Miscellaneous Tactics

Roger Bishop Jones

Date: 2012/11/24 20:22:24

Abstract

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

http://www.rbjones.com/rbjpub/pp/doc/t010.pdf

Id: t010.doc,v 1.22 2012/11/24 20:22:24 rbj Exp

Contents

References

[1] Roger Bishop Jones. Inductive, Co-Inductive and Psuedo-(Co-)Inductive Definitions in ProofPower. *RBJones.com*, 2010. `http://www.rbjones.com/rbjpub/pp/doc/t007.pdf`.

[2] Roger Bishop Jones. Introduction to Work in Progress. *RBJones.com*, 2010. `http://www.rbjones.com/rbjpub/pp/doc/t000.pdf`.

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 "⊑", Simple) ]
        end
handle _ => ();
val _ =     let open ReaderWriterSupport.PrettyNames;
    in add_new_symbols [ (["identical2"], Value "≡", 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.

```
SML
val pc_canon: string -> CANON -> CANON;
```

Description Creates a CANON which executes in a specific proof context.

```
SML
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*].

SML

val ⇔_**FC_T**: (*THM list* −> *TACTIC*) −> *THM list* −> *TACTIC*;

Description For doing forward chaining using *fc_* ⇔ _*canon*.

SML

val **all_⇒_intro_canon**: *CANON*;

Description This is *rule_canon all_* ⇒ _*intro*.

See Also *rule_canon*, *all_* ⇒ _*intro*

SML

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.

SML

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*.

SML

val **map_eq_sym_rule** : *THM* −> *THM*;
val **list_map_eq_sym_rule** : *THM list* −> *THM list*;
val **SYM_ASMS_T** : (*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_ASMS_T 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*

SML

```
val split_pair_conv : TERM -> THM;
val split_pair_rewrite_tac : TERM list -> THM list -> TACTIC;
val map_uncurry_conv : CONV;
val map_uncurry_rule : THM -> THM;
```

Description 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 ⌜*tm*⌝ yields the theorem $\vdash tm = (Fsttm, Sndtm)$.

split_pair_rewrite_tac, when supplied with a list of terms which have the type of ordered pairs, will expand each occurence 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 occurences of *Uncurry* in it by rewriting with the definition and beta reducing the result, and then eliminates all resulting terms of the form $(Fsttm, Sndtm)$ 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

```
(concl o map_uncurry_rule) (asm_rule ⌜∀x y (v, w)• A (x,y) (v,w) = x=v ∧ y=w⌝);
val it = ⌜∀ x y p• A (x, y) p = x = Fst p ∧ y = Snd p⌝ : TERM
```

SML

```
val rule_asm_tac : TERM -> (THM -> THM) -> TACTIC;
val rule_nth_asm_tac : int -> (THM -> THM) -> TACTIC;
```

Description For transforming assumptions in situ.

Definitions

```
fun rule_asm_tac term rule = DROP_ASM_T term (strip_asm_tac o rule);
fun rule_nth_asm_tac int rule = DROP_NTH_ASM_T int (strip_asm_tac o rule);
```

SML

```
val try : ('a -> 'a) -> ('a -> 'a);
```

Description Intended for application to rules, but more generally applicable, $tryfa$ is fa unless an exception is raised during its evaluation, in which case it is a.

Definition

```
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

```
set_goal([], ⌜∀x y z:ℝ• z = if x = y then (z +R y) *R x else x *R (z −R y)⌝);
a ℝ_top_anf_tac;
(* *** Goal "" *** *)

(* ?⊢ *) ⌜∀ x y z• z = (if x = y then x *R y +R x *R z else ∼R x *R y +R x *R z)⌝
```

Definition

```
val ℝ_top_anf_tac = conv_tac (TOP_MAP_C ℝ_anf_conv);
```

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 asssumption 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 asssumption before stripping it into the assumptions.

Example

```
set_goal([], ⌜x +R y +R z = if x = y ∧ y = z then x *R (ℕℝ 3) else x +R y +R z⌝);
a (cond_cases_tac ⌜x = y ∧ y = z⌝);
(* *** Goal "" *** *)

(*  2 *) ⌜x = y⌝
(*  1 *) ⌜y = z⌝

(* ?⊢ *) ⌜z + z + z = z * 3.⌝
```

Definition

```
fun COND_CASES_T x tt = CASES_T x (fn y => TRY (rewrite_tac [y]) THEN (tt y));
fun cond_cases_tac x = COND_CASES_T x strip_asm_tac;
local
fun less_suc_n_conv t =
    let val (_, [m,sn]) = strip_app t;
        val (_, [n,_]) = strip_app sn;
    in list_∀_elim [m, n] less_plus1_thm
    end;
in val less_cases_conv = (RIGHT_C plus1_conv) THEN_C less_suc_n_conv THEN_TRY_C (RIGHT
end;
```

SML

```
|end; (* of signature RbjTactics1 *)
```

SML

```
|structure RbjTactics1 : RbjTactics1 = struct
```

SML

```
|end; (* of structure RbjTactics1 *)
```

3 Stripping With Failure

SML

```
|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.

SML

```
|val check_asm_tac1 : THM -> TACTIC;
```

Description *check_asm_tac*1 is a similar to *check_asm_tac* but will fail rather than leave the goal unchanged.

*check_asm_tac*1 *thm* checks the form of the theorem, *thm*, and then takes the first applicable action from the following table:

thm	action
$\Gamma \vdash t$	proves goal if its conclusion is t
$\Gamma \vdash T$	as *fail_tac*
$\Gamma \vdash F$	proves goal
$\Gamma \vdash \neg t$	proves goal if t in assumptions, fails if $\neg t$ is in assumptions, else as *asm_tac*
$\Gamma \vdash t$	proves goal if $\neg t$ in assumptions, fails if t is in assumptions, else as *asm_tac*

During the search through the assumptions in the last two cases, *check_asm_tac*1 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_tac*1.

SML

```
val strip_asm_tac1 : THM -> TACTIC;
```

Description *strip_asm_tac*1 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*:

$$\begin{array}{l} sat\ (\vdash\ a\ \wedge\ b) \quad \rightarrow\ sat\ (\vdash\ a)\ THEN\ sat\ (\vdash\ b) \\ sat\ (\exists x \bullet a) \quad \rightarrow\ sat\ (a[x'/x] \vdash a[x'/x]) \\ sat\ (\vdash\ a\ \vee\ b)(\{\Gamma\}\ t) \quad \rightarrow\ sat\ (a \vdash a)\ (\{\Gamma\}\ t)\ ;\ sat\ (b \vdash b)\ (\{\Gamma\}\ t) \end{array}$$

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_tac*1 (q.v.) to discharge the goal or to generate additional assumptions.

See Also *STRIP_THM_THEN*, used to implement this function. *check_asm_tac*1, *strip_tac*, *strip_asm_conv*.

SML

```
val strip_asms_tac1 : THM list -> TACTIC;
```

Description *strip_asms_tac*1 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_tac*1 to each of the theorems, except that it will fail only if every application of *strip_asm_tac*1 fails, i.e. if the total effect is null.

See Also *STRIP_THM_THEN*1, used to implement this function. *check_asm_tac*1, *strip_tac*, *strip_asm_conv*.

SML

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

Description *t*1 *AND_OR_T t*2 has the same effect as ((*TRY t*1) *THEN t*2)*ORELSE t*1 but is faster. *AND_OR* is an alias for *AND_OR_T*.

See Also *THEN*, *ORELSE*, *TRY*

SML

```
val ∧_THEN_T1 : (THM -> TACTIC) -> (THM -> TACTIC);
val ∧_THEN1 : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description Similar in effect to $\wedge\_THEN$ 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 $\Gamma \vdash t1 \wedge t2$.

```
∧_THEN1 thmtac (Γ ⊢ t1 ∧ t2) = thmtac (Γ ⊢ t1) AND_OR thmtac (Γ ⊢ t2)
```

See Also $\wedge\_THEN$

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,        ∨_THEN,        SIMPLE_∃_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 -> 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

```
SOME [∀_tac, ∧_tac, ∀_tac]
    is equivalent to
∀_tac AND_OR ∧_tac AND_OR ∀_tac
```

See Also $EVERY\_T$

SML

```
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 *MAP_EVERY_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 *MAP_EVERY*

SML

```
end; (* of signature StripFail *)
```

SML

```
structure StripFail : StripFail = struct
```

SML

```
end; (* of structure StripFail *)
```

4 Theories, Proof Contexts and Consistency

4.1 Specifications

SML

```
signature PreConsisProof = sig
```

Description This signature provide the wherewithal to conduct a consistency proof for a HOL constant specification before introducing the specification, so that the specification can be seen to be consistent and will appear in the theory listing as if no consistency proof had been necessary.

The signature also provides some procedures to incorporate exception handling require when a document is required to create or to recreate a theory, and must therefore first delete things which are not necessarily present, similarly for proof contexts and other functions for manipulating proof contexts.

4.1.1 Doing Consistency Proofs before Axiomatic Descriptions

SML

```
val save_cs_∃_thm : THM -> THM;
```

Description This function may be used to provide to the system a theorem which establishes the consistency of a HOL constant specification about to be introduced.

To avoid getting theory listings in which the definitions of some constants are given using *ConstSpec* I like to do any necessary consistency proofs before introducing the constant specification which needs them. For this to do any good, the automatic consistency prover has to know that I done it.

If used in conjunction with the partial proof context *'savedthm_cs_∃_conv* the theorem will be used to establish the consistency of the specification avoid the need to place a consistency caveat on the stored form of the specification in the theory.

SML

```
(* Proof Context: 'savedthm_cs_∃_proof *)
```

Description 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*

4.1.2 Recursive Definitions without Constructors

SML

```
val CombI_prove_∃_rule : string -> TERM -> THM;
```

Description The provision for recursive definitions to be proven consistent is designed for recursion of types which have constructors, and is not therefore directly applicable to definitions by transfinite recursion in set theory.

However, with a small hack it can be applied to such definitions.

The basic observation is that it will work with a dummy constructor (*CombI* will do) inserted in the place where a real constructor would normally be. There are two problems with this. The first is that you have to put this constructor in your recursive definitions where they look pretty stupid. The second is that only one recursion pattern can be used with any dummy constructor, so if you want more than one recursion principle to be available you have to invent more dummy constructors, and your spec will look really stupid.

The fixes for these problemettes are, firstly to get the existence claim before entering the definition, for the definition with the constructor in it, and then rewrite it to eliminate the constructor and save it so that when you make the definition without the constructor the consistency result is to hand. The second problem is fixed by having a distinct partial proof context for each recursion principle which uses the dummy constructor so that only one recursion principle using the dummy constructor is in scope when the consistency proof is obtained.

A special rule is made available here to simplify the operation of this procedure.

The basic method is as follows (presumes some familiarity with the existing mechanisms):

- prove a recursion theorem corresponding to the pattern of recursion under consideration, but for the argument to the function being defined on which recursion is taking place add the "constructor" *CombI*.
- create a partial proof context just for the recursion principle (see: *add_∃_cd_thms*).
- use *CombI_prove_∃_rule* to obtain and save the required consistency result, supplying as a parameter to it the name of a partial proof context containing just the relevant recursion principle. This function expects to be given the term to be used for the definition except with the constructor *CombI* applied. It will save the result using *save_cs_∃_thm*.
- do the definition in the desired form (i.e. without the dummy constructor).

See Also *add_∃_cd_thm*, *save_cs_∃_thm*

4.1.3 Existential Proofs

SML

val ∃_⇒_***conv*** : *CONV*;

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

SML

val ***prove***_∃_⇒_***conv*** : *CONV*;

Description 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.

4.1.4 Force New Theory

SML

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 ProofPowertheory.

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*

4.1.5 Proof Contexts

SML

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 ProofPowerproof 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*

SML

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*

4.1.6 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 *)
```

4.2 Implementation

SML
```
structure PreConsisProof : PreConsisProof = struct
```

4.3 Partial Primitive Recursive Definitions

This functionality is now to be incorporated into ProofPowerand 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 commited the sources, we have to do this the hard way:

SML

```
val lget x = fst(hd x);
val _ = new_pc "build_predicates";
val _ = set_rw_eqn_cxt ((lget o get_rw_eqn_cxt) "'propositions" @
        (lget o get_rw_eqn_cxt) "'paired_abstractions")
        "build_predicates";
val _ = set_sc_eqn_cxt ((lget o get_sc_eqn_cxt) "'propositions" @
        (lget o get_sc_eqn_cxt) "'paired_abstractions")
        "build_predicates";
val _ = set_st_eqn_cxt ((lget o get_st_eqn_cxt) "'propositions" @
        (lget o get_st_eqn_cxt) "'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:

SML

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

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

SML

```
val full_strip_∧ : TERM -> TERM list = strip_leaves dest_∧;
```

We wish to "mark" some terms, to prevent stripping going too far. We use $pp'TS$ as a marker.

"mark" a term:

SML

```
local
    val ci = ⌜pp'TS:BOOL → BOOL⌝;
in
fun mark (tm:TERM):TERM = mk_app(ci,tm)
end;
```

SML

```
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:

SML

```
val ∃_⇒_conv : CONV = (
    let val ∃_⇒_lemma = prove_rule[]
            ⌜∀p q• (∃f•q ⇒ p f) ⇔ (q ⇒ ∃f•p f)⌝;
    in fn tm =>
        let val (f, b) = dest_simple_∃ tm;
            val (q, pf) = dest_⇒ b;
            val p = mk_simple_λ(f, pf);
            val thm1 = list_∀_elim[p, q] ∃_⇒_lemma;
            val thm2 = conv_rule(LEFT_C(BINDER_C (RIGHT_C β_conv))) thm1;
            val thm3 = conv_rule(RIGHT_C(RIGHT_C(BINDER_C β_conv))) thm2;
            val thm4 = simple_eq_match_conv thm3 tm;
        in thm4
        end
    end
);
```

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.

SML

```
val prove_∃_⇒_conv : CONV = (fn tm =>
    let val thm1 = tac_proof (([], tm),
                REPEAT strip_tac
            THEN conv_tac ∃_⇒_conv
            THEN ⇒_T discard_tac
            THEN conv_tac basic_prove_∃_conv);
        val thm2 = ⇔_t_intro thm1;
    in thm2
    end
);
```

A new value of type $refTHM$ called $saved\_cs\_\exists\_thm$ is used to store consistency results.

SML

```
val saved_cs_∃_thm = ref t_thm;
```

SML

```
fun save_cs_∃_thm thm = (saved_cs_∃_thm := thm; thm);
```

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:

SML

```
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;
```

Store the above in a proof context:

```
SML
val _ = force_new_pc "'prove_∃_⇒_conv";
val _ = set_cs_∃_convs [prove_∃_⇒_conv, ONCE_MAP_C (pure_once_rewrite_conv [let_def] THEN_C
val _ = commit_pc "'prove_∃_⇒_conv";
```

```
SML
val _ = force_new_pc "'savedthm_cs_∃_proof";
val _ = set_cs_∃_convs [prove_∃_⇒_conv, savedthm_cs_∃_conv] "'savedthm_cs_∃_proof";
val _ = set_pr_conv basic_prove_conv "'savedthm_cs_∃_proof";
val _ = set_pr_tac basic_prove_tac "'savedthm_cs_∃_proof";
val _ = commit_pc "'savedthm_cs_∃_proof";
```

```
SML
local   val lthy = get_current_theory_name ();
    val _ = open_theory "combin";
    val CombI_def = get_spec ⌜CombI⌝;
    val _ = open_theory lthy
in
fun CombI_prove_∃_rule pc tm =
    let val tmthm = pc_rule pc basic_prove_∃_rule tm
    in save_cs_∃_thm (rewrite_rule [CombI_def] tmthm)
    end
end;
```

```
SML
end; (* of structure PreConsisProof *)
```

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.

SML

val ***simple_⇒_unify_mp_rule1*** : *THM -> THM -> THM* ;

Description A unifying Modus Ponens rule for an implicative theorem.

Rule

$$\frac{\Gamma 1 \ \vdash \ \forall \ x1 \ ... \bullet \ t1 \ \Rightarrow \ t2; \quad \Gamma 2 \ \vdash \ \forall \ y1 \ ... \bullet \ t1'}{\Gamma 1 \ \cup \ \Gamma 2 \ \vdash \ \forall \ z1 \ ... \bullet \ t2'} \Rightarrow \_unify\_mp\_rule1$$

where $t1'$ is unifiable with $t1$. Type instantiation and substitution is permitted for the x_i in $t1$, the and y_i in $t1'$ and instantiation of the type variables in $t1$ which do not occur in $\Gamma 1$ and those in $t1'$ which do not occur in $\Gamma 2$. $t2'$ is obtained from $t2$ by applying to it the substitution to $t1$ required for its unification. The z_i will be the variables free in t2' which were not previously free either in $t2$ or $t1'$. 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 ‘Γ ⊢ ∀ x1 ... xn • u ⇒ v‘*

SML

val ***⇒_unify_mp_rule1*** : *THM -> THM -> THM* ;

Description A matching Modus Ponens rule for an implicative theorem, supporting paired abstraction.

Rule

$$\frac{\Gamma 1 \ \vdash \ \forall \ x1 \ ... \bullet \ t1 \ \Rightarrow \ t2; \quad \Gamma 2 \ \vdash \ \forall \ y1 \ ... \bullet \ t1'}{\Gamma 1 \ \cup \ \Gamma 2 \ \vdash \ \forall \ z1 \ ... \bullet \ t2'} \Rightarrow \_unify\_mp\_rule1$$

where $t1'$ is unifiable with $t1$. Type instantiation and substitution is permitted for the x_i in $t1$, the and y_i in $t1'$ and instantiation of the type variables in $t1$ which do not occur in $\Gamma 1$ and those in $t1'$ which do not occur in $\Gamma 2$. $t2'$ is obtained from $t2$ by applying to it the substitution to $t1$ required for its unification. The z_i will be the variables free in t2' which were not previously free either in $t2$ or $t1'$. No type instantiation or substitution will occur in the assumptions of either theorem.

Pairs are supported in the bindings.

Errors

7044	*Cannot match ?0 and ?1*
7045	*?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u ⇒ v‘*

SML

```
val unify_forward_chain_rule : THM list -> THM list -> THM list;
val ufc_rule : THM list -> THM list -> THM list;
```

Description This is a rule which uses a list of possibly universally quantified implications and a list of other theorems to infer new theorems, using $\Rightarrow$ *_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 $\Rightarrow$ *_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 $\ulcorner F \urcorner$ as its conclusion, the first such found wil be returned as a singleton list. In order to work well in conjunction with *fc_canon* and *ufc_tac* the theorems returned by $\Rightarrow$ *_unify_mp_rule1* are transformed as follows:

1. Theorems of the form: $\vdash \forall x_1 \ldots \bullet t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow \neg t_k \Rightarrow F$ 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 $\Rightarrow \backslash \neg t_k$.
3. All theorems are universally quantified over all the variables which appear free in their conclusions but not in their assumptions (using *all_∀_intro*).

Note that the use of $\Rightarrow$ *_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.

See Also *unify_forward_chain_tac*, *forward_chain_canon*.

SML

```
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 *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 *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 $\vdash$ *t1* $\Rightarrow$ (*t2* $\Leftrightarrow$ *t3*) for rewriting, one might use *UFC_T1 id_canon rewrite_tac*.

See Also *ufc_tac*, *asm_ufc_tac*, *bc_tac*, *UFC_T*.

SML

```
val UFC_T :
    (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_UFC_T :
    (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_UFC_T :
    (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_UFC_T :
    (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_UFC_⇔_T :
    (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_UFC_⇔_T :
    (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 tacticals with *UFC* are aliases for the corresponding ones with *UNIFY_FORWARD_CHAIN*.)

The description of *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 *ASM* and *ALL* in the name is exactly as for *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.

Examples *ufc_tac* is the same as: *UFC_T strip_asm_tac1*.

To rewrite the goal with the results of the forward inference one could use *UFC_T rewrite_tac*.

See Also *ufc_tac*, *asm_ufc_tac*, *UFC_T*1.

SML

```
val ufc_tac : THM list -> TACTIC;
val all_ufc_tac : THM list -> TACTIC;
val asm_ufc_tac : THM list -> TACTIC;
val all_asm_ufc_tac : THM list -> TACTIC;
val all_ufc_⇔_tac : THM list -> TACTIC;
val all_asm_ufc_⇔_tac : THM list -> TACTIC;
val all_ufc_⇔_rewrite_tac : THM list -> TACTIC;
val all_asm_ufc_⇔_rewrite_tac : THM list -> TACTIC;
```

Description These are tactics which use theorems whose conclusions are implications, or from which implications can be derived using the canonicalisation function *fc_canon*, q.v., to reason forwards from the assumptions of a goal.

The basic step is to take a theorem of the form $\Gamma \vdash t1 \Rightarrow t2$ and an assumption of the form $t1'$ where $t1'$ is unifiable with $t1$ and to deduce the corresponding instance of $t2'$. The new theorem, $\Delta \vdash t2'$ say, may then be stripped into the assumptions.

In the case of *ufc_tac* the implicative theorem is always derived from the list of theorems given as an argument. In the case of *asm_ufc_tac* the assumptions are also used. In all of the tactics the rule *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 ⌜F⌝ then the first such found will be stripped into the assumptions, and all other theorems discarded.

If one of the implications has the form $t1 \Rightarrow t2 \Rightarrow t3$ or $t1 \wedge t2 \Rightarrow t3$ and if assumptions matching $t1$ and $t2$ are available, *ufc_tac* or *asm_ufc_tac* will derive an intermediate implication $t2 \Rightarrow t3$ and *asm_ufc_tac* could then be used to derive $t3$. The variants with *all_* may be used to derive $t3$ directly without generating any intermediate implications in the assumptions. They work like the corresponding tactic without *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 *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 $\forall x \bullet x \in a \Rightarrow x \in b$. 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, *UFC_T1* and *ASM_UFC_T1* may be used to avoid the use of *fc_canon*.

For example, the tactic:

```
asm_ufc_tac[] THEN asm_ufc_tac[]
```

will prove the goal:

```
{p x, ∀x•p x ⇒ q x, ∀x•q x ⇒ r x} r x.
```

The variants with ⇔ in the name use $fc_{\Leftrightarrow c}anon$ instead of 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 $strip_asms_tac1$ and therefore fail if no new assumptions are added (unless the goal is discharged), except the ones whose name includes *rewrite* which attempt the rewrite the conclusion of the goal with the results instead of stripping them into the assumptions.

See Also *bc_tac*, *UFC_T*, *ASM_UFC_T*, *UFC_T1*, *ASM_UFC_T1*.

SML

```
end; (* of signature UnifyForwardChain *)
```

5.2 Implementation

SML

```
structure UnifyForwardChain : UnifyForwardChain = struct
```

In $unify\_ \Rightarrow \_mp_rule$ 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 elegible 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 quanitifiers on the antecedent). If this suceeds 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.

SML

```
end; (* of structure UnifyForwardChain *)
```

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.

SML

```
signature ParseComb = sig
```

Description This signature provides combinators for a simple parser. It is only for small bits of language, so it expects the input to be a list. The output, on the other hand is generated by a collection of constructors supplied as a parameter.

SML

```
exception parse_fail of int;

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

Description This *'a ptree* is an abstract type of parse trees. A *PtreePack* is a record type of constructors for parse trees.

SML

```

val alt_parse: 'a parser list -> 'a parser;
val seq_parse: 'a parser list -> 'a parser;
val opt_parse: 'a parser -> 'a parser;
val star_parse: 'a parser -> 'a parser;
val plus_parse: 'a parser -> 'a parser;
```

SML

```
end; (* of signature ParseComb *)
```

SML

```
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_plit:'a -> 'a ptree,
            mk_plist: 'a ptree list -> 'a ptree,
            mk_palt: int * 'a ptree -> 'a ptree,
            mk_popt: '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_plit = fn l => mk_tree (Ptlit l, []),
    mk_plist = fn ptl => mk_tree (Ptnone, ptl),
    mk_palt = fn (i, pt) => mk_tree (Ptint i, [pt]),
    mk_popt = fn pto => mk_tree (Ptnone, (fn Nil => [] | Value pt => [mk_tree (Ptnone, [pt])])pto )
};

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

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

fun opt_parse p (ptp as {mk_popt, ...}:'a pt_pack) li =
    let val (pt, nli) = p ptp li
    in (mk_popt (Value pt), nli)
    end handle parse_fail i => (mk_popt 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 *)
```

SML

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

SML

SML

end; (∗ *of signature Grammar* ∗)

7 Trawling for Useful Theorems

SML

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.

SML

```
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 assmuptions 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.

SML

```
end; (* of signature Trawling *)
```

SML

```
structure Trawling : Trawling = struct
```

SML

```
end; (* of structure Trawling *)
```

8 For Inductive and Coinductive Definitions

8.1 Some Handy SML functions

The following functions have been moved here from [1].

SML

```
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 _ => ⌜MLf⌝ MLa⌝⌝;

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;
```

SML

```
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 (into $\ulcorner F \urcorner$) false equations between set displays.

SML

```
infix symdiff;

fun x symdiff y = (x diff y) cup (y diff x);

fun dest_enum l =
    (fn DEnumSet els => els
    | D∅ t => []) (dest_term l);

fun enum_eq_sdiff t =
    let val DEq (lhs, rhs) = dest_term t
    in (dest_enum lhs) symdiff (dest_enum rhs)
    end;

fun false_enum_eq_conv t =
    let val (dt :: _) = enum_eq_sdiff t
    in
        tac_proof(([], ⌜⌞ML⌟t⌝ ⇔ F⌝),
            rewrite_tac [sets_ext_clauses]
            THEN ¬_in_tac
```

```
        THEN ∃_tac dt THEN prove_tac[])
    end handle _ => fail_conv t;

val false_enum_eq_tac = conv_tac (MAP_C false_enum_eq_conv);
```

9 INDEX