \) may be used to derive \( t_3 \) directly without generating any intermediate implications in the assumptions. They work like the corresponding tactic without \( \text{all} \) but any theorems which are derived which are themselves implications are not stripped into the assumptions but instead are used recursively to derive further theorems. When no new implications are derivable all of the non-implicative theorems derived during the process are stripped into the assumptions.

Note that the use of \( \text{fc\_canon} \) implies that conversions from the proof context are applied to generate implications. E.g., in an appropriate proof-context covering set theory, \( a \subseteq b \) might be treated as the implication \( \forall x\cdot x \in a \Rightarrow x \in b \). Also variables which appear free in a theorem are not considered as candidates for instantiation (in order to give some control over the number of results generated). The tacticals, \( \text{UFC\_T1} \) and \( \text{ASM\_UFC\_T1} \) may be used to avoid the use of \( \text{fc\_canon} \).

For example, the tactic:

\[ \text{asm\_ufc\_tac}[] \text{ THEN } \text{asm\_ufc\_tac}[] \]

will prove the goal:

\[ \{ p x, \forall x\cdot p x \Rightarrow q x, \forall x\cdot q x \Rightarrow r x \} r x. \]

The variants with \( \iff \) in the name use \( \text{fc\_canon} \) instead of \( \text{fc\_canon} \) for processing the rules so that a concluding equivalence is not broken into implications and the results of forward chaining can be used for rewriting (however, this still won’t work unless there are outer quantifiers to prevent the equivalence from being broken up when stripped into the assumptions).

All of these tactics add the results into the assumptions using \( \text{strip}_a, \text{sms}_a t a c1 \) and therefore fail if no new assumptions are added (unless the goal is discharged), except the ones whose name includes \( \text{rewrite} \) which attempt the rewrite the conclusion of the goal with the results instead of stripping them into the assumptions.

See Also \( \text{bc\_tac}, \text{UFC\_T}, \text{ASM\_UFC\_T}, \text{UFC\_T1}, \text{ASM\_UFC\_T1} \).
5.2 Implementation

In unify, \( \Rightarrow \text{rule} \) the two theorems will be unified as necessary to permit inference by modus ponens. Only variables universally quantified at the outer level will be candidates for instantiation, and in each of the premises only type variables which do not appear in the assumptions will be eligible for instantiation. The two theorems are stripped of their outer universal quantifiers and the antecedent of the first (which must be an implication) will also be stripped of universal quantifiers and will then be unified with the second (without permitting substitution for the quantifiers on the antecedent). If this succeeds the consequent is inferred (after adding quantifiers as necessary to the second theorem and instantiating the quantifiers as necessary in the first theorem). Then any variables which are free in the result but were previously bound are rebound.

6 Embedding Languages

In this section facilities for very simple deep embeddings are provided. In the intended applications an easily readable grammar is more important than an efficient parser, so we provide a simple recursive descent parser which is parameterised by a grammar coded into a HOL term.
structure ParseComb : ParseComb = struct

exception parse_fail of int;

datatype 'a tree = MkTree of 'a * 'a tree list;

type 'a ptree = int * 'a tree;

type 'a pt_pack = \{ mk_split: 'a ptree list -> 'a ptree,
            mk_split: 'a ptree list -> 'a ptree,
            mk_split: int * 'a ptree -> 'a ptree,
            mk_split: 'a ptree OPT -> 'a ptree\};

type 'a parser = 'a pt_pack -> 'a list -> 'a ptree * 'a list;

datatype 'a pttag = Ptint of int | Ptlit of 'a | Ptnone;

fun mk_ptp (mk_tree: ('a pttag) * 'b list -> 'b) = \{
    mk_split = fn l => mk_tree (Ptlit l, []),
    mk_psplit = fn ptl => mk_tree (Ptint p, ptl),
    mk_psplit = fn (i, pt) => mk_tree (Ptint i, [pt]),
    mk_psplit = fn pto => mk_tree (Ptint (fn Nil => [] | Value pt => [mk_tree (Ptint, [pt])]) pto)
\};

fun alt_parse (pl: 'a parser list) (ptp as {mk_psplit, ...}): 'a pt_pack (li: 'a list) =
    let fun aux (pl as (hp::tpl)) n j = (let val (pt, rli) = hp ptp li in (mk_psplit (n, pt), rli) end
        handle parse_fail k => aux tpl (n+1) (if j < k then j else k))
    in aux [] n i = raise parse_fail i
    in aux pl 0 (length li) end;

fun seq_parse pl (ptp as {mk_psplit, ...}): 'a pt_pack li =
    let fun aux (p, (ptl, li)) = let val (npt, rli) = p ptp li in (npt::ptl, rli) end
    in (mk_psplit (rev ptl, nli))
    end;

fun opt_parse p (ptp as {mk_psplit, ...}): 'a pt_pack li =
    let val (pt, nli) = p ptp li
    in (mk_psplit (Value pt), nli)
    end handle parse_fail i => (mk_psplit Nil, li);
fun star_parse (p'a parser) (ptp as {mk_plist, ...'}a pt_pack) li =
  let fun aux1 li = let val (pt, rli) = p ptp li
   in ([pt], rli)
   end handle parse_fail i => ([], li)
  fun aux2 li =
    let val (ptl, rli) = aux1 li
    in case ptl of
        [] => ([], rli)
    | ptl => let val (ptl2, sli) = aux2 rli
      in (ptl @ ptl2, sli)
      end
    end
  val (ptl, nli) = aux2 li
  in (mk_plist (rev ptl), nli)
  end;

fun plus_parse (p'a parser) (ptp as {mk_plist, ...'}a pt_pack) li =
  let fun aux1 li = let val (pt, rli) = p ptp li
   in ([pt], rli)
   end handle parse_fail i => ([], li)
  fun aux2 li =
    let val (ptl, rli) = aux1 li
    in case ptl of
        [] => ([], rli)
    | ptl => let val (ptl2, sli) = aux2 rli
      in (ptl @ ptl2, sli)
      end
    end
  val (ptl, nli) = aux2 li
  in (mk_plist (rev ptl), nli)
  end;
end; (* of structure ParseComb *)
signature Grammar = sig

Description  This signature provides access to grammars for a simple parser. A grammar consists of a list of phrase definitions. A phrase definition consists of a phrase name and an expression. An expression is either:

1. a literal (not sure what that is at the moment)
2. a choice among a list of expressions
3. a list of expressions
4. an expression to be permitted any number of times, with an optional list separator and an option to allow zero times.

We require functions for constructing, discriminating and destructing.

signature Trawling = sig

Description  The functions in this signature search the ancestors of the current theory for theorems which do something with the current goal, i.e. which rewrite the conclusion, backward chain from it, or forward chain from the assumptions.
datatype THMDET = Spec of TERM | Thm of (string * string);
val on_conc : (TERM -> 'a) -> 'a;
val on_asms : (TERM list -> 'a) -> 'a;
val rew_thms : TERM -> ((int * THMDET) * THM) list;
val rew_specs : TERM -> ((int * THMDET) * THM) list;
val bc_thms : TERM -> ((int * THMDET) * THM) list;
val fc_thms : TERM list -> ((int * THMDET) * THM) list;
val all_fc_thms : TERM list -> ((int * THMDET) * THM) list;
val todo : unit -> {bc: int, fc: int, rw: int};
val td_thml : THMDET list -> THM list;

Description on_conc and on_asms apply their arguments respectively to the conclusion or the
list of assumptions of the current goal.
rew_thms, rew_specs, bc_thms retrieve respectively theorems (thms) or specifications (specs)
which can be used to sucessfully rewrite (rew) or backchain from (bc) the term supplied as an
argument.
fc_thms and all_fc_thms, when supplied with a list of assumptions, retrieve theorems which will
yield results using fc_tac and all_fc_tac.
todo() returns a count of how many theorems or specifications are applicable to the current goal,
classified according to the method of application. bc = back chaining, fc = forward chaining, rw = rewriting.

8 For Inductive and Coinductive Definitions

8.1 Some Handy SML functions

The following functions have been moved here from [1].
fun lfoldl f a [] = a
| lfoldl f a (h::t) = lfoldl f (f (a, h)) t;

fun lfoldr f a [] = a
| lfoldr f a (h::t) = f (a, (lfoldr f h t));

fun list_s_enter [] d = d
| list_s_enter ((s,v)::t) d = list_s_enter t (s_enter s v d);

fun list_to_sdict l = list_s_enter l initial_s_dict;

fun list_pos e [] = 0
| list_pos e (h::t) = if h = e then 1 else let val p = list_pos e t
| in if p = 0 then 0 else p+1 end;

val strip_
  ->_type = strip_spine_right dest_
  ->_type;

fun list_mk_
  ->_type (h::t) = lfoldr mk_
  ->_type h t;

fun match_mk_app (f, a) = mk_app(f, a) handle _ => "\_\_\_\_f\_\_\_;l\_\_a\_\_\_";

fun list_match_mk_app (f, al) = lfoldl match_mk_app f al;

fun mk_tree_type ty = mk_ctype("TREE", [ty]);

fun mk_tree t tl = let val tt = type_of t
| in mk_app (mk_const("MkTree", mk_
  ->_type (mk_
  ->_type (tt, mk_ctype("LIST", [tt])), mk_tree_type tt)),
| tl)
| end;

fun dest_tree tr = dest_app tr;

fun list_mk_tree t tl = mk_tree t (mk_list tl);

fun dest_tree_list tr = let val (t, l) = dest_tree tr in (t, dest_list l) end;

fun gen_type_map cf vf ty =
let fun aux (VarType v) = vf v
    aux (CType (s, tl)) = cf s ((map (aux o dest_simple_type)) tl)
in aux (dest_simple_type ty)
end;

local fun front_last [e] = ([], e)
    front_last (f::t) =
    let val (f2, l) = front_last t
    in (f::f2, l)
    end
in fun front x = let val (f, l) = front_last x in f end
fun last x = let val (f, l) = front_last x in l end
fun right_rotate_list [] = []
    | right_rotate_list [e] = [e]
    | right_rotate_list x = let val (f, l) = front_last x in l :: f end
fun left_rotate_list [] = []
    | left_rotate_list [e] = [e]
    | left_rotate_list (h::t) = t @ [h]
end;

8.2 False Equations Between Set Displays

The following code defines a conversion for transforming \( (\text{⌜F⌝}) \) false equations between set displays.

SML

\[\text{infix symdiff;}\]

\[\text{fun x symdiff y} = (x \text{ diff } y) \text{ cup } (y \text{ diff } x);\]

\[\text{fun dest_enum l} =\]
\[\quad \text{(fn DEnumSet els => els}\]
\[\quad \quad \text{| DØ t => []}) (dest_term l);}\]

\[\text{fun enum_eq_sdiff t} =\]
\[\quad \text{let val DEq (lhs, rhs) = dest_term t}\]
\[\quad \text{in (dest_enum lhs) symdiff (dest_enum rhs)}\]
\[\text{end;}\]

\[\text{fun false_enum_eq_conv t} =\]
\[\quad \text{let val (dt :: _) = enum_eq_sdiff t}\]
\[\quad \text{in}\]
\[\quad \quad \text{tac-proof([[], \text{⌜F⌝})},\]
\[\quad \quad \text{rewrite_tac [sets_ext_clauses]}
\[\quad \quad \text{THEN } \text{in_tac}\]

26
THEN \exists\_tac\ dt \ THEN\ prove\_tac\\[\]\\)
end\ handle\ _\ =\ =>\ fail\_conv\ t;

val\ false\_enum\_eq\_tac = conv\_tac\ (MAP\ C\ false\_enum\_eq\_conv);
### INDEX

<table>
<thead>
<tr>
<th>Symbol/Word</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>'prove_3' =&gt; _conv</td>
<td>15</td>
</tr>
<tr>
<td>'savedthm_cs_3_proof'</td>
<td>11</td>
</tr>
<tr>
<td>[ \text{FC}_T ]</td>
<td>1</td>
</tr>
<tr>
<td>[ \text{T}_\text{canon} ]</td>
<td>4</td>
</tr>
<tr>
<td>[ \text{unify_mp_rule1} ]</td>
<td>16</td>
</tr>
<tr>
<td>[ \text{conv} ]</td>
<td>11</td>
</tr>
<tr>
<td>[ \text{lemma} ]</td>
<td>14</td>
</tr>
<tr>
<td>&amp;&amp;_THEN1</td>
<td>9</td>
</tr>
<tr>
<td>&amp;&amp;_THEN_T1</td>
<td>9</td>
</tr>
<tr>
<td>\R_top_anf_tac</td>
<td>4</td>
</tr>
<tr>
<td>abc_canon</td>
<td>4</td>
</tr>
<tr>
<td>abc_tac</td>
<td>11</td>
</tr>
<tr>
<td>add_pc_thms</td>
<td>11</td>
</tr>
<tr>
<td>add_pc_thms1</td>
<td>11</td>
</tr>
<tr>
<td>all_ =&gt; _intro_canon</td>
<td>4</td>
</tr>
<tr>
<td>all_asm_ufc &amp;&amp; _rewrite_tac</td>
<td>19</td>
</tr>
<tr>
<td>ALL_ASM_UF_C &amp;&amp; _T</td>
<td>18</td>
</tr>
<tr>
<td>all_asm_ufc &amp;&amp; _tac</td>
<td>19</td>
</tr>
<tr>
<td>all_asm_ufc_T</td>
<td>17</td>
</tr>
<tr>
<td>all_asm_ufc_T1</td>
<td>17</td>
</tr>
<tr>
<td>all_fc_thms</td>
<td>24</td>
</tr>
<tr>
<td>all_ufc &amp;&amp; _rewrite_tac</td>
<td>19</td>
</tr>
<tr>
<td>ALL_UF/_C &amp;&amp; _T</td>
<td>18</td>
</tr>
<tr>
<td>all_ufc &amp;&amp; _tac</td>
<td>19</td>
</tr>
<tr>
<td>ALL_UF/_C_T</td>
<td>17</td>
</tr>
<tr>
<td>ALL_UF/_C_T1</td>
<td>17</td>
</tr>
<tr>
<td>all_ufc_tac</td>
<td>19</td>
</tr>
<tr>
<td>alt_parse</td>
<td>20</td>
</tr>
<tr>
<td>AND_OR</td>
<td>8</td>
</tr>
<tr>
<td>AND_OR_T</td>
<td>8</td>
</tr>
<tr>
<td>asm_abc_tac</td>
<td>4</td>
</tr>
<tr>
<td>ASM_UF/_C_T</td>
<td>18</td>
</tr>
<tr>
<td>ASM_UF/_C_T1</td>
<td>17</td>
</tr>
<tr>
<td>asm_ufc_tac</td>
<td>19</td>
</tr>
<tr>
<td>be_thms</td>
<td>24</td>
</tr>
<tr>
<td>check_asm_tac</td>
<td>4</td>
</tr>
<tr>
<td>COND_CASES_T</td>
<td>7</td>
</tr>
<tr>
<td>cond_cases_tac</td>
<td>6</td>
</tr>
<tr>
<td>dest_tree</td>
<td>25</td>
</tr>
<tr>
<td>dest_tree_list</td>
<td>25</td>
</tr>
<tr>
<td>false_enum_eq_conv</td>
<td>26</td>
</tr>
<tr>
<td>false_enum_eq_tac</td>
<td>26</td>
</tr>
<tr>
<td>fe_thms</td>
<td>24</td>
</tr>
<tr>
<td>force_new_pc</td>
<td>14</td>
</tr>
<tr>
<td>force_new_theory</td>
<td>14</td>
</tr>
<tr>
<td>front</td>
<td>11</td>
</tr>
<tr>
<td>full_strip_&amp;</td>
<td>13</td>
</tr>
<tr>
<td>Grammar</td>
<td>23</td>
</tr>
<tr>
<td>last</td>
<td>26</td>
</tr>
<tr>
<td>left_rotate_list</td>
<td>26</td>
</tr>
<tr>
<td>less_cases_conv</td>
<td>6</td>
</tr>
<tr>
<td>less_cases_rule</td>
<td>6</td>
</tr>
<tr>
<td>LIST_AND_OR_T</td>
<td>9</td>
</tr>
<tr>
<td>list_map_eq_sym_rule</td>
<td>4</td>
</tr>
<tr>
<td>list_match_mk_sym_rule</td>
<td>4</td>
</tr>
<tr>
<td>list_mk_&amp;_type</td>
<td>25</td>
</tr>
<tr>
<td>list_mk_tree</td>
<td>25</td>
</tr>
<tr>
<td>list_mk_tree_type</td>
<td>25</td>
</tr>
<tr>
<td>map_list_AND_OR_T</td>
<td>10</td>
</tr>
<tr>
<td>map_uncurry_conv</td>
<td>5</td>
</tr>
<tr>
<td>map_uncurry_rule</td>
<td>5</td>
</tr>
<tr>
<td>map_uncurry_rule_2</td>
<td>5</td>
</tr>
<tr>
<td>mark_</td>
<td>13</td>
</tr>
<tr>
<td>match_mk_sym_rule</td>
<td>13</td>
</tr>
<tr>
<td>mk_tree</td>
<td>25</td>
</tr>
<tr>
<td>mk_tree_type</td>
<td>25</td>
</tr>
<tr>
<td>on_asms</td>
<td>24</td>
</tr>
<tr>
<td>on_conc</td>
<td>24</td>
</tr>
<tr>
<td>opt_parse</td>
<td>20</td>
</tr>
<tr>
<td>output_stats</td>
<td>12</td>
</tr>
<tr>
<td>ParseComb</td>
<td>20</td>
</tr>
<tr>
<td>pc_canon</td>
<td>3</td>
</tr>
<tr>
<td>plus_parse</td>
<td>20</td>
</tr>
<tr>
<td>PreConsisProof</td>
<td>10</td>
</tr>
<tr>
<td>prove_3 &amp;&amp; _conv</td>
<td>11</td>
</tr>
<tr>
<td>prove_3 &amp;&amp; _conv_1</td>
<td>14</td>
</tr>
<tr>
<td>proof_path</td>
<td>20</td>
</tr>
<tr>
<td>pf_tree</td>
<td>20</td>
</tr>
<tr>
<td>Rb_Tactics1</td>
<td>3</td>
</tr>
<tr>
<td>rew_specs</td>
<td>7</td>
</tr>
<tr>
<td>rew_thms</td>
<td>24</td>
</tr>
<tr>
<td>right_rotate_list</td>
<td>26</td>
</tr>
<tr>
<td>rule_asm_tac</td>
<td>5</td>
</tr>
<tr>
<td>rule_canon</td>
<td>3</td>
</tr>
<tr>
<td>rule_nth_asm_tac</td>
<td>5</td>
</tr>
<tr>
<td>save_cs_3_thm</td>
<td>10</td>
</tr>
<tr>
<td>savedthm_cs_3_conv</td>
<td>11</td>
</tr>
<tr>
<td>seq_parse</td>
<td>24</td>
</tr>
<tr>
<td>simple_ =&gt; _unify_mp_rule1</td>
<td>15</td>
</tr>
<tr>
<td>split_pair_conv</td>
<td>6</td>
</tr>
<tr>
<td>split_pair_rewrite_tac</td>
<td>6</td>
</tr>
<tr>
<td>star_parse</td>
<td>20</td>
</tr>
<tr>
<td>strip_ =&gt; _type</td>
<td>8</td>
</tr>
<tr>
<td>strip_asm_tac</td>
<td>8</td>
</tr>
<tr>
<td>strip_asm_tac1</td>
<td>8</td>
</tr>
<tr>
<td>strip_pair</td>
<td>13</td>
</tr>
<tr>
<td>STRIP_THM_THEN1</td>
<td>9</td>
</tr>
<tr>
<td>StripFail</td>
<td>7</td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>SYM_ASMS_T</td>
<td>4</td>
</tr>
<tr>
<td>symdiff</td>
<td>26</td>
</tr>
<tr>
<td>td_thml</td>
<td>24</td>
</tr>
<tr>
<td>THMDET</td>
<td>24</td>
</tr>
<tr>
<td>todo</td>
<td>24</td>
</tr>
<tr>
<td>Trawling</td>
<td>24</td>
</tr>
<tr>
<td>try</td>
<td>24</td>
</tr>
<tr>
<td>ufc_rule</td>
<td>16</td>
</tr>
<tr>
<td>UFC_T</td>
<td>18</td>
</tr>
<tr>
<td>UFC_T1</td>
<td>17</td>
</tr>
<tr>
<td>ufc_tac</td>
<td>19</td>
</tr>
<tr>
<td>unify_forward_chain_rule</td>
<td>16</td>
</tr>
<tr>
<td>UnifyForwardChain</td>
<td>15 20</td>
</tr>
</tbody>
</table>