```
**********************************
* PROLOG CROSS REFERENCE LISTING *
**********************************

IMPRESS Theorem Prover
```

PREDICATE	FILE	CALLED BY
accept/2	FACILE.	go/0
add_conjecture/2	FACILE.	set_theorem/2
add_goal/3	PROVER.	proof_proceed/6
append/3	utility	add_goal/3 establish_input_cond/4 horn/3 fold/3 unfold/3 validate/4
apply_ind_hyp/3	PROVER.	proof_proceed/6
base_establish/3	PROVER.	prove_base_case/3
base_instantiate/4	INST.	base_version/3
base_version/3	INST.	prove_base_case/3
best/3	INDUCT.	order_schemes/2
best/4	INDUCT.	best/3 best/4
bye/0	FACILE.	<user>
call_base/3	REFLEC.	prove_by_induction/2
call_step/3	REFLEC.	prove_by_induction/2
cands_to_schemes/3	INDUCT.	struct_scheme_list/2 cands_to_schemes/3
csensym/2	utility	skolemize/1 add_conjecture/2
choose/2	INDUCT.	struct_induction/2
conj_to_list/2	MISC.	fold/3 unfold/3 validate/4 interp/2 listify/2 set_theorem/2 lookup/2
conjecture/2	THEORM.	lookup/2
copy/2	MISC.	find_type/3 step_check/3 copy_list/2
copy_ar/2	MISC.	ind_hypothesis/3 call_step/3 base_version/3 step_version/4
copy_list/2	MISC.	copy/2 copy_list/2
ctol/3	MISC.	conj_to_list/2 ctol/3

db_call/1	VALIDA.	valid/4 db_call_list/1
db_call_list/1	VALIDA.	interp/2 db_call_list/1
def/2	DEFNS.	validate/4 valid/4
demo/1	FACILE.	demo/1
demo_head/1	PRINT.	demo_tactic/1
demo_tactic/1	PRINT.	base_establish/3 proof_proceed/6 fold_goal/4 unfold_hypothesis/4 apply_ind_hyp/3 establish_hyp/5 step_establish/6 struct_induction/2
detail/1	FACILE.	detail/1
detail_head/1	PRINT.	detail_tactic/2
detail_tactic/2	PRINT.	base_establish/3 ind_hypothesis/3 lemma_resolve/4 struct_induction/2 hor fold/3 unfold/3 lemma_sen/3 fixpt_semant/1 validate/4
divrun/0	THEORM.	<user>
establish_hyp/5	PROVER.	proof_proceed/6
establish_input_cond/4	PROVER.	step_establish/6
evenrun/0	THEORM.	<user>
fact/2	DEFNS.	valid/4
find_base/4	INDUCT.	find_base_element/3 find_base/4
find_base_element/3	INDUCT.	find_type/3
find_type/3	INDUCT.	struct_recursive/3
fixpoint_induction/2	INDUCT.	prove_by_induction/2
fixpt_semant/1	TACTIC.	prove_base_case/3 prove_step_case/3
fold/3	TACTIC.	fold_goal/4
fold_goal/4	PROVER.	proof_proceed/6
fs_list/1	TACTIC.	fixpt_semant/1 fs_list/1
get_theorem/2	FACILE.	accept/2
go/0	FACILE.	<user>
go/1	FACILE.	theorems/0 plusrun/0 timesrun/0 evenru divrun/0 go/1
horn/3	TACTIC.	fold_goal/4 apply_ind_hyp/3 call_step/
ind_hyp_input_cond/2	PROVER.	step_establish/6

ind_hypothesis/3	PROVER.	prove_step_case/3
ind_instantiate/2	INST.	ind_hypothesis/3
inst/5	INST.	instantiate/4 inst/5
instance/4	INST.	base_instantiate/4 step_instantiate/5 ind_instantiate/2
instantiate/4	INST.	instance/4
interp/2	VALIDA.	db_call/1
last_theorem/1	undefined	show/0 <user> redo/0
lemma_sen/3	TACTIC.	step_establish/6 establish_input_cond/4
lemma_resolve/4	PROVER.	step_establish/6
list_to_conJ/2	MISC.	list_to_conJ/2 plist_to_conJ/2 print_tactic/1
listify/2	MISC.	fold_soal/4 call_step/3
lookup/2	FACILE.	so/1 demo/1 detail/1 set_theorem/2
matched_literals/3	MISC.	establish_input_cond/4 valid/4 simple_rec/1
member/2	utility	ind_hypothesis/3 fold_soal/4 establish_input_cond/4 ind_hyp_input_cond/2 recursant/3 score/ choose/2 unfold/3 valid/4
merse/3	MISC.	lemma_resolve/4 merse/3
namewritef/2	PRINT.	theorem_portray/1 print_tactic/1
negate_and_skolemize/3	INST.	prove_base_case/3 prove_step_case/3
o_1/1	FACILE.	so/1 demo/1 detail/1
oops/0	FACILE.	<user>
order_schemes/2	INDUCT.	struct_induction/2 order_schemes/2
performant/3	PROVER.	fold_soal/4 unfold_hypothesis/4
plist_to_conJ/2	PRINT.	theorem_portray/1
plusrun/0	THEORM.	<user>
print_level/1	FACILE.	print_level/1 detail_tactic/2 demo_tactic/1
print_tactic/1	PRINT.	detail_tactic/2
process/1	FACILE.	so/0 so/1 demo/1 detail/1

proof_proceed/6	PROVER.	prove_step_case/3
prototype/3	INST.	step_check/3 base_instantiate/4 step_instantiate/5 ind_instantiate/2
prove/2	PROVER.	process/1 redo/0
prove_base_case/3	PROVER.	prove_by_induction/2
prove_by_induction/2	PROVER.	prove/2
prove_cases/3	INDUCT.	prove_by_induction/2
prove_step_case/3	PROVER.	prove_by_induction/2
pseudo_rec_def/2	DEFNS.	fold_goal/4
rec_def/3	DEFNS.	fold_goal/4 struct_recursive/3 reflective_induction/2 call_base/3 call_step/3 fold/3 unfold/3 unify_fx/1 db_call/1 interp/2 recursive/2 simple_rec/1
recursant/3	PROVER.	fold_goal/4 unfold_hypothesis/4
recursive/2	DEFNS.	unify_fx/1
redo/0	FACILE.	<user>
reflective_induction/2	REFLEC.	prove_by_induction/2
richard/0	THEORM.	<user>
score/3	INDUCT.	struct_scheme/3
select/3	utility	establish_hyp/5 performant/3 struct_scheme/3 call_step/3 horn/3 fold/ unfold/3 fs_list/1 base_instantiate/4 step_instantiate/5
show/0	FACILE.	bye/0
simple_rec/1	DEFNS.	proof_proceed/6 simple_rec/1
skolemize/1	INST.	unfold_hypothesis/4 find_type/3 valid/4 negate_and_skolemize/3 skolemize/1
step_check/3	INDUCT.	struct_recursive/3
step_establish/6	PROVER.	proof_proceed/6 step_establish/6
step_instantiate/5	INST.	step_version/4
step_performant/2	PROVER.	step_establish/6
step_version/4	INST.	prove_step_case/3
struct_candidates/3	INDUCT.	struct_scheme_list/2 struct_candidates/3
struct_induction/2	INDUCT.	prove_by_induction/2

struct_recursive/3	INDUCT.	struct_candidates/3
struct_schema_list/2	INDUCT.	struct_induction/2
struct_scheme/3	INDUCT.	cands_to_schemes/3
theorem_portray/1	PRINT.	demo/1 detail/1 print_tactic/1
theorems/0	THEORM.	<user>
timesrun/0	THEORM.	<user>
ttyprint/1	utility	show/0
unfold/3	TACTIC.	unfold_hypothesis/4 establish_input_cond/4 valid/4
unfold_hypothesis/4	PROVER.	proof_proceed/6
unify_fx/1	TACTIC.	fs_list/1
uninstantiate/2	INST.	establish_hyp/5 performant/3 recursant/3 struct_scheme/3 score/3 instance/4
valid/4	VALIDA.	valid_consequent/1 establish_hyp/5 validate/4
valid_consequent/1	PROVER.	prove/2
validate/4	VALIDA.	base_establish/3 proof_proceed/6 call_step/3 validate/4
var_list/1	MISC.	variable/1 var_list/1
variable/1	MISC.	unify_fx/1
write_proof/1	MISC.	process/1
writef/1	utility	prove_base_case/3 prove_step_case/3 write_proof/1 demo/1 detail/1 print_tactic/1 detail_head/1 demo_head/1
writef/2	utility	write_proof/1 so/1 set_theorem/2 namewritef/2

```
| count of the clauses in IMPRESS

done using Richard's Toolkit

ToolKit version 1 (7 December 82)
Help is available in the followins areas:
      help
      pp
      ixref
      count
      vcheck
Call sive_help(Area) for a list of topics in an Area.
Call sive_help(Area,Topic) for help about a specific topic.

yes
| ?- sive_help(count).
The topics in  count for which help is available are:
     purpose
     files
     data_base
Call sive_help( count,Topic) for help about a specific topic.

yes
| ?- count.
Next file: impres:filin
impres:impres.ops        0 clauses    0 predicates.
impres:prover.          36 clauses   21 predicates.
impres:induct.          28 clauses   17 predicates.
impres:reflec.           3 clauses    3 predicates.
impres:tactic.          11 clauses    7 predicates.
impres:valida.          13 clauses    5 predicates.
impres:inst.            22 clauses   12 predicates.
impres:defns.           19 clauses    6 predicates.
impres:theorm.          15 clauses    7 predicates.
impres:misc.            32 clauses   14 predicates.
impres:facile.          25 clauses   15 predicates.
impres:print.           36 clauses    8 predicates.
top.pl                   3 clauses    3 predicates.
  pres:filin.          245 clauses  120 predicates.
Next file:
Grand total:           245 clauses  120 predicates.

yes
| ?-
```

SUBFILE: IMPRES.SUB @16:28 13-SEP-1982 <005> (25)

```
impres.sub
impres.ccl
filin
impres.ops
prover
induct
reflec
tactic
valida
inst
defns
theorm
misc
facile
print



\\\\\
```

SUBFILE: IMPRES.CCL @15:46 13-SEP-1982 <005> (24)
impres.def
impres.ops
prover.
induct.
reflec.
tactic.
valida.
inst.
defns.
theorm.
misc.
facile.
print.

\\\\\

```
SUBFILE: FILIN. 015:27 13-SEP-1982 <005> (170)
/* FILIN : Impres read in file
                                                Leon
                                                Updated: 13 September 82
*/

:- [
                'impres:impres.ops',     % Operator declarations
                'impres:prover',         % Theorem-proving code
                'impres:induct',         % Induction schemes
                'impres:reflec',         % Reflective induction
                'impres:tactic',         % Fold/Unfold etc
                'impres:valida',         % Validation code
                'impres:inst',           % Instantiation of literals
                'impres:defns',          % Recursive definitions and other fac
                'impres:theorm',         % Standard theorems
                'impres:misc',           % Miscellaneous utilities
                'impres:facile',         % Facilities for input convenience
                'impres:print',          % Pretty output routines
                'top.pl'                 % Lawrence's access to DOPE
   ].

ok :- core_image, writehead, ttynl, reinitialise.

writehead :- display('IMPRESS    DAI  '),
             version_date(Date),
             display(Date), display(')'), ttynl.



\\\\\
```

```
SUBFILE: IMPRES.OPS @15:15 15-JUL-1982 <005> (53)
/* IMPRES.OPS : Operator declarations for the predicate library
                and supportins modules.

                                                        Leon
                                                        Updated: 15 July 82
*/

% General operators

  :- op(950,xfx,[ <==, <-->, <--, --> ]).
  :- op(850,xfy,&).

  :- op(50,fx,[demo,detail,sol]).



\\\\\
```

```
/* PROVER : The Proof component of IMPRESS

                                                        Leon
                                                        Updated: 7 October 82
*/

prove(Theorem,symbolic_evaluation) :-
        valid_consequent(Theorem),
        !.

prove(Theorem,Proof) :-
        prove_by_induction(Theorem,Proof),
        !.

prove(Theorem,Proof) :-
        special_method(Theorem,Proof),
        !.

prove(Theorem,axiom(Name)) :-
        axiom(Theorem,Name),
        !.

prove(Theorem,unable_to_prove).

%% Symbolic evaluation %%

valid_consequent([Head] <-- Body) :-
        valid(Head,[],[],_).

%% Theorem is an axiom

axiom(A <-- B,Name) :-
        list_to_conj(A,Ac),
        list_to_conj(B,Bc),
        pseudo_rec_def(Name,Ac <--> Bc),
        !.

%% INDUCTIVE PROOFS %%

prove_by_induction(Theorem,Proof) :-
        fixpoint_induction(Theorem,Scheme),      % Not currently implemented
        prove_cases(Theorem,Scheme,Proof).

prove_by_induction(Theorem,reflective(B,S)) :-
        reflective_induction(Theorem,Name),
        call_base(Theorem,Name,B),
        call_step(Theorem,Name,S),
        !.

prove_by_induction(Theorem,structural(B,S)) :-
        struct_induction(Theorem,Scheme),
        prove_base_case(Theorem,Scheme,B),
        prove_step_case(Theorem,Scheme,S).


/* BASE CASE */

prove_base_case(Theorem,Scheme,[fs(Base)|Valid]) :-
        base_version(Theorem,Scheme,Base),
        fixpt_semant(Base),
```

```
        negate_and_skolemize(Base,Goal,Assertions),
        base_establish(Goal,Assertions,Valid),
        writef('\nBase case proved').

base_establish(Goal,Assertions,Proof) :-
        demo_tactic(base_establish),
        validate(Goal,Assertions,[],Proof),
        detail_tactic(base_establish,[Goal]).

/* STEP CASE */

prove_step_case(Theorem,Scheme,[fs(Step)|Proof]) :-
        step_version(Theorem,Scheme,Step,Hypothesis),
        fixpt_semant(Step),
        negate_and_skolemize(Step,Goal,Assertions),
        ind_hypothesis(Theorem,Scheme,Ind_hyp),
        proof_proceed(Scheme,Goal,Assertions,Hypothesis,Ind_hyp,Proof),
        !,
        writef('\nStep case proved').

proof_proceed(Scheme,Goal,Assertions,Hypothesis,Ind_hyp,[Pf1,Ind_hyp,Pf2|Pf3]
        simple_rec(Goal),
        simple_rec(Hypothesis),
        !,
        demo_tactic(simply_recursive),
        fold_goal(Goal,NewGoal,[],Pf1),
        apply_ind_hyp(Ind_hyp,NewGoal,NewGoal1),
        establish_hyp(NewGoal1,Assertions,Scheme,Rest,Pf2),
        validate(Rest,Assertions,[],Pf3).

proof_proceed(Scheme,Goal,Assertions,Hypothesis,Ind_hyp,
                        [Hyp,Goal_pf,Perf,Ind_hyp,H_P|Res]) :-
        demo_tactic(s_i_proof_plan),
        unfold_hypothesis(Hypothesis,Hyp_performant,Hyp_recursant,Hyp),
        fold_goal(Goal,Goal_recursant,Goal_performant,Goal_pf),
        apply_ind_hyp(Ind_hyp,Goal_recursant,NewGoal),
        establish_hyp(NewGoal,Hyp_recursant,Scheme,NewGoal1,H_P),
        add_goal(NewGoal1,Goal_performant,NewGoal2),
        step_establish(NewGoal2,Assertions,Scheme,Hyp_performant,
                                                Hyp_recursant,Res).

ind_hypothesis(Theorem,Scheme,A <-- B) :-
        copy_ar(Theorem, A <-- B),
        ind_instantiate(Scheme,Literal),
        member(Literal,B),
        detail_tactic(ind_hypothesis,[A <-- B]).

/* Various open-ended tactics needed in the inductive proof plan */

fold_goal(Goal,Recursant,Performant,fold(Name)) :-
        member(Literal,Goal),
        functor(Literal,Name,_),
        rec_def(Name,_,_),
        !,
        demo_tactic(fold_goal),
        fold([] <-- Goal,Name,[] <-- NewGoal),
        recursant(Literal,NewGoal,Recursant),
        performant(Literal,NewGoal,Performant).

fold_goal(Goal,Recursant,Rest,pseudo_fold(Name)) :-
```

```
        member(Literal,Goal),
        functor(Literal,Name,_),
        pseudo_rec_def(Name,A <--> B),
        !,
        demo_tactic(fold_goal),
        listify(A<--B,Clause),
        horn([] <-- Goal,Clause,[] <-- NewGoal),
        recursant(Literal,NewGoal,Recursant),
        performant(Literal,NewGoal,Rest).

unfold_hypothesis(Hypothesis,Perf,Rec,unfold(Name)) :-
        demo_tactic(unfold_hypothesis),
        functor(Hypothesis,Name,_),
        unfold([Hypothesis]<--[],Name,List<--[]),
        skolemize(List),
        performant(Hypothesis,List,Perf),
        recursant(Hypothesis,List,Rec).

apply_ind_hyp(Ind_hyp,Goal,NewGoal) :-
        demo_tactic(apply_ind_hyp),
        horn([] <-- Goal,Ind_hyp,[] <-- NewGoal), !.

add_goal(Goal,Goal_performant,NewGoal) :-
        append(Goal,Goal_performant,NewGoal).

establish_hyp(Goal,Assertions,scheme(hypothesis(Literal,_,_),_,_),NewGoal,
                                                    Proof ) :-
        uninstantiate(Literal,Blank),
        select(Blank,Goal,NewGoal),
        valid(Blank,Assertions,_,Proof),
        demo_tactic(establish_hyp).

/* Establish the step goal however you can */

step_establish([],_,_,_,_,[]) :- !.

step_establish([Literal|Rest],Assertions,Scheme,Performant,
                                Recursant,[Proof|Rest_proof]) :-
        ind_hyp_input_cond(Literal,Scheme),
        !,
        demo_tactic(input_cond),
        establish_input_cond(Literal,Assertions,Performant,Proof),
        lemma_resolve(Proof,Performant,Rest,Goal),
        step_establish(Goal,Assertions,Scheme,Performant,Recursant,Rest_proo

step_establish([Literal|Rest],Assertions,Scheme,Performant,Recursant,
                                [performant(Literal)|Rest_proof]) :-
        step_performant(Literal,Performant),
        !,
        demo_tactic(step_performant),
        step_establish(Rest,Assertions,Scheme,Performant,Recursant,Rest_proo

step_establish([Literal|Rest],Assertions,Scheme,Performant,
                                Recursant,[Lemma|Rest_pf]) :-
        lemma_gen(Literal,Performant,Lemma),
        lemma_resolve(lemma(Lemma),Performant,Rest,Goal),
        !,
        step_establish(Goal,Assertions,Scheme,Performant,Recursant,Rest_pf).

%%% Development hack
```

```
step_establish(A,B,C,D,E,F) :- step_establish1(A,B,C,D,E,F).

establish_input_cond(Literal,Assertions,_,assert(Literal)) :-
        member(Literal,Assertions).

establish_input_cond(Literal,Assertions,_,unfold(Name)) :-
        member(Predicate,Assertions),
        matched_literals(Literal,Predicate,Name),
        unfold([]<--[Literal],Name,[]<--[Predicate]).

establish_input_cond(Literal,Assertions,Perf,lemma(Lemma)) :-
        member(Predicate,Assertions),
        matched_literals(Literal,Predicate,Name),
        append(Perf,[Predicate],Body),
        lemma_gen(Literal,Body,Lemma).

lemma_resolve(lemma(Lemma),Performant,Rest,Goal) :-
        !,
        merge(Performant,Rest,Goal),
        detail_tactic(lemma_resolve,[[]<--Goal]).

lemma_resolve(_,_,Goal,Goal).

ind_hyp_input_cond(Literal,scheme(_,Input_conds,_)) :-
        member(Literal,Input_conds).

%       ind_hyp_hyp(Literal,[Hyp_recursant]) :-
%               functor(Literal,Name,Arity),
%               functor(Hyp_recursant,Name,Arity).

performant(Literal,Body,Performant) :-
        uninstantiate(Literal,Blank),
        select(Blank,Body,Performant).

recursant(Literal,Body,[Blank]) :-
        uninstantiate(Literal,Blank),
        member(Blank,Body).

step_performant(Literal,[Literal]).
```

```
SUBFILE: INDUCT. @16:15 13-SEP-1982 <005> (763)
/* INDUCT: Induction schemes

                                                        Leon
                                                        Updated: 13 September 82
*/
%% STRUCTURAL INDUCTION %%

struct_induction(Theorem,Scheme) :-
        struct_schema_list(Theorem,Poss),
        order_schemes(Poss,List),
        choose(Scheme,List),
        demo_tactic(struct),
        detail_tactic(struct,[Scheme]).

struct_schema_list(Consequent<--Antecedent,Schemes) :-
        struct_candidates(Antecedent,[],List),
        cands_to_schemes(Consequent<--Antecedent,List,Schemes).

struct_candidates([],List,List) :- !.

struct_candidates([Literal|Rest],Sofar,List) :-
        struct_recursive(Literal,Ars,Type),
        !,
        struct_candidates(Rest,[cand(Literal,Ars,Type)|Sofar],List).

struct_candidates([_|Rest],Sofar,List) :-
        struct_candidates(Rest,Sofar,List).

cands_to_schemes(_,[],[]) :- !.

cands_to_schemes(Theorem,[Cand|RestC],[Scheme|RestS]) :-
        struct_scheme(Theorem,Cand,Scheme),
        cands_to_schemes(Theorem,RestC,RestS).

struct_scheme(Consequent<--Antecedent,cand(Literal,Ars,Type),
                  scheme(hypothesis(Literal,Ars,Type),Input_conds,Score)) :-
        uninstantiate(Literal,Hypothesis),
        select(Hypothesis,Antecedent,Input_conds),
        score(Literal,Consequent,Score),
        !.

score(Hypothesis,Consequent,1) :-
        uninstantiate(Hypothesis,Blank),
        member(Blank,Consequent),
        !.

score(_,_,2).

choose(Scheme,List) :- member(Scheme,List).

order_schemes([],[]) :- !.
order_schemes([S],[S]) :- !.

order_schemes([S|T],[Top|L]) :-
        best([S|T],Top,Rest),
        order_schemes(Rest,L).
```

```
best([S|T],Top,Rest) :- best(T,Top,S,Rest).

best([],Top,Top,Rest) :- !.

best([scheme(H,I,N)|Rest],Top,scheme(H_sofar,I_sofar,M),[scheme(H,I,N)|R]) :-
        N =< M,
        !,
        best(Rest,Top,scheme(H_sofar,I_sofar,M),R).

best([Scheme|Rest],Top,Worse_scheme,[Worse_scheme|R]) :-
        best(Rest,Top,Scheme,R).

struct_recursive(Literal,Pos,Type) :-
        functor(Literal,Fun,_),
        rec_def(Fun,Base,Step <-- _),
        find_type(Base,Pos,Type),
        step_check(Step,Pos,Type),
        !.

find_type(Base,Pos,Type) :-
        copy(Base,Copy),
        skolemize(Copy),
        Copy=..[Fun|Args],
        find_base_element(Args,Pos,Type).

step_check(Step,Pos,Type) :-    % Intended to check whether findins
        copy(Step,Literal),
        ars(Pos,Literal,Ars),           % of base_recursive is correct .
        prototype(step,Type,Ars).

find_base_element(Args,Pos,Type) :-
        find_base(Args,Pos,Type,1).

find_base([],_,_,_) :- !.

find_base([[]|_],Pos,list,Pos).
find_base([0|_],Pos,number,Pos).

find_base([_|Y],Pos,Type,N) :-
        M is N+1,
        find_base(Y,Pos,Type,M).

/* FIXPOINT INDUCTION (not currently implemented) */

prove_cases(_,_,_) :- fail.

%% FIXPOINT INDUCTION %%

fixpoint_induction(Theorem,Scheme) :- fail. % not currently implemented

/*
fixpoint_induction(Theorem,Scheme) :-
        fixpoint_scheme_list(Theorem,Poss),
        order_schemes(Poss,List),
        choose(Scheme,List).

fixpoint_schems_list(_,[],[]) :- !.

fixpoint_schems_list(Theorem,[Literal|Rest],[score(N,Scheme)|L]) :-
        inductable(Literal),
```

```
        !,
        fixpoint_scheme(Theorem,Literal,Scheme),
        score(Scheme,N),
        fixpoint_schema_list(Theorem,Rest,T).

fixpoint_schema_list(Theorem,[_|Rest],Schemes) :-
        fixpoint_schema_list(Theorem,Rest,Schemes).

inductable(Literal) :-
        functor(Literal,Name,_),
        rec_def(Name,_,_).

% Crude first approximation
fixpoint_scheme(Literal,[Fun],Scheme) :- functor(Literal,Fun,_).

prove_cases(_,[],_,_) :- !,
        ttynl,display('Theorem established by fixpoint induction').

prove_cases(Theorem,[Case|C],Scheme,[Proof|P]) :-
        pseudo_fold(Case,Theorem,Version,Hint),
        enter_version(Version,Goal,Assertions),
        prove(Goal,Assertions,Scheme,Hint,Proof),
        !,
        prove_cases(Theorem,C,Scheme,P).

enter_version(Consequent <-- Antecedent,Goal,Assertions,_) :-
        unify_and_skol(Consequent <-- Antecedent,Head <-- Body,_),
        enter(Body,Assertions,assertion),
        enter(Head,Goal,goal).
*/


\\\\\
```

```
SUBFILE: REFLEC. @12:2 7-SEP-1982 <005> (95)
/* REFLEC. :

                                                        Leon
                                                        Updated: 7 September 82
*/

reflective_induction([Head]<--[true],Name) :-
        functor(Head,Name,_),
        rec_def(Name,_,_).

call_base([Goal] <-- [true],Name,Goal) :-
        not not rec_def(Name,Copy,_),
        !.

call_step([Goal] <-- [true],Name,[horn(Goal)|Proof]) :-
        rec_def(Name,_,Clause),
        copy_ar(Goal,Copy),
        listify(Clause,L),
        horn([] <-- [Copy],L,[] <-- NewGoal),
        select(Goal,NewGoal,Rest),
        validate(Rest,[],_,Proof),
        !.



\\\\\
```

```
SUBFILE: TACTIC. @15:47 13-SEP-1982 <005> (431)
/* TACTIC. : Resolutions,Folds,Unfolds,etc

                                                        Leon
                                                        Updated: 27 August 82
*/
%% HORN CLAUSE RESOLUTION %%

horn(Ass <-- Goal,[Head] <-- Body,Ass <-- NewGoal)  :-
        select(Head,Goal,Rest),
        append(Rest,Body,NewGoal),
        detail_tactic(horn,[Ass <-- Goal,[Head] <-- Body,Ass <-- NewGoal]).

%% FOLDING %%

fold(Ass <-- Goal,Name,Ass <-- NewGoal) :-
        rec_def(Name,_,Head <-- Body),
        select(Head,Goal,Rest),
        conj_to_list(Body,B),
        !,
        append(Rest,B,NewGoal),
        detail_tactic(fold,[Ass <-- Goal,Name,Ass <-- NewGoal]).

%% UNFOLDING %%

unfold(Literal <-- Goal,Name,NewLit <-- Goal) :-
        rec_def(Name,_,Head <-- Body),  % Using rec_def as -->
        select(Head,Literal,Rest),
        conj_to_list(Body,B),
        !,
        append(Rest,B,NewLit),
        detail_tactic(unfold,[Literal <-- Goal,Name,NewLit <-- Goal]).

unfold([]<--Goal,Name,[]<--NewGoal) :-
        rec_def(Name,_,Head <-- Body),
        select(Body,Goal,Rest),
        conj_to_list(Head,H),
        !,
        append(Rest,H,NewGoal),
        detail_tactic(unfold,[[] <-- Goal,Name,[] <-- NewGoal]).

unfold([]<--[Goal],Name,[]<--NewGoal) :-
        rec_def(Name,_,Head <-- Body),
        conj_to_list(Body,B),
        member(Goal,B),
        !,
        conj_to_list(Head,NewGoal),
        detail_tactic(unfold,[[] <-- Goal,Name,[] <-- NewGoal]).

%% Lemma Generation %%

lemma_gen(Head,Body,[Head]<--Body) :-
        detail_tactic(lemma,[[Head] <-- Body]).

%% INFERENCE USING THE FIXPOINT SEMANTICS %%

fixpt_semant(Head <-- Body) :-
        fs_list(Body),
```

```
        fs_list(Head),
        detail_tactic(fixpt_semant,[Head <-- Body]).

fs_list(List) :-
        select(Literal,List,Rest),
        unify_fx(Literal),
        !,
        fs_list(Rest).

fs_list(List).

unify_fx(Literal) :-
        recursive(Literal,Name),
        not variable(Literal),
        (rec_def(Name,Literal,_) ; rec_def(Name,_,Literal <-- _)).

/*
fs_list([]) :- !.

fs_list([Literal|T]) :-
        recursive(Literal),
        unify_fx(Literal),
        !,
        fs_list(T).

fs_list([_|T]) :- fs_list(T).

unify_fx(Literal) :-
        Literal=..[Name|Args],
        var_list(Args),
        !.

unify_fx(Literal) :-
        functor(Literal,Name,_),
        rec_def(Name,Literal,_),
        !.

unify_fx(Literal) :-
        functor(Literal,Name,_),
        rec_def(Name,_,Literal <-- _).
*/


\\\\\
```

```
SUBFILE: VALIDA, @13:36 27-AUG-1982 <005> (531)
/* VALIDA : Validation Techniques

                                                        Leon
                                                        Updated: 20 July 82
*/

validate([],_,_,[]) :- !.                               % Vacuous case

validate([Literal|Rest],Assertions,Perf,[Proof|Rest_proof]) :-
        valid(Literal,Assertions,Hints,Proof),
        !,
        detail_tactic(valid,[Literal]),
        validate(Rest,Assertions,Perf,Rest_proof).

validate([Goal|Rest],Assertions,Hint,[def(Name)|Proof]) :-
        functor(Goal,Name,_),
        def(Name, Goal <--> NewGoal),
        detail_tactic(def,[Goal <-- NewGoal]),
        conj_to_list(NewGoal,N),
        append(N,Rest,List),
        validate(List,Assertions,Hint,Proof).

valid(Goal,_,_,true(Goal) ) :-
        skolemize(Goal),
        db_call(Goal),
        !.

valid(Goal,_,_,fact(Goal) ) :-
        functor(Goal,Name,_),
        fact(Name,Goal),
        !.

valid(Goal,Assertions,_,assert(Goal)) :-
        member(Goal,Assertions),
        !.

valid(Goal,Assertions,_,unfold(assert(Literal))) :-
        member(Literal,Assertions),
        matched_literals(Goal,Literal,Name),
        unfold([]<--[Goal],Name,[]<--[Literal]).

valid(Goal,Assertions,_,def(assert(Literal))) :-
        member(Literal,Assertions),
        functor(Literal,Name,_),
        def(Name,Literal <--> Goal).

db_call(Literal) :-
        functor(Literal,Name,_),
        rec_def(Name,_,_),
        interp(Literal,Name).

interp(Literal,Name) :-
        rec_def(Name,Literal,_).

interp(Literal,Name) :-
        rec_def(Name,_,Literal <-- Body),
        !,
```

```
        conj_to_list(Body,List),
        db_call_list(List).

db_call_list([]).

db_call_list([Literal|L]) :-
        db_call(Literal),
        db_call_list(L).

/* Old validation Code

validate(Goal,Assertions,Perf,[Lemma,A|L],L) :-
        good_member(Goal,Assertions,Literal,A),
        true_lemma(Goal <-- Literal & Perf,Lemma).

validate(Goal,Performant,Lemma,[Lemma|P],P) :-
        true_lemma(Goal <-- Performant,Lemma).

clobber(Goal,Goal,_,[]) :- !.
clobber(Goal,Assertion,_,back(Name)) :-
        rec_def_step(Name, Assertion <-- Goal), !.
clobber(Goal,Assertion,[Perf],Lemma) :-
        functor(Goal,Name,_),
        rec_def_step(Name, Assertion <-- Perf & Goal).

true_lemma(A <-- B,Name) :-
        add_conjecture(Name, A <-- B),
        ttynl,display('The following lemma is conjectured'),
        print_clause(Name).

goal_to_conj(Goal,Conj) :-
        atomic(Goal), !, is_clause(Goal,[],List), list_to_conj(List,Conj).
goal_to_conj(Goal,Conj) :- list_to_conj(Goal,Conj).

good_member(Goal,Assertions,Literal,A) :-
        member(A,Assertions),
        atomic(A),
        is_clause(A,[Literal],[]),
        match(Goal,Literal).

good_member(Goal,Assertions,Literal,Literal) :-
        member(Literal,Assertions),
        match(Goal,Literal).

match(Goal,Literal) :-  functor(Goal,Name,Arity), functor(Literal,Name,Arity)

valid(Goal,_,_,base(Goal) ) :-
        functor(Goal,Name,_),
        rec_def(Name,Goal,_),
*/



\\\\\
```

```
SUBFILE: INST. @16:19 13-SEP-1982 <005> (427)
/* INST. : Instantiation of Clauses and Literals

                                                        Leon
                                                        Updated: 13 September 82
*/

base_version(Theorem,Scheme,Version) :-
        copy_ar(Theorem,Blank),
        base_instantiate(Blank,Scheme,Type,Version).

step_version(Theorem,Scheme,Version,Hypothesis) :-
        copy_ar(Theorem,Blank),
        step_instantiate(Blank,Scheme,Hypothesis,Type,Version).

/* INSTANTIATIONS       */

base_instantiate(Head <-- Body,scheme(hypothesis(Literal,Pos,Type),_,_),Type,
                                        Head <-- [NewLit|Rest]) :-
        prototype(base,Type,Strins),
        instance(Literal,Strins,Pos,NewLit),
        select(NewLit,Body,Rest),       % Effectively instantiate Clause Body
        !.                              % Put Hypothesis at front of body

step_instantiate(Head <-- Body,scheme(hypothesis(Literal,Pos,Type),_,_),NewLi
                                        Type,Head <-- [NewLit|Rest]) :-
        prototype(step,Type,Strins),
        instance(Literal,Strins,Pos,NewLit),
        select(NewLit,Body,Rest),       % Same strategy as for base_case
        !.

ind_instantiate(scheme(hypothesis(Literal,Pos,Type),_,_),NewLit) :-
        prototype(ind,Type,Strins),
        instance(Literal,Strins,Pos,NewLit).

instance(Literal,Strins,Pos,NewLit) :-
        uninstantiate(Literal,Blank),
        instantiate(Blank,NewLit,Strins,Pos).

instantiate(Literal,NewLit,Strins,Pos) :-
        Literal=..[Func|Args],
        inst(Args,Newargs,Strins,Pos,1),
        NewLit=..[Func|Newargs].

inst([],_,_,_,_) :- !, display(' Failure in instantiate').

inst([H|T],[Strins|T],Strins,Pos,Pos) :-
        !,
        H = Strins.     % Instantiate H to Strins

inst([H|T],[H|NewT],Strins,Pos,Count) :-
        NewCount is Count + 1,
        inst(T,NewT,Strins,Pos,NewCount).

uninstantiate(Literal,Blank) :-
        functor(Literal,Name,Arity), functor(Blank,Name,Arity).

negate_and_skolemize(Head <-- Body,Head,Body) :-
```

```
        skolemize(Head),
        skolemize(Body).

skolemize(X) :- var(X),!,csensym(var,Name),X = Name.
skolemize([]) :- !.
skolemize([H|T]) :- !,skolemize(H),skolemize(T).
skolemize(X) :- atomic(X),!.
skolemize(X) :- X=..[_|Args],skolemize(Args).

/* PROTOTYPES */

prototype(base,list,[]).
prototype(step,list,[A|list]).

prototype(base,number,0).
prototype(step,number,s(number)).

prototype(ind,Type,Type).



\\\\\
```

```
SUBFILE: DEFNS. @15:35 10-SEP-1982 <005> (375)
/* Defns : Various definitions,etc. used as Theorems

                                                    Leon
                                                    Updated: 10 September 82
*/

/* RECURSIVE DEFINITIONS */

rec_def(append, append([],X,X), append([H|X],Y,[H|Z]) <-- append(X,Y,Z) ).

rec_def(length, length([],0), length([H|X],s(N)) <-- length(X,N) ).

rec_def(is_list, is_list([]), is_list([H|X]) <-- is_list(X) ).

rec_def(plus, plus(0,X,X), plus(s(X),Y,s(Z)) <-- plus(X,Y,Z) ).

rec_def(less, less(0,s(N)), less(s(X),s(Y)) <-- less(X,Y) ).

% Recursants are currently before performants to facilitate symbolic evaluati

rec_def(times, times(0,Y,0), times(s(X),Y,Z) <-- times(X,Y,W) & plus(W,Y,Z) ).

rec_def(divides, divides(X,0), divides(X,Z) <-- divides(X,Y) & plus(X,Y,Z)).

rec_def(even, even(0), even(s(s(X))) <-- even(X) ).

rec_def(isolate, isolate([],X=Exp,Exp),  isolate([N|Posn],Lhs=Rhs,Ans)
   <-- isolate(Posn,NewLhs=NewRhs,Ans) & isolax(N,Lhs=Rhs,NewLhs=NewRhs) ).

rec_def(position, position(X,X=Exp,[]),  position(X,Lhs=Rhs,[N|Posn])
   <-- position(X,NewLhs=NewRhs,Posn) & isolax(N,Lhs=Rhs,NewLhs=NewRhs) ).

/* DEFINITIONS */

def(solve, solve(Lhs=Rhs,X,X=Ans) <--> free_of(X,Ans) & equiv(Lhs=Rhs,X=Ans) )

def(free_of, free_of(X,Ans) <--> single_occ(X,X=Ans) ).

def(positive, positive(X) <--> less(0,X) ).

/* FACTS */

fact(equiv, equiv(X,X) ).

/* PSEUDO RECURSIVE DEFINITIONS */

pseudo_rec_def(solve,    solve(Lhs=Rhs,X,Soln)
        <--> equiv(Lhs=Rhs,NewLhs=NewRhs) & solve(NewLhs=NewRhs,X,Soln) ).

pseudo_rec_def(single_occ,  single_occ(X,Eqn)
        <--> isolax(N,Eqn,NewEqn) & single_occ(NewEqn) ).

/* Literal type checking */

recursive(Literal,Name) :-
        functor(Literal,Name,_),
        rec_def(Name,_,_).
```

```
simple_rec([Literal]) :- !, simple_rec(Literal).

simple_rec(Literal) :-
        functor(Literal,Name,_),
        rec_def(Name,_, A <-- B),
        matched_literals(A,B,Name),
        !.



\\\\\
```

```
SUBFILE: THEORM, @15:35 13-SEP-1982 <005> (228)
/* THEORM: Theorems proved by IMPRESS

                                                        Leon
                                                        Updated: 13 September 82
*/

[       conjecture(cade,     % Version once proved by IMPRESS
%               length(Z,N+M) <-- length(X,N) & length(Y,M) & append(X,Y,Z) )

conjecture(cade_paper,
length(Z,N) <-- length(X,L) & length(Y,M) & append(X,Y,Z) & plus(L,M,N) ),

conjecture(cade_talk, less(X,Z) <-- plus(X,Y,Z) & positive(Y) ),

conjecture(list, is_list(Z) <-- is_list(X) & is_list(Y) & append(X,Y,Z) ),

conjecture(assoc_append, append(X,Y,Z) <--
        append(U,V,X) & append(V,Y,W) & append(U,W,Z) ),

conjecture(trans_less, less(X,Z) <-- less(X,Y) & less(Y,Z) ),

conjecture(isolate_correct,
        solve(Lhs=Rhs,X,X=Ans) <-- position(X,Lhs=Rhs,Posn)
                & isolate(Posn,Lhs=Rhs,Ans) & single_occ(X,Lhs=Rhs) ),

conjecture(p0, plus(0,X,X) <-- true ),
conjecture(p1, plus(X,0,X) <-- true ),

theorems :- go [cade_paper,cade_talk,list,assoc_append,trans_less,
                        isolate_correct,p0,p1].

richard :- consult('impres:impres.tst').

plusrun :- go [p0,p1,p2,p3,p4,p5,p6].
timesrun :- go [t0,t1,t2,t3,t4,t5,t6,t7].
evenrun :- go [e0,e1,e2,e3].
divrun :- go [d0,d1,d2,d3].



\\\\\
```

```
SUBFILE: MISC. @16:22 13-SEP-1982 <005> (349)
/* MISC. : Various utilities for IMPRESS

                                                        Leon
                                                        Updated: 13 September 82
*/

/*      UTILITIES       */

write_proof(unable_to_prove) :-
        writef('\nNo suitable proof method currently available\n').
write_proof(symbolic_evaluation) :-
        writef('\nTheorem proved by symbolic evaluation\n').
write_proof(Proof) :-
        functor(Proof,Name,_),
        writef('\nTheorem proved by %t induction\n',[Name]).

variable(Literal) :- Literal =..[Fun|Args],var_list(Args).
var_list([]) :- !.
var_list([H|T]) :- var(H), !, var_list(T).

%% Converting between lists and conjunctions and vice versa

conj_to_list([],[]) :- !.
conj_to_list([X|Y],[X|Y]) :- !.

conj_to_list(Conj,List) :-
        !,
        ctol(Conj,List,[]).

ctol(A & B,L1,L3) :-
        !,
        ctol(A,L1,L2),
        ctol(B,L2,L3).

ctol(A,[A|L],L).

listify(A <-- B,C <-- D) :- conj_to_list(A,C),conj_to_list(B,D).

list_to_conj([X],X) :- !.
list_to_conj([H|T],H & C) :- !, list_to_conj(T,C).
list_to_conj(X,X).

%% Copying clauses and literals

copy(X,X) :- atomic(X), !.
copy(X,Y) :- var(X), !.
copy(X,Y) :- X=..[Fun|Args],copy_list(Args,NewArgs),Y=..[Fun|NewArgs].

copy_list([],[]) :- !.
copy_list([H|T],[NewH|NewT]) :- copy(H,NewH), copy_list(T,NewT).

copy_ar(Term,Newterm) :-
        asserta(cop(Term)),
        retract(cop(Newterm)).

%       skol_copy(Clause,Copy) :- copy_ar(Clause,Copy), skolemize(Copy).
```

```
merge([],X,X) :- !.
merge([X],Y,Z) :- !,merge(X,Y,Z).
merge(X,[],[X]) :- !.
merge(X,[X|Y],[X|Y]) :- !.
merge(X,[H|Y],[H|Z]) :- merge(X,Y,Z).

matched_literals(L1,L2,Name) :-
        functor(L1,Name,Arity),
        functor(L2,Name,Arity).

/* Old code

variable(H) :- var(H), !.
variable(H) :- atomic(H), !, fail.
variable(Literal) :- Literal=..[_|Args], !, var_list(Args).
*/



\\\\\
```

```
SUBFILE: FACILE. @12:38 7-SEP-1982 <005> (448)
/* FACILE : Some conveniences for IMPRESS

                                                        Lawrence & Leon
                                                        Updated: 7 September 82
*/

I% Run Interpreted %%


                            % Go from the terminal

go   :- ttynl, display('Prove: '), ttyflush,
        accept(Input,Theorem),
        process(Theorem).

go [] :- !.

go [A|B] :- !, writef('\nTrying to prove %t',[A]), go(A), go(B).

go Name :-
        o_l(low), !,
        lookup(Name,Theorem),
        process(Theorem).

demo [] :- !.

demo [A|B] :- !, demo(A), demo(B).

demo Name :-
        o_l(demo), !,
        lookup(Name,Theorem),
        writef('\nTrying to prove'),
        theorem_portray(Theorem),
        process(Theorem).

detail [] :- !.

detail [A|B] :- !, detail(A), detail(B).

detail Name :-
        o_l(detail), !,
        lookup(Name,Theorem),
        writef('\nTrying to prove'),
        theorem_portray(Theorem),
        process(Theorem).

process(Theorem) :-
        asserta( last_theorem(Theorem) ),
        prove(Theorem,Proof),   % In future theoremify(Theorem)
        write_proof(Proof),
        !.

accept(Input,Theorem) :- read(Input), set_theorem(Input,Theorem).

set_theorem(Input,Theorem) :- atomic(Input), lookup(Input,Theorem).

set_theorem(A <-- B,Head <-- Body) :-
```

```
        conj_to_list(A,Head),
        conj_to_list(B,Body),
        add_conjecture(Name, A <-- B ),
        writef('\nAttempting to prove %t',[Name]).

lookup(Input,Head <-- Body) :-
        conjecture(Input, A <-- B ),
        conj_to_list(A,Head),
        conj_to_list(B,Body),
        !.

lookup(Input,_)  :-
        ttynl,display('Sorry! No theorem currently of that name'),ttynl,
        fail.

                        % Show all the theorems

show :- ttynl, display('Theorems:'), ttynl, ttynl,
        last_theorem(Theorem),
        ttyprint(Theorem), ttynl,
        fail.

show.


                        % Redo last theorem

redo :- call( last_theorem(Theorem) ),
        !,
        prove(Theorem,_).


                        % Remove record of last theorem

oops :- retract( last_theorem(_) ),
        display('(Ok, I''ve forgotten it!)'), ttynl,
        !.


                        % Leave Impress, showing all the theorems

bye  :- log,
        show, ttynl,
        display('Goodbye'), ttynl,
        halt.


                        % Print levels
print_level(low).

o_l(X) :- retract(print_level(Y)), assert(print_level(X)).

print_level(demo) :- print_level(detail).

                        % Database access

add_conjecture(Name,Conj) :-
        csensym(theorem,Name),
        assert(conjecture(Name,Conj)).
```

\\\\

```
SUBFILE: PRINT. @12:7 7-SEP-1982 <005> (548)
/* PRINT : Pretty printer for clauses

                                                        Leon
                                                        Updated: 7 September 82
*/

                        % Print out a clause

theorem_portray(Head <-- Body) :-
        !,
        plist_to_conj(Body,R),
        plist_to_conj(Head,L),
        namewritef('  %t <-- %t',[L,R]).

theorem_portray(X) :- tab(2), write(X).

        namewritef(Format, Terms) :-
                numbervars(Terms, 0, _),
                writef(Format, Terms),
                fail.           . %  undo numbervars bindings
        namewritef(_, _).

plist_to_conj([],' ') :- !.
plist_to_conj([true],'  ') :- !.
plist_to_conj(List,ConJ) :- list_to_conj(List,ConJ).

detail_tactic(Tactic,Arslist) :-
        print_level(detail),
        !,
        detail_head(Tactic),
        print_tactic(Arslist).

detail_tactic(_,_).

demo_tactic(Tactic) :- print_level(demo), !, demo_head(Tactic).
demo_tactic(_).

print_tactic([Clause,output_hack,_]) :- !,
        theorem_portray(Clause).

print_tactic([Goal,Clause,Newsoal]) :-
        theorem_portray(Goal), print(' with '),
        theorem_portray(Clause), writef('   sives\n'),
        theorem_portray(Newsoal).

print_tactic([scheme(hypothesis(Literal,Ars,Type),Input_conds,_)]) :- !,
        list_to_conj(Input_conds,ConJ),
        namewritef('\n\nThe prosram hypothesis is %t
  which sussests induction on arsument %t of %t type.
The input_conditions are %t',[Literal,Ars,Type,ConJ]).

print_tactic([Goal]) :- theorem_portray(Goal).

detail_head(horn) :- writef('\n\nResolvins ').
detail_head(fold) :- writef('\n\nFoldins ').
detail_head(unfold) :- writef('\n\nUnfoldins ').
detail_head(fixpt_semant) :- writef('\n\nApplyins fixpoint semantics sives\n\n'
```

```
detail_head(lemma) :- writef('\n\nHypothesising the following lemma\n\n').
detail_head(struct).
detail_head(base_establish).
detail_head(ind_hypothesis) :- writef('\n\nThe induction hypothesis is\n\n').
detail_head(lemma_resolve) :-
        writef('\n\nResolving with the hypothesised lemma gives\n\n').
detail_head(valid) :- writef('\n\nValidating literal').
detail_head(def) :- writef('\n\nApplying the definition').

demo_head(struct) :- writef('\n\nStructural induction scheme chosen').
demo_head(base_establish) :- writef('\n\nEstablishing base goal').
demo_head(simply_recursive) :- writef('\n\nSimply recursive proof plan chosen'
demo_head(s_i_proof_plan) :- writef('\n\nStructural induction proof plan chose
demo_head(fold_goal) :- writef('\n\nFolding step output condition').
demo_head(unfold_hypothesis) :- writef('\n\nUnfolding program hypothesis').
demo_head(apply_ind_hyp) :- writef('\n\nApplying induction hypothesis').
demo_head(establish_hyp) :- writef('\n\nEstablishing hypothesis performant').
demo_head(step_performant) :- writef('\n\nEstablishing step output performant'
demo_head(input_cond) :- writef('\n\nEstablishing step input condition').
```

\\\\\

```
/* TWOOCC : The solving equations with 2 occurrences of the unknown theorem

                                                        Leon
                                                        Updated: 7 October 82
*/

conjecture(two_occ,
solve(L=R,X,Soln) <-- occ(X,L=R,s(s(0))) & collect(L=R,X,NewL=NewR)
        & position(X,NewL=NewR,P) & isolate(P,NewL=NewR,Soln)    ),

special_method(Theorem,special(two_occ)) :-
        negate_and_skolemize(Theorem,Goal,Assertions),
        fold_goal(Goal,Goal_rec,Goal_perf,_),
        conjecture(isolate_correct,C),
        listify(C,L),
        horn([]<--Goal_rec,L,[]<--G),
        append(G,Goal_perf,NewGoal),
        two_occ_establish(NewGoal,Assertions).

two_occ_establish(Goal,Assertions) :-
        blank_copy(Assertions,L),
        step_establish(Goal,Assertions,scheme(_,L,_),[],[],Pf).


step_establish1([Literal|Rest],Assertions,Scheme,P,R,[def(Name)|Pf]) :-
        functor(Literal,Name,_),
        def(Name,Literal <-- Body),
        conj_to_list(Body,L),
        append(L,Rest,Goal),
        step_establish(Goal,Assertions,Scheme,[],[],Pf).

conjecture(isolate_correct,
        solve(Lhs=Rhs,X,Soln) <-- position(X,Lhs=Rhs,Posn)
                & isolate(Posn,Lhs=Rhs,Soln) & single_occ(X,Lhs=Rhs) ).

z :- spy two_occ_establish, go two_occ.
```

```
/*  NEW : Test code for new theorems

                                                        Leon
                                                        Updated: 7 October 82
*/

def(single_occ, single_occ(X,Eqn) <-- occ(X,Eqn,s(0))    ),

def( equiv, equiv(Eqn,New) <-- isolate(Eqn,X,New)        ),

rec_def(occ, occ(X,Eqn,0) <-- free_of(X,Eqn),
occ(X,Eqn,s(N)) <-- collect(Eqn,X,New) & occ(X,New,N)    ),

def( equiv, equiv(Eqn,New) <-- collect(Eqn,X,New)        ),

def( equiv, equiv(Eqn,New) <-- attract(Eqn,X,New)        ),

conjecture(collect,  solve(Eqn,X,Soln) <--
```

```
SUBFILE: CLAUSE. @7:26 13-FEB-1982 <005> (399)
/* CLAUSE : Interface to database of clauses

                                                        Leon
                                                        Updated: 13 February 82
*/


                        % Access a named clause


set_rec_def(Name,Consequent,Antecedent) :-
        is_clause(Name,Consequent,Antecedent),
        !.

set_rec_def(Name,Consequent,Antecedent) :-
        add_rec_def(Name),
        is_clause(Name,Consequent,Antecedent).

pseudo_rec_def(Name, Clause_name) :-
        def(Name, A <--> B),
        uninstantiate(A,Blank),
        conj_to_list(B,Body),
        member(Blank,Body),
        !,
        add_conjecture(Clause_name, A <-- B).

                        % Add various types of clause to the database

enter([],[],_) :- !.

enter([Literal|T],[Name|NameT],assertion) :-
        add_assertion(Name,Literal),
        enter(T,NameT,assertion).

enter(Goal,Name,goal) :-
        add_goal(Name,Goal).

add_assertion(Name,Assertion) :-
        csensym(assertion,Name),
        add_fact( is_clause(Name,[Assertion],[]) ),
        add_fact( assertion(Name) ).

add_ind_hyp(Name, Consequent <-- Antecedent) :-
        csensym(induction_hypothesis,Name),
        add_fact( is_clause(Name,Consequent,Antecedent) ),
        add_fact( induction_hypothesis(Name) ).

add_goal(Name,Goal) :-
        csensym(goal,Name),
        add_fact( is_clause(Name,[],Goal) ),
        add_fact( goal(Name) ).

add_rec_def(Name) :-
        rec_def_step(Name, Head <-- Body ),
        conj_to_list(Body,L),
        add_fact( is_clause(Name,[Head],L) ),
        add_fact( rec_def(Name) ).
```

```
add_conjecture(Name, Head <-- Body ) :-
        csensym(conjecture,Name),
        conj_to_list(Body,L),
        add_fact( is_clause(Name,[Head],L) ),
        add_fact( conjecture(Name) ).

theoremify(Name) :-
        retract( conjecture(Name) ),
        !,
        assert( theorem(Name) ).


                        % Basic Clause handling

is_clause(Name,Consequent,Antecedent) :-
        recorded(Name, is_clause(Consequent,Antecedent), _),
        !.

add_fact( is_clause(Name,Consequent,Antecedent) ) :-
        !,
        recorda(Name, is_clause(Consequent,Antecedent), _).

add_fact(X) :- assert(X).

delete_clause(Name)
     :- recorded(Name, is_clause(_,_), ID),
        erase(ID),
        fail.

delete_clause(_).
```

\\\\\

```
*isolate
Clauses for Isolation soundness proof
Alan Bundy 5.2.80
use with checker etc */

/* Hypotheses */

clause(hyp1,[insleocc(x,a=b)],[]).

clause(hyp2,[position(x,a,[])],[]).

clause(hyp3,[isolate([],a=b,ans)],[]).


/* Goal */

clause(goal,[],[solve(x,A=b,ans)]).


/* Axioms */

clause(defn_solve,[solve(X,Eqn,Ans)],[free_of(X,Rhs),
                                      Ans=(X=Rhs),
                                      equiv(Ans,Eqn)]).

clause(free_of,[free_of(X,B)],[insleocc(X,A=B),
                                      position(X,A,Posn)]).

clause(ident,[equiv(A,B)], [A=B]).


/* Definitions */

defn(position(X,A,Posn), case([
    triple(Posn=[],[],X=A),
    triple(Posn=[N|Posn1],[_,_],_)])).

defn(isolate(Posn,Eqn,Ans),case([
        triple(Posn=[],[],Ans=Eqn),
        triple(Posn=[N|Posn1],[isolax(N,Eqn,New),isolate(Posn1,New,Ans1)],
                Ans=Ans1)])).
```

Leon Sterling

Note 64

Alan Bundy
20.2.80

PROVING THE CORRECTNESS OF ISOLATE

As a first step to the automatic derivation of PRESS methods (as advertised in the latest PRESS grant) I thought I would try to prove the correctness of isolate, namely the theorem:

```
solve(X,L=R,Ans) <-
        single_occ(X,L=R) & position(L,Posn)=X & isolate(Posn,Eqn)=Ans
```

So far I have made the following progress:

- Using a simplified version of the new definition of Isolation described in note 63 I have found a proof by hand.
- I have built an automatic proof checker and begun checking my hand proof.

My aim is to gradually automate the machine proof. To remove the simplifications from the definitions of Isolate and to try proving the correctness of other PRESS methods.

Axioms

The proof is by induction on the structure of 'posn'. We need the axioms/lemmas:

1. solve(X,Eqn,X=Rhs) <->
        free_of(X,Rhs) & equiv(Eqn,X=Rhs)
2. free_of(X,B) <-
        single_occ(X,A=B) & position(A,Posn)=X
3. equiv(A,A)
4. equiv(A,C) <- equiv(A,B) & equiv(B,C)
5. position(A,[])=A
6. posn(ars(N,Exp),Posn)=Term <- posn(Exp,[N|Posn])=Term

```
7. equiv(Old,New) <- isolax(N,Old,New)

8. singleocc(X,C=D) <- singleocc(X,L=R) &
                       isolax(N,L=R,C=D) &
                       position(L,[N|Posn])

9. isolate([],Ean) = Ean

10. isolax(N,L=R,ars(N,L)=cons(L,R,N)) &
    isolate(Posn,ars(N,L)=cons(L,R,N)) = Ans
            <- isolate([N|Posn],L=R) = Ans
```

Basis

The negation of the basis case provides the additional clauses:

```
(i) singleocc(x,l=r).

(ii) position(l=r,[]) = x

(iii) isolate([],l=r) = (x=ans)

(iv) <- solve(x,l=r,x=ans).
```

The proof of the basis case is as follows:

```
Resolving (iv) and 1
(a).    <- free_of(x,ans) & equiv(l=r,x=ans).

Resolving 2 with (a) produces:
(b)     <- singleocc(x,A=ans) & position(A,Posn)=x &
                              & equiv(l=r,x=ans).

Resolving (b) with 5 produces:
(c)     <- singleocc(x,x=ans) & equiv(l=r,x=ans).

Paramodulating (iii) into 9 gives:
(d)     (x=ans) = (l=r)

k
Paramodulating (d) into (c) (twice) gives:
(e)     <- singleocc(x,l=r) & equiv(l=r,l=r)

Resolving (e) with 3 gives
(f)     <- singleocc(x,l=r)

Resolving (f) with (i) produces the empty clause
```

(s) <- QED

Step

The negation of the step case provides the following additional clauses:

(i) solve(X,L=R,Ans) <-
single occ(X,L=R) & position(L,Posn)=X
& isolate(Posn,L=R)=Ans

(ii) single occ(x,l=r)

(iii) position(l,[n|Posn])=x

(iv) isolate([n|Posn],l=r) = (x=ans)

(v) <- solve(x,l=r,x=ans)

The proof of the step case is as follows:

Resolving (v) with 1 produces
(a) <- free_of(x,ans) & equiv(l=r, x=ans)

Resolving (a) with 4 produces
(b) <- free_of(x,ans) & equiv(l=r, B) & equiv(B, x=ans)

Resolving (b) with 7 produces
(c) <- free_of(x,ans) & isolax(N,l=r, B) & equiv(B, x=ans)

Resolving (c) with 1 produces
(d) <- solve(x,B,x=ans) & isolax(N, l=r, B)

Resolving (d) with (i) produces
(e) <- singleocc(x,C=D) & position(C,posn)=x &
isolate(posn,C=D) = (x=ans) & isolax(N, l=r, C=D)

Resolving (e) with 8 and mersing produces
(f) <- singleocc(x,l=r) & position(l,[N|posn])=x &
position(C,posn)=x &
isolate(posn,C=D) = (x=ans) & isolax(N, l=r, C=D)

Resolving (f) with (ii) produces
(g) <- position(l,[N|posn])=x & position(C,posn)=x &
isolate(posn,C=D) = (x=ans) & isolax(N,l=r,C=D)

Resolving (g) with 10 produces
(h) <- position(l,[N|posn])=x & position(ars(N,1),posn)=x &
isolate([N|posn],l=r) = (x=ans)

Resolving (h) with (iv) produces
(j) <- position(l,[n|posn])=x & position(ars(n,1),posn)=x

Resolving (j) with 6 and mersing produces
(k) <- position(l,[n|posn])=x

Resolving (k) with (iii) produces the empty clause
(l) <- QED

Another Proof of the Correctness of Isolate

Note 86

Jean Ctet:ine
6 Jenuery :99?

s a first step in buildins IMPRCSS, r proof is siven of the correctness of solate. This is an improved version of note 64, and it is expected that ltimstely IMPRCSS will produce a verw similer proof to the hend one given ere.

Axioms

The sxioms used in the proof sre siven.

```
ax(defn_solve,[solve(Ean,X,X=Ans)],[equiv(Ean,X=Ans),free_of(X,Ans)]).

ax(solve_back1,[equiv(Ean,X=Ans)],[solve(Ean,X,X=Ans)]).

ax(solve_back2,[free_of(X,Ans)],[solve(Ean,X,X=Ans)]).

ax(free,[free_of(X,B)],[single_occ(X,X=B)]).

ax(single,[single_occ(X,Exp)],[single_occ(X,New),isolax(N,New,Exp)]).

ax(single2,[single_occ(X,X=Ans)],
[single_occ(X,Ean),position(X,Ean,Posn),isolate(Posn,Ean,Ans)]).

ax(posn,[position(X,Ean,Posn)],[isolax(N,Ean1,Ean),
position(X,Ean1,[N|Posn])]).

ax(equivx,[equiv(Old,New)],[isolax(N,Old,New)]).

ax(equiv,[equiv(Old,X=Ans)],[isolate([],Old,Ans)]).

ax(equiv_ex,[equiv(A,B)],[equiv(A,C),equiv(C,B)]),

ax(isolate,[isolax(N,Old,New)],[isolate([N|Posn],Old,Ans)]),

ax(isolate2,[isolate(Posn,New,Ans)],[isolate([N|Posn],Old,Ans)]).
```

ne sxioms will be referred to by their name, which is the first srsument of redicate sx.

These sxioms will be psrt of a lsrse dutsbsse of facts that hopefully the schine will derive. At presest some swkwerdnesses sre presest. For exmple xioms defn_solve, solve_back1, and solve_back2 sre resl}y one iff theorem bout the solve clsuse. Simlerly the lest two sxioms sre psrt of sn iff

eorem about isolate. Also, exinm equiv is something of a patch and is used
the proof of the base case when the fact that isolate is a function rather
an a predicate.

The base case

First the proof of the base case is given. This is the theorem
lve(Ean,X,X=Ans) : single_occ(X,Ean),position(X,Fen,F),isolate(F,
n,Ans)].

Negating this theorem, and putting it into appropriate clausal form for the
tabase yields the following. Note variables have been instantiated to
emonic skolem constants.

```
ax(basehyp1,[single_occ(x,lbase=rbase)],[]).

    asehyp2,[position(x,lbase=rbase,[])],[]).

ax(basehyp3,[isolate([],lbase=rbase,ans)],[]).

ax(basegoal,[],[solve(x,lbase=rbase,ans)]).
```

The proof proceeds by linear resolution.

1. Resolve basegoal and defn. solve to form new clause

   <- equiv(lbase=rbase,x=ans) & free_of(x,ans)

2. Resolve with free to form new clause

   <- single_occ(x,x=ans) & equiv(lbase=rbase,x=ans)

3. Resolve with single2 to form new clause

   <- single_occ(x,Ean) & position(x,Ean,Posn) & isolate(Posn,Ean,ans)
   & equiv(lbase=rbase,x=ans)

4. Resolve with equiv to form new clause

   <- isolate(Posn,lbase=rbase,ans) & single_occr(x,Ean) &
   position(x,Ean,Posn) & isolate(Posn,Ean,ans)

5. Merge two occurrences of isolate(Posn,lbase=rbase,ans) to form

   <- single_occ(x,lbase=rbase) & position(x,lbase=rbase,Posn) &
   isolate(Posn,lbase=rbase,ans)

6. Resolve with basehyp1 to form

<- position(x,lbase=rbase,Posn) & isolate(Posn,lbase=rbase,ans)

7. Resolve with basehyp2 to form

<- isolate([],lbase=rbase,ans)

8. Resolve with basehyp3 to yield the empty clause, a contradiction, and the result is proved.

The step case

Now consider the step case. In this case the following clauses are added.

ax(stephyp1,[single_occ(x,lstep=rstep)],[]).

ax(stephyp2,[position(x,lstep=rstep,[n|posn])],[]).

ax(stephyp3,[isolate([n|posn],lstep=rstep,ans)],[]).

ax(stephyp4,[solve(X,Can,Ans)],[single_occ(X,Can),
sition(X,Can,Posn),isolate(Posn,Can,Ans)]).

ax(stepgoal,[],[solve(x,lstep=rstep,ans)]).

The proof proceeds as follows.

1. Resolve stepgoal and defn_solve to form new clause

<- equiv(lstep=rstep,x=ans) & free_of(x,ans)

2. Resolve with equiv_ex to form

<- equiv(lstep=rstep,Can) & equiv(Can,x=ans) & free_of(x,ans)

Resolve with solve_back1 to form

<- solve(x,Can,ans) & equiv(lstep=rstep,Can) & free_of(x,Ans)

4. Resolve with solve_back2 to form

<- solve(x,Can1,ans) & solve(x,Can,ans) & equiv(lstep=rstep,Can)

5. Merge two occurrences of solve(x,Can,ans)

<- solve(x,Can,ans) & equiv(lstep=rstep,Can)

6. Resolve with stephyp4 to form

<- single_occ(x,Can) & position(x,Can,Posn) & isolate(Posn,Can,ans)

& equiv(lstep=rstep,Can)

7. Resolve with single to form

<- single_occ(x,Can1) & isolsx(N,Can1,Can) & position(x,Can,posn) &
isolate(posn,Can,ans) & equiv(lstep=rstep,Can)

8. Resolve with equivx to form

<- isolsx(N2,lstep=rstep,Can2) & single_occ(x,Can1) &
isolsx(N,Can1,Can) & position(x,Can,posn) & isolate(posn,Can,ans)

9. Resolve with posn to form

< isolsx(N3,Can3,Can) & position(x,Can3,[N3|posn3]) &
isolsx(N2,lstep=rstep,Can2) & single_occ(x,Can1) &
isolsx(N,Can1,Can) & isolate(posn,Can,ans)

10. Merge two occurrences of isolsx(M,lstep=rstep,Cam) to form

<- position(x,lstep=rstep,[M|posn]) & isolsx(M,lstep=rstep,Cam) &
single_occ(x,Can1) & isolsx(N,Can1,Can) & isolate(posn,Can,ans)

11. Merge two occurrences of isolsx(MN,lstep=rstep,Camn) to form

< position(x,lstep=rstep,[MN|posn]) & single_occ(x,lstep=rstep) &
isolsx(MN,lstep=rstep,Camn) & isolate(posn,Camn,ans)

12. Resolve with isolate to form

< isolate([MN|Posn],lstep=rstep,Ans) &
position(x,lstep=rstep,[MN|posn]) & single_occ(x,lstep=rstep) &
isolate(posn,Camn,ans)

13. Resolve with isolate2 to form

<- isolate([N|posn],Can,ans) & isolate([MN|Posn],lstep=rstep,Ans) &
position(x,lstep=rstep,[MN|posn]) & single_occ(x,lstep=rstep)

14. Merge two occurrences of isolate([MN|posn],lstep=rstep,ans) to form

<- isolate([MN|posn],lstep=rstep,ans) &
position(x,lstep=rstep,[MN|posn]) & single_occ(x,lstep=rrtep)

15. Resolve with stephyp1 to form

< isolate([MN|posn],lstep=rstep,ans) &
position(x,lstep=rstep,[MN|posn])

16. Resolve with stephyp2 to form

< isolate([n|posn],lstep=rstep,ans)

17. Finally, resolve with stephwp3 to give a contradiction.

!centre(Talk given at Mecho meeting 22.5.81)

1. Aim. Basic objectives of IMPRESS theorem-prover in meta-theory of algebra etc.

2. Current objective is to generate some proof schema, collect them together and then implement some code. Suggestions welcome for important proofs to be considered. I'll start with a proof of the correctness of isolate (a suitably modified version of note 64).

3. Write up statement to be proved and give handout.

4. Current environment. Alan's proof checker modified a little, plus Lawrence's data base. Explain notation.

. The first decision the program has to make is how to prove the result. My initial assumption will be induction proofs, so what the program has to decide is how (or on what) to induct. In this case the program looks at the subgoals and tries to find a suitable candidate for an induction scheme. In the isolate example, there are 3 subgoals to be considered. Single_occ calls occ which is a utility (I'm not sure exactly where this infomation will be). Position is also a utility, and can be similarly ignored. However the axiom isolate which is the representation here of the isolate method very clearly recurses on a smaller and smaller Posn. So induction on the structure of Posn will be tried.

6. The next step is to set up the two cases that need to be proved, i.e. the base case and the step case. The base case is when we set Posn to be [], the empty list. (Next handout). Symbols are gensymmed for uninstantiated variables, (nice ones have been chosen on the sheet). And the proof flows reasonably naturally. Interestingly hyp2 is not used (is this bad?).

7. The step case leads to the second proof on the handout. And the proof is complete.

8. Implementation details are necessarily vague at the moment as is any perspective on the relationship of IMPRESS to Boyer-Moore, automatic programming or anything else.

C'est tout!

Version used for note 86

```
/*   Axioms to generate the isolation proof
       essentially given in note 64    11.5.81   Leon          */

ax(defn_solve,[solve(X,Eqn,Ans)],[equiv(Eqn,X=Ans),free_of(X,Ans)]).

ax(solve_back1,[equiv(Eqn,X=Ans)],[solve(X,Eqn,Ans)]).

ax(solve_back2,[free_of(X,Ans)],[solve(X,Eqn,Ans)]).

ax(free,[free_of(X,B)],[single_occ(X,X=B)]).

ax(single,[single_occ(X,Exp)],[single_occ(X,New),isolax(N,Exp,New)]).

ax(single2,[single_occ(X,X=Ans)],
       [single_occ(X,Eqn),position(X,Eqn,Posn),isolate(Posn,Eqn,Ans)]).

ax(posn,[position(X,Eqn,Posn)],[isolax(N,Eqn,Eqn1),
                                    position(X,Eqn1,[N|Posn])]).

ax(equivx,[equiv(New,Old)],[isolax(N,Old,New)]).

   equiv,[equiv(Old,X=Ans)],[isolate(Pos,Old,Ans)]).

ax(equiv_ex,[equiv(A,B)],[equiv(A,C),equiv(C,B)]).

ax(isolate,[isolax(N,Old,New)],[isolate([N|Posn],Old,Ans)]).

ax(isolate2,[isolate(Posn,New,Ans)],[isolate([N|Posn],Old,Ans)]).

/* Possible alternate axioms to generate proof

ax(equiv_sym,[equiv(A,B)],[equiv(B,A)]).

ax(isolat,[isolax(N,Old,New)],[isolate([N|Posn],Old,Ans),
                                        isolate([Posn],New,Ans)]).

ax(is_back,[isolate(Pos,Eqn,Ans)],[solve(X,Eqn,X=Ans)]).

*/


%       The negation of the base case yields

ax(basehyp1,[single_occ(x,lbase=rbase)],[]).

ax(basehyp2,[position(x,lbase=rbase,[])],[]).

ax(basehyp3,[isolate([],lbase=rbase,ans)],[]).

ax(basegoal,[],[solve(x,lbase=rbase,ans)]).


%       The negation of the induction step yields

ax(stephyp1,[single_occ(x,lstep=rstep)],[]).

ax(stephyp2,[position(x,lstep=rstep,[n|posn])],[]).

ax(stephyp3,[isolate([n|posn],lstep=rstep,ans)],[]).
```

```
ax(stephyp4,[solve(X,Eqn,Ans)],[single_occ(X,Eqn),
                position(X,Eqn,posn),isolate(posn,Eqn,Ans)]).

ax(stepgoal,[],[solve(x,lstep=rstep,ans)]).
```

STEP CASE 21/5/81

```
Prolog-10  version 3.2
Copyrisht (C) 1981 by D. Warren, F. Pereira and L. Byrd

! ?- resolve(indsoal,defn_solve).
We resolve indsoal and defn_solve to form new clause lemma6 which is:

        <-
equiv(lnew=rnew,x=ans) &
free_of(x,ans)
yes
! ?- resolve(lemma6,equiv_ex).
We resolve lemma6 and equiv_ex to form new clause lemma7 which is:

        <-
equiv(lnew=rnew,_124) &
equiv(_124,x=ans) &
free_of(x,ans)
yes

's example of hyper_resolution comes from lemma7 and defn_solve_back.

! ?- print_clause(hlemma1).

        <-
solve(x,_67,ans) &
equiv(lnew=rnew,_67)
yes
! ?- resolve(indhyp4,hlemma1).
We resolve indhyp4 and hlemma1 to form new clause lemma8 which is:

        <-
single_occ(x,_123) &
position(x,_123,[posn]) &
isolate([posn],_123,ans) &
equiv(lnew=rnew,_123)
yes
! ?- resolve(lemma8,single).
We resolve lemma8 and single to form new clause lemma9 which is:

        <-
single_occ(x,_124) &
equiv(_124,_123) &
position(x,_123,[posn]) &
isolate([posn],_123,ans) &
equiv(lnew=rnew,_123)
yes
! ?- merge(lemma9).
We merge two occurrences of equiv(lnew=rnew,_88) in lemma9 to form corollary2, wh

        <-
single_occ(x,lnew=rnew) &
position(x,_88,[posn]) &
isolate([posn],_88,ans) &
equiv(lnew=rnew,_88)
yes
! ?- resolve(corollary2,indhyp1).
We resolve corollary2 and indhyp1 to form new clause lemma10 which is:

        <-
```

```
position(x,_140,[posn]) &
isolate([posn],_140,ans) &
equiv(lnew=rnew,_140)
yes
| ?- resolve(lemma10,equivx).
We resolve lemma10 and equivx to form new clause lemma11 which is:

        <-
isolax(_124,lnew=rnew,_123) &
position(x,_123,[posn]) &
isolate([posn],_123,ans)
yes

This clause comes from hyper_resolving lemma11 and isolate, which is

ax(isolate,[isolax(N,Old,New),position(X,New,[P]),isolate([P],New,Ans)],
                        [position(X,Old,[N|P]),isolate([N|P],Old,Ans)]).

| ?- print_clause(hlemma2).

        <-
olate([_67|posn],lnew=rnew,ans) &
position(x,lnew=rnew,[_67|posn])
yes
| ?- resolve(hlemma2,indhyp2).
We resolve hlemma2 and indhyp2 to form new clause lemma12 which is:

        <-
isolate([n|posn],lnew=rnew,ans)
yes
| ?- resolve(lemma12,indhyp3).
We resolve lemma12 and indhyp3 to form new clause lemma13 which is:

        <-

yes
| ?- halt.
```

```
RESS    DAI                                    PROOFS : 19/5/81
 Version of 19 may 81


los-10  version 3.2
yrisht (C) 1981 by D. Warren, F. Pereira and L. Byrd

- resolve(goal,defn_solve).
resolve goal and defn_solve to form new clause lemma1 which is:

     <-
iv(l=r,x=ans) &
e_of(x,ans)

- resolve(lemma1,free).
resolve lemma1 and free to form new clause lemma2 which is:

     <-
sle_occ(x,x=ans) &
iv(l=r,x=ans)

esolve(lemma2,single).
resolve lemma2 and single to form new clause lemma3 which is:

     <-
sle_occ(x,_124) &
iv(_124,x=ans) &
iv(l=r,x=ans)

- merge(lemma3).
merge two occurrences of equiv(l=r,x=ans) in lemma3 to form corollary1, which is:

     <-
sle_occ(x,l=r) &
iv(l=r,x=ans)

- resolve(corollary1,equiv).
resolve corollary1 and equiv to form new clause lemma4 which is:

     <-
  e(_125,l=r,ans) &
sle_occ(x,l=r)

- resolve(lemma4,hyp1).
resolve lemma4 and hyp1 to form new clause lemma5 which is:

     <-
late(_140,l=r,ans)

- resolve(lemma5,hyp2).
resolve lemma5 and hyp2 to form new clause lemma6 which is:

     <-


**********************************************************************
 first three steps of the proof of the induction step are the same
for the base case, so the proof besins wioth lemma3 from above.
```

```
?- resolve(lemma3,indhyp1).
 resolve lemma3 and indhyp1 to form new clause lemma7 which is:

      <-
div(lnew=rnew,x=ans) &
div(l=r,x=ans)
;
?- swap2(lemma7).
orderins terms to form new clause corollary2 which is:

      <-
div(l=r,x=ans) &
div(lnew=rnew,x=ans)
;
?- resolve(corollary2,equiv_ex).
 resolve corollary2 and equiv_ex to form new clause lemma8 which is:

      <-
   l=r,_124) &
iv,_124,x=ans) &
div(lnew=rnew,x=ans)
;
?- merge(lemma8).
 merge two occurrences of equiv(lnew=rnew,x=ans) in lemma8 to form corollary3, whi

      <-
div(l=r,lnew=rnew) &
div(lnew=rnew,x=ans)
;
?- resolve(corollary3,equivx).
 resolve corollary3 and equivx to form new clause lemma9 which is:

      <-
plax(_124,l=r,lnew=rnew) &
div(lnew=rnew,x=ans)
;
?- resolve(lemma9,equiv).
    olve lemma9 and equiv to form new clause lemma10 which is:

      <-
plate(_125,lnew=rnew,ans) &
plax(_145,l=r,lnew=rnew)
;
?- resolve(lemma10,indhyp2).
 resolve lemma10 and indhyp2 to form new clause lemma11 which is:

      <-
plax(_141,l=r,lnew=rnew)
;
?- resolve(lemma11,indhyp3).
 resolve lemma11 and indhyp3 to form new clause lemma12 which is:

      <-

;
?- halt.
```

mid-April

```
Utilities package  (2 April 81)
Prolog-10  version 3.2

| ?- [filin].

press:press.ops consulted   30 words      0.03 sec.

press:misc consulted   644 words      0.22 sec.

press:match consulted   1743 words      0.73 sec.

checke consulted   996 words      0.41 sec.

isolat.new consulted   338 words      0.12 sec.

filin consulted   3813 words      1.66 sec.

yes
| ?- ok.
  PRESS    DAI
Prolog-10  version 3.2

| ?- resolve(soal,defn_solve).
We resolve soal and defn_solve to form new clause lemma1 which is:

        <-
equiv(x=ans,l=r) &
free_of(x,ans)
yes
| ?- resolve(equiv_sym,lemma1).
We resolve equiv_sym and lemma1 to form new clause lemma2 which is:

        <-
equiv(l=r,x=ans) &
free_of(x,ans)
yes
| ?- resolve(free,lemma3).

_0
| ?- resolve(free,lemma2).
We resolve free and lemma2 to form new clause lemma3 which is:

        <-
single_occ(x,x=ans) &
equiv(l=r,x=ans)
yes
| ?- resolve(single,lemma3).
We resolve single and lemma3 to form new clause lemma4 which is:

        <-
single_occ(x,_111) &
equiv(_111,x=ans) &
equiv(l=r,x=ans)
yes
| ?- merge(lemma4).
We merge two occurrences of equiv(l=r,x=ans) in lemma4 to form corr1, which is:

        <-
single_occ(x,l=r) &
equiv(l=r,x=ans)
```

```
yes
| ?- resolve(hyp1,corr1).
We resolve hyp1 and corr1 to form new clause lemma5 which is:

        <-
equiv(l=r,x=ans)
yes
| ?- resolve(equiv,lemma5).
We resolve equiv and lemma5 to form new clause lemma6 which is:

        <-
isolate([],l=r,ans)
yes
| ?- resolve(hyp2,lemma6).;
|    .

***syntax error***
resolve(hyp2,lemma6)
***here***
  ;.
     resolve(hyp2,lemma6).
We resolve hyp2 and lemma6 to form new clause lemma7 which is:

        <-

yes
| ?- halt.
```

A Lemma ?

Feb '8
for base case

```
| ?- resolve(goal,defn_solve).

We resolve goal and defn_solve to form new clause lemma1 which is:

        <-
free_of(x,_108) &
ans=(x=_108) &
equiv(ans,_117=b)
yes
| ?- resolve(ident,lemma1).

We resolve ident and lemma1 to form new clause lemma2 which is:

        <-
ans=(_116=b) &
free_of(x,_115) &
ans=(x=_115)
yes
| ?- merge(lemma2).

We merge two occurrences of ans=(x=b) in lemma2 to form corr1, which is:

        <-
free_of(x,b) &
ans=(x=b)
yes
| ?- resolve(corr1,free_of).

We resolve corr1 and free_of to form new clause lemma3 which is:

        <-
single occ(x,_107=b) &
position(x,_107,_108) &
ans=(x=b)
yes
| ?- resolve(lemma3,hyp1).

We resolve lemma3 and hyp1 to form new clause lemma4 which is:

        <-
position(x,a,_113) &
ans=(x=b)
yes
| ?- resolve(lemma4,hyp2).

We resolve lemma4 and hyp2 to form new clause lemma5 which is:

        <-
ans=(x=b)
yes
| ?- resolve(position+1,hyp2).

We resolve position+1 and hyp2 to form new clause lemma6 which is:
x=a
        <-

yes
| ?- paramod(lemma6,lemma5).

We paramodulate lemma6 into lemma5 to form new clause lemma7, which is:
```

```
        <-
ans=(a=b)
yes
! ?- resolve(lemma7,isolate1).

We resolve lemma7 and isolate1 to form new clause lemma8 which is:

        <-
isolate([],a=b,ans)
yes
! ?- resolve(lemma8,hyp3).

We resolve lemma8 and hyp3 to form new clause lemma9 which is:

        <-

yes
! ?- true.

yes
```

```
/* OCC : Provins the dual occurrence of the unknown in the equstion thm.
```

See note 139

```
                                                    Leon
                                                    Updated: 15 September 82
*/

solve(Eqn,X,Soln) <-- occ(X,Eqn,s(s(2))) & collect(Eqn,X,New)
        & position(X,New,P) & isolate(P,New,Soln)                    (1)

solve(Eqn,X,Soln) <-- position(X,Eqn,Posn)
                & isolate(Posn,Eqn,Soln) & single_occ(X,Eqn)         (2)

occ(X,Eqn,s(N)) <-- collect(Eqn,X,New) & occ(X,New,N)                (3)

single_occ(X,Eqn) <-- occ(X,Eqn,s(0))                                (4)

solve(Eqn,X,X=Ans) <-- free_of(X,Ans) & equiv(Eqn,X=Ans)             (5)

solve(Eqn,X,Soln) <-- equiv(Eqn,New) & solve(New,X,Soln)             (6)

equiv(Eqn,New) <-- collect(Eqn,X,New)                                (7)

Negating and skolemizing (1) gives

occ(x,eqn,s(s(2)))   <--                hyp1
collect(eqn,x,new)   <--                hyp2
position(x,new,p)    <--                hyp3
isolate(p,new,soln) <--                 hyp4
<-- solve(eqn,x,soln)                   goal

Resolving goal with (6) gives

<-- equiv(eqn,New) & solve(New,x,soln)                               (R1)

Resolving (R1) with (2) gives

<-- equiv(eqn,New) & position(x,New,P) & isolate(P,New,soln)
                        & single_occ(x,New)                          (R2)

Resolving (R2) with hyp4 gives

<-- equiv(eqn,new) & position(x,new,p) & single_occ(x,new)           (R3)

Resolving (R3) with hyp3 gives

<-- equiv(eqn,new) & single_occ(x,new)                               (R4)

Resolving (R4) with (7) gives

<-- collect(eqn,X,new) & single_occ(x,new)                           (R5)

Resolving (R5) with (4) gives

<-- collect(eqn,X,new) & occ(x,new,s(0))                             (R6)

Resolving hyp1 forward wrt (3) gives

collect(eqn,x,New) <--  and occ(x,New,s(0)) <--
Call the latter hyp1'.
```

Resolving (R6) with hyp2 gives

<-- occ(x,ea,s(0)) (R7)

Resolving R7 with hyp1' gives <-- | Hooray!

```
/* FILIN : Impres read in file - 11.5.81        */

:- [
                'press:press.ops'       % Operator declarations
   ].


:- compile([
                load,                   % Load axioms
                clause,                 % Clause database management
                print                   % Clause pretty printer
           ]).

:- [
                'press:misc',           % Press utilities
                'press:match',          % Matching code
                checke,                 % Alan's proof
   ].

:- load([
                axioms                  % Isolation clauses
         ]).


ok :- core_image, writehead, ttynl, reinitialise.

writehead :- write('IMPRESS    DAI').
```

```
/* AXIOMS. :

                                                    Leon
                                                    Updated: 2 November 81
*/

ax(isolate_correct,[solve(X,Eqn,Ans)],
        [single_occ(X,Eqn),position(X,Eqn,Posn),isolate(Posn,Eqn,Ans)]).

ax(isolate,[isolate([N|Posn],Eqn,Ans)],
                        [isolax(N,Eqn,Eqn1),isolate(Posn,Eqn,Ans)]).

ax(list,[is_list(Z)],[is_list(X),is_list(Y),append(X,Y,Z)]).

ax(append,[append([H|T],Y,Z)],[Z = [H|Zrest],append(T,Y,Zrest)]).

/*   Axioms to generate the isolation proof
        essentially given in note 64    11.5.81   Leon          */

iff_ax(defn_solve,[solve(X,Eqn,Ans)],[equiv(Eqn,X=Ans),free_of(X,Ans)]).

ax(free,[free_of(X,B)],[single_occ(X,X=B)]).

ax(single,[single_occ(X,Exp)],[single_occ(X,New),isolax(N,Exp,New)]).

ax(single2,[single_occ(X,X=Ans)],
        [single_occ(X,Eqn),position(X,Eqn,Posn),isolate(Posn,Eqn,Ans)]).

ax(posn,[position(X,Eqn,Posn)],[isolax(N,Eqn,Eqn1),
                                        position(X,Eqn1,[N|Posn])]).

ax(equivx,[equiv(New,Old)],[isolax(N,Old,New)]).

ax(equiv,[equiv(Old,X=Ans)],[isolate(Pos,Old,Ans)]).

ax(equiv_ex,[equiv(A,B)],[equiv(A,C),equiv(C,B)]).

iff_ax(isolate,[isolate([N|Posn],Old,Ans)],
                [isolate(Posn,New,Ans),isolax(N,Old,New)]).

/* possible alternate axioms to generate proof

ax(equiv_sym,[equiv(A,B)],[equiv(B,A)]).

ax(isolat,[isolax(N,Old,New)],[isolate([N|Posn],Old,Ans),
                                        isolate([Posn],New,Ans)]).

ax(is_back,[isolate(Pos,Eqn,Ans)],[solve(X,Eqn,X=Ans)]).

*/

%       The negation of the base case yields

ax(basehyp1,[single_occ(x,lbase=rbase)],[]).

ax(basehyp2,[position(x,lbase=rbase,[])],[]).
```

delete
convert to
AXIOMS.OCD

```
ax(basehyp3,[isolate([],lbase=rbase,ans)],[]).

ax(basegoal,[],[solve(x,lbase=rbase,ans)]).


%       The negation of the induction step yields

ax(stephyp1,[single_occ(x,lstep=rstep)],[]).

ax(stephyp2,[position(x,lstep=rstep,[n|posn])],[]).

ax(stephyp3,[isolate([n|posn],lstep=rstep,ans)],[]).

ax(stephyp4,[solve(X,Eqn,Ans)],[single_occ(X,Eqn),
                position(X,Eqn,posn),isolate(posn,Eqn,Ans)]).

stepgoal,[],[solve(x,lstep=rstep,ans)]).
```

```
SUBFILE: AXIOMS. @9:53 2-JUN-1981 <005> (320)
/*   Axioms to generate the isolation proof
     .   essentially given in note 64    11.5.81   Leon          */

iff_ax(defn_solve,[solve(X,Eqn,Ans)],[equiv(Eqn,X=Ans),free_of(X,Ans)]).

ax(free,[free_of(X,B)],[single_occ(X,X=B)]).

ax(single,[single_occ(X,Exp)],[single_occ(X,New),isolax(N,Exp,New)]).

ax(single2,[single_occ(X,X=Ans)],
        [single_occ(X,Eqn),position(X,Eqn,Posn),isolate(Posn,Eqn,Ans)]).

ax(posn,[position(X,Eqn,Posn)],[isolax(N,Eqn,Eqn1),
                                    position(X,Eqn1,[N|Posn])]).

ax(equivx,[equiv(New,Old)],[isolax(N,Old,New)]).

   equiv,[equiv(Old,X=Ans)],[isolate(Pos,Old,Ans)]).

ax(equiv_ex,[equiv(A,B)],[equiv(A,C),equiv(C,B)]).

iff_ax(isolate,[isolate([N|Posn],Old,Ans)],
                [isolate(Posn,New,Ans),isolax(N,Old,New)]).

/* Possible alternate axioms to generate proof

ax(equiv_sym,[equiv(A,B)],[equiv(B,A)]).

ax(isolat,[isolax(N,Old,New)],[isolate([N|Posn],Old,Ans),
                                        isolate([Posn],New,Ans)]).

ax(is_back,[isolate(Pos,Eqn,Ans)],[solve(X,Eqn,X=Ans)]).

*/


        The negation of the base case yields

ax(basehyp1,[single_occ(x,lbase=rbase)],[]).

ax(basehyp2,[position(x,lbase=rbase,[])],[]).

ax(basehyp3,[isolate([],lbase=rbase,ans)],[]).

ax(basegoal,[],[solve(x,lbase=rbase,ans)]).


%       The negation of the induction step yields

ax(stephyp1,[single_occ(x,lstep=rstep)],[]).

ax(stephyp2,[position(x,lstep=rstep,[n|posn])],[]).

ax(stephyp3,[isolate([n|posn],lstep=rstep,ans)],[]).

ax(stephyp4,[solve(X,Eqn,Ans)],[single_occ(X,Eqn),
                position(X,Eqn,posn),isolate(posn,Eqn,Ans)]).
```

```
ax(stepsobl,[],[solve(x,lstep=rstep,ans)]),
```

```
/* ISOLAT - Clauses for proving the correctness of isolation

                                                    Leon
                                                    Updated: 6 November 81
*/

skolemize([solve(X,Eqn,Ans)],[single_occ(X,Eqn),position(X,Eqn,[])],
          [solve(x,lbase=rbase,ans)],[single_occ(x,lbase=rbase),
                                                    position(x,lbase=rbase,[])]).

skolemize([solve(X,Eqn1,Ans)],[single_occ(X,Eqn),position(X,Eqn,Posn),
                          isolate(Posn,Eqn,Ans)],[solve(x,lstep=rstep,ans)],
        [single_occ(x,lstep=rstep),position(x,lstep=rstep,[n|posn]),
                          isolate([n|posn],lstep=rstep,ans)]).

    :form(isolate_correct,isolate,isolate(posn,_,_),[solve(x,lstep=rstep,ans)],
        [equiv(lstep=rstep,Eqn),single_occ(x,Eqn),position(x,Eqn,posn)]).

:- add_axiom(isolate_correct,[solve(Eqn,X,Ans)],
          [single_occ(X,Eqn),position(X,Eqn,Posn),isolate(Posn,Eqn,Ans)]).

hyp(isolate_correct,isolate).

rec_pred(isolate,3,list,1).

step(isolate([N|Posn],Eqn,Ans),isolax(N,Eqn,Neweqn)).

%       recur(isolate(Posn,Eqn,Ans),isolate([N|Posn],Oldeqn,Ans)).

condition(isolate_correct,single_occ(_,_)).

setup(isolate_correct,posn(_,_,_)).

    hyp(isolate_correct,_) :-
        add_lemma(stephyp,[solve(X,Eqn,Ans)],[single_occ(X,Eqn),
                position(X,Eqn,posn),isolate(posn,Eqn,Ans)]).

resolve_to_step([solve(x,lstep=rstep,ans)],
                          [solve(x,Eqn,ans),equiv(lstep=rstep,Eqn)]).

resolve_step([solve(x,Eqn,ans),equiv(lstep=rstep,Eqn)],[equiv(lstep=rstep,Eqn),
        single_occ(x,Eqn),position(x,Eqn,posn),isolate(posn,Eqn,ans)]).

resolve_recur([equiv(lstep=rstep,Eqn),single_occ(x,Eqn),position(x,Eqn,posn),
                          isolate(posn,Eqn,ans)],_,
        [equiv(lstep=rstep,Eqn),single_occ(x,Eqn),position(x,Eqn,posn)]).
```

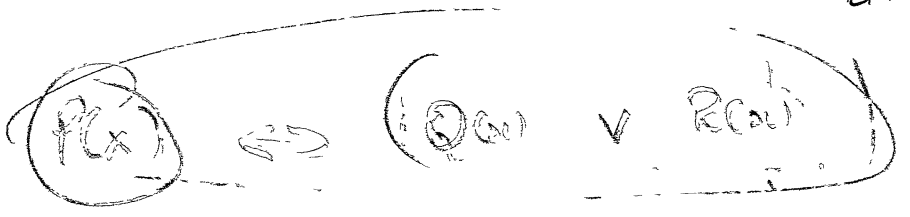
```
/* KLAUSE - Ho hum? */

/* Produce new clauses dynamically from definitions */

klause(Pred-N, [Reln], Body) :-
        defn(Reln,case(List)),
        functor(Reln,Pred,Arity),
        nmember(triple(Cond,Body,Ans), List, N),
        Cond, Ans.

klause(Pred+N, [Ans], [Reln|Body]) :-
        defn(Reln, case(List)),
        functor(Reln,Pred,Arity),
        nmember(triple(Cond,Body,Ans), List, N),
        Cond.
```

Alan's way of generating iff statements etc,



Q(x) => P(x)

R(x) => P(x)

P(x) => (Q(x) ∨ R(x))

```
SUBFILE: LOAD. @11:28 2-JUN-1981 <005> (245)
/* LOADAX : Load axioms from a file

                                                        Leon
                                                        Updated: 11 May 81
*/

/* EXPORT */

  :- public     load/1.


/* IMPORT */
/*
  UTIL:FILES    open/2
                close/2
*/

/* MODES */

  :- mode       load(+),
                loadins_axioms,
                process_axiom(+),
                read_next(?).



                        % Load from a list of files

load(V)
     :- var(V),
        !,
        errmess('Variable as file name').

load([]) :- !.

load([HD|TL])
     :- !,
        load(HD),
        load(TL).

load(File)
     :- open(Old,File),
        !,
        loadins_axioms,
        ttynl, display('Axioms loaded from '),
        display(File), ttynl,
        close(File,Old).

load(_).



                        % Load all the axioms from a file

loadins_axioms
     :- repeat,
```

```
        read_next(Axiom),
        process_axiom(Axiom),
        !.

process_axiom( end_of_file ) :- !.

process_axiom( iff_ax(Name,Left,Risht) )
     :- !,
        add_iff(Name,Left,Risht),
        fail.

process_axiom( ax(Name,Left,Risht) )
     :- !,
        add_axiom(Name,Left,Risht),
        fail.

process_axiom( Gash )
     :- ttynl, display('Rubbish ignored: '),
        display(Gash), ttynl,
        fail.
```

process_axiom(rec_def(Name, Base, Step)) :- fail,
process_axiom (Theorem) :- assert (Theorem).

```
                % Read next entry - discard variables

read_next(X)
     :- repeat,
        read(Y),
        ( nonvar(Y)  ;  errmess('Variable ignored'), fail ),
        !,
        X = Y.
```

process_axiom (rec_def (Name, Base, Step)) :-
    insert (Base),
    ~~insert (Step).~~
    conj_to_comma (Step, T)
    pair_to_clause (S, T, clause),
    insert (clause).

pair_to_clause (S, T, (S :- T))

insert ((A :- B)) :- !

conj_to_comma (A & B, (A, C)) :- !, conj_to_comma (B, C).
conj_to_comma (A, A).

```
SUBFILE: PRINT. @17:37 11-MAY-1981 <005> (125)
/* PRINT : Pretty printer for clauses
     .  .  '
                                                        Leon
                                                        Updated: 11 May 81
*/

/* EXPORT */

  :- public      print_clause/1,
                 print_clause/2.


/* MODES */

  :- mode        print_clause(+),
                 print_clause(?,?),
                 print_list(+,+).

  '

                          % Print out a clause

print_clause(Name) :- !,
        set_clause(Name,Pos,Nes),
        print_clause(Pos,Nes).



print_clause(Pos,Nes) :-
        print_list(Pos,v),
        writef('\n\t<-\n',[]),
        print_list(Nes,&), !.


print_list([],_) :- !.

   'nt_list([Hd],_) :- !,
        writef('%t',[Hd]).

print_list([Hd1,Hd2|Tl],Sym) :- !,
        writef('%t %t\n',[Hd1,Sym]), print_list([Hd2|Tl],Sym).



\\\\\
```

From Leon S[400,4321] on July 20, 1981 at 12:23 AM
Alan

I like the paper. It's good to have the terms of the meta-language written down, and my ideas of what IMPRESS does have been clarified. There are a few points however.

1. I'd like to suggest a slight reorganisation, which includes my quibble of the other day. What is basically involved is a swap of sections 3 and 4 so we can highlight more clearly the parallels between the two uses of meta-level inference. The isolate example and the proof of its correctness forms the thread through the paper.
So the emphasis in section 2 might be slightly different, illustrating isolation as a method of PRESS. Then section 3 would show how this leads to a guided proof (as is currently done in secn. 4). Section 4 would outline IMPRESS , mentioning the learning environment, and illustrating the derivation of properties about a PRESS procedure as an IMPRESS method. In this case setting up the correctness of isolate as a goal theorem to be proved. (Just a slight rearrangement of the 'rent secn.3). Then in section 5, after establishing the new language, show how meta-level inference leads to a proof of the theorem, hence deriving new control information.

Is the term meta-meta-level standard? If not perhaps we should explain it more.

2. The rewrite rule $\log_u v = w \rightarrow u = v^{1/w}$ is not very familiar to me. May I suggest $e^{\sin(x^2)} = \tan x$ instead as the example to illustrate isolation.

3. One case I'm not sure how to handle if we define position and isolate as you suggest is if the single occurrence of the unknown occurs on the right-hand side of the expression. How does the proof as is, work for 2 = x+3.

4. I'd prefer the tense of verbs in describing the Winston-type learning program to be conditional(or is it subjective?). That is use would rather than will. I've suggested an alternative version.

5. Perhaps the notational footnote about the PROLOG variable convention could be expanded into a notational point at the end of the introduction explaining the notation of the paper, PRESS language, etc. Can the use of v for or be avoided?

Leon

From Bundy Hps[400,405] on July 20, 1981 at 10:48 AM
Leon

Thanks for you comments on our paper, some of which I have incorporated in the draft, and the remainder I wish to argue about. The numbers refer to your points.

0. I am not averse to changing the title, but could you explain what you were trying to achieve with your amendment? For my part I am trying to sell 'Meta-Level Inference' as a technical term. Also I think that using PRESS in the title, when people cannot be expected to know what

it means, is a bit cryptic.

1. I originally tried to fit section 4 before section 3, but failed, because the PRESS clauses introduced in section 3 are needed to explain meta-level inference. As it is there is a nice juxtaposition between 2 use of m-l i in sections 4 and 5, which is announced as a theme of the paper in the abstract/intro, so I am fairly happy. I think people will understand what is meant by meta-meta-level.

2. I deliberately chose the example so it would look unfamiliar, and hence impress (no pun intended) people with the ability of PRESS. Do you think this was a tactical error?

3. The simplistic version of Isolation defined in the paper and proved correct by IMPRESS cannot handle unknowns on the RHS. I will say so in the paper. It is another simplifying assumption.

4. I changed the tense.

, I changed "v" to "or". What more do you think should be said about notation?

I only have this week to get this draft off, and do 100 other things, so I am keen to send the draft off with only minor alterations. However, we do not have to use this version as the research paper or the final workshop paper. I suggest that you take over the paper and rework it on points 0, 1 and 5, and that we review this together when I get back.

Alan

From Leon S[400,4321] on July 21, 1981 at 12:19 AM
Alan
The basic structure of both the collection and attraction clauses lends itself to our meta-level language.

```
collect(X,Old,New) :- collax(  ,Old,Old1), collect(X,Old1,New)
 ,
 ,tract(X,Old,New) :- attrax(  ,Old,Old1), attract(X,Old1,New)
```

Here collax and attrax are the performants, while collect and attract are the recursants.

The correct structures to induct on when dealing with collection and attraction are the number of occurrences and the size of the closeness tree respectively. Unfortunately, the way collect and attract are currently defined does not include these structures. But it looks relatively straightforward to redefine them. And this would be more consistent with the handling of isolation, which isn't called until certain conditions are met. These are the conditions we refer to in the meta-language - single_occ and position.
The relevant conditions for collection and attraction are some subset of least_dom, match and contains.

A plausible solve clause (or meta-level theorem) might then be something like

```
solve(X,Eqn,Soln) :- subterm(Sub,Eqn), least_dom(X,Sub),
                        collect(N,Sub,Sub1), subst(Sub=Sub1,Eqn,Eqn1),
                              solve(X,Eqn1,Soln).
```

That's not quite right but gives some flavour.
The hypothesis can clearly seen to be the last 3 terms,
and the conditions the first 2.

Perhaps it might be an idea to develop a dialect of PRESS
called pure PRESS in the analogous fashion to pure LISP.
Not a very serious suggestion but it might be interesting to setup
a slightly modified database of PRESS methods as the basis for
testing IMPRESS.

Leon

From Leon Hps[400,4321] on February 4, 1982 at 7:33 AM
A version of the code archived today proved the two
lisplike examples given in research paper No. 168.
I'm modifying the code so that it also proves the isolate
example, and incorporates suggestions from the PRESS meeting,etc.
This note is just to record the fact that the theorems in some
sense have been proved.

Leon

From Bundy Hps[400,405] on March 17, 1982 at 11:43 AM
Bernard & Leon
If we are to take our own propaganda seriously
then we shouldbe getting IMPRESS to learn esoteric
methods like trismethod (the AP tris thing) from say
one worked example. Doesthis sound feasible?
Bernard could Isee the code for trismethod?
Would this be a good focus for your thesis?
Alan
CC:Silver Hps,Sterling Hps

From Leon Hps[400,4321] on March 21, 1982 at 8:28 PM
After a concerted attack of debugging,
IMPRESS can live up to its name. It can now prove the following
5 theorems, and there is no reason why it should not be able
to handle variants.

1. length(Z,M) <-- length(X,L) & length(Y,N) & append(X,Y,Z)
& plus(L,N,M).

2. is_list(Z) <-- is_list(X) & is_list(Y) & append(X,Y,Z).

3. less(X,Z) <-- less(X,Y) & less(Y,Z).

4. append(X,Y,Z) <-- append(U,V,X) & append(V,Y,W) & append(U,W,Z).
i.e. associativity of append

5. solve(Eqn,X,Ans) <-- position(X,Eqn,Posn) & isolate(Posn,Eqn,Ans)
& single_occ(X,Eqn).

For anyone who would like to experiment, the code is in
impres.exe[400,4321].
Bugs are welcomed.

Leon
CC:Bundy Hps,Sterling Hps,Silver Hps,Okeefe Hps,Byrd Hps,Wallen Hps,Jenkins Hps

From Richard[400,422] on March 22, 1982 at 1:38 PM
Could you put something in Impress.Hlp telling users how to

define new functions, and perhaps data-types?

I came up with the idea of defining

```
times(0,X,0).
times(s(Y),X,Z) <-- times(Y,X,W) & plus(X,W,Z).

divides(A,B) <-> (Exists C) times(A,C,B).
```

and asking

```
divides(A,B) & plus(A,B,C) --> divides(A,C) ?
```

but I can't think of any way of expressing this.
Perhaps defining divides directly instead of using times, and providing as well that times(A,B,C) --> divides(A,C) & divides(B,C).

From Richard[400,422] on March 22, 1982 at 1:51 PM
Have a look at Impress.Log[400,422] and tell me what I'm doing wrong. In particular, why can't Impress find t1? (should it be conjecture(t1, times(X,0,0) <-- true) ??)
The little lemma invented for d2 is rather good, it happens to be true, and interesting. Congratulations.

From Richard[400,422] on March 22, 1982 at 6:57 PM
Just to see what it would do, I tried THOR on even(plus(x,x)). The definition was
even(x) = if x<2 then x=0 else even(sub1(sub1(x)))
I stopped it when it was down to Conjecture 1,1,2,2,2,2,2, and still going!! When I tried
even(x) = if x=0 then T else if x=1 then F else even(sub1(sub1(x)))
it !*****FAILED*****! miserably, no realising that it was valid!!

From Richard[400,422] on March 22, 1982 at 7:34 PM
I have managed to prove some elementary facts about plus. But not that 0+X=X. Oddly enough, defining all the arithmetic functions anew gave THOR the same trouble with this as with even. I think the trouble in IMPRESS may be that it needs induction on the second argument, and plus wants to induct of the first. All my attempts to prove things with Impress are in Impress.log.

From Richard[400,422] on March 23, 1982 at 8:32 PM
I've collected together three definitions (times, even, divides) and 24 theorems for Impress to prove. The file is Impress.Tst[400,422]. A log of Impress's efforts is Scra:Impress.Log. Would you care to rename Impress.Tst into your Impress area and take it off my hands? I get the impression that when you figure out when to unfold most of these theorems should be provable. But by no means all.

By the way, my earlier definition of divides was defective. When Impress conjectured that divides(X,0) it was what I meant to be true, but it isn't provable from that definition.

From Lincoln[400,4324] on April 19, 1982 at 7:27 PM
I have just had a bash at Impress and a peek through the code. Could you

1. I could not grasp how you actually prove that a goal is valid givensome axiom
ward
resolution. Maybe this is an unfair question ...but could you summarize it briefly ?

2. Is the process (above) complete ? i.e. will it always validate a logically t

3. By mistake I asked it to prove:
        is_list(Z) <-- is_list(X) & is_list(Y) & append(X,D,Z).

it succeeded by postulating a lemma something like:
        is_list(var25) <-- is_list(var26).

did it postulate that D was a list in order to prove the theorem(very clever)

4. Does the scheme used for induction take the validation procedure into account or doesn't it matter ? (i.e. does the sheme choosing routine + the validation form a complete process ?)

You can see why I am interested by the mention of completeness !!!!!!
Thanks for your indulgence
Linc

From Leon Hps[400,4321] on July 26, 1982 at 4:40 PM
Bug in commutativity of addition.

From Richard Hps[400,422] on August 5, 1982 at 6:50 PM
Impress:Problm.Tst
contains a new problem for Impress to tackle. I <<think>> the problem is that member is defined as in Prolog, i.e. there is a "base" case, but it is member(X, [X|_]); there is no base case on the data structure. I've tried all the examples in Impres.Tst; the distinguishing characteristic of the ones that fail seems to be that they require induction on a variable that Impress isn't prepared

that Impress doesn't want to do induciton on. Not having type information that will re-assure it that these arguments are in fact the sort of thing you can induct on, there doesn't seem to be any easy way for Impress to work out what induction pattern to use.

Is there any way of getting a more informative "proof" than "Theorem proved by foobar-induction"? Somehow, I can't work up much confidence in such messages, regardless of how well-justified I know they must be.

From Leon Hps[400,4321] on August 23, 1982 at 11:00 AM
There is a bug in fixpt_semant, which is preventing Richard's latest example from succeeding the goal fold_goal.

From Richard Hps[400,422] on August 26, 1982 at 9:39 PM
I have coded up some new problems for Impress. They are in the file

Inequa.Tst[400,4321,Impres]

and concern inequality. It is in fact possible to define inequality recursively (over the unary integers), though forcing it into Impress's current straitJacket is tricky, and the reason that only goals

ineq(0, X) <-- true

can be proved may be due to my failure to code it up properly. If one were permitted to have a predicate with TWO base cases in Impress, the proofs should be no more difficult than the similar proofs for less. (NB "u1" is not true; Impress correctly fails to prove it.)

To update

The completeness of Isolation
or
What the hell are we proving anyway?

Note 6 March 82 Leon Sterling

Notes 86 and 64 prescribed a proof of the 'correctness' of isolate. This proof was automated and formed the basis for the early versions of IMPRESS and research Papers 164 and 168. Clearly something interesting is happening and we are proving theorems. But I'm not clear what exactly the isolate proof is really about. Hence this note.

Consider the simplified PRESS clause for solve, invoking the isolate procedure.

Rod Burstall points out

```
solve(Eqn,X,X=Ans) <- single_occ(X,Eqn) & position(X,Eqn,Posn) & isolate(Posn,Eqn,Ans).
```

Intuitively, we also have termination. We would like completeness.

Viewed procedurally this is a call to isolate (my language preempted this in the sentence above), and the predicates single_occ and position are conditions governing the call of the procedure. All straightforward.

Viewed declaratively, this is a theorem in the Meta-Theory of Algebra. A proof of this theorem thus constitutes a proof of a meta-theorem of algebra. There are two points to be raised here. Firstly, we are not interested in Algebra meta-theorems per se. Thus what is interesting about the fact this theorem is proved? Secondly, what do we need to assume to prove this theorem and how does this reflect itself.

To consider the first question, isolate is a method of solving equations. It effectively works by resolution, which causes rewrite rules to be applied to an

isolate(N,Eqn, Ans) ← solve(Eqn,X,Ans) & single_occ & pos.

equation, transforming it into an equation where the solution is immediately read off. What are the classic questions about theorem proving methods? Correctness, Soundness, Termination and Completeness. Where does correctness fit into this triad?

ANOTHER IMPRESS PROOF

Note 139 17 September 82

Leon Sterling

One of the stated aims of the grant proposal for the 'Self-Improving Algebra System' is to derive new methods. In the proposal a method for solving equations with 2 occurrences of the unknown is given, namely collect once and then isolate. The relevant theorem, also given in working paper 55, is

```
solve(Eqn,X,Soln) <-- occ(X,Eqn,2) & collect(Eqn,X,New)
                        & isolate(X,New,Soln)
```

The proposed theorems necessary to derive this theorem were given as

```
solve(Eqn,X,Ans) <--  occ(X,Eqn,1) & isolate(X,Eqn,Ans)

occ(X,Eqn,1) <-- occ(X,Eqn1,2) & collect(Eqn1,X,Eqn)

equiv(Eqn,Eqn1) <-- collect(Eqn1,X,Eqn)

solve(Eqn,X,Soln) <-- equiv(Eqn,Eqn1) & solve(Eqn1,X,Soln)
```

I am pleased to report the vocabulary of IMPRESS is now rich enough for it to be possible to write down a proof plan to prove this theorem. The purpose of this note is to describe the proof plan and how the axioms can be encoded in the IMPRESS vocabulary in order to express the proof. No claim is made that the method of proof to be described is the neatest way of expressing this fact in the metatheory of algebra. Nor is the proof plan guaranteed to yield a general proof method. Rather the description is intended as a starting point for using IMPRESS to describe facts about PRESS and PRESS methods, and investigating the sort of representations needed for proving theorems about equation solving.

Firstly, let us make some slight changes to the original expression of the theorem, to bring it into line with the current view of PRESS. The predicate isolate is now defined with respect to a position list, which necessitates the introduction of an extra predicate, position, linking an equation with the position of its single occurrence. Also natural numbers are expressed in 'Peano notation' in order to enable recursive definitions and descriptions. For this example this means s(s(0)) is 2 and s(0) is 1. Also the form of the theorems needed is a little different in some cases.

The revised set of theorems used is

```
solve(L=R,X,Soln) <--
          occ(X,L=R,s(s(0))) & collect(L=R,X,NewL=NewR)
        & position(X,NewL=NewR,P) & isolate(P,NewL=NewR,Soln)      (1)

solve(L=R,X,Soln) <-- single_occ(X,L=R) & isolate(Posn,L=R,Soln)
                        & position(X,L=R,Posn)                     (2)

occ(X,Eqn,s(N)) <-- collect(Eqn,X,New) & occ(X,New,N)              (3)

single_occ(X,Eqn) <-- occ(X,Eqn,s(0))                              (4)

equiv(Eqn,New) <-- collect(Eqn,X,New)                              (5)
```

Some comments need be made on the form of these theorems. In order to prove 2, the correctness of isolation, it simplified matters considerably to explicitly represent the Eqn as L=R, i.e. put an equals into the predicate. Thus some clauses/axioms in IMPRESS are affected. So it was more convenient to explicitly have the equals sign in 1. (Somewhere there should probably be a meta-level predicate, equation(Eqn), whose definition would be equation(L=R).)

The occurrence information is expressed recursively, and one can (perhaps a little artificially) devise a base case, also. This allows the use of 3 as part of a recursive definition. The definition of occ used by IMPRESS for proving this theorem is

```
rec_def(occ, occ(X,Eqn,0) <-- free_of(X,Eqn),
          occ(X,Eqn,s(N)) <-- collect(Eqn,X,New) & occ(X,New,N).
```

Again our standard version of the correctness of isolation is given by the predicate single_occ, rather than occ, and this is reflected in the axiomatisation. Axiom 4 is then the obvious rewrite. Axiom 5 was also a rewrite. IMPRESS currently handles such rewrites by the definition type of axiom, and thus these 2 axioms can be encoded appropriately. Definitions are probably not the ideal method of expressing rewriting information, but better methods should appear on consideration of more examples.

The proof plan implementing the proof is now given. It is not claimed to be general purpose, but it is suggestive of how tactics can be grouped together.

```
prove_by_induction(Theorem,aaaaa(_)) :-
          negate_and_skolemize(Theorem,Goal,Assertions),
          fold_goal(Goal,Goal_rec,Goal_perf,_),
```

```
specified_resolve(isolate_correct,Goal_rec,G),
append(G,Goal_perf,NewGoal),
two_occ_establish(NewGoal,Assertions).
```

This plan has two added tactics, specified_resolve and two_occ_establish. The first is intended as a mechanism for telling the program which lemma to resolve with, a facility most theorem-proving programs have. The second written specifically for this is not as special purpose as it seems. It is just an interface to the method step_establish for proving a list of goals with some hints as to what to use. All that was necessary to get the method to work was the addition of another tactic (i.e. clause) to step_establish. This clearly indicates step_establish is based on a more general tactic which two_occ_establish could also use. (This will be considered when next improving IMPRESS.)

```
/* OCC : Proving the dual occurrence of the unknown in the equation thm.

                                                    Leon
                                                    Updated: 15 September 82
*/

solve(Eqn,X,Soln) <-- occ(X,Eqn,s(s(0))) & collect(Eqn,X,New)
        & position(X,New,P) & isolate(P,New,Soln)                    (1)

solve(Eqn,X,Soln) <-- position(X,Eqn,Posn)
                & isolate(Posn,Eqn,Soln) & single_occ(X,Eqn)         (2)

occ(X,Eqn,s(N)) <-- collect(Eqn,X,New) & occ(X,New,N)                (3)

single_occ(X,Eqn) <-- occ(X,Eqn,s(0))                                (4)

solve(Eqn,X,X=Ans) <-- free_of(X,Ans) & equiv(Eqn,X=Ans)             (5)

solve(Eqn,X,Soln) <-- equiv(Eqn,New) & solve(New,X,Soln)             (6)

equiv(Eqn,New) <-- collect(Eqn,X,New)                                (7)

Negating and skolemizing (1) gives

occ(x,eqn,s(s(0)))   <--                    hyp1
collect(eqn,x,new)   <--                    hyp2
position(x,new,p)    <--                    hyp3
isolate(p,new,soln) <--                     hyp4
<-- solve(eqn,x,soln)                       goal

Resolving goal with (6) gives

<-- equiv(eqn,New) & solve(New,x,soln)                               (R1)

Resolving (R1) with (2) gives

<-- equiv(eqn,New) & position(x,New,P) & isolate(P,New,soln)
                        & single_occ(x,New)                          (R2)

Resolving (R2) with hyp4 gives

<-- equiv(eqn,new) & position(x,new,p) & single_occ(x,new)           (R3)

Resolving (R3) with hyp3 gives

<-- equiv(eqn,new) & single_occ(x,new)                               (R4)

Resolving (R4) with (7) gives

<-- collect(eqn,X,new) & single_occ(x,new)                           (R5)

Resolving (R5) with (4) gives

<-- collect(eqn,X,new) & occ(x,new,s(0))                             (R6)

Resolving hyp1 forward wrt (3) gives

collect(eqn,x,New) <--  and occ(x,New,s(0)) <--
    the latter hyp1',
```

Link between unpacking of step 1 & step later with occurrence

Unit preference

Woody Bledsoe – reasoning against definitions (last resort)

PEEKING

Resolving (R6) with hyp2 gives

<-- occ(x,new,s(0)) (R7)

Resolving R7 with hyp1' gives <-- Hooray!

Note 58

Alan Bornins
3 December 1979

CONJECTURING AND PROVING NEW REWRITE RULES IN PRESS

This note is a collection of remarks on how a PRESS learning program might conjecture and prove new rewrite rules. The usual disclaimer: these ideas are in a preliminary state, and are incomplete and probably inconsistent.

The proposed organization of the program is as follows. As in the current PRESS, there would be sets of rewrite rules for isolation, collection, and attraction. (It might be useful to have some other sets as well.) To start things off, the program would be given an initial collection of rules, which it would automatically classify into the appropriate sets. The program would operate by conjecturing new rewrite rules, e.g. an isolation rule of the form "U+V=W --> U=...". Tasks would be generated to flesh out these conjectures into complete rules by selectively applying the existing rewrite rules to the filled-in part of the conjectured rule. The meta-level criteria for deciding which rewrite rules to apply would be different in the learning program from those in the current PRESS, since the present criteria are aimed at solving for an unknown. Also, a more powerful matcher could be employed (as in "Readings between the Lines" and Note 55).

Given this approach, among the questions that arise are: What should be the starting state of the program? What meta-level information should be used in deciding how to apply the existing rewrite rules? What powers are needed by the matcher? What control regime should the program use?

Starting State of the Program

Some possibilities for the starting state of the program are:

- Just the axioms of set theory (a la Lenat)
- the Peano Postulates
- the field axioms for real arithmetic, along with definitions of functions such as sin, cos, and log
- a fairly sophisticated set of rewrite rules, where the program would just fill in some gaps

I am inclined to use the "field axioms for real arithmetic, along with

function definitions" option. Here are some rationalizations for this choice: Giving the program less information seems to make its task too difficult. (Lenat's program discovered integers; expecting a program to also discover rationals and Dedikind cuts seems a bit much at this stage.) Starting with the field axioms seems feasible, although difficult. Just filling in some gaps might be easier, but wouldn't as interesting; also, judging the worth of the program would be more difficult -- how dependent would the performance of the program be on the particular choice of initial rules? However, it might be a good experimental set up during program development.

Suggesting Useful Conjectures

As described in Note 55, one way of suggesting new rewrite rules would be to generate all possible expressions in a best-first order, and try each expression as a possible left hand side of a rewrite rule. (One might want to try the expressions as right hand sides as well, applying existing rules to make them more complex.)

Another approach would be to give the learning program sample problems to solve, and have it generate new rules as the need arose.

Implementation of Expression Generators

As a digression from this handwaving, two versions of an expression generator that have actually been implemented will be described.

The first method uses an agenda mechanism. Each task on the agenda has an integer priority. The highest priority task is selected and removed from the agenda, and the predicate **do**(Task) is executed. The task is responsible for voluntarily giving up control at some point and allowing the agenda to start up another task. In the course of its execution, the given task may add new tasks to the agenda. Tasks are stored using the **record** predicate, where the key is the task's priority. The agenda keeps track of the current highest priority, so that it knows where to start looking for the next task to do. This priority is automatically updated as tasks are added and deleted.

In implementing the expression generator, three kinds of tasks were used: **expand**, **assign**, and **test** tasks. An expand task takes a variable in an expression and expands that variable into a subexpression. For example, 1+Var might be expanded to 1+(Var1*Var2). An assign task takes a variable and assigns an unknown or numeric constant to it. The priority mechanism is used to control the order in which the tasks are done. For example, assign tasks are initially given a high priority, so that some 'ground' expressions are generated; but the assignment of larger numeric constants are given lower priorities, so that the program doesn't simply generate the sequence "u, 0, 1, 2, 3, 4, 5, 6, ...".

Once all the variables in an expression have been assigned into, a test task is generated. Currently this task just prints a message on the screen.

A problem with the current implementation is that the number of tasks keeps increasing. An expand task picks a variable and expands it in all the ways that it knows about, substituting for it Var1+Var2, Var1*Var2, sin(Var), and so on. Each of the newly formed expressions is made into a task and added to the agenda. This constant increase in the number of tasks slows down the program as more expressions are generated. (Initially, the program was even slower, because it didn't give high enough priorities to finishing some assign tasks to get some output.) If this method is used, it should be changed so that there is always only one expand task that keeps track of the last expansion used.

The second generation method uses PROLOG backtracking, and is much simpler. (The initial program for this was designed and written by Fernando.) When **generate(E)** is called, E will be unified first with u; successive backtracking calls will produce more complex expressions. Each expression has a weight. The weight of a complex expression is the sum of the weight of its operator and the weights of each of its arguments. The weights of unknowns and the numeric constants 0, 1, and 2 are all 1; larger numeric constants are penalized with successively higher weights. First all expressions with weight 0 are generated, then all expressions with weight 1, and so on. To generate expressions with weight W, first all constants with this weight are generated; then all unary expressions of weight W, then all binary expressions of this weight. To generate e.g. all the binary expressions, a binary operator is selected. Then two non-negative integers that sum to W less the weight of the operator are chosen, and expressions of those two weights are found by a recursive call to **generate**. Successive backtracks produce new combinations of integers that sum to the appropriate number, then new operators.

Meta-Level Information for Generating Rules

Given an expression that might be the LHS or RHS of a new rewrite rule, how can plausible rules be conjectured and proven? Some methods:

- Try matching a more general rule (using the powerful matcher). For example, suppose we are trying to derive a collection rule for U*W+W. This expression could be matched against the distributive law to derive a new rule U*W+W --> (U+1)*W. A more complex example of the use of this method is given in Note 55.

- Evaluate expressions with constants. For example, we might derive the rule U+U --> 2*U using the distributive law:

U+U --> 1*U+1*U --> (1+1)*U --> 2*U

This method was used in changing 1+1 to 2.

- Use a weak form of attraction that allows the creation of extra terms. The exact nature of this rule isn't clear ... the desired effect is to allow e.g. the rule (U+V)*(U-V) --> U^2 - V^2 to be derived by a sequence of steps such as:

```
(U+V)*(U-V) --> U*(U-V) + V*(U-V)
      --> U*U - U*V + V*U - V*V
      --> U*U - V*V
      --> U^2 - V^2
```

- Use GPS/STRIPS style difference tables. I am viewing this as a very general but weak method that would be used as a last resort. (One might regard the isolation, collection, and attraction rules as more focussed versions of this method.) The difference table approach should probably be used only in deriving isolation rules, since for other kinds of rules not enough is known about the form of the RHS to allow the method to be applied effectively. An example of its use would be in deriving an isolation rule for U+V=W. The program would conjecture a rule of the form "U+V=W --> U=..." What is the difference between the two sides of the rewrite rule? The "..." on the right matches anything, so there is no problem with it. Using a bag representation, on the left of the equals sign we have an extra term "V". How can we get rid of an extra term from a + bag? The only known rule is U+0 --> U. What is the difference between U+V and U+0? The V has to be changed to a 0. What rewrite rule can do this? U+(-U) --> 0. So we try adding a (-V) to each side of the equation on the LHS, and after a bit more manipulation derive the rule "U+V=W --> U=W+(-V)". Wave, wave.

Capabilities of the Matcher

The powerful matcher described in "Reading between the Lines" would be adequate for the matching tasks that have arisen so far in thinking about the system. In fact, except for the trigonometric derivations, a less powerful matcher would be adequate -- one with PROLOG's capabilities, that also knew about the associative and commutative laws for addition and multiplication, and that could match 1*U or 0+U with U.

There would also have to be a kind of matcher to extract differences for use with the difference table method. However, this capability wouldn't be needed for the usual matching operation, and would be programmed into a separate procedure. Among the differences that the difference matcher should be able to detect are:

- Extra or missing term in a bag. This is the difference in the above example.

- Extra or missing function. For example, in matching "-U" and "U", the matcher should notice that there is an extra function "-" applied to "U".

Control Regimes

What control regime should the program use? The prime contenders are PROLOG backtracking and an agenda mechanism. Advantages of PROLOG backtracking are that it is easy to write programs that use it, and it is fast. Advantages of agendas are that they will allow the use of derivation methods that might never terminate. Also, there is some intrinsic interest in the use of agenda mechanisms.

^nother possible reason for using agendas is that some derivations may ve to wait for later results to be able to make any progress. This reason isn't very well thought out, but here goes. Consider the task of deriving an isolation rule from the conjecture "-U=V --> U=...". The difference table method would probably be used here. The rule "--U --> U" would be applicable. However, it might not exist when this derivation was first attempted ("-U" would be generated before "--U" by the expression generator). Therefore, the task to generate an isolation rule for "-U=V" would have to suspend itself and try again later.

This example leads to the observation that simple numerical priorities may not be the best way to organize an agenda ... it might be better for suspended tasks to have a reason indicating why they had been suspended. This could be stored in the form of a condition that, when satisfied, would permit the task to .be awakened. In the above example, the isolation rule task would have the condition that some rule for getting rid of minus signs existed. Of course, one reason for suspending a task might be "used up too many cpu cyles"; in this case the reactivation condition would be "more cycles available".

.ere are several reasons why this argument might not hold up. If the isolation rule for "-U=V" can be derived using the rule for "--U", and the "--U" rule can be derived from the starting rules, then the isolation rule can be derived directly. (However, the proof strategy used may not be complete, so the derivation wouldn't be found.) Another objection is that there is still no need to use agendas ... the program should simply set up a subtask of deriving some rule for getting rid of minus signs. Any useful rule discovered in the process would be saved. (However, how would the program know the form of the needed rule?)

Initial Set of Rewrite Rules for the Program

Below are listed some candidates for the initial set of rewrite rules for the learning program, which embody some basic laws for real arithmetic and trigonometry. Many rules in this initial set can be used in either direction; this has been indicated by the characters "<-->".

Addition

Commutative Law	r+s <--> s+r
Associative Law	r+(s+t) <--> (r+s)+t
Additive Identity	r+0 <--> r
Additive Inverse	r+(-r) <--> 0

Multiplication

Commutative Law	r*s <--> s*r
Associative Law	r*(s*t) <--> (r*s)*t
Multiplicative Identity	1*r <--> r
Multiplicative Inverse	if r=\=0 then r*(r^-1) <--> 1
Distributive Law	r*(s+t) <--> r*s+r*t

Exponents and Logarithms

```
r^m*r^n  <-->  r^(m+n)
r^1  <-->  r
r^0  <-->  1      (redundant?)
r^(-n)  <-->  (r^n)^-1      (redundant?)
log(u,v)=w  <-->  v=u^w
```

Trig Symbols

```
tan(u)  <-->  sin(u)/cos(u)
cosec(u)  <-->  1/sin(u)
sec(u)  <-->  1/cos(u)
cot(u)  <-->  1/tan(u)

sin(-u)  <-->  -sin(u)
cos(-u)  <-->  cos(u)

arcsin(x) = theta  <-->  theta=sin(x) & -90<=theta<=90
arccos(x) = theta  <-->  theta=cos(x) & 0<=theta<=180
arctan(x) = theta  <-->  theta=tan(x) & -90<theta<90

sin(a+b)  <-->  sin(a)*cos(b) + cos(a)*sin(b)
```

Rules for Equality?

```
x=x
```

```
if x=y then y=x
if x=y and y=z then x=z
for every function f, if x=y then f(x)=f(y)
```

Evaluation?

```
1+1  <-->  2 etc.
sin(0)  <-->  0 etc.
```

Rules for Logic-Valued Expressions?

```
true & X  <-->  X
false & X  <-->  false
etc.
```

Notes on the Choice of Initial Rewrite Rules

Commutivity and associativity of addition and multiplication will probably be built into the matcher. Would it also be useful to have declarative representations of these in a rewrite rule set?

Exponentiation ... the standard approach is first to define U^N for a positive integer n, then prove U^N*U^M --> U^(N+M) using induction. Then one insists that the rule hold for all rationals, and proceeds from there. I have put in U^N*U^M as an axiom. (What about irrational exponents?)

The rule for sin(a+b) is a rather sophisticated one. However, its derivation from the law of cosines or from the Pythagorean Theorem requires the use of geometric knowledge; for simplicity, it seems better for the moment not to insist that the program have geometric knowledge as well.

Possible Methodology for Program Development

A possible methodology for program development would be to check out generate and test parts of program separately. After both parts were working, the entire program could be compiled and left to run over the weekend.

Given that agendas are harder to program than backtracking, even if agendas are eventually to be used, it might be worthwhile first to write the program using backtracking, and convert it to use agendas after it was debugged.

```
**********************************
* PROLOG CROSS REFERENCE LISTING *
**********************************

Alan Bornins's Asenda Based Expression Generator
```

PREDICATE	FILE	CALLED BY
add_task(1)	asenda.	do(1) do(1)
concat(3)	utility	unknown(2)
constant(3)	subst.	do(1)
do(1)	check.	
do(1)	bind.	
do(1)	expand.	run(0)
expression(1)	expand.	do(1)
flag(3)	utility	set_task(1) new_priority(1)
set_task(1)	asenda.	run(0) set_task(1)
new_priority(1)	asenda.	add_task(1)
priority(2)	check.	
priority(2)	bind.	
priority(2)	expand.	add_task(1)
run(0)	asenda.	
subst_first(4)	subst.	subst_first(4) do(1) do(1)
trace(3)	utility	add_task(1) new_priority(1)
unknown(2)	subst.	constant(3)
writef(2)	utility	do(1)

LEARN

```
/* File in procedures for PRESS learning program
   Alan Borning's Agenda Based Conjecture Maker
   Use with MUTIL */

:- [agenda, expand, bind, subst, check].

:- flag(max_priority,_,0).

:- [test].
```

AGENDA

```
/* PROCEDURES TO MANAGE THE AGENDA */

/* The agenda is represented using the PROLOG internal data base
   predicates.  A task for a given priority P is recorded using
   P as its key.  The global variable "max_priority" holds the highest
   priority of all the current tasks.  The purpose of the kludgery is
   to prevent a recursive call each time a task is run. */


run :-
   repeat,
   set_task(Task),
   do(Task),
   fail.


set_task(Task) :-
   flag(max_priority,P,P),
   recorded(P,Task,ID),
   erase(ID),
   !.

set_task(nothing) :-
   flag(max_priority,0,0),
   !.

set_task(Task) :-
   flag(max_priority,P,P),
   P1 is P-1,
   flag(max_priority,_,P1),
   set_task(Task),
   !.


add_task(Task) :-
   trace('adding task %t to agenda\n',[Task],3),
   priority(Task,P),
   recordz(P,Task,_),
   new_priority(P).


new_priority(P) :-
   flag(max_priority,M,M),
   P > M, !,
   trace('changing max priority to %t\n',[P],3),
   flag(max_priority,_,P).

new_priority(P) :- !.
```

EXPAND

```
/* PROCEDURES THAT IMPLEMENT TASKS FOR EXPANDING EXPRESSIONS */

/* An "expand" task is represented using the following predicate:
                expand(priority,expression)
  */


/* ACCESS TO PARTS */

priority( expand(Priority,_), Priority).


/* PROCEDURES FOR DOING EXPAND TASK */

do( expand(Priority,Expr) ) :-
  /* first add a "bind" task to bind all vars in the existing expression */
   add_task( bind(3,Expr,1,0) ),
   /* now get each expression, substitute it in, and add a new expand
      task for the newly expanded expression */
   expression(E),
   subst_first(var,E,Expr,New_Expr),
   add_task( expand(2,New_Expr) ),
   fail.


/* all possible expansions ... */

expression( -var ).
expression( sin(var) ).

expression( var+var ).
expression( var*var ).
```

BIND

```
/* PROCEDURES THAT IMPLEMENT TASKS FOR BINDING VARIABLES IN EXPRESSIONS */

/* A "bind" task is represented using the following predicate:
                bind(priority,expression,var_index,last_bound)
   priority is the integer priority of the task
   expression is the current expression.  Remaining variables are
      represented by the atom 'var' in the expression.
   var_index is an integer -- the number of the next variable in the original
      expression that is being bound in this task
   last_bound is another integer, which indicates the last thing bound to
      the next occurence of 'var'.  This is used to decide which constant
      or unknown to substitute for 'var'. */


/* ACCESS TO PARTS */

priority( bind(P,_,_,_), P).



/* PROCEDURE FOR PERFORMING THE "BIND" TASK */

do( bind(Priority,Expr,Var_Index,Last_Bound) ) :-
   I is Last_Bound+1,
   (I>2 -> P1 is 1 ; P1 is 3),
   constant(I,Var_Index,C),
   /* if the following "subst_first" fails, then there are no
      more occurences of 'var' */
   subst_first(var,C,Expr,New_Expr),
   /* Success -- insert two new tasks in the agenda:
      one is a new "bind" task with the new value substituted
      for 'var'; the other is another "bind" task with
      an incremented value for Last_Bound */
   V1 is Var_Index+1,
   add_task( bind(P1,New_Expr,V1,0) ),
   add_task( bind(P1,Expr,Var_Index,I) ),
   !.

do( bind(Priority,Expr,_,_) ) :-
   /* there are no more variables to be bound -- add a
      "check_sat" task to the agenda */
   P1 is Priority+1,
   add_task( check_sat(P1,Expr) ),
   !.
```

```
/* PROCEDURES TO SUBSTITUTE NEW VARIABLE NAMES
   AND GENERATE NAMES FOR UNKNOWNS */


/* subst_first(Old,New,Old_Expr,New_Expr) substitutes New for
   the first occurence of Old in Old_Expr and returns the result
   in New_Expr */

subst_first(Old,New,Old,New) :- !.

subst_first(Old,New,Old_Expr,New_Expr) :-
   atomic(Old_Expr), !,
   fail.

subst_first(Old,New,[H|T],[NH|T]) :-
   subst_first(Old,New,H,NH), !.

subst_first(Old,New,[H|T],[H|NT]) :-
   subst_first(Old,New,T,NT), !.

subst_first(Old,New,[H|T],[H|T]) :-
   !, fail.

subst_first(Old,New,Old_Expr,New_Expr) :-
   Old_Expr=..[P|Args],
   subst_first(Old,New,Args,New_Args),
   New_Expr=..[P|New_Args].



/* constant(I,Max_Unknown,C) returns the Ith canonical constant in C,
   given the maximum unknown wanted in Max_Unknown */

constant(I,Max_Unknown,C) :-
   I =< Max_Unknown,
   unknown(I,C).

constant(I,Max_Unknown,true) :-
   I =:= Max_Unknown+1.

constant(I,Max_Unknown,false) :-
   I =:= Max_Unknown+2.

constant(I,Max_Unknown,Int) :-
   I > Max_Unknown+2,
   Int is I-Max_Unknown-3.



/* unknown(I,Name) returns a canonical name for the Ith unknown */

unknown(1,u).
unknown(2,v).
unknown(3,w).
unknown(4,x).
unknown(5,y).
unknown(6,z).

unknown(I,Name) :-
   I>6,
```

```
J is I-6,
concat(u,J,Name).
```

CHECK.

```
/* PROCEDURES THAT IMPLEMENT THE TESTING OF EXPRESSIONS */

/* a "check_sap" task is represented by the predicate
                check_sap(priority,expression)
 */



/* ACCESS TO PARTS */

priority( check_sap(Priority,_), Priority).



/* PERFORMING TASKS */

do( check_sap(Priority,Expr) ) :-
   writef('checking expression %t \n',[Expr]),
   !.


  ' nothing ) :-
  writef('nothing to do\n',[]),
   !.
```

TEST

```
:- add_task( expand(1,var) ).

:- tlim(5).
```

Agenda Based Output

```
yes
| ?- run.
adding task bind(3,var,1,0) to agenda
changing max priority to 3
adding task expand(2,-var) to agenda
adding task expand(2,sin(var)) to agenda
adding task expand(2,var+var) to agenda
adding task expand(2,var*var) to agenda
adding task check_sap(4,var) to agenda
changing max priority to 4
checking expression var
adding task bind(3,-var,1,0) to agenda
changing max priority to 3
adding task expand(2,-(-var)) to agenda
adding task expand(2,-sin(var)) to agenda
adding task expand(2,-(var+var)) to agenda
adding task expand(2,-var*var) to agenda
adding task check_sap(4,-var) to agenda
changing max priority to 4
checking expression -var
  ding task bind(3,sin(var),1,0) to agenda
  anging max priority to 3
adding task expand(2,sin(-var)) to agenda
adding task expand(2,sin(sin(var))) to agenda
adding task expand(2,sin(var+var)) to agenda
adding task expand(2,sin(var*var)) to agenda
adding task check_sap(4,sin(var)) to agenda
changing max priority to 4
checking expression sin(var)
adding task bind(3,var+var,1,0) to agenda
changing max priority to 3
adding task expand(2,-var+var) to agenda
adding task expand(2,sin(var)+var) to agenda
adding task expand(2,var+var+var) to agenda
adding task expand(2,var*var+var) to agenda
adding task check_sap(4,var+var) to agenda
changing max priority to 4
checking expression var+var
adding task bind(3,var*var,1,0) to agenda
  anging max priority to 3
  ding task expand(2,(-var)*var) to agenda
adding task expand(2,sin(var)*var) to agenda
adding task expand(2,(var+var)*var) to agenda
adding task expand(2,var*var*var) to agenda
adding task check_sap(4,var*var) to agenda
changing max priority to 4
checking expression var*var
adding task bind(3,-(-var),1,0) to agenda
changing max priority to 3
adding task expand(2,-(-(-var))) to agenda
adding task expand(2,-(-sin(var))) to agenda
adding task expand(2,-(-(var+var))) to agenda
adding task expand(2,-(-var*var)) to agenda
adding task check_sap(4,-(-var)) to agenda
changing max priority to 4
checking expression -(-var)
adding task bind(3,-sin(var),1,0) to agenda
changing max priority to 3
adding task expand(2,-sin(-var)) to agenda
```

```
adding task expand(2,-sin(sin(var))) to agenda
adding task expand(2,-sin(var+var)) to agenda
adding task expand(2,-sin(var*var)) to agenda
adding task check_sap(4,-sin(var)) to agenda
changing max priority to 4
checking expression -sin(var)
adding task bind(3,-(var+var),1,0) to agenda
changing max priority to 3
adding task expand(2,-(-var+var)) to agenda
adding task expand(2,-(sin(var)+var)) to agenda
adding task expand(2,-(var+var+var)) to agenda
adding task expand(2,-(var*var+var)) to agenda
adding task check_sap(4,-(var+var)) to agenda
changing max priority to 4
checking expression -(var+var)
adding task bind(3,-var*var,1,0) to agenda
changing max priority to 3
adding task expand(2,-(-var)*var) to agenda
adding task expand(2,-sin(var)*var) to agenda
adding task expand(2,-(var+var)*var) to agenda
adding task expand(2,-var*var*var) to agenda
  ding task check_sap(4,-var*var) to agenda
  anging max priority to 4
checking expression -var*var
adding task bind(3,sin(-var),1,0) to agenda
changing max priority to 3
adding task expand(2,sin(-(-var))) to agenda
adding task expand(2,sin(-sin(var))) to agenda
adding task expand(2,sin(-(var+var))) to agenda
adding task expand(2,sin(-var*var)) to agenda
adding task check_sap(4,sin(-var)) to agenda
changing max priority to 4
checking expression sin(-var)
adding task bind(3,sin(sin(var)),1,0) to agenda
changing max priority to 3
adding task expand(2,sin(sin(-var))) to agenda
adding task expand(2,sin(sin(sin(var)))) to agenda
adding task expand(2,sin(sin(var+var))) to agenda
adding task expand(2,sin(sin(var*var))) to agenda
adding task check_sap(4,sin(sin(var))) to agenda
  anging max priority to 4
  ecking expression sin(sin(var))
adding task bind(3,sin(var+var),1,0) to agenda
changing max priority to 3
adding task expand(2,sin(-var+var)) to agenda
adding task expand(2,sin(sin(var)+var)) to agenda
adding task expand(2,sin(var+var+var)) to agenda
adding task expand(2,sin(var*var+var)) to agenda
adding task check_sap(4,sin(var+var)) to agenda
changing max priority to 4
checking expression sin(var+var)
adding task bind(3,sin(var*var),1,0) to agenda
changing max priority to 3
adding task expand(2,sin((-var)*var)) to agenda
adding task expand(2,sin(sin(var)*var)) to agenda
adding task expand(2,sin((var+var)*var)) to agenda
adding task expand(2,sin(var*var*var)) to agenda
adding task check_sap(4,sin(var*var)) to agenda
changing max priority to 4
checking expression sin(var*var)
```

```
adding task bind(3,-var+var,1,0) to agenda
changing max priority to 3
adding task expand(2,-(-var)+var) to agenda

[ Break  (level 1) ]
| ?- [ End Break  (level 1) ]
adding task expand(2,-sin(var)+var) to agenda
adding task expand(2,-(var+var)+var) to agenda
adding task expand(2,-var*var+var) to agenda
adding task check_sap(4,-var+var) to agenda
changing max priority to 4
checking expression -var+var
adding task bind(3,sin(var)+var,1,0) to agenda
changing max priority to 3
adding task expand(2,sin(-var)+var) to agenda
adding task expand(2,sin(sin(var))+var) to agenda
adding task expand(2,sin(var+var)+var) to agenda
adding task expand(2,sin(var*var)+var) to agenda
adding task check_sap(4,sin(var)+var) to agenda
changing max priority to 4
checking expression sin(var)+var
  ding task bind(3,var+var+var,1,0) to agenda
  anging max priority to 3
adding task expand(2,-var+var+var) to agenda
adding task expand(2,sin(var)+var+var) to agenda
adding task expand(2,var+var+var+var) to agenda
adding task expand(2,var*var+var+var) to agenda
adding task check_sap(4,var+var+var) to agenda
changing max priority to 4
checking expression var+var+var
adding task bind(3,var*var+var,1,0) to agenda
changing max priority to 3
adding task expand(2,(-var)*var+var) to agenda
adding task expand(2,sin(var)*var+var) to agenda
adding task expand(2,(var+var)*var+var) to agenda
adding task expand(2,var*var*var+var) to agenda
adding task check_sap(4,var*var+var) to agenda
changing max priority to 4
checking expression var*var+var
adding task bind(3,(-var)*var,1,0) to agenda
  anging max priority to 3
  ding task expand(2,-(-var)*var) to agenda
adding task expand(2,(-sin(var))*var) to agenda
adding task expand(2,-(var+var)*var) to agenda
adding task expand(2,(-var*var)*var) to agenda
adding task check_sap(4,(-var)*var) to agenda
changing max priority to 4

[ Break  (level 1) ]
| ?- abort.
[ Execution aborted ]

| ?- core     50176  (20992 lo-segs + 29184 hi-segs)
heap     15872 =  13307 in use +   2565 free
global    1175 =     16 in use +   1159 free
local     1024 =     16 in use +   1008 free
trail      511 =      0 in use +    511 free
    0.01 sec. for 1 GCs gaining 379 words
    0.01 sec. for 1 local shift and 1 trail shift
    3.56 sec. runtime
```

```
**********************************
* PROLOG CROSS REFERENCE LISTING *
**********************************

        Alan Bornins's Backtrackins Based Generator
```

PREDICATE	FILE	CALLED BY
binary(2)	tree	expression(4)
concat(3)	utility	unknwn(2)
=ression(2)	tree	
expression(4)	tree	senerate(2) expression(4)
senerate(1)	tree	so(0)
senerate(2)	tree	senerate(1) senerate(2)
so(0)	tree	
losical(2)	tree	expression(2)
number(2)	tree	expression(2)
sum(3)	tree	expression(4) sum(3)
unary(2)	tree	expression(4)
unknown(4)	tree	expression(4) unknown(4)
..knwn(2)	tree	unknown(4)

TREES

```
/* Generate all possible expressions by backtracking.

   Each expression is given a weight.  First the weight 1 expressions
   are generated; then the weight 2; and so on.  The weight of an
   atomic expression is defined by the clauses for "unknwn",
   "number", and "logical".  The weight of a complex expression is the
   sum of the weight for its operator and the weights of each argument.

   The arguments "Old_Max" and "New_Max" in the various clauses are used
   to control the generation of unknowns, to avoid generating expressions
   that are equivalent except for different names for unknowns.
   "Old_Max" is passed in, and indicates that unknowns up to the
   Old_Max-th one can be used.  "New_Max" is then returned. */


generate(Expr) :- generate(1,Expr).

generate(Weight,Expr) :- expression(Weight,Expr,1,_).
generate(Weight,Expr) :-
   W1 is Weight+1,
   generate(W1,Expr).

expression(Weight,Expr,Old_Max,New_Max) :-
   unknown(Weight,Expr,Old_Max,New_Max).
expression(Weight,Expr) :- number(Weight,Expr).
expression(Weight,Expr) :- logical(Weight,Expr).

expression(Weight,Expr,Old_Max,New_Max) :-
   /* hack to speed things up for small weights -- given
      current weights, this can't succeed for Weight<2 */
   Weight >= 2,
   unary(Op,Op_Weight),
   W1 is Weight - Op_Weight,
   W1 >= 1,
   expression(W1,E1,Old_Max,New_Max),
   Expr =.. [Op,E1].

expression(Weight,Expr,Old_Max,New_Max) :-
   Weight >= 3,
   binary(Op,Op_Weight),
   W1 is Weight - Op_Weight,
   W1 >= 1,
   sum(I,J,W1),
   expression(I,E1,Old_Max,Max1),
   expression(J,E2,Max1,New_Max),
   Expr =.. [Op,E1,E2].


/* sum(I,J,N) produces by backtracking all I and J that sum to N */

sum(N,0,N).
sum(I,J,N) :-
   N>0,
   N1 is N-1,
   sum(I,J1,N1),
   J is J1+1.
```

```
unknown(1,Expr,Old_Max,New_Max) :-
    unknwn(Old_Max,Expr),
    New_Max is Old_Max+1.

unknown(1,Expr,Old_Max,Old_Max) :-
    Old_Max>1,
    J is Old_Max-1,
    unknown(1,Expr,J,_).


/* unknwn(I,Name) returns a canonical name for the Ith unknown */

unknwn(1,u) :- !.
unknwn(2,v) :- !.
unknwn(3,w) :- !.
unknwn(4,x) :- !.
unknwn(5,y) :- !.
unknwn(6,z) :- !.

  knwn(I,Name) :-
    !,
    I>6,
    J is I-6,
    concat(u,J,Name).



/* The weighting for constants gives preference to 0,1, and 2;
    numbers greater than 2 are heavily penalized. */

number(1,0).
number(1,1).
number(1,2).

number(N,N) :-
    N>2.

logical(1,true).
  ical(1,false).



unary(-,2).
unary(sin,2).
unary(cos,2).
unary(tan,4).

binary(+,1).
binary(*,1).
binary(^,4).
binary(=,2).


go :- generate(E), write(E), nl, fail.
```

Backtracking Based Output

```
yes
| ?- [tree].

tree consulted   726 words      0.28 sec.

yes
| ?- so.
u
-u
sin(u)
cos(u)
u+v
u+u
u*v
u*u
u=v
u=u
-(-u)
-sin(u)
-cos(u)
   +v)
 .u+u)
-u*v
-u*u
sin(-u)
sin(sin(u))
sin(cos(u))
sin(u+v)
sin(u+u)
sin(u*v)
sin(u*u)
cos(-u)
cos(sin(u))
cos(cos(u))
cos(u+v)
cos(u+u)
cos(u*v)
cos(u*u)
tan(u)
  +v
 u+u
sin(u)+v
sin(u)+u
cos(u)+v
cos(u)+u
u+v+w
u+v+v
u+v+u
u+u+v
u+u+u
u*v+w
u*v+v
u*v+u
u*u+v
u*u+u
u+(-v)
u+(-u)
u+sin(v)
u+sin(u)
```

```
u+cos(v)
u+cos(u)
u+(v+w)
u+(v+v)
u+(v+u)
u+(u+v)
u+(u+u)
u+v*w
u+v*v
u+v*u
u+u*v
u+u*u
(-u)*v
(-u)*u
sin(u)*v
sin(u)*u
cos(u)*v
cos(u)*u
(u+v)*w
(u+v)*v
(u+v)*u
  u)*v
 +u)*u
u*v*w
u*v*v
u*v*u
u*u*v
u*u*u
u*(-v)
u*(-u)
u*sin(v)
u*sin(u)
u*cos(v)
u*cos(u)
u*(v+w)
u*(v+v)
u*(v+u)
u*(u+v)
u*(u+u)
u*(v*w)
  v*v)
 (v*u)
u*(u*v)
u*(u*u)
-(u=v)
-(u=u)
sin(u=v)
sin(u=u)
cos(u=v)
cos(u=u)
(u=v)+w
(u=v)+v
(u=v)+u
(u=u)+v
(u=u)+u
u+(v=w)
u+(v=v)
u+(v=u)
u+(u=v)
u+(u=u)
```

```
(u=v)*w
(u=v)*v
(u=v)*u
(u=u)*v
(u=u)*u
u*(v=w)
u*(v=v)
u*(v=u)
u*(u=v)
u*(u=u)
u^v
u^u
-u=v
-u=u
sin(u)=v
sin(u)=u
cos(u)=v
cos(u)=u
u+v=w
u+v=v
u+v=u
  u=v
  u=u
u*v=w
u*v=v
u*v=u
u*u=v
u*u=u
u= -v
u= -u
u=sin(v)
u=sin(u)
u=cos(v)
u=cos(u)
u=v+w
u=v+v
u=v+u
u=u+v
u=u+u
u=v*w
  v*v
  v*u
u=u*v
u=u*u
-(-(-u))
-(-sin(u))
-(-cos(u))
-(-(u+v))
-(-(u+u))
-(-u*v)
-(-u*u)
-sin(-u)
-sin(sin(u))
-sin(cos(u))
-sin(u+v)
-sin(u+u)
-sin(u*v)
-sin(u*u)
-cos(-u)
-cos(sin(u))
```

```
-cos(cos(u))
-cos(u+v)
-cos(u+u)
-cos(u*v)
-cos(u*u)
-tan(u)
-(-u+v)
-(-u+u)
-(sin(u)+v)
-(sin(u)+u)
-(cos(u)+v)
-(cos(u)+u)
-(u+v+w)
-(u+v+v)
-(u+v+u)
-(u+u+v)
-(u+u+u)
-(u*v+w)
-(u*v+v)
-(u*v+u)
-(u*u+v)
  u*u+u)
  u+(-v))
-(u+(-u))
-(u+sin(v))
-(u+sin(u))
-(u+cos(v))
-(u+cos(u))
-(u+(v+w))
-(u+(v+v))
-(u+(v+u))
-(u+(u+v))
-(u+(u+u))
-(u+v*w)
-(u+v*v)
-(u+v*u)
-(u+u*v)
-(u+u*u)
-(-u)*v
-(-u)*u
  in(u)*v
 _in(u)*u
-cos(u)*v
-cos(u)*u
-(u+v)*w
-(u+v)*v
-(u+v)*u
-(u+u)*v
-(u+u)*u
-u*v*w
-u*v*v
-u*v*u
-u*u*v
-u*u*u
-u*(-v)
-u*(-u)
-u*sin(v)
-u*sin(u)
-u*cos(v)
-u*cos(u)
```

```
-u*(v+w)
-u*(v+v)
-u*(v+u)
-u*(u+v)
-u*(u+u)
-u*(v*w)
-u*(v*v)
-u*(v*u)
-u*(u*v)
-u*(u*u)
sin(-(-u))
sin(-sin(u))
sin(-cos(u))
sin(-(u+v))
sin(-(u+u))
sin(-u*v)
sin(-u*u)
sin(sin(-u))
sin(sin(sin(u)))
sin(sin(cos(u)))
sin(sin(u+v))
  ⁿ(sin(u+u))
  ⌟n(sin(u*v))
sin(sin(u*u))
sin(cos(-u))
sin(cos(sin(u)))
sin(cos(cos(u)))
sin(cos(u+v))
sin(cos(u+u))
sin(cos(u*v))
sin(cos(u*u))
sin(tan(u))
sin(-u+v)
sin(-u+u)
sin(sin(u)+v)
sin(sin(u)+u)
sin(cos(u)+v)
sin(cos(u)+u)
sin(u+v+w)
sin(u+v+v)
   ⁿ(u+v+u)
  ⌟n(u+u+v)
sin(u+u+u)
sin(u*v+w)
sin(u*v+v)
sin(u*v+u)
sin(u*u+v)
sin(u*u+u)
sin(u+(-v))
sin(u+(-u))
sin(u+sin(v))
sin(u+sin(u))
sin(u+cos(v))
sin(u+cos(u))
sin(u+(v+w))
sin(u+(v+v))
sin(u+(v+u))
sin(u+(u+v))
sin(u+(u+u))
sin(u+v*w)
```

```
sin(u+v*v)
sin(u+v*u)
sin(u+u*v)
sin(u+u*u)
sin((-u)*v)
sin((-u)*u)
sin(sin(u)*v)
sin(sin(u)*u)
sin(cos(u)*v)
sin(cos(u)*u)
sin((u+v)*w)
sin((u+v)*v)
sin((u+v)*u)
sin((u+u)*v)
sin((u+u)*u)
sin(u*v*w)
sin(u*v*v)
sin(u*v*u)
sin(u*u*v)
sin(u*u*u)
sin(u*(-v))
   ?(u*(-u))
  .n(u*sin(v))
sin(u*sin(u))
sin(u*cos(v))
sin(u*cos(u))
sin(u*(v+w))
sin(u*(v+v))
sin(u*(v+u))
sin(u*(u+v))
sin(u*(u+u))
sin(u*(v*w))
sin(u*(v*v))
sin(u*(v*u))
sin(u*(u*v))
sin(u*(u*u))
cos(-(-u))
cos(-sin(u))
cos(-cos(u))
cos(-(u+v))
   .(-(u+u))
  ./s(-u*v)
cos(-u*u)
cos(sin(-u))
cos(sin(sin(u)))
cos(sin(cos(u)))
cos(sin(u+v))
cos(sin(u+u))
cos(sin(u*v))
cos(sin(u*u))
cos(cos(-u))
cos(cos(sin(u)))
cos(cos(cos(u)))
cos(cos(u+v))
cos(cos(u+u))
cos(cos(u*v))
cos(cos(u*u))
cos(tan(u))

: Break  (level 1) ]
```

```
| ?- abort.
[ Execution aborted ]

| ?- core     48128  (18944 lo-ses + 29184 hi-ses)
heap     13824 =  12097 in use +    1727 free
global    1175 =     16 in use +    1159 free
local     1024 =     16 in use +    1008 free
trail      511 =      0 in use +     511 free
    0.00 sec. for 1 trail shift
   11.25 sec. runtime
XREF Prolog Cross Referencer  (6 Jul 81)
XREF Prolog Cross Referencer  (6 Jul 81)
```

The use of Lemmas in Proving Theorems

13/1/83

(Assuming IMPRESS styles)

One of the problems Richard posed IMPRESS was the transitivity of division (TD) i.e.

divides (X, Z) ← divides (X, Y) & divides (Y, Z).

Using his axiomatization of divides, i.e.
divides (X, 0) ← and divides (X, Z) ← divides (X, Y) & plus (X, Y, Z)

TD is hard to prove.

The definition of divides in algebra is

a | b if ∃q s.t. bc = qa.

divides (X, Z) ← times (X, W, Z)

Using this definition to expand TD, we get as the theorem to prove

~~divides~~

times (X, W, Z) ← times (X, V, Y) & times (Y, U, Z).

This can be proved using the associativity of times as a lemma i.e.

times (X, YZ, XYZ) ← times (X, Y, XY) & times (Y, Z, YZ) & times (XY, Z, XYZ).

The function properties of times can then be used to help prove the theorem with our 'normal refutation procedure'. A much more elegant solution can be found by using the lemma 'forwards'.

The technique is to use the hypotheses of the theorem to produce the conclusion of the lemma. Let us see an example.

Negating + skolemizing the theorem gives

- times $(x, V, y) \leftarrow$ and times $(y, W, z) \leftarrow$ as hypotheses + ~~to~~ $\leftarrow$ times $(x, W z)$ as the conclusion.

~~Now~~ The hypotheses only match the conditions of associativity in one way, & i.e. $x \to X$, $y \to XY$, $z \to XYz$.

Using the function properties of times gives V as $v(x,y)$ & W as $w(y,z)$, which in the matching corresponds to

$$v(x,y) \to Y \text{ and } w(y,z) \to Z$$

The lemma has become

$$\text{times}(x, YZ, xz) \leftarrow \text{times}(v(x,y), w(y,z), YZ).$$

(~~This~~ is precisely what is needed in the 'mathematical' proof of the theorem, which goes as follows.

"$z = xyw$, and $y = xv \Rightarrow z = xvw$ & the times property holds for x, z.")

The proof then proceeds with

$\leftarrow$ times (x, Wz, z) resolving against the new lemma to give $\leftarrow$ times $(v(x,y), w(y,z), Yz)$ which is true due to the function properties of times, again.

divides (s(s(0)), X) ← even (X)

← divides (s(s(0)), x) { even (x)

← times (A, s(s(0))), x)

← times (s(s(0)), A, x)

← times (s(0), A, Y) & plus (Y, A, x)

← times (0, A, W) & plus (W, A, X) & plus (X, A, x)

← times (0, A, 0)

Induction :- divides (s(s(0))), even(0) o K

Then we don't go

divides (s(s(x)), s(s(x))) ← divides ( )

How to do twice? etc.

---

lesseq (A, c) ← lesseq (A, B) & less eq (B, c)

← lesseq (a, c)

(← plus (X, a, c))

lesseq (a, b) ← plus (a, F(a,b), b)

lesseq (b, c) ← plus (b, F(b,c), c)

comm. heuristic fails No!

(Use to ~~separate~~ evaluate first)

[← plus (X, Y, W) & [plus (X, Z, a)] & plus (XY, Z, c)]

work on most ground first.

plus (c,

← plus (a, X, c)

← plus (a, Y, XY) & plus (Y, Z, X) & plus (XY, Z, c)

F(a,b), b F(a,b), F(b,c), g(b, F(b,c), c)

↓

fn. properties

Alan,

Here are some typical theorems one might want to prove in an elementary number theory. The axioms assumed are also given. It is certainly true that guidance is necessary to avoid the infinite loops inherent when introducing commutative and associative properties. The heuristics being developed seem to ensure no loopins occurs.

The only theorem I can7t prove is

even(Z) <-- plus(X,Y,Z) & even(X) & even(Y).

This seems to need mutual induction on X and Y and that is not immediately expressible as the axioms are now. What is certainly true is that proof plans do seem to happen, but the ranse of induction schemes are by no means covered with the primitive form currently in IMPRESS.

TEST THEOREMS

divides(s(0),X) <-- (1)

divides(s(s(0)),X) <-- even(X) (2)

divides(X,Z) <-- divides(X,Y) & divides(Y,Z) (3)

lessea(X,X) <-- (4)

lessea(X,Z) <-- lessea(X,Y) & lessea(Y,Z) (5)

even(Z) <-- plus(X,Y,Z) & even(X) & even(Y) (6)

even(Z) <-- times(X,Y,Z) & even(X) (7)

PROOFS

(1) by construction

After skolemization

<-- divides(s(0),x)

Resolvins with existence definition

<-- times(A,s(0),x)

" " commutativity

<-- times(s(0),A,x)

" " recursive definition of times

<-- times(0,A,W) & plus(W,A,x)

" " "

<-- plus(0,A,x)

" " plus

<--

(1) by induction

Base Case: <-- divides(0,0)

Immediately resolves (!)

<--

Step Case: divides(s(0),num) <-- induction hypothesis

<-- divides(s(0),s(num)) soal

Note that our induction scheme doesn't cover this example. For example, what would a program hypothesis be in this case. In fact doins these examples brousht home how limited the currently implemented induction scheme is. The readins of Boyer-Moore sussests many extensions, as does the

proof of these theorems. Nonetheless, continuing with virtual induction schemes throughout, ...,

```
                        Unfolding
        <-- divides(s(0),X) & plus(X,s(0),s(num)
                        Using the skolemized hypothesis
        <-- plus(num,s(0),s(num))
                        Using commutativity
        <-- plus(s(0),num,s(num))
                        Evaluating
        <--
```

(2) by construction

```
even(x) <--

        <-- divides(s(s(0)),x)
                Existence definition
        <-- times(A,s(s(0)),x)
                Commutativity
        <-- times(s(s(0)),A,x)
                Evaluating
        <-- times(s(0),A,W) & plus(W,A,x)
                Evaluating
        <-- times(0,A,V) & plus(V,A,W) & plus(W,A,x)
                Evaluating
        <-- plus(0,A,W) & plus(W,A,x)
                Evaluating
        <-- plus(A,A,x)
                Using skolemized hypothesis
        <--
```

(2) by induction

Base case: <-- divides(s(s(0)),0) Immediately proven.

Again, choosing the induction scheme is difficult. The best one here is to instantiate with s(s(num)) and assume the induction hypothesis for num

```
Step case:  divides(s(s(0)),num) <-- even(num)  induction hypothesis
            even(s(s(num))) <--                 skolemized hypothesis
            <-- divides(s(s(0)),s(s(num)))
                        Unfolding
            <-- divides(s(s(0)),X) & plus(s(s(0)),X,s(s(num)))
                        Using inductionhypothesis
            <-- even(num) & plus(s(s(0)),num,s(s(num)))
                        Using skolemized hypothesis
            <-- plus(s(s(0)),num,s(s(num)))
                        Evaluating
            <--
```

Axioms For Elementary Number Theory

RECURSIVE DEFINITIONS

```
plus(0,X,X) <--
plus(s(X),Y,s(Z)) <-- plus(X,Y,Z)

lesseq(0,X) <--
lesseq(s(X),s(Y)) <-- lesseq(X,Y)

times(0,X,0) <--
times(s(X),Y,Z) <-- times(X,Y,W) & plus(W,Y,Z)

divides(X,0) <--
divides(X,Z) <-- divides(X,Y) & plus(X,Y,Z)

even(0) <--
even(s(s(X))) <-- even(X)
```

EXISTENTIAL DEFINITIONS

```
lesseq(A,B) <-- plus(C,A,B)
plus(diff(A,B),A,B) <-- lesseq(A,B)

divides(X,Z) <-- times(A,X,Z)
times(quot(A,B),A,B) <-- divides(A,B)

even(X) <-- plus(Half,Half,X)
plus(half(X),half(X),X) <-- even(X)

square(X) <-- times(Root,Root,X)          % Note that this has no obvious
times(root(X),root(X),X) <-- square(X)    %  recursive definition
```

FUNCTION PROPERTIES

```
plus(X,Y,sum(X,Y)) <--

times(X,Y,prod(X,Y)) <--
```

COMMUTATIVITY AND ASSOCIATIVITY

```
plus(X,Y,Z) <-- plus(Y,X,Z)
plus(X,YZ,XYZ) <-- plus(X,Y,XY) & plus(Y,Z,YZ) & plus(XY,Z,XYZ)

times(X,Y,Z) <-- times(Y,X,Z)
times(X,YZ,XYZ) <-- times(X,Y,XY) & times(Y,Z,YZ) & times(XY,Z,XYZ)
```

```
%   The theorem to be proved is that set inclusion (with sets implemented
%   as lists) is transitive.

%   Definition of member

rec_def(member,
        member(X, [X|_]),       %   "base" case
        member(X, [_|T]) <-- member(X, T)
).

%   Definition of includes

rec_def(includes,
        includes(S, []),
        includes(S, [X|T]) <-- member(X, S) & includes(S, T)
).

%   The theorem

conjecture(includes_is_transitive,
        includes(A, C) <-- includes(A, B) & includes(B, C)
).
```