Can Logic Programming Execute as Fast as Imperative Programming?

Peter Lodewijk Van Roy

Volume 2

Appendices

Appendix A

User manual for the Aquarius Prolog compiler

1. Introduction

The Aquarius Prolog compiler reads clauses and directives from stdin and outputs Prolog-readable compiled code to stdout as one fact per instruction. The output is assembly code for the Berkeley Abstract Machine (BAM). Directives hold starting from the next predicate that is input. Clauses do not have to be contiguous in the input stream, however, the whole stream is read before compilation starts.

This manual is organized into ten sections. Section 2 documents the compiler's directives. Section 3 gives the compiler's options. Section 4 gives a short overview of the dataflow analysis done by the compiler. Section 5 gives the type declarations accepted by the compiler. Section 6 summarizes the differences between Aquarius Prolog and the Edinburgh standard. Section 7 gives an example showing how to use the compiler. Section 8 describes the method used to compile specialized entry points to increase the efficiency of built-ins. Section 9 describes the assembly language interface. Section 10 describes how to define BAM assembly macros.

2. Directives

The directives recognized by the Aquarius compiler are given in Table 1.

3. Options

The Aquarius compiler's options are given in three categories: high-level (these options control actions of the compiler at the Prolog level), architecture-dependent (these options are constant for a particular architecture), and low-level (mainly useful for debugging purposes). The default options are set for the VLSI-BAM processor. The options are given in Tables 2, 3, and 4.

4. Dataflow analysis

Dataflow analysis is enabled with the `analyze` option. It generates ground, nonvar, recursively dereferenced and uninitialized variable types which are merged with the programmer's types. Both uninitialized memory and uninitialized register types are generated. Entry declarations (given by `entry` directives) are used to drive the analysis. Predicates of arity zero are always used as entry declarations. The quality of the generated types is such that compilation time, execution time, and code size are all significantly reduced. Therefore it is recommended always to compile with analysis. The whole program is kept in memory during the analysis.

All `mode`, `entry`, and `op` directives are executed before the analysis starts. Other directives are executed after the analysis and before compilation. The directives `default` and `clear` interfere with dataflow analysis, so they should be given only when the `analyze` option is disabled.

4.1. Dataflow analysis and dynamic code

The compiler makes the distinction between static and dynamic code. Static code is completely known at compile-time and is subject to analysis. Dynamic code is created at run-time by the built-in predicates `assert/1`, `retract/1`, and their cousins. It is not analyzed. There are two cases to consider:

(1) A dynamic predicate calls a static predicate. In this case, there must be an entry declaration giving the worst-case type of the call for each static predicate that might be called by a dynamic predicate.

Leaving out this declaration may result in incorrect compilation.

(2) A static predicate calls a dynamic predicate. The analyzer will assume worst-case types for the dynamic predicate unless it has a type declaration.

The most common uses of dynamic code are as databases of facts, or as rules that only call a limited set of static predicates. For these uses, there is no problem in integrating analyzed static code with dynamic code.

4.2. Dataflow analysis and the `call/1` built-in

The `call/1` built-in predicate can call any predicate in the program with any modes, and it is not possible in general to determine these predicates and their modes at compile-time. However, most programs that use `call/1` will call one of a known set of predicates or will call a dynamic predicate. There are three cases to consider:

(1) If the set of predicates that may be arguments of `call/1` is known by the programmer, then these predicates should be given entry declarations with worst-case modes. (This case can be written more efficiently by writing a new predicate that directly calls one of the set, and avoids calling `call/1`.)

(2) If the predicates that may be arguments of `call/1` are dynamic, then analysis is correct without entry declarations. This is true because dynamic predicates are not analyzed.

(3) If any predicate in the program may be an argument of `call/1` and nothing is known about the modes then analysis is useless and it should not be done.

5. Types

The Aquarius compiler accepts type declarations for a predicate. Using types results in a significant improvement in code quality. Types are represented as `(Head:-Formula)` where `Head` contains only variables and `Formula` is a logical conjunction. Almost any Prolog test can be used in a type formula. Possible type formulas are given in Table 5. This representation for types is simple, yet powerful enough to represent much important information in a compact way. The representation generalizes the declarations of Dec-10 Prolog. For example, the Dec-10 declaration:

```
:- mode(concat(+,+,-))
```

is expressed here as:

```
:- mode((concat(A,B,C):-nonvar(A),nonvar(B),var(C))).
```

6. Differences with Edinburgh Prolog

Aquarius Prolog recognizes new type-checking built-ins which are not part of the Edinburgh Prolog standard as embodied by C-Prolog. The new built-ins and their definitions in standard Prolog are given in Table 6.

7. An example of the compiler's use

The following example shows how the compiler is used:

```
% /hprg2/Bam/Compiler/compiler  % Run the compiler.
                                % Code is entered directly.
:-mode((a(A):-nonvar(A))).      % Enter the type.
a(a).                           % Enter a simple two-fact predicate.
a(b).
ˆD                              % End-of-file.
                                % The output follows:

% Cputime between start and finish is 1.383

procedure(a/1).
    deref(r(0),r(0)).
    hash(atomic,r(0),2,l(a/1,1)).
    fail.
label(l(a/1,1)).
    pragma(hash_length(2)).
    pair(a,l(a/1,3)).
    pair(b,l(a/1,4)).
label(l(a/1,3)).
label(l(a/1,4)).
    return.
```

8. Entry specialization for more efficient built-ins

The directive `modal_entry(Head,EntryTree)` adds a discrimination tree of entry points for the predicate `Head`. This directive is used by the system to implement more efficient built-ins. It is not normally needed by programmers, although they can take advantage of it for other predicates. The compiler uses the discrimination tree to choose the most efficient entry point for each call of a predicate depending on the type formula that is true at the predicate's call. The syntax of the discrimination tree in modal_entry is:

```
tree(entry(EntryHead)).
tree(mode(Formula,TrueTree,FalseTree)) :-
    tree(TrueTree), tree(FalseTree).
```

`EntryHead` is the entry point that replaces `Head` and `Formula` is a type formula. Compilation of a the predicate `Head` proceeds by following a path down the discrimination tree. If the formula valid when `Head` is called implies `Formula` then the `TrueTree` is followed. Otherwise the `FalseTree` is followed. Tree traversal stops when an entry point `entry(EntryHead)` is encountered. At that point the original call is replaced by `EntryHead`.

9. Interfacing with BAM assembly language routines

Prolog predicates can efficiently call routines written in BAM assembly code (the compiler's output) or in the target machine's assembly language (for example, VLSI-BAM, MIPS, or MC68020 assembly code). The interface with both low-level languages is provided through the five-argument type declaration. This declaration has the following form:

```
:- mode(Head, Require, Before, After, Survive).
```

`Head` is the head of the predicate. `Require` is the required type formula, i.e. the formula made true by the compiler. All uninitialized variable types (both uninitialized memory and uninitialized register) must be part of the required formula. `Before` is the type formula known to be valid before the call. `After` is the type formula known to be valid after the call. `Survive` is the register survive flag. If the flag is `y` then the predicate must not alter the values of any argument registers (except those used to return a result). It must save and restore any argument registers it needs. The predicate is called with a `simple_call` instruction and must return with a `simple_return` instruction (or its equivalent in

VLSI-BAM processor assembly). A simple call may not be nested. It is more efficient than a standard call because it does not need an environment frame around it in the calling routine.

If the survive flag is `n` then the predicate is assumed to invalidate all argument register values. In this case the argument registers are available as scratch registers and the calling routine will create an environment frame.

Efficient parameter passing is implemented by using uninitialized variables. These are of two kinds: uninitialized memory and uninitialized register variables. An uninitialized memory variable is a pointer to an empty memory cell. Binding to it is a store to memory. An uninitialized register variable is an empty register. Binding to it is a move to the register. No trailing or dereferencing is needed in either case.

Declaring an argument to have a uninitialized register type means that the output of the routine is stored in the corresponding argument register. Similarly, an uninitialized memory type requires the output to be stored to the location pointed to by the argument register. Inputs and outputs must be put in separate registers.

10. Defining BAM assembly language macros

It is possible to define macros in the Prolog source that are expanded into BAM assembly instructions. The advantages of macros are that they do not have call-return overhead, that unnecessary shuffling of data between registers is avoided, and that the full range of low-level compiler optimizations is performed on them. A macro definition has the following form:

```
:- macro((Head :- Body)).
```

where `Head` is the head of the predicate that will be expanded and `Body` is a series of BAM instructions. For example:

```
:- mode(quad(A,B), uninit_reg(B), true, deref(B), y).
:- macro((quad(A,B) :- add(A,A,X), add(X,X,B))).
```

The macro definition is preceded by a mode declaration telling that the second argument is the output.

Macro definitions must obey the following rules:

(1) All legal BAM instructions and addressing modes are allowed in the macro definition including user instructions, except as noted below. User instructions are never generated by the compiler, but they are recognized and optimized in macro definitions. Labels are given as ground terms or as Prolog variables. The latter are given unique ground values by the compiler. Registers are given as user registers (e.g. `r(h)` and `r(t2)`) or as Prolog variables (e.g. `X` and `Y`). The latter are allocated by the compiler. Do not use numbered registers (`r(0)`, `r(1)`, ...).

(2) The macro definition must be preceded by a mode declaration. The exit modes must be valid upon exiting the macro. All head arguments that return results must be of uninitialized register type.

(3) The macro may not alter any of the head arguments except those returning a result.

(4) The second argument of the `deref(X,Y)` instruction must be a new variable, i.e. it must not have a value upon entering the macro. Failing to obey this constraint will lead to incorrect behavior on backtracking.

(5) It is not recommended to create choice points inside macros since it is not known how many registers are live.

Table 1 – Compiler directives

Directive	Action
`:- help.`	Print a summary of these directives.
`:- default.`	Set the default options for the VLSI-BAM processor and clear all type declarations and modal entries.
`:- mips.`	Ensure compatibility with the MIPS processor. This directive should occur only once in a file. It sets the option align(1), disables the option split_integer, and sets all other options to their default values. It clears all type declarations and modal entries.
`:- vlsi_plm.`	Ensure compatibility with the re-microcoded VLSI-PLM. This directive should occur only once in a file. It sets the options high_reg(6) and align(1), disables the option split_integer, and sets all other options to their default values. It clears all type declarations and modal entries. Trail checks and shifts are compiled differently.
`:- clear.`	Clear all type declarations and modal entries.
`:- option(Options).`	Add the options in `Options` to the current options. `Options` may be a single option or a list of options.
`:- notoption(Options).`	Remove the options in `Options` from the current options. `Options` may be a single option or a list of options.
`:- printoption.`	Print a list of the currently active options.
`:- mode((Head:-Formula)).`	Type declaration for a predicate. The type information is remembered until new types are given for that predicate or until all type information is cleared. This declaration is not used as a starting point for dataflow analysis. However, the types generated by dataflow analysis are used to supplement the declaration, and an error message is given if there is a contradiction.
`:- entry((Head:-Formula)).`	Type declaration for a predicate—same as above. This declaration is also used as a starting point in dataflow analysis.
`:- mode(H,R,B,A,S).`	Detailed type declaration for a predicate. This declaration is useful for interfacing with assembly language. H is the head, R is the required type formula (made true by the compiler before each call), B is the before type formula (assumed true before each call), A is the after type formula (assumed true after each call), S is the survive flag (y/n depending on whether the call lets registers survive). The after type formula is used by dataflow analysis to improve the generated types.
`:- entry(H,R,B,A,S).`	Detailed type declaration for a predicate—same as above. This declaration is also used as a starting point in dataflow analysis.
`:- modal_entry(H,T).`	Optional discrimination tree of efficient entry points for the predicate H. The tree T contains type formulas used to replace each call of the predicate by a more efficient entry point.
`:- macro((Head:-Body)).`	Macro definition. The head is expanded into a sequence of BAM assembly instructions.
`:- include(FileName).`	Insert the text of the file `FileName`. This directive may be nested up to the system limit of simultaneous open files.
`:- pass(Anything).`	Pass the input ‘‘`:- pass(Anything).`’’ unaltered to the output in Prolog-readable form.
`:- version.`	Print the creation date of this version of the compiler.
`:- op(A,B,C).`	Declare an operator in Prolog. Pass the input ‘‘`:- op(A,B,C).`’’ unaltered to the output in Prolog-readable form.

Table 2 – High-level compiler options

Option	Default	Description
`select_limit(L)`	L=1	Perform selection for up to L arguments. Selection is done according to the enrichment heuristic. See Chapter 4 section 6.2.
`analyze`	off	Perform dataflow analysis for all predicates in the input stream. This option enables analysis of the entire input stream, no matter where it occurs in the stream. The starting points for analysis are the entry declarations and all predicates of arity zero. The types obtained from the analysis are merged with the programmer's types. The predicates are then compiled with the merged types.
`compile`	on	Compile the input. When this option is disabled, the entry types generated by the dataflow analyzer for the source predicates are output as valid Prolog-readable type declarations.
`factor`	on	Do factoring source transformation. With this transformation similar compound terms in adjacent heads are only unified once. Often this gives faster code.
`comment`	on	Give information about what the compiler is doing.
`same_number_solutions`	on	Keep the same number of solutions on backtracking as standard Prolog. Relaxing the semantics by removing this option results in better code in some cases.
`same_order_solutions`	on	Keep the same order of solutions on backtracking as standard Prolog. Relaxing the semantics by removing this option results in better code in some cases.
`depth_limit(D)`	D=2	Nesting depth limit on unification goals. Unifications deeper than this limit are transformed to remain within this limit. This transformation is used because compilation time and code size for deeply nested unifications would otherwise increase as the square of the size of the unification.
`short_block(S)`	S=6	Threshold on basic block length for shuffle optimization.

Table 3 – Architecture-dependent compiler options

Option	Default	Description
`low_reg(L)`	L=0	Lowest numbered machine register.
`high_reg(H)`	H=100	Highest numbered machine register. In the VLSI-BAM processor, registers higher than `r(15)` are mapped into memory.
`low_perm(P)`	P=0	Lowest numbered permanent variable.
`hash_size(H)`	H=5	Minimum size of a hash table.
`align(K)`	K=2	Align all compound terms to start on a multiple of K.
`uni`	on	Generate unify_atomic instruction to unify with an atomic term.
`split_integer`	on	Use separate tags for negative and nonnegative integers.

Table 4 – Low-level compiler options

Option	Default	Description
`system(X)`	quintus	The system running the compiler (other value: cprolog).
`write`	on	Write the object code when compilation is complete.
`peep`	on	Do peephole optimization.
`stats(S)`	off	Print timing statistics during compilation. S is one of the following atoms, or a list of them: `t` (top level of compilation), `c` (compilation of a single procedure), `p` (peephole optimization), `s` (selection algorithm—extraction of determinism), `d` (deterministic code generation).
`debug`	off	Print debugging messages during compilation.

Table 5 – Type formulas

Type	Meaning
`nonvar(A)`	A is a nonvariable term, i.e. its main functor is instantiated. Nothing is implied about its arguments.
`ground(A)`	A is a ground term, i.e. it contains no unbound variables.
`var(A)`	A is an unbound variable.
`uninit(A)`	A is an uninitialized memory variable. At the Prolog level, this means that A is an unbound variable known not to be aliased to another variable. In the implementation, A is a pointer to an empty memory cell. Binding to this variable is a simple store, without dereferencing or trailing.
`uninit_reg(A)`	A is an uninitialized register variable. At the Prolog level, this has the same meaning as an uninitialized memory variable. In the implementation, A is an empty machine register. This type increases the efficiency of parameter passing by returning a value directly in a register. It is useful for interfacing with assembly language.
`deref(A)`	A is dereferenced.
`rderef(A)`	A is recursively dereferenced, i.e. A is dereferenced and all subterms of A are recursively dereferenced.
`structure(A)`	A is a structure.
`list(A)`	A is a list, i.e. a cons cell or nil.
`cons(A)`	A is a cons cell, i.e. a non-nil list.
`compound(A)`	A is a structure or a cons cell.
`functor(A,F,N)`	A is the structure F with arity N.
`atom(A)`	A is an atom.
`atomic(A)`	A is atomic, i.e. a number or an atom.
`simple(A)`	A is atomic or an unbound variable.
`integer(A)`	A is an integer.
`float(A)`	A is a floating point number.
`number(A)`	A is an integer or a float.
`negative(A)`	A is a negative integer.
`nonnegative(A)`	A is a nonnegative integer.
`A>0`	A is a positive integer.
`A==x`	A is the atom `x`.
`true`	Nothing is known about the type.
`fail`	This means ‘‘execution can never reach this point.’’
`(F1,F2)`	This means ‘‘F1 and F2,’’ where F1 and F2 are type formulas.
`(F1;F2)`	This means ‘‘F1 or F2,’’ where F1 and F2 are type formulas.
`not(F)`	This means ‘‘not F,’’ where F is a type formula.

Table 6 – New type-checking predicates in Aquarius Prolog

Predicate	Prolog Definition
`nil(A)`	`:- nonvar(A), A=[].`
`cons(A)`	`:- nonvar(A), A=[_|_].`
`list(A)`	`:- nonvar(A), (A=[] ; A=[_|_]).`
`compound(A)`	`:- nonvar(A), \+atomic(A).`
`structure(A)`	`:- nonvar(A), \+atomic(A), \+A=[_|_].`
`ground(A)`	`:- nonvar(A), functor(A, _, N), ground(N, A).`
`simple(A)`	`:- (var(A) ; atomic(A)).`
`negative(A)`	`:- integer(A), A<0.`
`nonnegative(A)`	`:- integer(A), A>=0.`
`is_list(A)`	`:- (var(A), ! ; A=[] ; A=[_|B], is_list(B)).`
`is_partial_list(A)`	`:- (var(A), ! ; A=[_|B], is_partial_list(B)).`
`is_proper_list(A)`	`:- (var(A),!,fail;A=[];A=[_|B],is_proper_list(B)).`

The following clauses are part of the definition:

```
ground(N, _) :- N=:=0.
ground(N, A) :- N=\=0, arg(N, A, X), ground(X), N1 is N-1, ground(N1, A).
```

Appendix B

Formal specification of the Berkeley Abstract Machine syntax

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Formal specification of the Berkeley Abstract Machine (BAM) syntax
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California
% May be used and modified for non-commercial purposes if this notice is kept.
% Written by Peter Van Roy.

% This file is an executable Prolog program that checks the syntactic
% correctness of BAM instructions.  The predicate instr(I) is true if I is
% a legal BAM instruction.  In addition to instructions output by the Aquarius
% compiler, this predicate also accepts the user instructions of the BAM,
% which allow the run-time system to be written completely in BAM assembly.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Check correctness of a sequence of BAM instructions ***

% Create saved state:
% Note: In C-Prolog this must be started up in a system
% equal to in size or larger than the one which created it.
main :- save(check, 1), prompt(_, ’’), read(Instr), pipe(Instr, 0, 0), halt.
main :- halt.

% Pipe working loop:
pipe(end_of_file, M, N) :- !,
   T is M+N,
   write(’*** Checked ’),write(T),write(’ instructions; ’),
   write(M),write(’ correct and ’),write(N),write(’ incorrect.’),nl.
pipe(Instr, M, N) :-
   (instr(Instr)
   -> M1 is M+1, N1=N
   ;  M1=M, N1 is N+1,
      write(’*** Incorrect: ’),write(Instr),nl
   ),
   !, read(NewInstr), pipe(NewInstr, M1, N1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** BAM Instructions ***

% 1. Unification support instructions:
instr(deref(V,W))          :- var_i(V), var_i(W).
instr(equal(EA,A,L))       :- ea_e(EA), arg_i(A), lbl(L).
instr(unify(V,W,F,G,L))    :- var_i(V), var_i(W), nv_flag(F),nv_flag(G),lbl(L).
instr(trail(V))            :- var_i(V).
instr(move(EA,VI))         :- ea_m(EA), var_i(VI).
instr(push(EA,R,N))        :- ea_p(EA), hreg(R), pos(N).
instr(adda(R,S,T))         :- numreg(R), numreg(S), hreg(T).
instr(pad(N))               :- pos(N).
instr(unify_atomic(V,I,L)) :- var_i(V), an_atomic(I), lbl(L).
instr(fail).

% 2. Conditional control flow instructions:
instr(switch(T,V,A,B,C))   :- a_tag(T), var_i(V), lbl(A), lbl(B), lbl(C).
instr(choice(I/N,Rs,L))    :- pos(I), pos(N), I=<N, lbl(L), regs(Rs).
instr(test(Eq,T,V,L))      :- eq_ne(Eq), var_i(V), a_tag(T), lbl(L).
instr(jump(C,A,B,L))       :- cond(C), numarg_i(A), numarg_i(B), lbl(L).
instr(move(CH,V))          :- a_var(V), choice_ptr(CH).
instr(cut(V))              :- a_var(V).
instr(hash(T,R,N,L))       :- hash_type(T), reg(R), pos(N), lbl(L).
instr(pair(E,L))           :- an_atomic(E), lbl(L).

% 3. Arithmetic instructions:
instr(add(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sub(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(mul(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(div(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(mod(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(and(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr( or(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(xor(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(not(A,V))            :- numarg_i(A),              a_var(V).
instr(sll(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sra(A,B,V))          :- numarg_i(A), numarg_i(B), a_var(V).
instr(sll). /* vlsi_plm only */
instr(sra). /* vlsi_plm only */

% 4. Procedural instructions:
instr(procedure(N/A))      :- atom(N), natural(A).
instr(call(N/A))           :- atom(N), natural(A).
instr(return).
instr(simple_call(N/A))    :- atom(N), natural(A).
instr(simple_return).
instr(label(L))            :- lbl(L).
instr(jump(L))             :- lbl(L).
instr(allocate(Perms))     :- natural(Perms).
instr(deallocate(Perms))   :- natural(Perms).
instr(nop).

% 5. Pragma information for translator and reorderer:
instr(pragma(P))           :- pragma(P).

% 6. Additions to BAM for the assembly language programmer:
instr(I)                   :- user_instr(I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Additions to BAM for the assembly language programmer ***

% This section describes the parts of the BAM language that are never output
% by the compiler, but only used by the BAM assembly programmer.  This is used
% to write the run-time system in BAM code, so that it is as portable as
% possible.  Additional instructions are jump to register address, convert
% tagged atom or integer to untagged integer (ord), its inverse (val), and
% non-trapping full-word unsigned comparison, non-trapping full-word
% arithmetic, and trailing for backtrackable destructive assignment.

user_instr(jump_reg(R))      :- reg(R).
user_instr(jump_nt(C,A,B,L)) :- cond(C), numarg_i(A), numarg_i(B), lbl(L).
user_instr(ord(A,B))         :- arg(A), a_var(B).
user_instr(val(T,A,V))       :- a_tag(T),    numarg_i(A), a_var(V).
user_instr(add_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sub_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(and_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr( or_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(xor_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(not_nt(A,V))      :- numarg_i(A),              a_var(V).
user_instr(sll_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sra_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(trail_bda(X))     :- a_var(X).

% Additional registers:
% See Implementation Manual for list of existing registers.
user_reg(r(A)) :- atom(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Pragmas ***

% A variable is a multiple of N.
% Inserted just before loads in readmode unification.
pragma(align(V,N))         :- a_var(V), pos(N).

% Inserted just before a sequence of pushes in writemode unification.
% (The pushes may be interleaved with non-memory moves.)
pragma(push(term(Size)))   :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).

% Specify the tag of a variable.
% (This is useful for processors without explicit tag support.)
pragma(tag(V,T))           :- a_var(V), a_tag(T).

% Length of a hash table.
pragma(hash_length(Len))   :- pos(Len).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Tags ***

a_tag(tatm). /* atom */
a_tag(tint). /* integer */
a_tag(tneg). /* negative integer */
a_tag(tpos). /* nonnegative integer */
a_tag(tstr). /* structure */
a_tag(tlst). /* cons cell */
a_tag(tvar). /* variable */

atom_tag(tatm).

pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Addressing modes ***

heap_ptr(r(h)).
choice_ptr(r(b)).

reg(r(I))          :- int(I).
reg(U)             :- user_reg(U).

hreg(R)            :- reg(R).
hreg(R)            :- heap_ptr(R).

perm(p(I))         :- natural(I).

an_atomic(I)       :- int(I).
an_atomic(T^A)     :- atom(A), atom_tag(T).
an_atomic(T^(F/N)) :- atom(F), pos(N), atom_tag(T).

a_var(Reg)         :- reg(Reg).
a_var(Perm)        :- perm(Perm).

arg(Arg)           :- a_var(Arg).
arg(Arg)           :- an_atomic(Arg).

var_i(Var)         :- a_var(Var).
var_i([Var])       :- a_var(Var).

arg_i(Arg)         :- var_i(Arg).
arg_i(Arg)         :- an_atomic(Arg).

numreg(Arg)        :- reg(Arg).
numreg(Arg)        :- int(Arg).

numarg_i(Arg)      :- var_i(Arg).
numarg_i(Arg)      :- int(Arg).

var_off([Var])     :- a_var(Var).
var_off([Var+I])   :- a_var(Var), pos(I).

% Effective address for equal:
ea_e(Var)          :- a_var(Var).
ea_e(VarOff)       :- var_off(VarOff).

% Effective address for move:
```

```
ea_m(Arg)          :- arg(Arg).
ea_m(VarOff)       :- var_off(VarOff).
ea_m(Tag^H)        :- pointer_tag(Tag), heap_ptr(H).

% Effective address for push:
ea_p(Arg)          :- arg_i(Arg).
ea_p(Tag^H)        :- pointer_tag(Tag), heap_ptr(H).
ea_p(Tag^(H+D))    :- pointer_tag(Tag), pos(D), heap_ptr(H).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Miscellaneous ***

eq_ne(eq). /* Equal */
eq_ne(ne). /* Not equal */

cond(lts). /* Signed less than */
cond(les). /* Signed less than or equal */
cond(gts). /* Signed greater than */
cond(ges). /* Signed greater than or equal */
cond(eq).  /* Equal */
cond(ne).  /* Not equal */

hash_type(atomic).
hash_type(structure).

lbl(fail).
lbl(N/A)      :- atom(N), natural(A).
lbl(l(N/A,I)) :- atom(N), natural(A), natural(I).

nv_flag(nonvar).
nv_flag(var).
nv_flag('?').

% A list of register numbers:
% (May contain the value 'no' as well)
regs([]).
regs([R|Set]) :- (int(R); R=no), regs(Set).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Utilities ***

ground(X) :- nonvar(X), functor(X, _, N), ground(N, X).

ground(N, _) :- N=:=0.
ground(N, X) :- N=\=0, arg(N, X, A), ground(A), N1 is N-1, ground(N1, X).

int(N)     :- integer(N).
natural(N) :- integer(N), N>=0.
pos(N)     :- integer(N), N>0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix C

Formal specification of the Berkeley Abstract Machine semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Formal specification of the Berkeley Abstract Machine (BAM) semantics
% Copyright (C) 1990 Peter Van Roy and Regents of the University of California
% May be used and modified for non-commercial purposes if this notice is kept.
% Written by Peter Van Roy.

% The specification is a Prolog program that defines the meaning of BAM in
% terms of its execution in a simple memory model.  It runs BAM code directly
% from the output of the Aquarius compiler.

% The specification does not include the user instructions of the BAM since
% their behavior depends on the target machine.

% The specification is written in the Extended DCG notation.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Meaning of registers:
% r(b)      Index to most recent choice point.
% r(e)      Index to current environment.
% r(tr)     Top of trail stack.
% r(h)      Top of heap stack.
% r(hb)     Value of r(h) at last choice point creation.
% r(pc)     Code address.
% r(cp)     Continuation pointer for code.
% r(tmp_cp) Temporary continuation pointer for code, used only in simple_call.
% r(retry)  Retry address for backtracking, only exists inside choice points.
% r(I)      Argument and temporary register.
% p(I)      Location on current environment.

% Types stored in registers:
% r(e) Contains values of registers {r(e),r(cp)} U {p(0), ..., p(N-1)},
%      where N is the size of the environment.
% r(b) Contains values of registers {r(e),r(cp),r(tr),r(b),r(h),r(retry)} U RS,
%      where RS is a subset of {r(0), r(1), ...}.
% r(tr)        Contains a natural number.
% r(h), r(hb)  Contain words with a pointer tag.
% r(pc), r(cp) Contain natural numbers or symbolic labels.
% r(tmp_cp)    Contains a symbolic label.
% r(retry)     Contains a symbolic label.
% r(I)         Contains a word.
% p(I)         Contains a word.

% Comments:
% A word is either an integer or a structure of the form Tag^Value where Value
% is a natural number except if Tag=tatm, in which case Value is an atom or a
% structure (F/N) where F is an atom and N is a natural number.
% A symbolic label is either the atom 'fail', or the structure F/N, or the
% structure l(F/N,I), where F is an atom and N and I are natural numbers.
% r(cp) is stored in environments, allowing nested calls.
% r(tmp_cp) is not stored in environments, allowing only one level of call.
% However, no environment is needed in a predicate containing a simple_call.
% There are no explicit stacks for environments or choice points; registers
% r(e) and r(b) each contain a set of register values.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Accumulator declarations:

% Accumulators:

acc_info(code,   T, In, Out, table_command(T,In,Out)).
acc_info(lblmap, T, In, Out, table_command(T,In,Out)).
acc_info(regs,   T, In, Out, table_command(T,In,Out)).
acc_info(trail,  T, In, Out, table_command(T,In,Out)).
acc_info(heap,   T, In, Out, heap_table_command(T,In,Out)).
acc_info(count,  T, In, Out, (Out is T+In)).

% Predicate declarations:

% Top level:
pred_info(            execute, 0, [regs,heap,trail,code,lblmap,count]).
pred_info(         instr_loop, 0, [regs,heap,trail,code,lblmap,count]).
pred_info(     instr_loop_end, 1, [regs,heap,trail,code,lblmap,count]).
pred_info(              instr, 1, [regs,heap,trail,code,lblmap]).
pred_info(         numeric_pc, 2, [lblmap]).

% Addressing modes:
pred_info(               heap, 3, [     heap]).
pred_info(                reg, 3, [regs     ]).
pred_info(               perm, 3, [regs     ]).
pred_info(              a_var, 3, [regs     ]).
pred_info(              var_i, 3, [regs,heap]).
pred_info(                arg, 2, [regs     ]).
pred_info(              arg_i, 2, [regs,heap]).
pred_info(             numreg, 2, [regs     ]).
pred_info(             numarg, 2, [regs,heap]).
pred_info(            var_off, 2, [regs,heap]).
pred_info(            imm_tag, 2, [regs     ]).
pred_info(               ea_e, 2, [regs,heap]).
pred_info(               ea_m, 2, [regs,heap]).
pred_info(               ea_p, 2, [regs,heap]).

% Instruction utilities:
pred_info(          deref_rtn, 2, [regs,heap,trail]).
pred_info(     deref_rtn_cont, 3, [regs,heap,trail]).
pred_info(          equal_rtn, 3, [regs,heap,trail]).
pred_info(         switch_rtn, 5, [regs,heap,trail]).
pred_info(           test_rtn, 4, [regs,heap,trail]).
pred_info(      jump_cond_rtn, 4, [regs,heap,trail]).
pred_info(        hash_lookup, 3, [regs,heap,trail,lblmap,code]).
pred_info(      hash_lookup_2, 3, [regs,heap,trail,lblmap,code]).
pred_info(      hash_indirect, 3, [     heap      ]).
pred_info(   save_choice_regs, 2, [regs,heap,trail]).
pred_info(restore_choice_regs, 2, [regs,heap,trail]).
pred_info(        detrail_rtn, 2, [regs,heap,trail]).
pred_info(          trail_rtn, 1, [regs,heap,trail]).
pred_info(          cmp_trail, 2, [regs,heap,trail]).
pred_info(          unify_rtn, 3, [regs,heap,trail]).
pred_info(        unify_rtn_2, 3, [regs,heap,trail]).
pred_info(        unify_rtn_2, 5, [regs,heap,trail]).
pred_info(   unify_rtn_args_2, 6, [regs,heap,trail]).
pred_info(   unify_rtn_args_3, 7, [regs,heap,trail]).
pred_info(          unify_atm, 3, [regs,heap,trail]).
pred_info(          unify_end, 2, [regs,heap,trail]).
pred_info(       unify_varvar, 2, [regs,heap,trail]).
pred_info(           get_size, 3, [     heap      ]).
pred_info(              arith, 4, [regs,heap      ]).

pred_info(          write_rtn, 0, [regs,heap,trail]).
pred_info(          write_rtn, 1, [regs,heap,trail]).
pred_info(          write_arg, 2, [regs,heap,trail]).
pred_info(         write_args, 3, [regs,heap,trail]).

% Implement the accumulator commands:
table_command(ins(I,Val), In,  In) :- ins(In, I, Val).
table_command(get(I,Val), In,  In) :- get(In, I, Val).
table_command(set(I,Val), In, Out) :- set(In, I, Val, Out).

% Mask off tag before looking up heap entry:
heap_table_command(ins(_^I,Val), In,  In) :- ins(In, I, Val).
heap_table_command(get(_^I,Val), In,  In) :- get(In, I, Val).
heap_table_command(set(_^I,Val), In, Out) :- set(In, I, Val, Out).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Initialization and runtime options ***

:- dynamic(bamspec_option/1).

main :-
   save(bamspec, 1),
   prompt(_, ''),
   ( copyright,
     execute
   ; error(['Sorry, the executable BAM specification has failed.'])
   ),
   halt.
main :-
   halt.

copyright :-
   write('Berkeley Abstract Machine (BAM) Executable Specification'), nl,
   write('Copyright (C) 1990 Peter Van Roy and '),
   write('Regents of the University of California'), nl, nl.
```

```
flag_print(I) :- bamspec_option(print), !, write('Executing '), write(I), nl.
flag_print(_).

% Look up symbolic label to get a numeric PC:
numeric_pc(PC,  PC) -->> {integer(PC)}, !.
numeric_pc(PC, NPC) -->> [get(PC,NPC)]:lblmap.

% Read in the instructions and create the code array and label map:
% The code array gives the instruction corresponding to each PC value.
% The label map gives the PC value corresponding to each symbolic label.
read_code(Code, LblMap) :-
   read(Instr),
   read_code(Instr, 0, Code, LblMap).

read_code(end_of_file, _, Code, LblMap) :- !, seal(Code), seal(LblMap).
read_code((:-Option), PC, Code, LblMap) :- !,
   asserta(bamspec_option(Option)),
   read(NextInstr),
   read_code(NextInstr, PC, Code, LblMap).
read_code(Instr, PC, Code, LblMap) :-
   ins(Code, PC, Instr),
   insert_lblmap(Instr, LblMap, PC),
   PC1 is PC+1,
   read(NextInstr),
   read_code(NextInstr, PC1, Code, LblMap).

% Add an entry to the label map:
insert_lblmap(label(L),     LblMap, PC) :- !, ins(LblMap, L, PC).
insert_lblmap(procedure(P), LblMap, PC) :- !, ins(LblMap, P, PC).
insert_lblmap(_, _, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Top level execution ***

execute :-
   write('Reading BAM code'), nl,
   read_code(Code, LblMap),
   write('Starting execution'), nl,
   execute(leaf, Regs, leaf, _, leaf, _, Code, _, LblMap, _, 0, N),
   write('Executed '), write(N), write(' instructions.'), nl,
   print_array(Regs).

execute(File) :-
   seeing(OldFile),
   see(File),
   read_code(Code, LblMap),
   seen,
   see(OldFile),
   execute(leaf, Regs, leaf, _, leaf, _, Code, _, LblMap, _, 0, N),
   write('Executed '), write(N), write(' instructions.'), nl,
   print_array(Regs).
```

```
execute -->>
   [set(r(e),leaf)]:regs,
   [set(r(b),leaf)]:regs,
   [set(r(h),tvarˆ0)]:regs,
   [set(r(tr),0)]:regs,
   [set(r(pc),0)]:regs,
   [set(r(cp),global_success/0)]:regs,
   instr(choice(1/2,[],global_failure/0)),
   instr_loop.

% Instruction execution loop:
instr_loop -->>
   [get(r(pc),PC)]:regs,
   !,
   instr_loop_end(PC).
instr_loop -->>
   error([’Attempt to execute beyond existing code.’]).

instr_loop_end(write/1) -->> !, write_rtn, instr(return), instr_loop.
instr_loop_end(nl/0) -->> !, nl, instr(return), instr_loop.
instr_loop_end(global_success/0) -->> !,
   write(’*** Global success ***’), nl.
instr_loop_end(global_failure/0) -->> !,
   write(’*** Global failure ***’), nl.
instr_loop_end(fail) -->>
   instr(fail),
   !,
   instr_loop.
instr_loop_end(PC) -->>
   numeric_pc(PC, NPC),
   % Fetch:
   [get(NPC,Instr)]:code,
   NPC1 is NPC+1,
   [set(r(pc),NPC1)]:regs,
   % Execute:
   [1]:count,
   flag_print(Instr),
   instr(Instr),
   !,
   instr_loop.
instr_loop_end(PC) -->>
   error([’Program counter is ’,PC]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** BAM Instructions ***

% 1. Unification support instructions:
instr(deref(V,W)) -->>
   var_i(get,  V, X),
   deref_rtn(X, Y),
   var_i(set, W, Y).
instr(equal(EA,A,L)) -->>
   ea_e(EA, X),
   arg_i(A, Y),
   {lbl(L)},
   equal_rtn(X, Y, L).
instr(unify(V,W,F,G,L)) -->>
   var_i(get, V, X),
   var_i(get, W, Y),
   {nv_flag(F)},
   {nv_flag(G)},
   {lbl(L)},
   unify_rtn(X, Y, L).
instr(unify_atomic(V,I,L)) -->>
   var_i(get, V, X),
   {an_atomic(I)},
   {lbl(L)},
   unify_rtn(X, I, L).
instr(trail(V)) -->>
   var_i(get, V, X),
   trail_rtn(X).
instr(move(EA,VI)) -->>
   ea_m(EA, X),
   var_i(set, VI, X), !.
instr(push(EA,R,N)) -->>
   ea_p(EA, X),
   {hreg(R)},
   [get(R,Y)]:regs,
   [set(Y,X)]:heap,
   {pos(N)},
   add_word(Y, N, YN),
   [set(R,YN)]:regs.
instr(adda(R,S,T)) -->>
   {hreg(R)},
   [get(R,X)]:regs,
   numreg(S, Off),
   add_word(X, Off, NX),
   {hreg(T)},
   [set(T,NX)]:regs.
instr(pad(N)) -->>
   [get(r(h),H)]:regs,
   {pos(N)},
   add_word(H, N, NewH),
   [set(r(h),NewH)]:regs.

% 2. Conditional control flow instructions:
instr(choice(1/N,Rs,L)) -->> {pos(N), N>1, regs(Rs), lbl(L)}, !,
   save_choice_regs(Rs, NewB),
   {ins(NewB, r(retry), L)},
   [get(r(tr),TR)]:regs, {ins(NewB, r(tr), TR)},
   [get(r(e),  E)]:regs, {ins(NewB, r(e),   E)},
   [get(r(cp),CP)]:regs, {ins(NewB, r(cp), CP)},
   [get(r(b),  B)]:regs, {ins(NewB, r(b),   B)},
   [get(r(h),  H)]:regs, {ins(NewB, r(h),   H)},
   {seal(NewB)},
   [set(r(hb), H)]:regs,
   [set(r(b),NewB)]:regs.
```

```
instr(choice(I/N,Rs,L)) -->> {pos(N), pos(I), 1<I, I<N, regs(Rs), lbl(L)}, !,
   [get(r(b),B)]:regs,
   restore_choice_regs(Rs, B),
   {set(B,r(retry),L,NewB)},
   [set(r(b),NewB)]:regs.
instr(choice(N/N,Rs,L)) -->> {pos(N), regs(Rs), lbl(L)}, !,
   [get(r(b),B)]:regs,
   restore_choice_regs(Rs, B),
   {get(B,r(b),NewB)},
   [set(r(b),NewB)]:regs,
   {get(NewB,r(h),H)},
   [set(r(hb),H)]:regs.
instr(fail) -->>
   [get(r(b),B)]:regs,
   {get(B,r(h),H)},
   [set(r(h),H)]:regs,
   {get(B,r(e),E)},
   [set(r(e),E)]:regs,
   {get(B,r(cp),CP)},
   [set(r(cp),CP)]:regs,
   [get(r(tr),CurTR)]:regs,
   {get(B,r(tr),OldTR)},
   detrail_rtn(CurTR, OldTR),
   {get(B,r(retry),L)},
   [set(r(pc),L)]:regs.
instr(switch(T,V,A,B,C)) -->>
   {a_tag(T)},
   var_i(get, V, X),
   extract_tag(X, TX),
   {lbl(A), lbl(B), lbl(C)},
   switch_rtn(T, TX, A, B, C).
instr(test(Eq,T,V,L)) -->>
   {a_tag(T)},
   var_i(get, V, X),
   extract_tag(X, TX),
   {eq_ne(Eq)},
   {lbl(L)},
   test_rtn(Eq, T, TX, L).
instr(jump(C,A,B,L)) -->>
   {cond(C)},
   numarg(A, XA), {extract_value(XA, VA), check_int(XA)},
   numarg(B, XB), {extract_value(XB, VB), check_int(XB)},
   {lbl(L)},
   jump_cond_rtn(C, VA, VB, L).
instr(move(r(b),V)) -->>
   [get(r(b),B)]:regs,
   a_var(set, V, B).
instr(cut(V)) -->>
   a_var(get, V, X),
   [set(r(b),X)]:regs,
   {get(X,r(h),H)},
   [set(r(hb),H)]:regs.
instr(hash(T,R,N,L)) -->> hash_type(T), pos(N), lbl(L),
   reg(get, R, X),
    hash_indirect(T, X, Y),
    [get(L,PC)]:lblmap,
    hash_lookup(PC, Y, N).
instr(pair(_,_)) -->>
    {error(['Attempt to execute inside a hash table.'])}.

% 3. Arithmetic instructions:
instr(add(A,B,V)) -->> arith(add, A, B, V).
instr(sub(A,B,V)) -->> arith(sub, A, B, V).
instr(mul(A,B,V)) -->> arith(mul, A, B, V).
instr(div(A,B,V)) -->> arith(div, A, B, V).
instr(mod(A,B,V)) -->> arith(mod, A, B, V).
instr(and(A,B,V)) -->> arith(and, A, B, V).
instr( or(A,B,V)) -->> arith( or, A, B, V).
instr(xor(A,B,V)) -->> arith(xor, A, B, V).
instr(not(A,V))   -->> arith(not, A, 0, V).
instr(sll(A,B,V)) -->> arith(sll, A, B, V).
instr(sra(A,B,V)) -->> arith(sra, A, B, V).

% 4. Procedural instructions:
instr(procedure(N/A)) -->> {atom(N), natural(A)}.
instr(call(N/A)) -->> {atom(N), natural(A)},
    [get(r(pc),PC)]:regs,
    [set(r(cp),PC)]:regs,
    [set(r(pc),N/A)]:regs.
instr(return) -->>
    [get(r(cp),PC)]:regs,
    [set(r(pc),PC)]:regs.
instr(simple_call(N/A)) -->> {atom(N), natural(A)},
    [get(r(pc),PC)]:regs,
    [set(r(tmp_cp),PC)]:regs,
    [set(r(pc),N/A)]:regs.
instr(simple_return) -->>
    [get(r(tmp_cp),PC)]:regs,
    [set(r(pc),PC)]:regs.
instr(label(L)) -->> {lbl(L)}.
instr(jump(L)) -->> {lbl(L)},
    [set(r(pc),L)]:regs.
instr(allocate(N)) -->>
    {natural(N)},
    [get(r(e),E)]:regs,
    {ins(NewE, r(e), E)},
    [get(r(cp),CP)]:regs,
    {ins(NewE, r(cp), CP)},
    {seal(NewE)},
    [set(r(e),NewE)]:regs.
instr(deallocate(N)) -->>
    {natural(N)},
    [get(r(e),E)]:regs,
    {get(E,r(e),NewE)},
    {get(E,r(cp),NewCP)},
    [set(r(e),NewE)]:regs,
    [set(r(cp),NewCP)]:regs.
instr(nop) -->> [].
```

```
% 5. Pragma information for translator and reorderer:
% Pragmas are no-ops in the execution.
instr(pragma(P)) -->> {pragma(P)}, !.

% 6. Additions to BAM for the assembly language programmer:
% The meaning of these instructions depends on the underlying architecture,
% so they are not included in this specification.  See the Implementation
% Manual for a discussion of their use.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Pragmas ***

% A variable is a multiple of N.
% Inserted just before loads in readmode unification.
pragma(align(V,N))         :- a_var(V), pos(N).

% Inserted just before a sequence of pushes in writemode unification.
% (The pushes may be interleaved with non-memory moves.)
pragma(push(term(Size)))   :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).

% Specify the tag of a variable.
% (This is useful for processors without explicit tag support.)
pragma(tag(V,T))           :- a_var(V), a_tag(T).

% Length of a hash table.
pragma(hash_length(Len))   :- pos(Len).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Tags ***

a_tag(tatm). /* atom */
a_tag(tint). /* integer */
a_tag(tneg). /* negative integer */
a_tag(tpos). /* nonnegative integer */
a_tag(tstr). /* structure */
a_tag(tlst). /* cons cell */
a_tag(tvar). /* variable */

atom_tag(tatm).

atomic_tag(tatm).
atomic_tag(tint).
atomic_tag(tneg).
atomic_tag(tpos).

pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Addressing modes ***

% Both read and write access:

heap(get, W, X) -->> {ptr_word(W)}, [get(W,X)]:heap.
heap(set, W, X) -->> {ptr_word(W)}, [set(W,X)]:heap.

ptr_word(Tˆ_) :- pointer_tag(T).

reg(get, R, X) -->> {reg(R)}, [get(R,X)]:regs.
reg(set, R, X) -->> {reg(R)}, [set(R,X)]:regs.

reg(r(I)) :- int(I), !.

hreg(R) :- reg(R), !.
hreg(r(h)).

perm(get, P, X) -->> {perm(P)}, [get(r(e),E)]:regs, {get(E,P,X)}.
perm(set, P, X) -->> {perm(P)}, [get(r(e),E)]:regs, {set(E,P,X,NewE)},
             [set(r(e),NewE)]:regs.

perm(p(I)) :- natural(I).

a_var(WR, V, X) -->> reg(WR, V, X), !.
a_var(WR, V, X) -->> perm(WR, V, X).

a_var(Reg)  :- reg(Reg), !.
a_var(Perm) :- perm(Perm).

var_i(WR, [V], X) -->> a_var(get, V, W), heap(WR, W, X), !.
var_i(WR,   V, X) -->> a_var(WR, V, X).

% Read access only:

% An int is its own value:
int(N) :- integer(N).

% An atomic is its own value:
an_atomic(I)       :- int(I), !.
an_atomic(TˆA)     :- atom(A), atom_tag(T), !.
an_atomic(Tˆ(F/N)) :- atom(F), pos(N), atom_tag(T).

arg(Arg, Arg) -->> {an_atomic(Arg)}, !.
arg(Arg,   X) -->> a_var(get, Arg, X).

arg_i(Arg, Arg) -->> {an_atomic(Arg)}, !.
arg_i(Arg,   X) -->> var_i(get, Arg, X).

numreg(Arg, Arg) -->> {int(Arg)}, !.
numreg(Arg,   X) -->> reg(get, Arg, X).
```

```
numarg(Arg, Arg) -->> {int(Arg)}, !.
numarg(Arg,   X) -->> var_i(get, Arg, X).

var_off([Var+I], X) -->> a_var(get, Var, T), !,
             {pos(I)}, add_word(T, I, T2), [get(T2,X)]:heap.
var_off([Var],   X) -->> a_var(get, Var, T), [get(T,X)]:heap.

% Creating immediate tagged pointer objects:
imm_tag(Tag^(r(h)+D), W) -->> {pointer_tag(Tag)}, !,
   [get(r(h),T)]:regs,
   {pos(D)}, add_word(T, D, X),
   insert_tag(Tag, X, W).
imm_tag(Tag^r(h), W) -->> {pointer_tag(Tag)}, !,
   [get(r(h),X)]:regs,
   insert_tag(Tag, X, W).

% Effective address for equal:
ea_e(Var,    X) -->> a_var(get, Var, X), !.
ea_e(VarOff, X) -->> var_off(VarOff, X).

% Effective address for move:
ea_m(Arg,    X) -->> arg(Arg, X), !.
ea_m(VarOff, X) -->> var_off(VarOff, X), !.
ea_m(T^r(h), X) -->> imm_tag(T^r(h), X).

% Effective address for push:
ea_p(Arg,    X) -->> arg_i(Arg, X), !.
ea_p(T^Y,    X) -->> imm_tag(T^Y, X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Miscellaneous ***

eq_ne(eq). /* Equal */
eq_ne(ne). /* Not equal */

cond(lts). /* Signed less than */
cond(les). /* Signed less than or equal */
cond(gts). /* Signed greater than */
cond(ges). /* Signed greater than or equal */
cond(eq).  /* Equal */
cond(ne).  /* Not equal */

hash_type(atomic).
hash_type(structure).

lbl(fail).
lbl(N/A)      :- atom(N), natural(A).
lbl(l(N/A,I)) :- atom(N), natural(A), natural(I).

nv_flag(nonvar).
nv_flag(var).
nv_flag('?').
```

```
% A list of register numbers:
% (May contain the value ’no’ as well)
regs([]).
regs([R|Set]) :- (int(R); R=no), !, regs(Set).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dereference utilities:

deref_rtn(X, X) -->> {nonvartag(X)}, !.
deref_rtn(X, Y) -->>
   [get(X,X2)]:heap,
   deref_rtn_cont(X, X2, Y).

deref_rtn_cont(X, X, Y) -->> !, {Y=X}.
deref_rtn_cont(_, X, Y) -->> deref_rtn(X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Equal routine:

equal_rtn(X, X, _) -->> !.
equal_rtn(_, _, L) -->> [set(r(pc),L)]:regs.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Switch and test routines:

switch_rtn(_, tvar, A,_,_) -->> !, [set(r(pc),A)]:regs.
switch_rtn(T,   TX, _,B,_) -->> {equivalent_tag(T,TX)},!, [set(r(pc),B)]:regs.
switch_rtn(_,    _, _,_,C) -->> [set(r(pc),C)]:regs.

test_rtn(Eq, T, TX, L) -->> {test_true(Eq, T, TX)}, !, [set(r(pc),L)]:regs.
test_rtn( _, _,  _, _) -->> [].

test_true(eq, T, TX) :- equivalent_tag(T, TX).
test_true(ne, T, TX) :- \+equivalent_tag(T, TX).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Arithmetic utilities:

arith(Op, A, B, V) -->>
   numarg(A, XA), {extract_value(XA, VA), check_int(XA)},
   numarg(B, XB), {extract_value(XB, VB), check_int(XB)},
   arith_operation(Op, VA, VB, VC),
   a_var(set, V, VC).

arith_operation(add, VA, VB, VC) :- VC is VA+VB.
arith_operation(sub, VA, VB, VC) :- VC is VA-VB.
arith_operation(mul, VA, VB, VC) :- VC is VA*VB.
arith_operation(div, VA, VB, VC) :- VC is VA//VB.
arith_operation(mod, VA, VB, VC) :- VC is VA mod VB.
arith_operation(and, VA, VB, VC) :- VC is VA /\ VB.
arith_operation( or, VA, VB, VC) :- VC is VA \/ VB.
arith_operation(xor, VA, VB, VC) :- VC is (VA /\ \(VB)) \/ (VB /\ \(VA)).
arith_operation(not, VA,  _, VC) :- VC is \(VA).
arith_operation(sll, VA, VB, VC) :- VC is VA<<VB.
arith_operation(sra, VA, VB, VC) :- VC is VA>>VB.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Conditional jump:

jump_cond_rtn(C, VA, VB, L) -->> {jump_true(C, VA, VB)}, !, [set(r(pc),L)]:regs.
jump_cond_rtn(_,  _,  _, _) -->> [].

jump_true(lts, VA, VB) :- VA@<VB.
jump_true(gts, VA, VB) :- VA@>VB.
jump_true(les, VA, VB) :- VA@=<VB.
jump_true(ges, VA, VB) :- VA@>=VB.
jump_true( eq, VA, VB) :- VA==VB.
jump_true( ne, VA, VB) :- VA\==VB.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Hash table utilities:

hash_lookup(PC, X, N) -->>
   {PC1 is PC+1},
   [get(PC1,pragma(hash_length(N)))]:code,
   {PC2 is PC1+1},
   {PCN is PC1+N},
   hash_lookup_2(PC2, PCN, X).

hash_lookup_2(PC, PCN, _) -->> {PC>PCN}, !.
hash_lookup_2(PC, PCN, X) -->> {PC=<PCN},
   [get(PC,pair(E,L))]:code,
   {E=X},
   !,
   [set(r(pc),L)]:regs.
hash_lookup_2(PC, PCN, X) -->> {PC=<PCN},
   {PC1 is PC+1},
   hash_lookup_2(PC1, PCN, X).

% Indirection needed for structures because main functor is in memory:
hash_indirect(atomic,    X, X) -->> [].
hash_indirect(structure, X, Y) -->> [get(X,Y)]:heap.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Choice point and fail utilities:

save_choice_regs([], _) -->> [].
save_choice_regs([no|Rs], B) -->> !,
   save_choice_regs(Rs, B).
save_choice_regs([I|Rs], B) -->>
   [get(r(I),R)]:regs,
    {ins(B, r(I), R)},
    save_choice_regs(Rs, B).

restore_choice_regs([], _) -->> [].
restore_choice_regs([no|Rs], B) -->> !,
    restore_choice_regs(Rs, B).
restore_choice_regs([I|Rs], B) -->>
    {get(B, r(I), R)},
    [set(r(I),R)]:regs,
    restore_choice_regs(Rs, B).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Trailing and detrailing:

trail_rtn(X) -->>
    [get(r(hb),HB)]:regs,
    cmp_trail(X, HB).

cmp_trail(X, HB) -->> {less_trail(X, HB)}, !,
    [get(r(tr),TR)]:regs,
    [set(TR,X)]:trail,
    {TR1 is TR+1},
    [set(r(tr),TR1)]:regs.
cmp_trail(_, _) -->> [].

less_trail(_^X, _^Y) :- X<Y.

% Restore to unbound the variables on the trail between OldTR and CurTR.
detrail_rtn(CurTR, OldTR) -->> {CurTR=<OldTR}, !.
detrail_rtn(CurTR, OldTR) -->> {CurTR>OldTR},
    {CurTR1 is CurTR-1},
    [get(CurTR1,V)]:trail,
    [set(V,V)]:heap,
    detrail_rtn(CurTR1, OldTR).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% General unification routine:

unify_rtn(W1, W2, L) -->>
    unify_rtn_2(W1, W2, Flag),
    unify_end(Flag, L).

unify_end(success, _) -->> [].
% For later: detrailing if L\fail.
unify_end(fail, L) -->> [set(r(pc),L)]:regs.

unify_rtn_2(W1, W2, Flag) -->>
    {extract_tag_value(W1, T1, V1)},
    {extract_tag_value(W2, T2, V2)},
    unify_rtn_2(T1, V1, T2, V2, Flag).

unify_rtn_2(tvar, V1, NTag, V2, success) -->> {NTag\==tvar}, !,
    trail_rtn(tvar^V1),
    {make_word(NTag, V2, Word)},
    [set(tvar^V1,Word)]:heap.
unify_rtn_2(NTag, V2, tvar, V1, success) -->> {NTag\==tvar}, !,
    trail_rtn(tvar^V1),
    {make_word(NTag, V2, Word)},
    [set(tvar^V1,Word)]:heap.
unify_rtn_2(tvar, V1, tvar, V2, success) -->> !,
    unify_varvar(V1, V2).
% Matching atomic tags:
unify_rtn_2(ATag, V1, BTag, V2, Flag) -->>
    {atomic_tag(ATag)},
    {atomic_tag(BTag)},
    {equivalent_tag(ATag, BTag)},
    !,
    unify_atm(V1, V2, Flag).
% Non-matching nonvariable tags:
unify_rtn_2(ATag, _, BTag, _, fail) -->>
    {ATag\==tvar, BTag\==tvar},
    {\+equivalent_tag(ATag, BTag)},
    !.
% Matching pointer tags (recursive case):
unify_rtn_2(ATag, V1, ATag, V2, Flag) -->>
    {pointer_tag(ATag)},
    get_size(ATag, V1, Sz),
    unify_rtn_args_2(0, Sz, ATag, V1, V2, Flag).

% The term’s Size is the maximum offset needed to traverse the term in memory.
get_size(tlst, _, 1) -->> [].
get_size(tstr, V, N) -->>
    [get(tstr^V,Func)]:heap,
    {Func=(tatm^(_/N))}.

unify_rtn_args_2(N, Sz, _, _, _, success) -->> {N>Sz}, !.
unify_rtn_args_2(N, Sz, T, V, W, Flag) -->> {N=<Sz}, !,
    {VN is V+N},
    {WN is W+N},
    [get(T^VN,VX)]:heap, deref_rtn(VX, DVX),
    [get(T^WN,WX)]:heap, deref_rtn(WX, DWX),
    unify_rtn_2(DVX, DWX, F),
    {N1 is N+1},
    unify_rtn_args_3(F, N1, Sz, T, V, W, Flag).

% Continue with other arguments if argument unification succeeded:
unify_rtn_args_3(fail, _, _, _, _, _, fail) -->> [].
unify_rtn_args_3(success, N1, Sz, T, V, W, Flag) -->>
    unify_rtn_args_2(N1, Sz, T, V, W, Flag).

% Unifying value parts of two atomic terms with equivalent tag:
unify_atm(V, V, success) -->> !.
unify_atm(_, _, fail) -->> [].

% Unifying two variables: bind youngest to oldest, trail youngest.
unify_varvar(V1, V2) -->> {V1>V2}, !,
   trail_rtn(tvar^V1),
   [set(tvar^V1,tvar^V2)]:heap.
unify_varvar(V1, V2) -->> {V1=<V2}, !,
   trail_rtn(tvar^V2),
   [set(tvar^V2,tvar^V1)]:heap.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Simple type utilities:

ground(X) :- nonvar(X), functor(X, _, N), ground(N, X).

ground(N, _) :- N=:=0, !.
ground(N, X) :- N=\=0, arg(N, X, A), ground(A), N1 is N-1, ground(N1, X).

natural(N) :- integer(N), N>=0.
pos(N)  :- integer(N), N>0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Word, tag, and value manipulation utilities:

% This takes into account the relationship between tpos, tneg and tint.
% For integers it extracts tpos or tneg tags and the absolute value
% of the integer.  It creates the correct integer, given the tpos, tneg
% or tint tags.

equivalent_tag(T,       T) :- !.
equivalent_tag(tint, tpos) :- !.
equivalent_tag(tint, tneg).

extract_tag(N, tpos) :- integer(N), N>=0, !.
extract_tag(N, tneg) :- integer(N), N<0, !.
extract_tag(T^_,  T).

extract_value(N,   N) :- int(N), N>=0, !.
extract_value(N,   M) :- int(N), N<0, !, M is -N.
extract_value(_^V, V).

extract_tag_value(W, T, V) :-
   extract_tag(W, T),
   extract_value(W, V).

nonvartag(I) :- int(I), !.
nonvartag(T^_) :- \+T=tvar.

% Only used for pointer tags:
insert_tag(T, _^V, T^V).

make_word(tint, I, I) :- !.
make_word(tpos, I, I) :- !.
make_word(tneg, N, I) :- !, I is -N.
make_word(T, V, T^V).
```

```
add_word(T^I, J, T^K) :- K is I+J.

% Eventually, print out value of PC:
check_int(I) :- int(I), !.
check_int(_) :-
   error(['Operand of conditional is not an integer.']).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Table utilities:

% This code implements a mutable array, represented as a binary tree.

% Insert a value in logarithmic time and constant space:
% This predicate is used in this program only to create the array,
% although it can also be used to access array elements.
ins(T, I, V) :- hash(I, H), ins_2(T, H, V).

ins_2(node(N,W,L,R), I, V) :- ins_2(N, W, L, R, I, V).

ins_2(N, V, _, _, I, V) :- I=N, !.
ins_2(N, _, L, R, I, V) :-
        compare(Order, I, N),
        ins_2(Order, I, V, L, R).

ins_2(<, I, V, L, _) :- ins_2(L, I, V).
ins_2(>, I, V, _, R) :- ins_2(R, I, V).

% Access a value in logarithmic time and constant space:
% This predicate cannot be used to create the array incrementally,
% but it is faster than ins/3.
get(T, I, V) :- hash(I, H), get_2(T, H, V).

get_2(node(N,W,L,R), I, V) :-
        compare(Order, I, N),
        get_3(Order, I, V, W, L, R).

get_3(<, I, V, _, L, _) :- get_2(L, I, V).
get_3(=, _, V, W, _, _) :- V=W.
get_3(>, I, V, _, _, R) :- get_2(R, I, V).

% Update an array in logarithmic time and space:
set(T, I, V, U) :- hash(I, H), set_2(T, H, V, U).

set_2(leaf,          I, V, node(I,V,leaf,leaf)).
set_2(node(N,W,L,R), I, V, node(N,NW,NL,NR)) :-
        compare(Order, I, N),
        set_3(Order, I, V, W, L, R, NW, NL, NR).

set_3(<, I, V, W, L, R, W, NL, R) :- set_2(L, I, V, NL).
set_3(=, _, V, _, L, R, V, L, R).
set_3(>, I, V, W, L, R, W, L, NR) :- set_2(R, I, V, NR).

% Prevent any further insertions in the array:
```

```
seal(leaf).
seal(node(_,_,L,R)) :- seal(L), seal(R).

% Print values of array in sorted order:
print_array(Term) :-
   flat_array(Term, 2, Flat),
   print_list(Flat).

print_list([]).
print_list([(A->B)|L]) :-
   write(A), put(9), write('= '), write(B), nl,
   print_list(L).

flat_array(Term, N, Sort) :-
   N>0, N1 is N-1,
   flat_array(Term, N1, Flat, []), !,
   sort(Flat, Sort).
flat_array(leaf,          N,    []) :- N=:=0, !.
flat_array(node(_,_,_,_), N, '...') :- N=:=0, !.
flat_array(Term, _, Term).

flat_array(leaf, _) --> [].
flat_array(node(H,T,L,R), N) -->
   flat_array(L, N),
   {hash(H, I)},
   {flat_array(T, N, F)},
   [(I->F)],
   flat_array(R, N).

% Invertible hash function:
% Bit inversion of the integer components of a ground term.  Other parts are
% unchanged.  This one inverts the low 16 bits.  It can be changed by changing
% the last argument of bit_invert/3.
hash(I, H) :- integer(I), !, bit_invert(I, H, 16).
hash(T, H) :- functor(T, Na, Ar), functor(H, Na, Ar), hash_2(Ar, T, H).

hash_2(0, _, _) :- !.
hash_2(N, T, H) :- N>0,
   arg(N, T, X),
   arg(N, H, Y),
   hash(X, Y),
   N1 is N-1,
   hash_2(N1, T, H).

bit_invert(0, 0, _) :- !.
bit_invert(N, I, B) :- N>0,
   L is N>>1,
   R is N/\1,
   B1 is B-1,
   bit_invert(L, LI, B1),
   I is R*(1<<B) + LI.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Error handling:

error(L) :-
   write('*** Error: '),
   error_loop(L),
   write(' ***'), nl.

error_loop([]).
error_loop([M|L]) :- write(M), error_loop(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primitive version of write:

write_rtn -->>
   [get(r(0),X)]:regs,
   write_rtn(X).

write_rtn(tvar^V) -->> !, {write('_'), write(V)}.
write_rtn(I) -->> {int(I)}, !, {write(I)}.
write_rtn(tatm^(F/N)) -->> !, {write(''''), write(F/N), write('''')}.
write_rtn(tatm^A) -->> !, {write(A)}.
write_rtn(tlst^V) -->> !,
   {W is V+1},
   [get(tlst^V,Head)]:heap,
   [get(tlst^W,Tail)]:heap,
   deref_rtn(Head, DHead),
   deref_rtn(Tail, DTail),
   {write('[')},
   write_rtn(DHead),
   {write('|')},
   write_rtn(DTail),
   {write(']')}.
write_rtn(tstr^V) -->> !,
   [get(tstr^V,tatm^(F/N))]:heap,
   {write(F), write('(')},
   write_arg(V, 1),
   write_args(2, N, V),
   {write(')')}.

write_args(I, N, _) -->> {I>N}, !.
write_args(I, N, V) -->> {I=<N}, !,
   {I1 is I+1},
   {write(',')},
   write_arg(V, I),
   write_args(I1, N, V).

write_arg(V, I) -->>
   {W is V+I},
   [get(tstr^W,X)]:heap,
   deref_rtn(X, DX),
   write_rtn(DX).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix D

Semantics of the Berkeley Abstract Machine

1. Introduction

This appendix gives an English-language description of the semantics of the Berkeley Abstract Machine (BAM) as comments attached to a Prolog specification of its syntax. The BAM is intended to operate on the same data structures as the Warren Abstract Machine (WAM), therefore some familiarity with the WAM is an advantage. The semantics are represented by short descriptions supplemented by pseudo-code and examples where necessary.

The BAM is designed to be simple and easily translated to most general-purpose processors. Many of its optimizations apply to any processor, for example the streamlined choice point management and the use of write-once permanent variables to simplify trailing. Although the first target is the VLSI-BAM processor, we have built translators for other processors including the MIPS and the MC68020. Pragmas give information that is used to obtain the best translation for different processors.

The instruction set is divided in six categories, each in a different section. Each section starts with a box giving the syntax of the instructions presented in that section. This is followed by a description of the instructions' actions. Section 2 gives the unification instructions. Section 3 gives the conditional control flow instructions. Section 4 gives the arithmetic instructions. Section 5 gives the procedural control flow instructions. Section 6 gives the pragmas, which contain information that allows better translation. Section 7 gives the user instructions, additions to the BAM that are never output by the compiler but are intended for the BAM assembly programmer. The last section defines the syntax and semantics of the addressing modes used in the instructions.

In explaining the semantics, a few assumptions are made about the data representation. An infinite number of registers is assumed; the translator should map registers of sufficiently large index to memory. A tagged architecture is assumed; i.e. each word contains a tag and a value field which are treated as separate entities in some instructions and as a unit in other instructions. A load-store architecture is assumed; almost any architecture has a subset of instructions that satisfy this assumption. The actual details of the translation to the target architecture are not given since they depend on the characteristics of the architecture. These characteristics include the number of registers, the addressing modes, hardware support for certain features (tagging, dereferencing, trailing, etc.), the precise format of choice points and environments, and so forth.

2. Unification instructions

Unification syntax

```
instr(deref(V,W))         :- var_i(V), var_i(W).
instr(equal(EA,A,L))      :- ea_e(EA), arg_i(A), lbl(L).
instr(unify(V,W,F,G,L))   :- var_i(V),var_i(W),nv_flag(F),nv_flag(G),lbl(L).
instr(trail(V))           :- var_i(V).
instr(move(EA,VI))        :- ea_m(EA), var_i(VI).
instr(push(EA,R,N))       :- ea_p(EA), hreg(R), pos(N).
instr(adda(R,S,T))        :- numreg(R), numreg(S), hreg(T).
instr(pad(N))             :- pos(N).
instr(unify_atomic(V,I,L)):- var_i(V), an_atomic(I), lbl(L).
instr(fail).
```

`deref(V,W)`	Dereference the argument V and store the result in W. The argument V is unchanged. This is the only instruction which dereferences its argument. All other instructions assume that their arguments are dereferenced. Giving the dereference instruction two arguments simplifies the implementation of write-once permanent variables and makes a fast implementation of trailing possible.
`equal(X,Y,L)`	Compare X to Y and branch to L if they are not equal. The comparison is a full word operation, equivalent to ‘‘eq’’ in Lisp. It is assumed that X and Y are dereferenced.
`unify(X,Y,T,U,L)`	Perform a general unification of X and Y, and branch to L if it fails. Always binds oldest variables to the youngest. In the failure case all bindings are undone. It is assumed that X and Y are dereferenced. The two parameters T and U are added as an optimization, and may be safely ignored. They are flags (with values `'?'`, `var`, or `nonvar`) that say whether it is known if X and Y are variables or nonvariables. With this information a better translation to the target processor can be done.
`trail(X)`	Push the address of X on the trail stack if the trail condition X<`r(hb)` is satisfied. It is assumed that X is a dereferenced unbound variable, i.e. it has a `tvar` tag. Only one comparison is necessary for the trail check. The state register `r(hb)` points to the heap location which was the top of the heap when the most recent choice point was created.
`move(X,Y)`	Move X to Y. Depending on the addressing mode, this instruction does a load or store or creates a tagged value.
`push(X,R,N)`	Push X on the stack with stack pointer R, then increment R by N. This instruction is used for write mode unification.
`adda(X,Y,R)`	Add X and Y into R. This is a full word operation which never traps, unlike the arithmetic instructions in section 4. This instruction is used to allocate space for uninitialized variables. The second argument Y is an offset which is scaled properly by the translator (i.e. it is unchanged for the VLSI-BAM since it is word-addressed, and it is multiplied by 4 for the MIPS, since it is byte-addressed).

`pad(N)`	Add N words to the heap pointer `r(h)`. This is a full word operation which never traps, unlike the arithmetic instructions in section 4. It is used to ensure the correct alignment of compound terms. The space reserved by pad will never be stored to. If the increment is a multiple of the alignment then the pad disappears. The increment is scaled properly by the translator (see previous description of `adda`).
`unify_atomic(X,Y,L)`	Unify the variable X with the atomic term Y, and branch to L if it fails. It is assumed that X is dereferenced. The `unify_atomic` instruction is a special case of general unification that is added to reduce code size in the VLSI-BAM processor. There is a compiler option to enable or disable the generation of this instruction.
`fail`	Untrail all variable bindings and jump to the retry address. Do not restore argument registers. Argument registers are restored by the choice point management instructions.

3. Conditional control flow instructions

Clause selection syntax

```
instr(switch(T,V,A,B,C))  :- a_tag(T), var_i(V), lbl(A),lbl(B),lbl(C).
instr(choice(I/N,Rs,L))   :- pos(I), pos(N), I=<N, lbl(L), regs(Rs).
instr(test(Eq,T,V,L))     :- eq_ne(Eq), var_i(V), a_tag(T), lbl(L).
instr(jump(C,A,B,L))      :- cond(C), numarg_i(A), numarg_i(B),lbl(L).
instr(move(CH,V))         :- a_var(V), choice_ptr(CH).
instr(cut(V))             :- a_var(V).
instr(hash(T,R,N,L))      :- hash_type(T), reg(R), pos(N), lbl(L).
instr(pair(E,L))          :- an_atomic(E), lbl(L).
```

`switch(T,R,A,B,C)`	A three-way branch: branch to the label A, B, C depending on whether the tag of R is `tvar`, T, or any other value. The label `fail` is not an address, but denotes a branch to the global failure routine. It is assumed that R is dereferenced.
`choice(I/N,RS,L)`	The choice point management instruction for choosing clause I out of N clauses. Choice points are of variable size. The semantics of choice depends on I as follows:

I=1	Create a choice point with retry address L. Save in it the registers listed in RS.
1<I<N	Restore the registers mentioned in RS from the choice point, ignoring `no` terms. The `no` terms make it possible to know the position of the registers in the choice point without an explicit size field in the choice point. Update the retry address to L.
I=N	Restore the registers mentioned in RS, ignoring `no` terms. Remove the choice point. (L will always be `fail` when I=N.)

The above notation is consistent with three possible implementations (in order of decreasing efficiency): (1) The implementation given above, in which only those registers listed in RS are saved and restored, and the choice point does not have a size field. Restoring registers is done by the choice instructions, not by the fail instruction. The compiler does an effort to minimize the set of registers mentioned in RS. (2) Saving all registers up to the maximum register listed in RS. In this case the choice points are of variable size, and the `no` terms in RS are ignored. The notation is consistent with choice points containing a size field. (3) Always saving and restoring all registers. In this case the choice points are of fixed size, the RS argument is ignored, and the fail instruction restores the registers. In this case the semantics correspond to the try, retry, and trust instructions of the WAM.

`test(E,T,X,L)`

Branch to label L if the tag of X is equal/not equal to T. Equality/nonequality is selected by the value of E. The label `fail` is not an address, but denotes a branch to the global failure routine. It is assumed that X is dereferenced.

`jump(C,X,Y,L)`

Compare X and Y and jump to L if the comparison is true. The kind of comparison is given by C. This instruction traps if either argument is not an integer. The label `fail` is not an address, but denotes a branch to the global failure routine.

`cut(X)`

Implement the cut operation. Move X into the `r(b)` register; also move the value of `r(h)` in this choice point into the `r(hb)` register. The latter move is an optimization that reduces the number of trailed variables, but is not needed for correctness. The compiler ensures that X contains a pointer to the choice point which was most recent when the current predicate was entered.

`hash(T,R,N,L)`

Look up register R in a hash table located at label L. The hash table contains atomic terms (when T=`atomic`) or the main functors of structures (when T=`structure`). If R is not in the hash table, then execution falls through to the next instruction. Otherwise execution continues at the label contained in the hash table. When T=`structure` the compiler guarantees that R points to a structure. The following is an example of hash table code:

```
 ...
hash(Type,Reg,N,Lbl). ; Hash Reg into table at Lbl
 ...                  ; Fall through if not present


label(Lbl).           ; The hash table
hash_length(N).       ; Length of the hash table
pair(E1,L1).          ; N entries
pair(E2,L2).
 ...
pair(Ei,Li).          ; Jump to Li if Reg = Ei
 ...
pair(EN,LN).
```

`pair(E,L)`

A hash table entry. E is either an atom or the main functor of a structure. The label L is the address where execution continues if the supplied value matches E.

4. Arithmetic instructions

Arithmetic syntax	
`instr(add(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(sub(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(mul(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(div(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(and(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr( or(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(xor(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(not(A,V))`	`:- numarg_i(A),              a_var(V).`
`instr(sll(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`
`instr(sra(A,B,V))`	`:- numarg_i(A), numarg_i(B), a_var(V).`

All arithmetic instructions assume that their operands are dereferenced and destructively overwrite the result register. All perform operations on integers with correct tag and return a result with correct tag, trapping if either operand or the result is not a integer. Arithmetic semantics are:

`add(X,Y,Z)`	Z ← X+Y
`sub(X,Y,Z)`	Z ← X-Y
`mul(X,Y,Z)`	Z ← X*Y
`div(X,Y,Z)`	Z ← X/Y
`and(X,Y,Z)`	Z ← X and Y (bitwise and)
`or(X,Y,Z)`	Z ← X or Y (bitwise or)
`xor(X,Y,Z)`	Z ← X xor Y (bitwise exclusive or)
`sll(X,Y,Z)`	Z ← X << Y (logical shift of X left Y places)
`sra(X,Y,Z)`	Z ← X >> Y (arithmetic shift of X right Y places)
`not(X,Z)`	Z ← not X (bitwise invert X into Z)

5. Procedural control flow instructions

Procedural syntax	
`instr(procedure(N/A))`	`:- atom(N), natural(A).`
`instr(call(N/A))`	`:- atom(N), natural(A).`
`instr(return).`	
`instr(simple_call(N/A))`	`:- atom(N), natural(A).`
`instr(simple_return).`	
`instr(label(L))`	`:- lbl(L).`
`instr(jump(L))`	`:- lbl(L).`
`instr(allocate(Perms))`	`:- natural(Perms).`
`instr(deallocate(Perms))`	`:- natural(Perms).`

`procedure(P)`	The entry point of procedure P.
`call(N/A)`	Call the procedure `N/A`, assuming a fixed location for the arguments. The arguments of `N/A` are sequentially loaded into argument registers. By default the registers used are numbered from zero, i.e. `r(0)`, `r(1)`, ... This call is used for all user-defined predicates. It may be nested, but must be surrounded by an allocate-deallocate pair when used in the body of a predicate.
`return`	Return from a call.

`simple_call(N/A)`	Simple call of the procedure `N/A`, assuming the same argument passing as `call(N/A)`. This is a one-level call; it may not be nested. It does not require a surrounding allocate-deallocate pair. It can be implemented by saving the return address in a fixed register. This instruction is useful for interfacing with assembly routines.
`simple_return`	Return from a simple call.
`label(L)`	Denotes a branch destination. The label `fail` is not an address, but denotes a branch to the global failure routine.
`jump(L)`	Jump unconditionally to label L. The label may be to the first instruction of another procedure `N/A` or it may be internal to the current procedure. The label `fail` is not an address, but denotes a branch to the global failure routine.
`allocate(N)`	Create an environment of size N on the local stack, i.e. a new set of N permanent variables which are denoted by `p(I)`. Typically, the only state registers stored in the environment are `r(e)` and `r(cp)`. The environment must NOT contain the `r(b)` register.
`deallocate(N)`	Remove the top-most environment (which is of size N) from the local stack.

6. Pragmas

Pragma syntax

```
instr(pragma(Pragma)) :- pragma(Pragma).

pragma(align(V,N))          :- a_var(V), pos(N).
pragma(push(term(Size)))    :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).
pragma(tag(V,T))            :- a_var(V), a_tag(T).
pragma(hash_length(Len))    :- pos(Len).
```

`align(V,N)`	At this point the contents of register or permanent V are a multiple of N. This information helps the reordering stage to generate double-word load instructions for the VLSI-BAM processor.
`hash_length(N)`	N is the length of the hash table starting at this point.
`push(term(S))`	At this point a block of push instructions is about to create a term of size S on the heap.
`push(cons)`	At this point a cons cell (of size two words) is about to be created on the heap. This information helps the reordering stage to generate double-word push instructions for the VLSI-BAM processor.
`push(structure(A))`	At this point a structure of arity A is about to be created on the heap. This information helps the reordering stage to generate double-word push instructions for the BAM processor.
`push(variable)`	At this point an unbound, initialized variable is about to be created on the heap.
`hash_length(N)`	This is the start of a hash table of length N.

`tag(V,T)`	The contents of variable V have tag T. This pragma precedes a load or a store with address V. It is used to make loads and stores efficient for processors which do not have explicit tag support.

7. User instructions

This section describes the parts of the BAM language that are never output by the compiler, but only used by the BAM assembly programmer. This is used to write the run-time system in BAM code, so that it is as portable as possible. Additional instructions are jump to register address, creating and decomposing tagged words, non-trapping full-word arithmetic, non-trapping full-word unsigned comparison, and trailing for backtrackable destructive assignment. Additional registers are used in implementing the run-time system, and can be mapped to memory locations.

Additional instructions

```
instr(I) :- user_instr(I).

user_instr(jump_reg(R))      :- reg(R).
user_instr(jump_nt(C,A,B,L)):- cond(C),numarg_i(A),numarg_i(B),lbl(L).
user_instr(ord(A,B))         :- arg(A), a_var(B).
user_instr(val(T,A,V))       :- a_tag(T),    numarg_i(A), a_var(V).
user_instr(add_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sub_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(and_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr( or_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(xor_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(not_nt(A,V))      :- numarg_i(A),              a_var(V).
user_instr(sll_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(sra_nt(A,B,V))    :- numarg_i(A), numarg_i(B), a_var(V).
user_instr(trail_bda(X))     :- a_var(X).

user_reg(r(A))               :- atom(A).
```

`jump_reg(R)`	Jump unconditionally to the address stored in register R.
`jump_nt(C,A,B,L)`	Compare A and B and jump to L if the comparison is true. The kind of comparison is given by C. This instruction does a full word comparison and never traps. The label `fail` is not an address, but denotes a branch to the global failure routine.
`ord(A,B)`	Store in B the machine integer that corresponds to the atom or integer in A. This function strips the tag from A, and therefore depends on the target machine and the program that is compiled. It is used to convert atoms and integers into table indices.
`val(T,A,V)`	Create a tagged word in B by combining the tag T and the machine integer in A. This function is the inverse of `ord(A,B)`: In the sequence `ord(A1,B), val(T,B,A2)` the argument `A2` will receive an identical value to `A1` if `T` is the tag of `A1`.

`add_nt(A,B,V)`
`sub_nt(A,B,V)`
`and_nt(A,B,V)`
` or_nt(A,B,V)`
`xor_nt(A,B,V)`
`not_nt(A,B,V)`
`sll_nt(A,B,V)`
`sra_nt(A,B,V)`
	These arithmetic instructions destructively overwrite the result register All perform operations on full words, return a full word, and never trap. See the previous section on arithmetic for a description of the operations performed.

`trail_bda(X)`	Push the address and value of X on the trail stack if the trail condition X<`r(hb)` is satisfied. It is assumed that X is dereferenced. When detrailing, the old value of X is restored. This is used to implement backtrackable destructive assignment. Only one comparison is necessary for the trail check. The state register `r(hb)` points to the heap location which was the top of the heap when the most recent choice point was created.

8. Instruction arguments

This section defines the syntax of the instructions' arguments.

Addressing modes for equal, move and push

```
% Effective address for equal:
ea_e(Var)        :- a_var(Var).
ea_e(VarOff)     :- var_off(VarOff).

% Effective address for move:
ea_m(Arg)        :- arg(Arg).
ea_m(VarOff)     :- var_off(VarOff).
ea_m(Tag^H)      :- pointer_tag(Tag), heap_ptr(H).

% Effective address for push:
ea_p(Arg)        :- arg_i(Arg).
ea_p(Tag^H)      :- pointer_tag(Tag), heap_ptr(H).
ea_p(Tag^(H+D))  :- pointer_tag(Tag), pos(D), heap_ptr(H).
```

Other addressing modes

```
heap_ptr(r(h)).
choice_ptr(r(b)).

reg(r(I))          :- int(I).
reg(T)             :- user_reg(T).

hreg(R)            :- reg(R).
hreg(R)            :- heap_ptr(R).

perm(p(I))         :- natural(I).

an_atomic(I)       :- int(I).
an_atomic(TˆA)     :- atom(A), atom_tag(T).
an_atomic(Tˆ(F/N)) :- atom(F), pos(N), atom_tag(T).

a_var(Reg)         :- reg(Reg).
a_var(Perm)        :- perm(Perm).

arg(Arg)           :- a_var(Arg).
arg(Arg)           :- an_atomic(Arg).

var_i(Var)         :- a_var(Var).
var_i([Var])       :- a_var(Var).

arg_i(Arg)         :- var_i(Arg).
arg_i(Arg)         :- an_atomic(Arg).

numreg(Arg)        :- reg(Arg).
numreg(Arg)        :- int(Arg).

numarg_i(Arg)      :- var_i(Arg).
numarg_i(Arg)      :- int(Arg).

var_off([Var])     :- a_var(Var).
var_off([Var+I])   :- a_var(Var), pos(I).
```

Tag syntax

```
a_tag(tatm). /* atom */
a_tag(tint). /* integer */
a_tag(tneg). /* negative integer */
a_tag(tpos). /* nonnegative integer */
a_tag(tstr). /* structure */
a_tag(tlst). /* cons cell */
a_tag(tvar). /* variable */

atom_tag(tatm).

pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).
```

Conditionals syntax

```
eq_ne(eq).
eq_ne(ne).

cond(eq).  /* Equal */
cond(ne).  /* Not equal */
cond(lts). /* Signed less than */
cond(les). /* Signed less than or equal */
cond(gts). /* Signed greater than */
cond(ges). /* Signed greater than or equal */
```

Miscellaneous syntax

```
hash_type(atomic).
hash_type(structure).

lbl(fail).
lbl(N/A)      :- atom(N), natural(A).
lbl(l(N/A,I)) :- atom(N), natural(A), int(I).

nv_flag(nonvar).
nv_flag(var).
nv_flag('?').

% A list of register numbers:
% (May contain the value 'no' as well)
regs([]).
regs([R|Set]) :- (int(R); R=no), regs(Set).
```

Utility predicates

```
ground(X) :- nonvar(X), functor(X, _, N), ground(N, X).

ground(N, _) :- N=:=0.
ground(N, X) :- N=\=0, arg(N,X,A), ground(A), N1 is N-1, ground(N1,X).

int(N)     :- integer(N).
natural(N) :- integer(N), N>=0.
pos(N)  :- integer(N), N>0.
```

Appendix E

Extended DCG notation: A tool for applicative programming in Prolog

1. Introduction

This appendix describes a preprocessor that simplifies purely applicative programming in Prolog. The preprocessor generalizes Prolog's Definite Clause Grammar (DCG) notation to allow programming with multiple accumulators. It has been an indispensable tool in the development of the Aquarius Prolog compiler. Its use is transparent in versions of Prolog that conform to the Edinburgh standard. The preprocessor and a user manual are available by anonymous ftp to arpa.berkeley.edu.

It is desirable to program in a purely applicative style, i.e. within the pure logical subset of Prolog. In that case a predicate's meaning depends only on its definition, and not on any outside information. This has two important advantages. First, it greatly simplifies verifying correctness. Simple inspection is often sufficient. Second, since all information is passed locally, it makes the program more amenable to parallel execution. However, in practice the number of arguments of predicates written in this style is large, which makes writing and maintaining them difficult. Two ways of getting around this problem are (1) to encapsulate information in compound structures which are passed in single arguments, and (2) to use global instead of local information. Both of these techniques are commonly used in imperative languages such as C, but neither is a satisfying way to program in Prolog, for the following reasons:

- Because Prolog is a single-assignment language, modifying encapsulated information requires a time-consuming copy of the entire structure. Sophisticated optimizations could make this efficient, but compilers implementing them do not yet exist.
- Using global information destroys the advantages of programming in an applicative style, such as the ease of mathematical analysis and the suitability for parallel execution.

A third approach with neither of the above disadvantages is extending Prolog to allow an arbitrary number of arguments without increasing the size of the source code. The extended Prolog is translated into standard Prolog by a preprocessor. This report describes an extension to Prolog's Definite Clause Grammar notation that implements this idea.

2. Definite Clause Grammar (DCG) notation

DCG notation was developed as the result of research in natural language parsing and understanding [Pereira & Warren 1980]. It allows the specification of a class of attributed unification grammars with semantic actions. These grammars are strictly more powerful than context-free grammars. Prologs that conform to the Edinburgh standard [Clocksin & Mellish 1981] provide a built-in preprocessor that translates clauses written in DCG notation into standard Prolog.

An important Prolog programming technique is the accumulator [Sterling & Shapiro 1986]. The DCG notation implements a single implicit accumulator. For example, the DCG clause:

```
term(S) --> factor(A), [+], factor(B), {S is A+B}.
```

is translated internally into the Prolog clause:

```
term(S,X1,X4) :- factor(A,X1,X2), X2=[+|X3], factor(B,X3,X4), S is A+B.
```

Each predicate is given two additional arguments. Chaining together these arguments implements the accumulator.

3. Extending the DCG notation

The DCG notation is a concise and clear way to express the use of a single accumulator. However, in the development of large Prolog programs I have found it useful to carry more than one accumulator. If written explicitly, each accumulator requires two additional arguments, and these arguments must be chained together. This requires the invention of many arbitrary variable names, and the chance of introducing errors is large. Modifying or extending this code, for example to add another accumulator, is tedious.

One way to solve this problem is to extend the DCG notation. The extension described here allows for an unlimited number of named accumulators, and handles all the tedium of parameter passing. Each accumulator requires a single Prolog fact as its declaration. The bulk of the program source does not depend on the number of accumulators, so maintaining and extending it is simplified. For single accumulators the notation defaults to the standard DCG notation.

Other extensions to the DCG notation have been proposed, for example Extraposition Grammars [Pereira 1981] and Definite Clause Translation Grammars [Abramson 1984]. The motivation for these extensions is natural-language analysis, and they are not directly useful as aids in program construction.

4. An example

To illustrate the extended notation, consider the following Prolog predicate which converts infix expressions containing identifiers, integers, and addition (+) into machine code for a simple stack machine, and also calculates the size of the code:

```
expr_code(A+B, S1, S4, C1, C4) :-
     expr_code(A, S1, S2, C1, C2),
     expr_code(B, S2, S3, C2, C3),
     C3=[plus|C4],      /* Explicitly accumulate ’plus’ */
     S4 is S3+1.        /* Explicitly add 1 to the size */
expr_code(I, S1, S2, C1, C2) :-
     atomic(I),
     C1=[push(I)|C2],
     S2 is S1+1.
```

This predicate has two accumulators: the machine code and its size. A sample call is `expr_code(a+3+b,0,Size,Code,[])`, which returns the result:

```
Size = 5
Code = [push(a),push(3),plus,push(b),plus]
```

With DCG notation it is possible to hide the code accumulator, although the size is still calculated explicitly:

```
expr_code(A+B, S1, S4) -->
     expr_code(A, S1, S2),
     expr_code(B, S2, S3),
     [plus],       /* Accumulate ’plus’ in a hidden accumulator */
     {S4 is S3+1}.  /* Explicitly add 1 to the size */
expr_code(I, S1, S2) -->
     {atomic(I)},
     [push(I)],
     {S2 is S1+1}.
```

The extended notation hides both accumulators:

```
expr_code(A+B) -->>
     expr_code(A),
     expr_code(B),
     [plus]:code,   /* Accumulate 'plus' in the code accumulator */
     [1]:size.      /* Accumulate 1 in the size accumulator */
expr_code(I) -->>
     {atomic(I)},
     [push(I)]:code,
     [1]:size.
```

The translation of this version is identical to the original definition. The preprocessor needs the following declarations:

```
acc_info(code, T, Out, In, (Out=[T|In]))./* Accumulator declarations */
acc_info(size, T, In, Out, (Out is In+T)).

pred_info(expr_code, 1, [size,code]).    /* Predicate declaration */
```

For each accumulator this declares the accumulating function, and for each predicate this declares the name, arity (number of arguments), and accumulators it uses. The order of the `In` and `Out` arguments determines whether accumulation proceeds in the forward direction (see `size`) or in the reverse direction (see `code`). Choosing the proper direction is important if the accumulating function requires some of its arguments to be instantiated.

5. Concluding remarks

An extension to Prolog's DCG notation that implements an unlimited number of named accumulators was developed to simplify purely applicative Prolog programming. Comments and suggestions for improvements are welcome.

6. References

[Abramson 1984]

H. Abramson, ‘‘Definite Clause Translation Grammars,’’ *Proc. 1984 International Symposium on Logic Programming,* 1984, pp 233-240.

[Clocksin & Mellish 1981]

W.F. Clocksin and C.S. Mellish, ‘‘Programming in Prolog,’’ *Springer-Verlag,* 1981.

[O'Keefe 1988]

R. A. O'Keefe, ‘‘Practical Prolog for Real Programmers,’’ *Tutorial 8, Fifth International Conference Symposium on Logic Programming, Aug. 1988.*

[Pereira 1981]

F. Pereira, ‘‘Extraposition Grammars,’’ *American Journal of Computational Linguistics,* 1981, vol. 7, no. 4, pp 243-255.

[Pereira & Shieber 1981]

F. Pereira and S. Shieber, ‘‘Prolog and Natural-Language Analysis,’’ *CSLI Lecture Notes 10,* 1987.

[Pereira & Warren 1980]

F. Pereira and D.H.D. Warren, ‘‘Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks,’’ *Journal of Artificial Intelligence,* 1980, vol. 13, no. 3, pp 231-278.

[Sterling & Shapiro 1986]

L. Sterling and E. Shapiro, ‘‘The Art of Prolog,’’ *MIT Press,* 1986.

Extended DCG notation: A tool for applicative programming in Prolog

User Manual

1. Introduction

This manual describes a preprocessor for Prolog that adds an arbitrary number of arguments to a predicate without increasing the size of the source code. The hidden arguments are of two kinds:

(1) Accumulators, useful for results that are calculated incrementally in many predicates. An accumulator expands into two additional arguments per predicate.

(2) Passed arguments, used to pass global information to many predicates. A passed argument expands into a single additional argument per predicate.

The preprocessor has been tested under C-Prolog and Quintus Prolog. It is being used by the author in program development, and is believed to be relatively bug-free. However, it is still being refined and extended. The most recent version is available by anonymous ftp to arpa.berkeley.edu or by contacting the author. Please let me know if you find any bugs. Comments and suggestions for improvements are welcome.

2. Using the preprocessor

The preprocessor is implemented in the file `accumulator.pl`. It must be consulted or compiled before the programs that use it. In Prologs that conform to the Edinburgh standard, such as C-Prolog or Quintus Prolog, the user-defined predicate `term_expansion/2` is called when consulting or compiling each clause that is read. With this hook the use of the preprocessor is transparent.

Clauses to be expanded are of the form `(Head-->>Body)` where `Head` and `Body` are the head and body of the clause. The head is always expanded with all of its hidden arguments. Table 1 summarizes the expansion rules for body goals. In the table, `Goal` denotes any goal in a clause body, `Acc` denotes an accumulator, `Pass` denotes a passed argument, and `Arg` denotes either an accumulator or a passed argument. Hidden arguments of body goals that are not in the head have default values which can be overridden. For compatibility with DCG notation the accumulator `dcg` is available by default. If-then-else is not handled in this version.

The preprocessor assumes the existence of a database of information about the hidden parameters and the predicates to be expanded. Three relations are recognized: a declaration for each predicate, each accumulator, and each passed argument. These relations can be put at the beginning of each file (in which case their scope is the file) or stored in a separate file that is consulted first (in which case their scope is the whole program).

A short example gives a flavor of what the preprocessor does:

```
% Declare the accumulator ’castor’:
acc_info(castor, _, _, _, true).

% Declare the passed argument ’pollux’:
pass_info(pollux).

% Declare three predicates using these hidden arguments:
pred_info(p, 1, [castor,pollux]).
pred_info(q, 1, [castor,pollux]).
```

Table 1 — Expansion rules for the preprocessor

Body goal	Action
`{Goal}`	Don't expand any hidden arguments of `Goal`.
`Goal`	Expand all of the hidden parameters of `Goal` that are also in the head. Those hidden parameters not in the head are given default values.
`Goal:L`	If `Goal` has no hidden arguments then force the expansion of all arguments in `L` in the order given. If `Goal` has hidden arguments then expand all of them, using the contents of `L` to override the expansion. `L` is either a term of the form `Acc`, `Acc(Left,Right)`, `Pass`, `Pass(Value)`, or a list of such terms. When present, the arguments `Left`, `Right`, and `Value` override the default values of arguments not in the head.
`List:Acc`	Accumulate a list of terms in the accumulator `Acc`.
`List`	Accumulate a list of terms in the accumulator `dcg`.
`X/Arg`	Unify `X` with the left term for the accumulator or passed argument `Arg`.
`Acc/X`	Unify `X` with the right term for accumulator `Acc`.
`X/Acc/Y`	Unify `X` with the left and `Y` with the right term for the accumulator `Acc`.
`insert(X,Y):Acc`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `Acc`. This is useful when the value of the accumulator changes radically because `X` and `Y` may be the arguments of an arbitrary relation.
`insert(X,Y)`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `dcg` . This inserts the difference list `X-Y` into the accumulated list.

```
pred_info(r, 1, [castor,pollux]).

% The program:
p(X) -->> Y is X+1, q(Y), r(Y).
```

This example declares one accumulator, one passed argument, and three predicates using them. The program consists of a single clause. The preprocessor is used as follows: (bold-face denotes user input)

```
% cprolog
C-Prolog version 1.5
| ?- ['accumulator.pl'].
accumulator.pl consulted 9780 bytes 1.7 sec.

yes
| ?- ['example.pl'].
example.pl consulted 668 bytes 0.25 sec.

yes
| ?-
```

Now the predicate `p(X)` has been expanded. We can see what it looks like with the `listing` command:

```
| ?- listing(p).

p(X, S1, S3, P) :- Y is X+1, q(Y, S1, S2, P), r(Y, S2, S3, P).
```

(Variable names have been changed for clarity.) The arguments `S1`, `S2`, and `S3`, which implement the accumulator `castor`, are chained together. The argument `P` implements the passed argument. It is added as an extra argument to each predicate.

In object-oriented terminology the declarations of hidden parameters correspond to classes with a single method defined for each. Declarations of predicates specify the inheritance of the predicate from multiple classes, namely each hidden parameter.

3. Declarations

3.1. Declaration of the predicates

Predicates are declared with facts of the following form:

```
pred_info(Name, Arity, List)
```

The predicate `Name/Arity` has the hidden parameters given in `List`. The parameters are added in the order given by `List` and their names must be atoms.

3.2. Declaration of the accumulators

Accumulators are declared with facts in one of two forms. The short form is:

```
acc_info(Acc, Term, Left, Right, Joiner)
```

The long form is:

```
acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart)
```

In most cases the short form gives sufficient information. It declares the accumulator `Acc`, which must be an atom, along with the accumulating function, `Joiner`, and its arguments `Term`, the term to be accumulated, and `Left` & `Right`, the variables used in chaining.

The long form of `acc_info` is useful in more complex programs. It contains two additional arguments, `LStart` and `RStart`, that are used to give default starting values for an accumulator occurring in a body goal that does not occur in the head. The starting values are given to the unused accumulator to ensure that it will execute correctly even though its value is not used. Care is needed to give correct values for `LStart` and `RStart`. For DCG-like list accumulation both may remain unbound.

Two conventions are used for the two variables used in chaining depending on which direction the accumulation is done. For forward accumulation, `Left` is the input and `Right` is the output. For reverse accumulation, `Right` is the input and `Left` is the output.

To see how these declarations work, consider the following program:

```
% Example illustrating the difference between
% forward and reverse accumulation:

% Declare the accumulators:
acc_info(fwd, T, In, Out, Out=[T|In]).   % Forward accumulator.
acc_info(rev, T, Out, In, Out=[T|In]).   % Reverse accumulator.

% Declare the predicates using them:
pred_info(flist, 1, [fwd]).
pred_info(rlist, 1, [rev]).

% flist(N, [], List) creates the list [1, 2, ..., N]
flist(0) -->> [].
flist(N) -->> N>0, [N]:fwd, N1 is N-1, flist(N1).

% rlist(N, List, []) creates the list [N, ..., 2, 1]
rlist(0) -->> [].
rlist(N) -->> N>0, [N]:rev, N1 is N-1, rlist(N1).
```

This defines two accumulators `fwd` and `rev` that both accumulate lists, but in different directions. The joiner of both accumulators is the unification `Out=[T|In]`, which adds `T` to the head of the list `In` and creates the list `Out`. In accumulator `fwd` the output `Out` is the left argument and the input `In` is the right argument. This builds the list in ascending order. Switching the arguments, as in the accumulator `rev`, builds the list in reverse. A sample execution gives these results:

```
| ?- flist(10, [], List).

List = [1,2,3,4,5,6,7,8,9,10]

yes
| ?- rlist(10, List, []).

List = [10,9,8,7,6,5,4,3,2,1]

yes
| ?-
```

If the joining function is not reversible then the accumulator can only be used in one direction. For example, the accumulator `add` with declaration:

```
acc_info(add, I, In, Out, Out is I+In).
```

It can only be used as a forward accumulator. Attempting to use it in reverse results in an error because the argument `In` of the joiner is uninstantiated. The reason for this is that the predicate `is/2` is not pure logic: it requires the expression in its right-hand side to be ground.

3.3. Declaration of the passed arguments

Passed arguments are declared as facts in one of two forms. The short form is:

```
pass_info(Pass)
```

The long form is:

```
pass_info(Pass, PStart)
```

In most cases the short form is sufficient. It declares a passed argument `Pass`, that must be an atom. The long form also contains the starting value `PStart` that is used to give a default value for a passed argument in a body goal that does not occur in the head. Most of the time this situation does not occur.

4. Tips and techniques

Usually there will be one clause of `pred_info` for each predicate in the program. If the program becomes very large, the number of clauses of `pred_info` grows accordingly and can become difficult to keep consistent. In that case it is useful to remember that a single `pred_info` clause can summarize many facts. For example, the following declaration:

```
pred_info(_, _, List).
```

gives all predicates the hidden parameters in `List`. This keeps programming simple regardless of the number of hidden parameters.

Appendix F

Source code of the C and Prolog benchmarks

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* C version of tak benchmark */

#include <stdio.h>

int tak(x,y,z)
int x, y, z;
{
  int a1, a2, a3;
  if (x <= y) return z;
  a1 = tak(x-1,y,z);
  a2 = tak(y-1,z,x);
  a3 = tak(z-1,x,y);
  return tak(a1,a2,a3);
}

main()
{
  printf("%d\n", tak(24, 16, 8));
}

-------------------------------------------------------------------------------

/* Prolog version of tak benchmark */

main :- tak(24,16,8,X), write(X), nl.

tak(X,Y,Z,A) :- X =< Y, Z = A.
tak(X,Y,Z,A) :- X > Y,
        X1 is X - 1, tak(X1,Y,Z,A1),
        Y1 is Y - 1, tak(Y1,Z,X,A2),
        Z1 is Z - 1, tak(Z1,X,Y,A3),
        tak(A1,A2,A3,A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* C version of fib benchmark */

#include <stdio.h>

int fib(x)
int x;
{
  if (x <= 1) return 1;
  return (fib(x-1)+fib(x-2));
}
```

```
main()
{
  printf("%d\n", fib(30));
}

--------------------------------------------------------------------------------

/* Prolog version of fib benchmark */

main :- fib(30,N), write(N), nl.

fib(N,F) :- N =< 1, F = 1.
fib(N,F) :- N > 1,
        N1 is N - 1, fib(N1,F1),
        N2 is N - 2, fib(N2,F2),
        F is F1 + F2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* C version of hanoi benchmark */

#include <stdio.h>

han(n,a,b,c)
{
   int n1;

   if (n<=0) return;
   n1 = n-1;
   han(n1,a,c,b);
   han(n1,c,b,a);
}

main()
{
  han(20,1,2,3);
}

--------------------------------------------------------------------------------

/* Prolog version of hanoi benchmark */

main :- han(20,1,2,3).

han(N,_,_,_) :- N=<0.
han(N,A,B,C) :- N>0,
        N1 is N - 1,
        han(N1,A,C,B),
        han(N1,C,B,A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* C version of quicksort benchmark */
```

```
#include <stdio.h>

int ilist[50] = {27,74,17,33,94,18,46,83,65, 2,
                 32,53,28,85,99,47,28,82, 6,11,
                 55,29,39,81,90,37,10, 0,66,51,
                  7,21,85,27,31,63,75, 4,95,99,
                 11,28,61,74,18,92,40,53,59, 8};

int list[50];

qsort(l, r)
int l, r;
{
   int v, t, i, j;

   if (l<r) {
        v=list[l]; i=l; j=r+1;
        do {
            do i++; while (list[i]<v);
            do j--; while (list[j]>v);
            t=list[j]; list[j]=list[i]; list[i]=t;
        } while (j>i);
        list[i]=list[j]; list[j]=list[l]; list[l]=t;
        qsort(l,j-1);
        qsort(j+1,r);
   }
}

main()
{
   int i, j;

   for(j=0; j<10000; j++) {
       for(i=0;i<50;i++) list[i]=ilist[i];
       qsort(0,49);
   }
   for(i=0; i<50; i++) printf("%d ",list[i]);
   printf("\n");
}

--------------------------------------------------------------------------------

/* Prolog version of quicksort benchmark */

main :- range(1,I,9999), qsort(_), fail.
main :- qsort(S), write(S), nl.

range(L,L,H).
range(L,I,H) :- L<H, L1 is L+1, range(L1,I,H).

qsort(S) :- qsort([27,74,17,33,94,18,46,83,65, 2,
                   32,53,28,85,99,47,28,82, 6,11,
                   55,29,39,81,90,37,10, 0,66,51,
                    7,21,85,27,31,63,75, 4,95,99,
                    11,28,61,74,18,92,40,53,59, 8],S,[]).

qsort([X|L],R,R0) :-
        partition(L,X,L1,L2),
        qsort(L2,R1,R0),
        qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([Y|L],X,[Y|L1],L2) :- Y=<X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- Y>X,  partition(L,X,L1,L2).
partition([],_,[],[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Appendix G

Source code of the Aquarius Prolog compiler

‘‘A dissertation is 1% ink and 99% pulp.’’
– D. von Tischtiegel

Files in the compiler

File	Description	Page
`accumulator.pl`	Extended DCG preprocessor	228
`accumulator_cleanup.pl`	Cleanup file needed for preprocessor	234
`analyze.pl`	Dataflow analyzer	235
`clause_code.pl`	Clause compiler	263
`conditions.pl`	Formula manipulation utilities	284
`compiler.pl`	Top level of compiler, includes type enrichment	305
`expression.pl`	Compile arithmetic expressions	322
`factor.pl`	Factoring transformation	324
`flatten.pl`	Flattening transformation	334
`inline.pl`	Inline replacement	337
`mutex.pl`	Mutual exclusion and implication of formulas	339
`peephole.pl`	BAM transformations (except synonym)	357
`preamble.pl`	Part of standard form transformation	374
`proc_code.pl`	Predicate compiler	379
`regalloc.pl`	Register allocator	387
`segment.pl`	Head-body segmentation and goal reordering	395
`selection.pl`	Determinism extraction	401
`standard.pl`	Standard form transformation	411
`synonym.pl`	Synonym optimization	417
`tables.pl`	Compilation tables	424
`testset.pl`	List of test sets	434
`transform_cut.pl`	Cut transformation	440
`unify.pl`	Unification compiler	443
`utility.pl`	Utility predicates	460

accumulator.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California.
% All rights reserved.  This program may be freely used and modified for
% non-commercial purposes provided this copyright notice is kept unchanged.
% Written by Peter Van Roy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Multiple hidden parameters: an extension to Prolog’s DCG notation.
% Version: July 16, 1989

% Comments, suggestions, flames, manifestos, and bug reports are most welcome.
% Please send to:
%  Peter Van Roy
%  508-10 Evans Hall
%  University of California
%  Berkeley, CA 94720
% E-mail: vanroy@ernie.berkeley.edu

:- op(1200, xfx, [’-->>’]).   % Same as ’:-’.
:- op( 850, xfx, [’:’]).      % Slightly tighter than ’,’ and ’\+’.

% The predicate term_expansion/2 implements the extended translation.
% If loaded into Prolog along with the appropriate acc_info, pass_info,
% and pred_info facts it will be used automatically when consulting programs.

term_expansion((H-->>B), (TH:-FTB)) :-
   functor(H, Na, Ar),
   ’_has_hidden’(H, HList),
   ’_new_goal’(H, HList, HArity, TH),
   ’_create_acc_pass’(HList, HArity, TH, Acc, Pass),
   ’_flat_conj’(B, FB),
   ’_expand_body’(FB, TB, Na/Ar, HList, Acc, Pass),
   ’_flat_conj’(TB, FTB), !.

’_expand_body’(true, true, _, _, Acc, _) :- ’_finish_acc’(Acc).
’_expand_body’((G,B), (TG,TB), NaAr, HList, Acc, Pass) :-
   ’_expand_goal’(G, TG, NaAr, HList, Acc, NewAcc, Pass),
   ’_expand_body’(B, TB, NaAr, HList, NewAcc, Pass).

% Expand a single goal:
’_expand_goal’({G}, G, _, _, Acc, Acc, _) :- !.
’_expand_goal’(insert(X,Y), LeftA=X, _, _, Acc, NewAcc, _) :-
   ’_replace_acc’(dcg, LeftA, RightA, Y, RightA, Acc, NewAcc), !.
’_expand_goal’(insert(X,Y):A, LeftA=X, _, _, Acc, NewAcc, _) :-
   ’_replace_acc’(A, LeftA, RightA, Y, RightA, Acc, NewAcc), !.
% Force hidden arguments in L to be appended to G:
’_expand_goal’((G:A), TG, _, HList, Acc, NewAcc, Pass) :-
   \+’_list’(G),
   ’_has_hidden’(G, []), !,
   ’_make_list’(A, AList),
   ’_new_goal’(G, AList, GArity, TG),
   ’_use_acc_pass’(AList, GArity, TG, Acc, NewAcc, Pass).
% Use G’s regular hidden arguments & override defaults for those arguments
% not in the head:
’_expand_goal’((G:A), TG, _, HList, Acc, NewAcc, Pass) :-
   \+’_list’(G),
   ’_has_hidden’(G, GList), GList\==[], !,
   ’_make_list’(A, L),
   ’_new_goal’(G, GList, GArity, TG),
   ’_replace_defaults’(GList, NGList, L),
   ’_use_acc_pass’(NGList, GArity, TG, Acc, NewAcc, Pass).
’_expand_goal’((L:A), Joiner, NaAr, _, Acc, NewAcc, _) :-
   ’_list’(L), !,
   ’_joiner’(L, A, NaAr, Joiner, Acc, NewAcc).
’_expand_goal’(L, Joiner, NaAr, _, Acc, NewAcc, _) :-
   ’_list’(L), !,
   ’_joiner’(L, dcg, NaAr, Joiner, Acc, NewAcc).
’_expand_goal’((X/A), true, _, _, Acc, Acc, _) :-
   var(X), nonvar(A),
   ’_member’(acc(A,X,_), Acc), !.
’_expand_goal’((X/A), true, _, _, Acc, Acc, Pass) :-
   var(X), nonvar(A),
   ’_member’(pass(A,X), Pass), !.
’_expand_goal’((A/X), true, _, _, Acc, Acc, _) :-
   var(X), nonvar(A),
   ’_member’(acc(A,_,X), Acc), !.
’_expand_goal’((X/A/Y), true, _, _, Acc, Acc, _) :-
   var(X), var(Y), nonvar(A),
   ’_member’(acc(A,X,Y), Acc), !.
’_expand_goal’((X/Y), true, NaAr, _, Acc, Acc, _) :-
   write(’*** Warning: in ’),write(NaAr),write(’ the term ’),write(X/Y),
   write(’ uses a non-existent hidden parameter.’),nl.
% Defaulty cases:
’_expand_goal’(G, TG, HList, _, Acc, NewAcc, Pass) :-
   ’_has_hidden’(G, GList), !,
   ’_new_goal’(G, GList, GArity, TG),
   ’_use_acc_pass’(GList, GArity, TG, Acc, NewAcc, Pass).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Operations on the Acc and Pass data structures:

% Create the Acc and Pass data structures:
% Acc contains terms of the form acc(A,LeftA,RightA) where A is the name of an
% accumulator, and RightA and LeftA are the accumulating parameters.
% Pass contains terms of the form pass(A,Arg) where A is the name of a passed
% argument, and Arg is the argument.
’_create_acc_pass’([], _, _, [], []).
’_create_acc_pass’([A|AList], Index, TGoal, [acc(A,LeftA,RightA)|Acc], Pass) :-
   ’_is_acc’(A), !,
   Index1 is Index+1,
   arg(Index1, TGoal, LeftA),
   Index2 is Index+2,
   arg(Index2, TGoal, RightA),
   ’_create_acc_pass’(AList, Index2, TGoal, Acc, Pass).
```

```
'_create_acc_pass'([A|AList], Index, TGoal, Acc, [pass(A,Arg)|Pass]) :-
   '_is_pass'(A), !,
   Index1 is Index+1,
   arg(Index1, TGoal, Arg),
   '_create_acc_pass'(AList, Index1, TGoal, Acc, Pass).
'_create_acc_pass'([A|AList], Index, TGoal, Acc, Pass) :-
   \+'_is_acc'(A),
   \+'_is_pass'(A),
   write('*** Error: '),write(A),
   write(' is not a hidden parameter.'),nl.

% Use the Acc and Pass data structures to create the arguments of a body goal:
% Add the hidden parameters named in GList to the goal.
'_use_acc_pass'([], _, _, Acc, Acc, _).
% 1a. The accumulator A is used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
   '_replace_acc'(A, LeftA, RightA, MidA, RightA, Acc, MidAcc), !,
   Index1 is Index+1,
   arg(Index1, TGoal, LeftA),
   Index2 is Index+2,
   arg(Index2, TGoal, MidA),
   '_use_acc_pass'(GList, Index2, TGoal, MidAcc, NewAcc, Pass).
% 1b. The accumulator A is not used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
   '_acc_info'(A, LStart, RStart), !,
   Index1 is Index+1,
   arg(Index1, TGoal, LStart),
   Index2 is Index+2,
   arg(Index2, TGoal, RStart),
   '_use_acc_pass'(GList, Index2, TGoal, Acc, NewAcc, Pass).
% 2a. The passed argument A is used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
   '_is_pass'(A),
   '_member'(pass(A,Arg), Pass), !,
   Index1 is Index+1,
   arg(Index1, TGoal, Arg),
   '_use_acc_pass'(GList, Index1, TGoal, Acc, NewAcc, Pass).
% 2b. The passed argument A is not used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
   '_pass_info'(A, AStart), !,
   Index1 is Index+1,
   arg(Index1, TGoal, AStart),
   '_use_acc_pass'(GList, Index1, TGoal, Acc, NewAcc, Pass).
% 3. Defaulty case when A does not exist:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, Acc, Pass) :-
   write('*** Error: the hidden parameter '),write(A),
   write(' does not exist.'),nl.

% Finish the Acc data structure:
% Link its Left and Right accumulation variables together in pairs:
'_finish_acc'([]).
'_finish_acc'([acc(_,Link,Link)|Acc]) :- '_finish_acc'(Acc).

% Replace elements in the Acc data structure:
```

```
% Succeeds iff replacement is successful.
'_replace_acc'(A, L1, R1, L2, R2, Acc, NewAcc) :-
   '_member'(acc(A,L1,R1), Acc), !,
   '_replace'(acc(A,_,_), acc(A,L2,R2), Acc, NewAcc).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Specialized utilities:

% Given a goal Goal and a list of hidden parameters GList
% create a new goal TGoal with the correct number of arguments.
% Also return the arity of the original goal.
'_new_goal'(Goal, GList, GArity, TGoal) :-
   functor(Goal, Name, GArity),
   '_number_args'(GList, GArity, TArity),
   functor(TGoal, Name, TArity),
   '_match'(1, GArity, Goal, TGoal).

% Add the number of arguments needed for the hidden parameters:
'_number_args'([], N, N).
'_number_args'([A|List], N, M) :-
   '_is_acc'(A), !,
   N2 is N+2,
   '_number_args'(List, N2, M).
'_number_args'([A|List], N, M) :-
   '_is_pass'(A), !,
   N1 is N+1,
   '_number_args'(List, N1, M).

% Give a list of G's hidden parameters:
'_has_hidden'(G, GList) :-
   functor(G, GName, GArity),
   pred_info(GName, GArity, GList).
'_has_hidden'(G, []) :-
   functor(G, GName, GArity),
   \+pred_info(GName, GArity, _).

% Succeeds if A is an accumulator:
'_is_acc'(A)  :- atomic(A), !, '_acc_info'(A, _, _, _, _, _, _).
'_is_acc'(A)  :- functor(A, N, 2), !, '_acc_info'(N, _, _, _, _, _, _).

% Succeeds if A is a passed argument:
'_is_pass'(A) :- atomic(A), !, '_pass_info'(A, _).
'_is_pass'(A) :- functor(A, N, 1), !, '_pass_info'(N, _).

% Get initial values for the accumulator:
'_acc_info'(AccParams, LStart, RStart) :-
   functor(AccParams, Acc, 2),
   '_is_acc'(Acc), !,
   arg(1, AccParams, LStart),
   arg(2, AccParams, RStart).
'_acc_info'(Acc, LStart, RStart) :-
   '_acc_info'(Acc, _, _, _, _, LStart, RStart).
```

```
% Isolate the internal database from the user database:
'_acc_info'(Acc, Term, Left, Right, Joiner, LStart, RStart) :-
   acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart).
'_acc_info'(Acc, Term, Left, Right, Joiner, _, _) :-
   acc_info(Acc, Term, Left, Right, Joiner).
'_acc_info'(dcg, Term, Left, Right, Left=[Term|Right], _, []).

% Get initial value for the passed argument:
% Also, isolate the internal database from the user database.
'_pass_info'(PassParam, PStart) :-
   functor(PassParam, Pass, 1),
   '_is_pass'(Pass), !,
   arg(1, PassParam, PStart).
'_pass_info'(Pass, PStart) :-
   pass_info(Pass, PStart).
'_pass_info'(Pass, _) :-
   pass_info(Pass).

% Calculate the joiner for an accumulator A:
'_joiner'([], _, _, true, Acc, Acc).
'_joiner'([Term|List], A, NaAr, (Joiner,LJoiner), Acc, NewAcc) :-
   '_replace_acc'(A, LeftA, RightA, MidA, RightA, Acc, MidAcc),
   '_acc_info'(A, Term, LeftA, MidA, Joiner, _, _), !,
   '_joiner'(List, A, NaAr, LJoiner, MidAcc, NewAcc).
% Defaulty case:
'_joiner'([Term|List], A, NaAr, Joiner, Acc, NewAcc) :-
   write('*** Warning: in '),write(NaAr),
   write(' the accumulator '),write(A),
   write(' does not exist.'),nl,
   '_joiner'(List, A, NaAr, Joiner, Acc, NewAcc).

% Replace hidden parameters with ones containing initial values:
'_replace_defaults'([], [], _).
'_replace_defaults'([A|GList], [NA|NGList], AList) :-
   '_replace_default'(A, NA, AList),
   '_replace_defaults'(GList, NGList, AList).

'_replace_default'(A, NewA, AList) :-  % New initial values for accumulator.
   functor(NewA, A, 2),
   '_member'(NewA, AList), !.
'_replace_default'(A, NewA, AList) :-  % New initial values for passed argument.
   functor(NewA, A, 1),
   '_member'(NewA, AList), !.
'_replace_default'(A, NewA, _) :-      % Use default initial values.
   A=NewA.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generic utilities:

% Match arguments L, L+1, ..., H of the predicates P and Q:
'_match'(L, H, _, _) :- L>H, !.
'_match'(L, H, P, Q) :- L=<H, !,
   arg(L, P, A),
   arg(L, Q, A),
   L1 is L+1,
   '_match'(L1, H, P, Q).

% Flatten a conjunction and terminate it with 'true':
'_flat_conj'(Conj, FConj) :- '_flat_conj'(Conj, FConj, true).

'_flat_conj'(true, X, X).
'_flat_conj'((A,B), X1, X3) :-
   '_flat_conj'(A, X1, X2),
   '_flat_conj'(B, X2, X3).
'_flat_conj'(G, (G,X), X) :-
   \+G=true,
   \+G=(_,_).

'_member'(X, [X|_]).
'_member'(X, [_|L]) :- '_member'(X, L).

'_list'(L) :- nonvar(L), L=[_|_], !.
'_list'(L) :- L==[], !.

'_append'([], L, L).
'_append'([X|L1], L2, [X|L3]) :- '_append'(L1, L2, L3).

'_make_list'(A, [A]) :- \+'_list'(A), !.
'_make_list'(L,   L) :-   '_list'(L), !.

% replace(Elem, RepElem, List, RepList)
'_replace'(_, _, [], []).
'_replace'(A, B, [A|L], [B|R]) :- !,
   '_replace'(A, B, L, R).
'_replace'(A, B, [C|L], [C|R]) :-
   \+C=A, !,
   '_replace'(A, B, L, R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

accumulator_cleanup.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% For Quintus Prolog only:
% Used to clean up declarations between compilations of files using
% the extended DCG preprocessor.

:- abolish([pass_info/1,pass_info/2,acc_info/5,acc_info/7,pred_info/3]).
:- multifile pass_info/1.
:- multifile pass_info/2.
:- multifile acc_info/5.
:- multifile acc_info/7.
:- multifile pred_info/3.

% Dummy clauses so the compiler won’t complain of nonexistence:

pass_info(’_dummy’).

pass_info(’_dummy’, _).

acc_info(’_dummy’, _, _, _, _).

acc_info(’_dummy’, _, _, _, _, _, _).

pred_info(’_dummy’, 0, _, _, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

analyze.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Dataflow analysis version 4

% This module does a flow analysis, deriving 'uninit', 'ground', 'nonvar', and
% 'rderef' modes by doing an abstract interpretation of the source code.
% It traverses the call tree and keeps track of entry and exit modes.
% It iterates and propagates modes until the least fixed point is reached.
% The module is a transformation pass in the Aquarius compiler.

% The entry mode lattice is (for each argument of all predicates):

%             any
%            /   \
%       nonvar   rderef
%       /    \   /   \
%   ground  non+drf  uninit
%       \    /       /
%       gnd+drf     /
%            \     /
%            unknown

% The exit mode lattice is identical (there is no 'uninit' element for non-unify
% goals).  The rderef lattice element corresponds to the mode rderef(X) which
% means that all subterms of X are dereferenced.  It propagates just like
% ground(X).  It is useful to reduce the large amount of dereferencing which is
% going on (usually between 10% and 20% of the execution time with analyzer 2!).

% These simple lattices are easily implemented for four reasons: (1) neither
% uninits nor grounds are affected by aliasing, and (2) uninit, grounds, and
% rderefs all propagate indefinitely into explicit unifications with compound
% terms, (3) nonvars and grounds, once true, do not become false again, and
% (4) there is no explicit representation for aliasing--modes are calculated
% only for unaliased variables (e.g. uninit and rderef) and for cases where
% aliasing has no effect (e.g. ground and nonvar).

% Notes:
% 1. A set of entry points must exist to do the analysis.
% 2. In each pass, this algorithm traverses the call tree starting from those
%    predicates whose entry modes have changed or which contain a goal whose
%    exit modes have changed.  The algorithm differs from O'Keefe's in the
%    data structures used.
% 3. Only modes "generated" within the strees given are propagated.  Except for
%    the entry modes, all modes "already existing" are ignored.  This allows
%    the compiler to correct programmer's errors.
% 4. Predicates of arity zero are always used as entry points in the analysis.
% 5. To get better results for exit modes, this algorithm back-propagates
%    the results at the end of the clause to the beginning, through the
%    unifications in the clause.  For example, part1(A,B) :- A=[X|L], B=[X|L1],
%    part2(L,L1).  Let A be ground at entry.  If L1 returns ground from part2
%    then B should return ground from part1 as well.

% For later:
% 1. Back propagate univ, i.e. in A=..[X|Y], if X & Y are ground then A is too.
%    Can use existing unify(_,_,_) data structure.  Generalize?
% 2. The mode rlist(X) (recursive list) propagates similarly to ground(X) also.
%    It should be added to the exit mode analyzer too.
% 3. This version does not use cuts to get better modes.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Accumulator declarations:

pass_info(pred).
pass_info(dup).

acc_info(gnd,    T, In, Out, set_command(T,In,Out), [],  _).
acc_info(nonvar, T, In, Out, set_command(T,In,Out), [],  _).
acc_info(uninit, T, In, Out, set_command(T,In,Out), [],  _).
acc_info(rderef, T, In, Out, set_command(T,In,Out), [],  _).
acc_info(sf,     S, In, Out, unionv(S,In,Out),      [],  _).
acc_info(uni,    U, Out, In, Out=[U|In],             _, []).
acc_info(set,    T, In, Out, set_command(T,In,Out), [],  _).

acc_info(count,  I, In, Out, (Out is In+I)).
acc_info(change, I, In, Out, set_command(I, In, Out)).
acc_info(occurs, T, In, Out, table_command(T,In,Out)).
acc_info(entry,  T, In, Out, table_command(T,In,Out)).
acc_info(exit,   T, In, Out, table_command(T,In,Out)).
acc_info(defs,   T, In, Out, table_command(T,In,Out)).
acc_info(spec,   T, In, Out, table_command(T,In,Out)).
acc_info(uregs,  T, In, Out, table_command(T,In,Out)).
acc_info(fast,   T, In, Out, table_command(T,In,Out)).
acc_info(preds,  P, Out, In, Out=[P|In]).

% Predicate declarations:

% *** Initialization:
pred_info(    init_strees, 1, [entry,exit,defs,occurs]).
pred_info(      init_disj, 1, [entry,exit,defs,occurs,pred]).
pred_info(      init_conj, 1, [entry,exit,defs,occurs,pred]).
pred_info(      init_goal, 1, [entry,exit,defs,occurs,pred]).
pred_info(     entry_init, 1, [entry]).
pred_info( entry_init_one, 1, [entry]).
% *** Top level analysis:
pred_info(analyze_closure, 1, [entry,exit,defs,spec,occurs]).
pred_info(     trav_preds, 1, [entry,exit,defs,spec,change]).
% *** Traversal of call tree:
pred_info(      trav_pred, 4,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf]).
pred_info(      trav_disj, 8,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf]).
pred_info(      trav_conj, 5,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf,uni,dup]).
```

```
pred_info(      trav_goal, 6,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf,uni,dup]).
pred_info(      trav_call, 6,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf]).
pred_info(       trav_def, 7,
     [entry,exit,defs,spec,change,gnd,nonvar,uninit,rderef,sf]).
% *** Update of entry & exit mode tables:
pred_info(    update_entry, 3, [entry,   change,gnd,nonvar,uninit,rderef,sf]).
pred_info(    update_entry, 4, [entry,   change]).
pred_info(     update_exit, 1, [exit,    change,gnd,nonvar,rderef,sf]).
pred_info(     update_exit, 3, [exit,    change]).
% *** Utilities:
pred_info(     trav_goal_n, 2, [nonvar]).
pred_info(        trav_non, 2, [nonvar]).
pred_info(     trav_goal_u, 3, [gnd,uninit,rderef,sf]).
pred_info(       trav_unif, 4, [gnd,uninit,rderef,sf]).
pred_info(     trav_goal_d, 4, [gnd,uninit,rderef,sf]).
pred_info(       trav_drf1, 4, [gnd,uninit,rderef,sf]).
pred_info(       trav_drf2, 5, [gnd,uninit,rderef,sf]).
pred_info(        new_dref, 2, [    uninit,rderef,sf]).
pred_info(        add_dref, 1, [    uninit,rderef,sf]).
pred_info(     new_drf_gnd, 4, [gnd,nonvar,uninit,rderef,sf]).
pred_info(     new_drf_set, 2, [gnd,       uninit,rderef,sf]).
pred_info(       calc_exit, 2, [gnd,nonvar,rderef]).
pred_info(      calc_entry, 2, [gnd,nonvar,uninit,rderef,sf]).
pred_info(  back_propagate, 1, [gnd,nonvar,rderef]).
pred_info(  back_prop_g_cl, 1, [gnd]).
pred_info(  back_prop_g_cl, 2, [gnd]).
pred_info(     back_prop_g, 2, [gnd,count]).
pred_info(  back_prop_d_cl, 1, [rderef]).
pred_info(  back_prop_d_cl, 2, [rderef]).
pred_info(     back_prop_d, 2, [rderef,count]).
pred_info(  back_prop_n_cl, 1, [nonvar]).
pred_info(  back_prop_n_cl, 2, [nonvar]).
pred_info(     back_prop_n, 2, [nonvar,count]).
pred_info(        make_uni, 3, [uninit,sf,uni,dup]).
pred_info(  dref_prop_flag, 2, [uninit,sf,uni,dup]).
pred_info(       new_preds, 1, [occurs,change]).
pred_info(        new_sets, 2, [gnd,nonvar,uninit,rderef,sf]).
pred_info(       save_sets, 1, [gnd,nonvar,uninit,rderef,sf]).
pred_info(    restore_sets, 1, [gnd,nonvar,uninit,rderef,sf]).
pred_info(       spec_goal, 5, [gnd,nonvar,uninit,rderef,sf,spec]).
pred_info(      spec_goal2, 6, [gnd,nonvar,uninit,rderef,sf,spec]).
pred_info(     spec_update, 7, [gnd,nonvar,uninit,rderef,sf,spec]).
% *** Used in conversion of uninit to uninit_reg:
pred_info(    convert_closure, 1, [defs,occurs,fast,uregs]).
pred_info(         conv_preds, 1, [defs,       fast,uregs,change]).
pred_info(   calc_convert_set, 3, [defs,       fast,uregs,change,set]).
pred_info(     last_goal_ureg, 3, [defs,       fast,uregs,change,set]).
pred_info(   update_fast_goal, 3, [defs,       fast,uregs,change]).
pred_info(        update_ureg, 3,                  [uregs,change]).
pred_info(init_convert_strees, 1, [entry,preds,fast,uregs]).
pred_info(         enter_fast, 2, [fast]).
```

```
% Implement the accumulator commands:

set_command(sub(X), In, Out) :- excludev(X, In, Out). % diffv(In, [X], Out).
set_command(add(X), In, Out) :- includev(X, In, Out).
set_command(sub_set(X), In, Out) :- diffv(In, X, Out).
set_command(add_set(X), In, Out) :- unionv(X, In, Out).

table_command(get(I,Val), In,  In) :- get(In, I, Val).
table_command(fget(I,Val),In,  In) :- fget(In, I, Val).
table_command(set(I,Val), In, Out) :- fset(In, I, Val, Out).
table_command(add(I,X), In, Out) :-
   get(In, I, S1),
   includev(X, S1, S),
   fset(In, I, S, Out), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Add the derived modes to the strees and to the stored modes:

% This adds information to pre-existing modes of the strees.
create_mode_strees(Ss, ASs, Entry, Exit, UReg) :-
   create_mode_strees(Ss, ASs, Entry, Exit, UReg, top).

create_mode_strees([], [], _, _, _, _) :- !.
create_mode_strees([S|Ss], [AS|ASs], Entry, Exit, UReg, F) :-
   create_mode_stree(S, AS, Entry, Exit, UReg, F),
   create_mode_strees(Ss, ASs, Entry, Exit, UReg, F).

create_mode_stree(stree(NaAr,HD,(H:-OldF),OH,DList,SD),
         stree(NaAr,HD,(H:-NewF),OH,ADList,SD), Entry,Exit,UReg,F):- !,
   lattice_modes_table(NaAr, Entry, H, Flow),
   lattice_modes_table(NaAr, Exit, H, ExFlow),
   new_formula(H, Flow, ExFlow, HD, UReg, NewReq, NewBef, NewAft),
   add_mode_option(analyze_mode(H,NewReq,NewBef,NewAft)),
   flat_conj((NewReq,NewBef), NewF),
   write_mode(F, H),
   create_mode_strees(DList, ADList, Entry, Exit, UReg, nontop).
create_mode_stree(D, D, _, _, _, _) :- directive(D).

% Calculate new Req+Bef+Aft modes according to the following schema:

% OldReq         OldBef         OldAft
%   |              |              |
%   |\(split)      |              |
%   | \            |              |
%   |  \-unbound-\ |              |
%   |             \|(combine)     |
%   |              |              |
%   | subsume      | update       | update
%   | Flow         | Flow         | ExFlow
%   |              |              |
%   |             /|(split)       |
%   |  /-uninit--/ |              |
%   | /            |              |
%   |/(combine)    |              |
%   |              |              |
% NewReq         NewBef         NewAft

new_formula(H, Flow, ExFlow, HD, UReg, NewReq, NewBef, NewAft) :-
   require(H, OldReq),
   before(H, OldBef),
   split_unbound(OldReq, OldU, OldReq2),
   combine_formula(OldU, OldBef, OldBef2),
   update_mode(Flow, OldBef2, H, OldBef3),
   squeeze_conj(OldBef3, OldBef4),
   convert_uninit(HD, UReg, OldBef4, OldBef5),
   split_uninit(OldBef5, NewU, NewBef),
   logical_subsume(Flow, OldReq2, OldReq3),
   combine_formula(NewU, OldReq3, NewReq),
   after(H, OldAft),
   update_mode(ExFlow, OldAft, H, OldAft2),
   squeeze_conj(OldAft2, NewAft).

% Add the derived formula to an existing formula:
update_mode(NewF, true, _, NewF) :- !.
update_mode(true, OldF, _, OldF) :- !.
update_mode((Goal,Conj), OldF, Head, NewF) :- !,
   update_one(Goal, OldF, Head, MidF),
   update_mode(Conj, MidF, Head, NewF).

update_one(ground(X), OldF, Head, NewF) :- implies(OldF, unbound(X)), !,
   incorrect_mode(X, Head, ground(X), OldF, NewF).
update_one(nonvar(X), OldF, Head, NewF) :- implies(OldF, unbound(X)), !,
   incorrect_mode(X, Head, nonvar(X), OldF, NewF).
update_one(uninit(X), OldF, Head, NewF) :- implies(OldF, nonvar(X)), !,
   incorrect_mode(X, Head, uninit(X), OldF, NewF).
update_one(uninit_reg(X), OldF, Head, NewF) :- implies(OldF, nonvar(X)), !,
   incorrect_mode(X, Head, uninit_reg(X), OldF, NewF).
update_one(ground(X), OldF, _, OldF) :- pred_exists(ground(X), OldF), !.
update_one(nonvar(X), OldF, _, OldF) :- pred_exists(nonvar(X), OldF), !.
update_one(uninit(X), OldF, _, OldF) :- pred_exists(uninit(X), OldF), !.
update_one(uninit_reg(X), OldF, _, OldF) :- pred_exists(uninit_reg(X), OldF), !.
update_one(rderef(X), OldF, _, OldF) :- pred_exists(rderef(X), OldF), !.
update_one(uninit(X), OldF, Head, NewF) :- pred_exists(var(X), OldF), !,
   NewF = (uninit(X),MidF),
   split_from_formula(X, OldF, MidF, BadF),
   squeeze_conj(BadF, SquF),
   warning(Head, ['Mode ',SquF,' of ',Head,' replaced by ',uninit(X)]).
update_one(uninit_reg(X), OldF, Head, NewF) :- pred_exists(var(X), OldF), !,
   NewF = (uninit_reg(X),MidF),
   split_from_formula(X, OldF, MidF, BadF),
   squeeze_conj(BadF, SquF),
   warning(Head, ['Mode ',SquF,' of ',Head,' replaced by ',uninit_reg(X)]).
update_one(ground(X), OldF, _, (ground(X),OldF)) :- !.
update_one(nonvar(X), OldF, _, (nonvar(X),OldF)) :- !.
update_one(uninit(X), OldF, _, (uninit(X),OldF)) :- !.
update_one(uninit_reg(X), OldF, _, (uninit_reg(X),OldF)) :- !.
update_one(rderef(X), OldF, _, (rderef(X),OldF)) :- !.
```

```
incorrect_mode(X, Head, GoodF, OldF, (GoodF,MidF)) :-
   split_from_formula(X, OldF, MidF, BadF),
   squeeze_conj(BadF, SquF),
   warning(Head, ['Mode ',SquF,' of ',Head,' is incorrect.',nl,
              'Compilation continued with corrected mode ',GoodF]).

% Write the mode:
% If it is a top-level mode and no compile, then output it in
% Prolog-readable form.  Otherwise, output it as a comment.
write_mode(top, Head) :- \+compile_option(compile), !,
   require(Head, R), before(Head, B), after(Head, A), survive(Head, S),
   w(':- '),inst_writeq(mode(Head,R,B,A,S)),wn('.').
write_mode(_, Head) :- \+compile_option(compile), !,
   require(Head, R), before(Head, B), after(Head, A), survive(Head, S),
   w('%  '),inst_writeq(mode(Head,R,B,A,S)),nl.
write_mode(_, Head) :- compile_option(compile), !,
   require(Head, R), before(Head, B), after(Head, A), survive(Head, S),
   w('% '),inst_writeq(mode(Head,R,B,A,S)),nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Convert uninit(X) modes into uninit_reg(X) modes:

% This is done only for uninitialized variables which occur up to once in the
% last non-survive goal or the survive goals beyond it, and do not occur before
% that in the body of each clause.
% The last non-survive goal, if it exists, must have an uninit_reg mode in the
% same argument position as those uninit. vars, otherwise no tail recursion
% optimization can be done.  A closure calculation is done to guarantee this &
% not to lose optimization opportunities.
% (An easy approximation is to allow the conversion only for recursive calls,
% but this loses the optimization for predicates with cuts and factoring.)

% There are two entry points to this transformation:
% convert_uninit: uses the pre-calculated URegTable to convert one predicate.
% convert_uninit_strees: closure calculation which gives URegTable.

% *** Convert the allowed arguments of a single predicate to uninit_reg.
% Relies on the information gathered by convert_uninit_strees.
convert_uninit((Head:-Disj), URegTable, InF, OutF) :-
   compile_option(analyze_uninit_reg), !,
   functor(Head, Na, Ar),
   get(URegTable, Na/Ar, ConvSet),
   convert_form(InF, OutF, ConvSet).
convert_uninit(_, _, InF, InF).

convert_form((A,B), (CA,CB), ConvSet) :- !,
   convert_form(A, CA, ConvSet),
   convert_form(B, CB, ConvSet).
convert_form(uninit(X), uninit_reg(X), ConvSet) :- inv(X, ConvSet), !.
convert_form(T, T, _).

% *** Calculate for each predicate the set of arguments that may be uninit_reg.
% The results are returned in OutUReg.
convert_uninit_strees(Strees, Entry, Defs, Occurs, OutUReg) :-
   compile_option(analyze_uninit_reg), !,
   stats(a,1),
   init_convert(Strees, Entry, Preds, Fast, InUReg),
   stats(a,2),
   convert_closure(Preds, Defs, _, Occurs, _, Fast, _, InUReg, OutUReg).
convert_uninit_strees(Strees, _, _, _, OutUReg) :-
   seal(OutUReg).

convert_closure([]) -->> [].
convert_closure(PropPreds) -->> {cons(PropPreds)}, !,
   {stats(a,3)},
   conv_preds(PropPreds):change([],ChgSet),
   {stats(a,4)},
   {length(ChgSet, N)},
   {comment(['Uninit(reg) conversion pass--changed ',N,' predicates.'])},
   new_preds(ChgSet):change([],NewPreds),
   convert_closure(NewPreds).

conv_preds([]) -->> [].
conv_preds([NaAr|Preds]) -->>
   [fget(NaAr,UReg)]:uregs,
   [fget(NaAr,(Head:-Disj))]:defs,
   {stats(a,data4(NaAr))},
   calc_convert_set(Disj, Head, NaAr):set(UReg, NewUReg),
   {stats(a,5)},
   update_ureg(NaAr, UReg, NewUReg),
   {stats(a,6)},
   conv_preds(Preds).

% Find the set of uninit(mem) variables that satisfy the condition to be
% converted to uninit(reg) variables:
calc_convert_set(fail, _, _) -->> !.
calc_convert_set((Conj;Disj), Head, NaAr) -->>
   S/set, {cons(S)},
   {split_conj_begin_end(Conj, Begin, End)},
   {last_conj(End, Goal)},
   !,
   {varset(Begin, BVars)},
   [sub_set(BVars)]:set,
   {term_dupset(End, EDups)},
   % For later: can relax this Dups condition slightly:
   [sub_set(EDups)]:set,
   {stats(a,data7(NaAr))},
   last_goal_ureg(Goal, Head, NaAr),
   {stats(a,8)},
   % Removing this goal slows chat_parser by 1%:
   update_fast_goal(Goal, Head, NaAr),
   {stats(a,9)},
   calc_convert_set(Disj, Head, NaAr).
calc_convert_set((_;_), _, _) -->> [].

% Last_goal_ureg is used in the calculation of Head's set of uninit. regs.
% If the last goal in the clause is a non-survive goal, then the head’s
% uninitialized register arguments must be uninitialized register arguments
% of the last goal and in the same argument position.  This guarantees no
% extraneous move instructions destroying the possibility for TRO in the
% clause.
% If the last goal in the clause is a survive goal, then there are no extra
% conditions on the uninitialized register arguments since no TRO is possible
% anyway.
% If the last goal is not defined in the program being analyzed, then no
% uninitialized register arguments are possible.
last_goal_ureg(Goal,    _,    _) -->> {survive(Goal)}, !.
last_goal_ureg(Goal, Head, _/Ar) -->>
   S/set, {cons(S)},
   {functor(Goal, N, A)},
   [fget(N/A,UR)]:uregs,
   [fget(N/A,(G:-_))]:defs,
   !,
   % Rename UR’s variables to those of Goal:
   {stats(a,data10(N/A,G,Goal,UR))},
   {map_args(G, UR, Goal, URSet)},
   {stats(a,11)},
   {stats(a,data12(URSet))},
   intersectv(URSet):set,
   {stats(a,13)},
   {min_integer(A, Ar, Min)},
   {stats(a,data14(A,Ar,Min))},
   match_corresponding_args(1, Min, Head, Goal):set,
   {stats(a,15)}.
last_goal_ureg(_, _, _) -->> insert(_,[]):set.

% Additional optimization in the calculation of Goal’s set of uninit. regs.
% (This predicate can be removed without affecting correctness.)
% If the last goal is "fast" (i.e. it calls only builtins & survive goals) then
% update uregs for its definition too, by removing those ureg arguments whose
% position doesn’t match the head.  This avoids having to do move instructions
% AFTER the last goal returns.  Doing such moves is bad, in particular:
% 1. It removes the last call optimization (LCO) for that goal.
% 2. It may increase the number of environments created.
% If the last goal is not fast then it is not effective to keep LCO,
% since execution time of the last goal can become very large, so the few
% cycles gained by LCO aren’t an advantage.
% Note that uninit(mem) variables are better than uninit(reg)’s if an argument
% is moved.  This is because they can be passed INTO a predicate,
% whereas uninit(reg) variables must always be passed OUT OF a predicate.
% The string/1 predicate in chat_parser is an example of this, because it is
% factored.  It creates an environment when $fac_string/6 uses uninit(reg) args.
update_fast_goal(Goal, Head, _/Ar) -->>
   {\+survive(Goal)},
   {functor(Goal, N, A)},
   [fget(N/A,_)]:fast,
   [fget(N/A,UR)]:uregs,
   [fget(N/A,(G:-_))]:defs,
   !,
   {map_args(G, UR, Goal, URSet)},
   {min_integer(A, Ar, Min)},
   {match_corresponding_args(1, Min, Head, Goal, URSet, NewURSet)},
   {map_args(Goal, NewURSet, G, NUR)},
   update_ureg(N/A, UR, NUR).
update_fast_goal(_, _, _) -->> [].

% Return the arguments of Set which are in the same position in Head and Goal.
match_corresponding_args(N, M, _, _, _, []) :- N>M, !.
match_corresponding_args(N, M, Head, Goal, Set, Out) :- N=<M,
   arg(N, Head, X), inv(X, Set),
   arg(N, Goal, Y), X==Y,
   !,
   Out = [X|Next],
   N1 is N+1,
   match_corresponding_args(N1, M, Head, Goal, Set, Next).
match_corresponding_args(N, M, Head, Goal, Set, Out) :- N=<M,
   N1 is N+1,
   match_corresponding_args(N1, M, Head, Goal, Set, Out).

% Split a conjunction into two parts, where the second part is the largest
% conjunction of which all goals but the first don’t kill argument registers.
split_conj_begin_end(true, true, true).
split_conj_begin_end((Goal,Conj), true, (Goal,Conj)) :- all_survive(Conj), !.
split_conj_begin_end((Goal,Conj), (Goal,Begin), End) :-
   split_conj_begin_end(Conj, Begin, End).

all_survive(true).
all_survive((Goal,Conj)) :- survive(Goal), all_survive(Conj).

update_ureg(NaAr, UReg, NewUReg) -->> {UReg\==NewUReg}, !,
   [set(NaAr,NewUReg)]:uregs,
   [add(exit(NaAr))]:change.
update_ureg(_, _, _) -->> [].

% Create the initial values of Preds and URegTable:
init_convert(Strees, Entry, Preds, Fast, URegTable) :-
   init_convert_strees(Strees, Entry, _, Preds, [], _, Fast, _, URegTable),
   seal(Fast),
   seal(URegTable).

init_convert_strees([]) -->> [].
init_convert_strees([S|Ss]) -->>
        {S=stree(NaAr,(Head:-Disj),_,_,DList,_)}, !,
   [NaAr]:preds,
   enter_fast(NaAr, Disj),
   [fget(NaAr,Entry)]:entry,
   {get_set(uninit, Head, Entry, UReg)},
   [get(NaAr,X)]:uregs,
   {enter_ureg(X, UReg)},
        init_convert_strees(DList),
        init_convert_strees(Ss).
init_convert_strees([D|Ss]) -->>
        {directive(D)}, !,
        init_convert_strees(Ss).
```

```
enter_fast(NaAr, Disj) -->> {fast_routine(Disj)}, !, [get(NaAr,dummy)]:fast.
enter_fast(_, _) -->> [].

enter_ureg(X, UReg) :- var(X), !, X=UReg.
enter_ureg(X, _) :- nonvar(X).

% Succeeds for the definition of a "fast" routine, i.e. the routine calls only
% builtins and survive goals so that its execution time is almost independent
% of its arguments.
fast_routine((A;B)) :- !, fast_routine(A), fast_routine(B).
fast_routine((A,B)) :- !, fast_routine(A), fast_routine(B).
fast_routine(Goal) :- survive(Goal), !.
fast_routine(Goal) :- builtin(Goal), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Initialize entry and exit mode tables, a table of predicate definitions,
% and a table of predicates in which each predicate occurs.
% (Could create only one table as initial value for both entry & exit modes)

init_tables(Strees, Entry, Exit, Defs, Occurs) :-
   init_strees(Strees, _, Entry, _, Exit, _, Defs, _, Occurs),
   seal(Entry),
   seal(Exit),
   seal(Defs),
   seal(Occurs),
   !.

init_strees([]) -->> [].
init_strees([S|Ss]) -->>
   {S=stree(NaAr,(Head:-Disj),_,_,DList,_)}, !,
   {bottom_call(NaAr, B)},
   [get(NaAr,B)]:entry,
   [get(NaAr,B)]:exit,
   [get(NaAr,Def)]:defs,
   {enter_def(Def, (Head:-Disj), NaAr)},
   init_disj(Disj):pred(NaAr),
   init_strees(DList),
   init_strees(Ss).
init_strees([D|Ss]) -->>
   {directive(D)}, !,
   init_strees(Ss).

init_disj(fail) -->> [].
init_disj((Conj;Disj)) -->> init_conj(Conj), init_disj(Disj).

init_conj(true) -->> [].
init_conj((Goal,Conj)) -->> init_goal(Goal), init_conj(Conj).

init_goal(Goal) -->> {call_p(Goal)}, !,
   {functor(Goal, Na, Ar)},
   {bottom_call(Na/Ar, B)},
   [get(Na/Ar,B)]:entry,
   [get(Na/Ar,B)]:exit,
   Pred/pred,
   [add(Na/Ar,Pred)]:occurs.
init_goal(Goal) -->> {unify_p(Goal)}, !.

% Enter definition in table & give warning for multiple definitions:
enter_def(Def, NewDef, _) :- var(Def), !, NewDef=Def.
enter_def(Def, _,   NaAr) :- nonvar(Def), !,
   warning(['The predicate ',NaAr,' has multiple definitions.',nl,
        'Only the first definition will be used.']).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Create and traverse entry point data structures:

% Through the entry points' modes the analysis is kept consistent with
% external calls.

entry_data(Strees, Defs, EntData, SortPreds) :-
   (bagof(E, entry_data(E), Bag) -> true ; Bag=[]),
   filter_defs(Bag, EntData, EntDecl, Defs),
   entry_zero(Strees, EntPreds, EntDecl),
   sort(EntPreds, SortPreds). % Remove duplicates.

% Get all definitions of arity zero.  They are always entry points.
entry_zero([]) --> [].
entry_zero([stree(Na/0,_,_,_,DL,_)|Ss]) --> !, [Na/0],
   entry_zero(Ss), entry_zero(DL).
entry_zero([stree(_/N,_,_,_,DL,_)|Ss]) --> {N>0}, !,
   entry_zero(Ss), entry_zero(DL).
entry_zero([_|Ss]) --> entry_zero(Ss).

% Calculate the variable sets corresponding to the entry point's formula:
entry_data(entry(Head,Vars,Gnd,Non,Uni,Drf)) :-
   compile_option(entry(Head,Formula)),
   varset(Head, Vars),
   ground_set(Formula, Gnd),
   nonvar_set(Formula, Non),
   uninit_set(Formula, Uni),
   rderef_set(Formula, Drf).

% Keep only entries which are definitions:
filter_defs([], [], [], _).
filter_defs([E|In], [E|Out], [Na/Ar|Preds], Defs) :-
   E=entry(Head,_,_,_,_,_),
   functor(Head, Na, Ar),
   get(Defs, Na/Ar, _),
   !,
   filter_defs(In, Out, Preds, Defs).
filter_defs([_|In], Out, Preds, Defs) :-
   filter_defs(In, Out, Preds, Defs).

% Do one analysis pass over all entry points:
% Initialize tables for all entry points:
```

```
entry_init([]) -->> [].
entry_init([E|EntData]) -->> entry_init_one(E), entry_init(EntData).

entry_init_one(entry(Head,V,G,N,U,D)) -->>
   update_entry(Head, _, _)
    :[gnd(G,_),nonvar(N,_),uninit(U,_),rderef(D,_),sf(V,_),change([],_)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Top level of analysis:

analyze_strees(Strees, AStrees) :- compile_option(analyze), !,
   comment(['Starting dataflow analysis']),
   analyze(Strees, AStrees).
analyze_strees(Strees, Strees).

% During traversal of call tree update the entry & exit mode tables and
% keep track of the predicates whose entry and exit modes are changed.
% Repeat traversal step with the predicates that need propagating (i.e. those
% whose entry mode is changed & those which contain a goal whose exit mode
% is changed) until there are no more changes.
analyze(Strees, CStrees) :-
   stats(an,1),
   init_tables(Strees, Ent, Ex, Defs, Occurs),
   entry_data(Strees, Defs, EntData, EntPreds),
   cons(EntPreds), !,
   entry_init(EntData, Ent, MidEnt),
   stats(an,2),
   analyze_closure(EntPreds, MidEnt, OutEnt, Ex, OutEx, Defs, _,
           _, Spec, Occurs,_),
   stats(an,3),
   % Replace goals by their most specialized equivalents:
   seal(Spec),
   spec_strees(Strees, AStrees, Spec),
   stats(an,4),
   % Calculate which modes can be converted to uninit_reg:
   convert_uninit_strees(AStrees, OutEnt, Defs, Occurs, OutUReg),
   stats(an,5),
   % Update the modes:
   wn('% Modes generated:'),
   create_mode_strees(AStrees, BStrees, OutEnt, OutEx, OutUReg),
   stats(an,6),
   % Re-unravel the heads with the new modes (gives better selection):
   re_unr_strees(BStrees, CStrees),
   stats(an,7).
analyze(Strees, Strees) :-
   warning(['There are no usable entry points, so no flow analysis was done.']).

% ChgSet is the set of all predicates whose entry or exit modes have changed.
analyze_closure([]) -->> !.
analyze_closure(EntPreds) -->> cons(EntPreds), !,
   trav_preds(EntPreds):change([],ChgSet),
   {stats(an,2.5)},
   {length(ChgSet, N)},
   {comment(['Analysis pass--changed ',N,' entry and exit modes.'])},
   new_preds(ChgSet):change([],NewPreds),
   analyze_closure(NewPreds).

% Calculate from ChgSet the set of predicates that need further traversal:
% (This predicate also used in convert_uninit_strees)
new_preds([]) -->> !.
new_preds([entry(NaAr)|ChgSet]) -->> !,
   [add(NaAr)]:change,
   new_preds(ChgSet).
new_preds([exit(NaAr)|ChgSet]) -->> [fget(NaAr,Occurs)]:occurs, !,
   [add_set(Occurs)]:change,
   new_preds(ChgSet).
new_preds([_|ChgSet]) -->>
   new_preds(ChgSet).

% Top level call: Traverse predicates & update exit mode for each predicate:
trav_preds([]) -->> !.
trav_preds([NaAr|Preds]) -->>
   [fget(NaAr,Entry)]:entry,
   [fget(NaAr,Def)]:defs, !,
   {copy(Def,(Head:-Disj))},
   {stats(trav_pred,NaAr)},
   trav_pred(Disj, Head, Entry, NaAr),
   trav_preds(Preds).
% The predicate is not defined here (it is probably a builtin):
trav_preds([_|Preds]) -->>
   trav_preds(Preds).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Traversal of the call tree:

% Traverse a single predicate & its subtree of the call tree:
trav_pred(Disj, Head, Entry, NaAr) -->>
   new_sets(Head, Entry),
   OldDrf/rderef,
   OldGnd/gnd,
   trav_disj(Disj, Head, NaAr, 1, [], OutGnd, OutNon, OutDrf),
   [add_set(OutGnd)]:gnd,
   [add_set(OutNon)]:nonvar,
   {intersectv(OldDrf, OldGnd, D1)},
   {unionv(D1, OutDrf, D)},
   insert(_,OutDrf):rderef,
   update_exit(Head).

% Give each choice the same gnd, rderef, sf, and uninit values.
% Return the set of arguments ground at the output.
trav_disj(fail, Head, _, _, _, All, All, All) -->> !,
   {varset(Head, All)}.
% Don't redo an active clause: (unless current entry worse than stored?)
trav_disj((Conj;Disj), Head, NaAr, Num, CallList, OutGnd, OutNon, OutDrf) -->>
   {member(caller(NaAr,Num), CallList)}, !,
   {Num1 is Num+1},
   trav_disj(Disj, Head, NaAr, Num1, CallList, OutGnd, OutNon, OutDrf).
trav_disj((Conj;Disj), Head, NaAr, Num, CallList, OutGnd, OutNon, OutDrf) -->>
   save_sets(X),
   {term_dupset(Conj, Dups)},
   {stats(trav_conj,d(NaAr,Num))},
   trav_conj(Conj, NaAr, Num, 1, [caller(NaAr,Num)|CallList])
       :[uni(Unis,[]),dup(Dups)],
   back_propagate(Unis),
   NewGnd/gnd,
   NewNon/nonvar,
   NewDrf/rderef,
   restore_sets(X),
   {Num1 is Num+1},
   trav_disj(Disj, Head, NaAr, Num1, CallList, NewGnd2, NewNon2, NewDrf2),
   {intersectv(NewGnd, NewGnd2, OutGnd)},
   {intersectv(NewNon, NewNon2, OutNon)},
   {intersectv(NewDrf, NewDrf2, OutDrf)}.

% Back propagate gnd, rderef, and nonvar sets at end of clause to the head:
% Do closure on the clause’s unifications at the end of the clause
% to find any additional arguments that exit as gnd or rderef terms.
% This routine is not needed for correctness, but it improves results.
back_propagate(Unis) -->>
   back_prop_d_cl(Unis),
   back_prop_g_cl(Unis),
   Gnd/gnd, [add_set(Gnd)]:nonvar,
   back_prop_n_cl(Unis).

% Closure on rderef back propagation:
back_prop_d_cl(InUnis) -->>
   back_prop_d(InUnis, MidUnis):count(0,N),
   back_prop_d_cl(N, MidUnis).

back_prop_d_cl(0, _) -->> !.
back_prop_d_cl(N, Unis) -->> {N>0}, !, back_prop_d_cl(Unis).

back_prop_d([], []) -->> !.
back_prop_d([unify(yes, X, Y, YVars)|InUnis], OutUnis) -->>
   Drf/rderef, {subsetv(YVars, Drf)}, !,
   [add(X)]:rderef,
   [1]:count,
   back_prop_d(InUnis, OutUnis).
back_prop_d([unify(no, _, _, _)|InUnis], OutUnis) -->> !,
   back_prop_d(InUnis, OutUnis).
back_prop_d([U|InUnis], [U|OutUnis]) -->> back_prop_d(InUnis, OutUnis).

% Closure on gnd back propagation:
back_prop_g_cl(InUnis) -->>
   back_prop_g(InUnis, MidUnis):count(0,N),
   back_prop_g_cl(N, MidUnis).

back_prop_g_cl(0, _) -->> !.
back_prop_g_cl(N, Unis) -->> {N>0}, !, back_prop_g_cl(Unis).
```

```
back_prop_g([], []) -->> !.
back_prop_g([unify(_, X, Y, YVars)|InUnis], OutUnis) -->>
   {nonvar(Y)}, Gnd/gnd, {subsetv(YVars, Gnd)}, !,
   [add(X)]:gnd,
   [1]:count,
   back_prop_g(InUnis, OutUnis).
back_prop_g([unify(_, X, Y, YVars)|InUnis], OutUnis) -->>
   Gnd/gnd, {inv(X, Gnd)}, !,
   [add_set(YVars)]:gnd,
   [1]:count,
   back_prop_g(InUnis, OutUnis).
back_prop_g([U|InUnis], [U|OutUnis]) -->> back_prop_g(InUnis, OutUnis).

% Closure on nonvar back propagation:
back_prop_n_cl(InUnis) -->>
   back_prop_n(InUnis, MidUnis):count(0,N),
   back_prop_n_cl(N, MidUnis).

back_prop_n_cl(0, _) -->> !.
back_prop_n_cl(N, Unis) -->> {N>0}, !, back_prop_n_cl(Unis).

back_prop_n([], []) -->> !.
back_prop_n([unify(_, X, Y, _)|InUnis], OutUnis) -->>
   {var(X), var(Y)},
   Non/nonvar, {inv(Y, Non)}, !,
   [add(X)]:nonvar,
   [1]:count,
   back_prop_n(InUnis, OutUnis).
back_prop_n([U|InUnis], [U|OutUnis]) -->> back_prop_n(InUnis, OutUnis).

trav_conj(true, _, _, _, _) -->> !,
   % Any uninits remaining are initialized & thus become dereffed:
   Uni/uninit, [add_set(Uni)]:rderef.
trav_conj((Goal,Conj), NaAr, I, J, CallList) -->>
   {varset(Goal, Vars)},
   {stats(trav_goal,d(NaAr,I,J))},
   trav_goal(Goal, NaAr, I, J, CallList, Vars),
   {J1 is J+1},
   trav_conj(Conj, NaAr, I, J1, CallList).

trav_goal(Goal, NaAr, I, J, CallList, Vars) -->>
   {call_p(Goal)}, !,
   trav_call(Goal, NaAr, I, J, CallList, Vars),
   [sub_set(Vars)]:uninit,
   [Vars]:sf.
% Handling unify goals is done in three sections:
% trav_goal_n and trav_non (which updates the nonvar set),
% trav_goal_d and trav_drf1 & trav_drf2 (which updates the rderef set), and
% trav_goal_u and trav_unif (which updates the uninit and gnd sets).
% Each of these sections tries to get the best possible result.  This
% is implemented using backtracking to switch the unify arguments.
trav_goal(A=B, _, _, _, _, Vars) -->>
   make_uni(A, B, _),
   make_uni(B, A, _),
   {term_dupset(A=B, Dups)},
   trav_goal_d(A, B, Vars, Dups),
   trav_goal_u(A, B, Vars),
   Gnd/gnd, [add_set(Gnd)]:nonvar,
   trav_goal_n(A, B),
   % How do duplicate variables affect the Deref set?  Not at all.
   % How do circular terms affect the Deref set?
   [sub_set(Dups)]:uninit,
   Uni/uninit, [add_set(Uni)]:rderef,
   [Vars]:sf.

% Collect the unify goals in unify/3 data structures:
make_uni(X, Y, YVs) -->> {var(X)}, !,
   dref_prop_flag(X, Flag),
   {varset(Y, YVs)}, [unify(Flag, X, Y, YVs)]:uni.
make_uni(X, Y, _) -->> [].

% If (X is notin sf or X is uninit) and (X occurs only once in the body)
% then we can propagate rderef exit modes to X in back_propagate:
dref_prop_flag(X,  no) -->> Dups/dup, {inv(X, Dups)}, !.
dref_prop_flag(X, yes) -->> SF/sf, {\+inv(X,SF)}, !.
dref_prop_flag(X, yes) -->> Uni/uninit, {inv(X,Uni)}, !.
dref_prop_flag(_,  no) -->> [].

% 1. Backtracking between split_unify_v and trav_non:
trav_goal_n(A, B) -->>
   {split_unify_v(A, B, X, T)},
   trav_non(X, T),
   !.
% Default case:
trav_goal_n(_, _) -->> [].

% Update the nonvar set:
% This handles only the cases that are not taken care of in the gnd set.
trav_non(X, T) -->>
   {nonvar(T)},
   !,
   [add(X)]:nonvar.
trav_non(X, T) -->>
   {var(T)},
   Non/nonvar,
   {inv(T,Non)},
   !,
   [add(X)]:nonvar.

% 2. Backtracking between split_unify_v and trav_drf1 & trav_drf2:
trav_goal_d(A, B, Vars, Dups) -->>
   {split_unify_v(A, B, X, T)},
   {varset(T, TVars)},
   trav_drf1(X, T, TVars, Vars),
   !.
trav_goal_d(A, B, Vars, Dups) -->>
   {split_unify_v(A, B, X, T)},
   {varset(T, TVars)},
   trav_drf2(X, T, TVars, Vars, Dups),
   !.
% Default case:
trav_goal_d(A, B, Vars, Dups) -->>
   !,
   Gnd/gnd,
   intersectv(Gnd):rderef.

% Traverse a unify goal to get new Deref set:
% Try to find a successful clause in trav_drf1 before trav_drf2, because
% trav_drf1 gives a more powerful result.
% Relies on the fact that bindings always go from new variables to old.
trav_drf1(X, T, TVars, Vars) -->>
   SF/sf,
   {\+inv(X, SF)},
   !,
   new_dref(X, TVars),
   add_dref(TVars).
trav_drf1(X, T, TVars, Vars) -->>
   Uni/uninit,
   {inv(X, Uni)},
   !,
   new_dref(X, TVars),
   add_dref(TVars).

trav_drf2(X, T, TVars, _, _) -->>
   Drf/rderef,
   {inv(X, Drf)},
   Gnd/gnd,
   {inv(X, Gnd)},
   !,
   add_dref(TVars).
trav_drf2(X, T, TVars, _, Dups) -->>
   {Dups=[]},
   Drf/rderef,
   {inv(X, Drf)},
   SF/sf,
   {intersectv(SF, TVars, Old)},
   Uni/uninit,
   {subsetv(Old, Uni)},
   !,
   add_dref(TVars),
   [sub(X)]:rderef.

% If X is new (uninit or notin sf) then add it to rderef
% if all its arguments are rderef (i.e. (TVars n SF) c (Drf u Uni)).
new_dref(X, TVars) -->>
   SF/sf,
   {intersectv(TVars, SF, TSF)},
   Drf/rderef,
   Uni/uninit,
   {unionv(Drf, Uni, DU)},
   {subsetv(TSF, DU)},
   !,
   [add(X)]:rderef.
new_dref(X, TVars) -->> [].

% If (1) X is new (uninit or notin sf), or (2) X is ground and rderef,
% or (3) X is rderef and (SF n TVars) c Uni and there are no duplicate
% variables in {X} u TVars (because duplicate vars can create extra links),
% then can add TVars-(SF-Uni) to rderef:
add_dref(TVars) -->>
   SF/sf,
   Uni/uninit,
   {diffv(SF, Uni, ND)},
   {diffv(TVars, ND, D)},
   [add_set(D)]:rderef.

% 3. Backtracking between split_unify_v and trav_unif:
trav_goal_u(A, B, Vars) -->>
   {split_unify_v(A, B, X, T)},
   {varset(T, TVars)},
   trav_unif(X, T, TVars, Vars),
   !.
% Default case:
trav_goal_u(A, B, Vars) -->>
   [sub_set(Vars)]:uninit.

% Traverse a unify goal to get new Gnd and Uni sets:
% The manipulations done with the arguments of unify goals are the heart
% of the analysis.  The order of clauses in this predicate is important!
% This predicate fails if no conditions are satisfied.
trav_unif(X, T, TVars, Vars) -->>
   Gnd/gnd, {subsetv(TVars, Gnd)},
   !,
   [add(X)]:gnd,
   [sub(X)]:uninit.
trav_unif(X, T, TVars, Vars) -->>
   Uni/uninit, {inv(X, Uni)},
   !,
   SF/sf, {diffv(Vars, SF, V1)},
   [add_set(V1)]:uninit,
   [sub(X)]:uninit,
   {intersectv(Vars, SF, V3)},
   [sub_set(V3)]:uninit.
% Added Aug. 24, 1990:
trav_unif(X, T, TVars, Vars) -->>
   SF/sf, {\+inv(X,SF)},
   !,
   {diffv(Vars, SF, V1)},
   [add_set(V1)]:uninit,
   [sub(X)]:uninit,
   {intersectv(Vars, SF, V3)},
   [sub_set(V3)]:uninit.
trav_unif(X, T, TVars, Vars) -->>
   Gnd/gnd, {inv(X, Gnd)},
   !,
   [add_set(Vars)]:gnd,
   [sub_set(Vars)]:uninit.

% Traverse a non-unify goal to get new Gnd set:
% 1. Goal has a definition:
trav_call(Goal, _, _, _, CallList, Vars) -->>
   {functor(Goal, N, A)},
   [fget(N/A,Def)]:defs,
   !,
   update_entry(Goal, NewEntry, Flag),
   trav_def(Goal, N/A, Def, CallList, NewEntry, Flag, Vars).
% 2. Goal has no definition:
%    Find the most efficient entry point for Goal with the modal_entry tree.
%    Then update Gnd with the after modes of this entry.
%    After modes exist for builtins and if mode declarations were given,
%    even if no more efficient entry was found.
% Note: update_entry in trav_goal is irrelevant for this clause.
trav_call(Goal, NaAr, I, J, _, _) -->>
   spec_goal(NaAr, I, J, Goal, EGoal),
   {after(EGoal, After)},
   % Update rderef set to be in accord with the builtin’s exit mode:
   {bindset(EGoal, BS)},
   G/gnd,
   {diffv(BS, G, ND)},
   [sub_set(ND)]:rderef,
   {rderef_set(After, Drf)},
   [add_set(Drf)]:rderef,
   % Update gnd set to be in accord with the builtin’s exit mode:
   {ground_set(After, Gnd)},
   [add_set(Gnd)]:gnd,
   % Update nonvar set to be in accord with the builtin’s exit mode:
   {nonvar_set(After, Non)},
   [add_set(Non)]:nonvar,
   update_exit(EGoal).

% Traverse a goal that has a definition:
% 1. Stored entry mode of Goal is equal or worse and Goal is not in call list.
%    In this case, just update exit mode with no traversal.
%    This step needed to avoid having to traverse the whole call tree, which
%    has a worst-case size exponential in the program size.
trav_def(Goal, N/A, _, CallList, Entry, no, Vars) -->>
   {compile_option(nomem)},
   !,
   [fget(N/A,Exit)]:exit,
   {get_set(rderef, Goal, Exit, ExitDrf)},
   {get_set(ground, Goal, Exit, ExitGnd)},
   {get_set(nonvar, Goal, Exit, ExitNon)},
   new_drf_gnd(ExitDrf, ExitGnd, ExitNon, Vars).
trav_def(Goal, N/A, _, CallList, Entry, no, Vars) -->>
   {\+compile_option(nomem)},
   {\+member(caller(N/A,_), CallList)},
   !,
   [fget(N/A,Exit)]:exit,
   {get_set(rderef, Goal, Exit, ExitDrf)},
   {get_set(ground, Goal, Exit, ExitGnd)},
   {get_set(nonvar, Goal, Exit, ExitNon)},
   new_drf_gnd(ExitDrf, ExitGnd, ExitNon, Vars).
% 2. Stored entry mode of Goal is better or Goal is in the call list
%    & a definition exists: traverse to get new exit mode.
trav_def(Goal, N/A, Def, CallList, Entry, Flag, Vars) -->>
   {copy(Def,(Head:-Disj))},
   % 1. Set up correct state for the call:
   save_sets(X),
   new_sets(Head, Entry),
   % 2. Do the call:
   {stats(trav_disj,N/A)},
   trav_disj(Disj, Head, N/A, 1, CallList, OutGndH, OutNonH, OutDrfH),
   % 3. Restore state of caller:
   restore_sets(X),
   % 4. Calculate new exit mode:
   {map_argvars(Head, OutDrfH, Goal, OutDrfG)},
   {map_argvars(Head, OutGndH, Goal, OutGndG)},
   {map_args(Head, OutNonH, Goal, OutNonG)},
   new_drf_gnd(OutDrfG, OutGndG, OutNonG, Vars),
   update_exit(Goal).

% Calculate new rderef & gnd sets from the exit sets of a predicate:
% New rderef set = (Drf n Gnd) u (ExitDrf n (Uni u Drf u (Vars-SF)))
% New gnd set = (Gnd u ExitGnd)
% New nonvar set = (Non u ExitNon)
new_drf_gnd(ExitDrf, ExitGnd, ExitNon, Vars) -->>
   new_drf_set(ExitDrf, Vars),
   [add_set(ExitGnd)]:gnd,
   [add_set(ExitNon)]:nonvar.

new_drf_set([], _) -->> !,
   Gnd/gnd,
   intersectv(Gnd):rderef.
new_drf_set(ExitDrf, Vars) -->> cons(ExitDrf), !,
   insert(Drf,NewDrf):rderef,
   Gnd/gnd,
   Uni/uninit,
   SF/sf,
   {unionv(Uni, Drf, D1)},
   {diffv(Vars, SF, NV)},
   {unionv(NV, D1, D2)},
   {intersectv(ExitDrf, D2, D3)},
   {intersectv(Gnd, Drf, D4)},
   {unionv(D3, D4, NewDrf)}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Support for specialized entries:

% The call to a goal is replaced by the most specific entry possible,
% depending on its input mode.  This is useful to speed up builtins,
% although it can be used for any predicate that is in the modal_entry tree
% (for example, this can be used to speed up any library).
% It is implemented with the accumulator 'spec', which holds the most
% specialized entry for each call that has a modal_entry.  The index of spec
% is index(NaAr,ClauseNum,GoalNum), which uniquely identifies each goal in
% the program.

% For later: better interaction between required/needed uninit mem/reg?

% During analysis, replace Goal by the more specialized EGoal,
% if that is possible:
spec_goal(NaAr, ClNum, GNum, Goal, EGoal) -->>
   {modal_entry(Goal, _)},
   !,
   {Index = index(NaAr,ClNum,GNum)},
   [get(Index,Val)]:spec,
   functor(Goal, N, A),
   calc_entry(Goal, Entry),
   spec_goal2(Goal, N/A, Index, Val, Entry, EGoal).
spec_goal(_, _, _, Goal, Goal) -->> [].

% Get the previous specialization:
spec_goal2(Goal, NA, Index, Val, Entry, EGoal) -->> {var(Val)}, !,
   {lattice_modes_entry(NA, Entry, Goal, F)},
   {efficient_entry(Goal, EGoal, F)},
   Val = value(Goal,Entry,EGoal).
spec_goal2(Goal, NA, Index, Val, Entry, EGoal) -->> {nonvar(Val)}, !,
   {copy(Val, value(Goal,LEntry,LGoal))},
   spec_update(Goal, NA, Index, LEntry, LGoal, Entry, EGoal).

% Update the specialization & its entry lattice:
spec_update(Goal, NA, Index, LEntry, LGoal, Entry, EGoal) -->>
   {LEntry\==Entry},
   !,
   {lattice_modes_entry(NA, Entry, Goal, F)},
   {efficient_entry(Goal, EGoal, F)},
   [set(Index,value(Goal,Entry,EGoal))]:spec.
spec_update(Goal, NA, Index, LEntry, LGoal, LEntry, LGoal) -->> [].

% After analysis, convert all calls to their specialized equivalent:
% (The traversal code is a bit tedious, it would be nice to have a preprocessor
% here to do the traversing!)
spec_strees([], [], _).
spec_strees([S|Strees], [SS|SStrees], Spec) :-
   spec_stree(S, SS, Spec),
   spec_strees(Strees, SStrees, Spec).

spec_stree(stree(NaAr,(Head:-Disj), M,OH, DL,SD),
      stree(NaAr,(Head:-SDisj),M,OH,SDL,SD), Spec) :- !,
   spec_disj(Disj, SDisj, NaAr, 1, Spec),
   spec_strees(DL, SDL, Spec).
spec_stree(D, D, _) :- directive(D).

spec_disj(fail, fail, _, _, _).
spec_disj((Conj;Disj), (SConj;SDisj), NaAr, I, Spec) :-
   spec_conj(Conj, SConj, NaAr, I, 1, Spec),
   I1 is I+1,
   spec_disj(Disj, SDisj, NaAr, I1, Spec).

spec_conj(true, true, _, _, _, _).
spec_conj((Goal,Conj), (SGoal,SConj), NaAr, I, J, Spec) :-
   spec_goal(Goal, SGoal, NaAr, I, J, Spec),
   J1 is J+1,
   spec_conj(Conj, SConj, NaAr, I, J1, Spec).

spec_goal(Goal, SGoal, NaAr, I, J, Spec) :-
   get(Spec, index(NaAr,I,J), Val), !,
   copy(Val, value(Goal,_,SGoal)).
spec_goal(Goal, Goal, _, _, _, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% High-level utilities to calculate, propagate, and update modes:

save_sets(state(G,N,U,D,S)) -->>
   G/gnd,
   N/nonvar,
   U/uninit,
   D/rderef,
   S/sf.

restore_sets(state(G,N,U,D,S)) -->>
   insert(_,G):gnd,
   insert(_,N):nonvar,
   insert(_,U):uninit,
   insert(_,D):rderef,
   insert(_,S):sf.

% Assumes that Head contains only variables.
% (If not, get_set(nonvar) is incorrect because nonvar is non-recursive.)
new_sets(Head, Entry) -->>
   insert(_,G):gnd,
   insert(_,N):nonvar,
   insert(_,U):uninit,
   insert(_,D):rderef,
   insert(_,S):sf,
   {get_set(ground, Head, Entry, G)},
   {get_set(nonvar, Head, Entry, N)},
   {get_set(uninit, Head, Entry, U)},
   {get_set(rderef, Head, Entry, D)},
   {varset(Head, S)}.

% Map a set of variables between two goals:
% Maps from G1’s argument to G2’s argument only if the latter is a variable.
map_args(G1, Set1, G2, Set2) :-
   functor(G1, Na, Ar),
   map_args(1, Ar, G1, Set1, G2, Bag2, []),
   sort(Bag2, Set2).

map_args(I, Ar, G1, Set1, G2) -->
   {I=<Ar, arg(I, G1, A), inv(A, Set1)},
   {arg(I, G2, X), var(X)},
   !,
   [X],
   {I1 is I+1},
   map_args(I1, Ar, G1, Set1, G2).
map_args(I, Ar, G1, Set1, G2) --> {I=<Ar}, !,
   {I1 is I+1},
   map_args(I1, Ar, G1, Set1, G2).
map_args(I, Ar, _, _, _) --> {I>Ar}, !.

% Map a set of variables between two goals:
% Maps from each argument of G1 to all variables in G2’s corresponding argument.
map_argvars(G1, Set1, G2, Set2) :-
   functor(G1, Na, Ar),
   map_argvars(1, Ar, G1, Set1, G2, Bag2, []),
   sort(Bag2, Set2).

map_argvars(I, Ar, G1, Set1, G2) --> {I=<Ar, arg(I, G1, A), inv(A, Set1)}, !,
   {arg(I, G2, X)},
   varbag(X),
   {I1 is I+1},
   map_argvars(I1, Ar, G1, Set1, G2).
map_argvars(I, Ar, G1, Set1, G2) --> {I=<Ar}, !,
   {I1 is I+1},
   map_argvars(I1, Ar, G1, Set1, G2).
map_argvars(I, Ar, _, _, _) --> {I>Ar}, !.

% Get the set of head arguments corresponding to the lattice value Type:
get_set(Type, Head, Mode, TypeSet) :-
   functor(Head, Na, Ar),
   get_bag(1, Ar, Type, Head, Mode, Bag, []),
   sort(Bag, TypeSet).

get_bag(I, Ar, T, Head, Mode) -->
   {I=<Ar, arg(I, Mode, T1), greater_eq(T, T1)},
   !,
   {arg(I, Head, X)},
   varbag(X),
   {I1 is I+1},
   get_bag(I1, Ar, T, Head, Mode).
get_bag(I, Ar, T, Head, Mode) --> {I=<Ar}, !,
   {I1 is I+1},
   get_bag(I1, Ar, T, Head, Mode).
get_bag(I, Ar, _, _, _) --> {I>Ar}, !.

% Update the entry mode & return the most general entry mode.
% (Does NOT modify gnd or uninit sets)
update_entry(Goal, NewEntry, Flag) -->>
   calc_entry(Goal, Entry),
   {functor(Goal, Na, Ar)},
   [fget(Na/Ar,OldEntry)]:entry,
   {lub_call(OldEntry, Entry, NewEntry)},
   update_entry(Na/Ar, OldEntry, NewEntry, Flag).
update_entry(NaAr, OldEntry, NewEntry, yes) -->>
   {OldEntry\==NewEntry}, !,
   [set(NaAr,NewEntry)]:entry,
   [add(entry(NaAr))]:change.
update_entry(_, _, _, no) -->> [].

% Update the exit mode & return the most general exit mode.
update_exit(Goal) -->>
   {functor(Goal, Na, Ar)},
   [fget(Na/Ar,OldExit)]:exit,
   !,
   calc_exit(Goal, Exit),
   {lub_call(OldExit, Exit, NewExit)},
   update_exit(Na/Ar, OldExit, NewExit).
update_exit(_) -->> [].

update_exit(NaAr, OldExit, NewExit) -->>
   {OldExit\==NewExit}, !,
   [set(NaAr,NewExit)]:exit,
   [add(exit(NaAr))]:change.
update_exit(_, _, _) -->> [].

% Calculate the exit mode corresponding to the current values of the sets:
calc_exit(Goal, Exit) -->>
   {functor(Goal, Na, Ar)},
   {functor(Exit, Na, Ar)},
   Gnd/gnd,
   Non/nonvar,
   Drf/rderef,
   {calc_exit_2(1, Ar, Gnd, Non, Drf, Goal, Exit)}.

calc_exit_2(I, Ar, _, _, _, _, _) :- I>Ar, !.
calc_exit_2(I, Ar, Gnd, Non, Drf, Goal, Exit) :- I=<Ar, !,
   arg(I, Goal, X),
   arg(I, Exit, Y),
   varset(X, XVars),
   subset_flag(XVars, Gnd, Gf),
   subset_flag(XVars, Drf, Df),
   calc_exit_arg(Gf, Df, Y1),
   fix_nonvar(X, Non, Y1, Y),
   I1 is I+1,
   calc_exit_2(I1, Ar, Gnd, Non, Drf, Goal, Exit).

% Get an exit mode lattice value:
% Flags:      Gnd  Drf
calc_exit_arg(yes, yes, gnddrf) :- !. % Argument known to be ground & rderef.
calc_exit_arg(yes,  no, ground) :- !. % Argument known to be ground.
calc_exit_arg( no, yes, rderef) :- !. % Argument known to be rderef.
calc_exit_arg( no,  no, any) :- !.    % Argument could be anything.

% Calculate the entry mode corresponding to the current values of the sets:
calc_entry(Goal, Entry) -->>
   {term_dupset(Goal, Dups)},
   Gnd/gnd,
   Non/nonvar,
   Uni/uninit,
   Drf/rderef,
   SF/sf,
   {functor(Goal, Na, Ar)},
   {functor(Entry, Na, Ar)},
   {calc_entry_2(1, Ar, Gnd, Non, Dups, Uni, Drf, SF, Goal, Entry)}.

% Calculate the entry mode, given more information:
calc_entry_2(I, Ar, _, _, _, _, _, _, _, _) :- I>Ar, !.
calc_entry_2(I, Ar, Gnd, Non, Dups, Uni, Drf, SF, Goal, Entry) :- I=<Ar,!,
   arg(I, Goal, X),
   arg(I, Entry, Y),
   varset(X, XVars),
   subset_flag(XVars, Gnd, Gf),
   var_flag(X, Vf),
   diffv(SF, Uni, T1),
   unionv(T1, Dups, Old),
   membership_flag(X, Old, Of),
   intersectv(XVars, SF, XVarsSF),
   subset_flag(XVarsSF, Drf, Df),
   calc_entry_arg(Gf, Vf, Of, Df, Y1),
   fix_nonvar(X, Non, Y1, Y),
   I1 is I+1,
   calc_entry_2(I1, Ar, Gnd, Non, Dups, Uni, Drf, SF, Goal, Entry).

% Get an entry mode argument lattice value:
% Flags:       Gnd, Var, Old, Drf.
calc_entry_arg(yes,   _,   _, yes, gnddrf) :- !. % Ground & rderef.
calc_entry_arg(yes,   _,   _,  no, ground) :- !. % Ground argument.
calc_entry_arg( no,  no,   _, yes, rderef) :- !. % Non-var is rderef.
calc_entry_arg( no,  no,   _,  no, any) :- !. % Non-var is not rderef.
calc_entry_arg( no, yes,  no,   _, uninit) :- !. % Non-dup uninit.
calc_entry_arg( no, yes, yes, yes, rderef) :- !. % Init rderef.
calc_entry_arg( no, yes, yes,  no, any) :- !. % Initialized.

% Fix up the mode to add the nonvar information:
fix_nonvar(X, _, Y1, Y) :- nonvar(X), !,
   add_nonvar_info(yes, Y1, Y).
fix_nonvar(X, Non, Y1, Y) :- var(X), !,
   membership_flag(X, Non, Nf),
   add_nonvar_info(Nf, Y1, Y).

add_nonvar_info( no,      Y,      Y) :- !.
add_nonvar_info(yes, rderef, nondrf) :- !.
add_nonvar_info(yes,    any, nonvar) :- !.
add_nonvar_info(yes,      Y,      Y) :- !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Lattice utilities:

% Calculate the least upper bound of two arguments on the argument lattice:
lub(unknown,       X,      X) :- !.
lub(      X, unknown,      X) :- !.
lub(    any,       _,    any) :- !.
lub(      _,     any,    any) :- !.
lub(      X,       X,      X) :- !.
lub( nonvar,  ground, nonvar) :- !.
lub( ground,  nonvar, nonvar) :- !.
lub( nonvar,  nondrf, nonvar) :- !.
lub( nondrf,  nonvar, nonvar) :- !.
lub( nonvar,  gnddrf, nonvar) :- !.
lub( gnddrf,  nonvar, nonvar) :- !.
lub( ground,  nondrf, nonvar) :- !.
lub( nondrf,  ground, nonvar) :- !.
lub( ground,  gnddrf, ground) :- !.
lub( gnddrf,  ground, ground) :- !.
lub( nondrf,  gnddrf, nondrf) :- !.
lub( gnddrf,  nondrf, nondrf) :- !.
lub( rderef,  nondrf, rderef) :- !.
lub( nondrf,  rderef, rderef) :- !.
lub( rderef,  gnddrf, rderef) :- !.
lub( gnddrf,  rderef, rderef) :- !.
lub( rderef,  uninit, rderef) :- !.
lub( uninit,  rderef, rderef) :- !.
lub( uninit,  gnddrf, rderef) :- !.
lub( gnddrf,  uninit, rderef) :- !.
lub( uninit,  nondrf, rderef) :- !.
lub( nondrf,  uninit, rderef) :- !.
lub( nonvar,  rderef,    any) :- !.
lub( rderef,  nonvar,    any) :- !.
lub( nonvar,  uninit,    any) :- !.
lub( uninit,  nonvar,    any) :- !.
lub( ground,  rderef,    any) :- !.
lub( rderef,  ground,    any) :- !.
lub( ground,  uninit,    any) :- !.
lub( uninit,  ground,    any) :- !.
lub(A,B,any) :- error(['Bug in lub with ',lub(A,B,_)]).

% Greater than or equal function on lattice:
greater_eq(ground, gnddrf) :- !.
greater_eq(rderef, gnddrf) :- !.
greater_eq(rderef, nondrf) :- !.
greater_eq(nonvar, gnddrf) :- !.
greater_eq(nonvar, nondrf) :- !.
greater_eq(nonvar, ground) :- !.
greater_eq(     T,      T) :- !.

% Bottom element of the argument lattice:
bottom(unknown).

% Calculate the least upper bound of two modes on the call lattice:
lub_call(Call1, Call2, Lub) :-
   functor(Call1, Na, Ar),
   functor(Call2, Na, Ar),
   functor(Lub, Na, Ar),
   lub_call(1, Ar, Call1, Call2, Lub).
lub_call(I, Ar, _, _, _) :- I>Ar, !.
lub_call(I, Ar, Call1, Call2, Lub) :- I=<Ar, !,
   arg(I, Call1, X1),
   arg(I, Call2, X2),
   arg(I, Lub, X),
   lub(X1, X2, X),
   I1 is I+1,
   lub_call(I1, Ar, Call1, Call2, Lub).

% Create a bottom call lattice value:
bottom_call(Na/Ar, Bottom) :-
   functor(Bottom, Na, Ar),
   bottom_call(1, Ar, Bottom).

bottom_call(I, Ar, Bottom) :- I>Ar, !.
bottom_call(I, Ar, Bottom) :- I=<Ar, !,
   bottom(B),
   arg(I, Bottom, B),
   I1 is I+1,
   bottom_call(I1, Ar, Bottom).

% Create modes for a predicate from the final table:
% Given a table of lattice entries:
lattice_modes_table(Na/Ar, Table, Head, Formula) :-
   functor(Head, Na, Ar),
   get(Table, Na/Ar, Entry),
   lattice_modes_call(1, Ar, Entry, Head,    mem, Formula, true).

% Given the lattice entry itself: (this is used in specialization)
lattice_modes_entry(Na/Ar, Entry, Head, Formula) :-
   lattice_modes_call(1, Ar, Entry, Head, either, Formula, true).

lattice_modes_call(I, Ar, _, _, _) --> {I>Ar}, !.
lattice_modes_call(I, Ar, Value, Head, UT) --> {I=<Ar}, !,
   {arg(I, Value, T)},
   {arg(I, Head, X)},
   lattice_modes_arg(T, UT, X),
   {I1 is I+1},
   lattice_modes_call(I1, Ar, Value, Head, UT).

lattice_modes_arg(uninit, mem, X) --> !, co(uninit(X)).
lattice_modes_arg(uninit,   T, X) --> !, co(uninit(T,X)).
lattice_modes_arg(ground,   _, X) --> !, co(ground(X)).
lattice_modes_arg(rderef,   _, X) --> !, co(rderef(X)).
lattice_modes_arg(gnddrf,   _, X) --> !, co(ground(X)), co(rderef(X)).
lattice_modes_arg(nonvar,   _, X) --> !, co(nonvar(X)).
lattice_modes_arg(nondrf,   _, X) --> !, co(nonvar(X)), co(rderef(X)).
lattice_modes_arg( Other,   _, X) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% General utilities:
```

```
% Return a flag to show set membership:
membership_flag(X, Set, yes) :- inv(X, Set), !.
membership_flag(X,   _,  no).

% Return a flag to show if an argument’s variables are in the set:
subset_flag(Vars, Set, yes) :- subsetv(Vars, Set), !.
subset_flag(_,   _,  no).

% Return a flat to show if an argument’s variables are disjoint from the set:
disjoint_flag(Vars, Set, yes) :- disjointv(Vars, Set), !.
disjoint_flag(X,   _,  no).

% Return a flag to show if an argument is a variable:
var_flag(X, yes) :- var(X), !.
var_flag(X,  no) :- nonvar(X), !.

% Calculate the set of head arguments that are variables:
var_args(Goal, Set) :-
   Goal=..[_|RawBag],
   filter_vars(RawBag, Bag),
   sort(Bag, Set).

% Calculate the set of variables in Goal that occur more than once:
% Also calculate the set or bag of variables in Goal.
% (Also used in clause_code.pl)
term_dupset(Goal, DupSet) :-
   term_dupset_varbag(Goal, DupSet, _).

term_dupset_varset(Goal, DupSet, VarSet) :-
   term_dupset_varbag(Goal, DupSet, VarBag),
   sort(VarBag, VarSet).

term_dupset_varbag(Goal, DupSet, VarBag) :-
   varbag(Goal, VarBag),
   filter_dups(VarBag, DupSet).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

clause_code.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Clause code generation:

% This module takes care of calls to predicates which may require
% uninitialized variables.  The code to do this is complex.

% This module does a case analysis to generate code for uninitialized
% variables.  There are 4 x 3 possibilities: The predicate requires an init,
% an uninit(reg), or an uninit(mem) variable.  It also is given one of these
% three.

% Originally, all new variables in unified terms were created as uninit
% in var mode.  Better is only to create as uninit those variables which
% are required by a goal to be uninitialized.  This is done through the
% passed argument aun, the set of required uninitialized variables
% for all arguments in a body, which is passed from body into goal and unify.
% Meaning: aun = variables that are allowed to be uninitialized.
% Only variables in this set are allowed to be created as uninit, since they
% can be fruitfully used as uninit.

% Notes:
% 1. Any variable X that is var(X) which is passed to a user-defined predicate
%    must have the var(X) formula removed after the predicate, since we don’t
%    know if the predicate instantiates any arguments.  All other tests remain
%    true, because they start with an instantiated variable which can’t change.
% 2. Steps 1,2,3 could be interleaved: pre - call_move - post for each
%    argument.  If done, this must be signaled to regalloc.
% 3. In combination with procedure selection code generation, this module
%    could use more information on which registers contain which arguments.
%    Presently it assumes that procedure selection changes no argument
%    registers.
% 4. This module does not instantiate any variables in the input clause.
% 5. The calls to ’univ’ can be rewritten in a more verbose style using
%    indexing.  Transformation should be able to take care of this.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Entry 1: Accepts a list of clauses in body((H:-B),Entries) form.
% These clauses are given labels to identify them.
% This routine accepts the list of bodies created by segment.pl.
% For each body, there is a list of entries with a label and a mode formula
% valid at the entry.
% Small blocks of code are created for each entry to initialize the uninits
% that need it, i.e. those that aren’t required uninit for the body or else
% that don’t occur in some other entry.
clause_code_list([], _) --> !.
% The body has been called at least once:
clause_code_list([body((H:-B),Entries)|Bodies], DList) -->
   {nonvar(Entries)}, !,
   {entry_comment(H, B, Entries)},
   {split_uninit_list(Entries, UMSets, URSets, UForms, IForms)},

   % Calculate & initialize the extra uninits:
   % {uninit_allowset(B, DList, UReq)},
   % {intersectv_list([UReq|USets], UIn)},
   {intersectv_list(UMSets, UMIn)},
   {intersectv_list(URSets, URIn)},
   {functor(H, N, A)},
   insert_uninit(UMSets, H, UMIn, Entries, N/A),
   % Init. of uninit(reg) is not necessary as long as their use
   % is restricted to returning last call arguments.

   % Calculate the new formula for the body:
   {intersect_formula_list(IForms, Finit)},
   {uninit_origins(UForms, Origins)},
   {make_uninit_mem_formula(UMIn, Origins, MidF, Finit)},
   {make_uninit_reg_formula(URIn, NewF, MidF)},

   [label(N/A)],
   clause_code((H:-B), DList, return, NewF),
   clause_code_list(Bodies, DList).
% The body has not been called at all,
% therefore it is not necessary to compile it:
clause_code_list([body((H:-B),Entries)|Bodies], DList) -->
   {var(Entries)}, !,
   clause_code_list(Bodies, DList).

entry_comment(H, B, Entries) :- compile_option(entry_comment), !,
   length(Entries, _),
   comment(H, ['(',B,') is called with ',Entries]).
entry_comment(_, _, _) :- \+compile_option(entry_comment), !.

% From the entries, extract: (1) sets of uninit vars, (2) uninit formulas, and
% (3) remaining formulas.
split_uninit_list([], [], [], [], []).
split_uninit_list([entry(_,F)|Fs], [UMVs|UMSets], [URVs|URSets], [UF|UFs],
          [IF|IFs]) :-
   split_uninit(F, UF, IF),
   uninit_set_type(mem, UF, UMVs),
   uninit_set_type(reg, UF, URVs),
   split_uninit_list(Fs, UMSets, URSets, UFs, IFs).

% Insert code for each entry to initialize the correct uninit variables,
% and then jump to the body.
insert_uninit([], _, _, [], _) --> [].
insert_uninit([Us|USets], Head, UIn, [entry(Lbl,_)|Es], NaAr) -->
   {diffv(Us, UIn, Xs)},
   [label(Lbl)],
   init_code_block(Xs, Head),
   [jump(NaAr)],
   insert_uninit(USets, Head, UIn, Es, NaAr).

init_code_block([], _) --> [].
init_code_block([X|Xs], Head) -->
   {find_arg(N, Head, X)}, !,
   {low_reg(L), R is L+N-1},
   pragma_tag(R, var),
   [move(R,[R])],
   init_code_block(Xs, Head).
init_code_block([X|Xs], Head) -->
   error(Head, ['Variable ',X,' is not initialized.']),
   init_code_block(Xs, Head).

% Make a new formula containing the right uninit modes:
make_uninit_mem_formula([], _) --> [].
make_uninit_mem_formula([X|Vars], Origins) -->
   {member_origin(X-Set, Origins)}, !,
   co(uninit(mem,X,Set)),
   make_uninit_mem_formula(Vars, Origins).
make_uninit_mem_formula([X|Vars], Origins) -->
   co(uninit(X)),
   make_uninit_mem_formula(Vars, Origins).

make_uninit_reg_formula([]) --> [].
make_uninit_reg_formula([X|Vars]) -->
   co(uninit(reg,X)),
   make_uninit_reg_formula(Vars).

member_origin(X-Set, [Y-S|_]) :- X==Y, !, Set=S.
member_origin(X-Set, [_|Origins]) :- member_origin(X-Set, Origins).

% Calculate the origin sets (the arguments of the unifications where they
% could have been created) for all uninitialized variables:
% This is necessary because each entry to the body may have created the same
% uninitialized variable in a different unification.  Whenever a variable is
% passed as an argument then all uninits containing it in their origin sets
% have to be initialized.
uninit_origins(Fs, Origins) :-
   origin_fragments(Fs, Fragments, []),
   keysort(Fragments, Sort),
   origin_merge(Sort, Origins).

% List all known origin sets of uninitialized variables:
origin_fragments([]) --> [].
origin_fragments([F|Fs]) --> origin_formula(F), origin_fragments(Fs).

origin_formula((A,B)) --> !, origin_formula(A), origin_formula(B).
origin_formula(uninit(mem,X,Set)) --> !, [X-Set].
origin_formula(Goal) --> [].

% Merge origin set fragments together:
origin_merge([], []).
origin_merge([X-Set|Frags], Origins) :-
   origin_merge(Frags, X, S, Origins, []).

origin_merge([], X, S) --> [X-S].
origin_merge([Y-T|Frags], X, S) --> {X==Y}, !,
   {append(T, S, Sp)},
   origin_merge(Frags, X, Sp).
origin_merge([Y-T|Frags], X, S) --> {X\==Y}, !,
   [X-S],
   origin_merge(Frags, Y, T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Entry 2: Accepts a single clause in any form.
% Return Flag:
%   RFlag = return: return to caller after execution.
%   RFlag = jump(L):    jump to L after execution.
clause_code(Clause, DList, RFlag, InF) -->
   clause_code(Clause, DList, RFlag, InF, OutF).

clause_code(Clause, DList, RFlag, InF, OutF) -->
   {standard_form(Clause, Head, Body)},
   clause_code_3((Head:-Body), DList, RFlag, InF, OutF).

% Generate code and do register allocation for a clause in standard form.
clause_code_3(Cl, DList, RFlag, fail, fail) --> !, [fail].
clause_code_3(Cl, DList, RFlag, InF, OutF) -->
   {\+InF=fail},
   [A],
   {copy([Cl|InF], [CopyCl|CopyInF])},
   clause(CopyCl, DList, NCl, CopyInF, CopyOutF),
   {copy([CopyCl|CopyOutF], [Cl|OutF])},
   [D],
   return_jump(RFlag),
   {clause_allocate(NCl, EnvNeeded, NumPerms)},
   {environment(EnvNeeded, NumPerms, A, D)}, !.

% Handle distinction dummy/nodummy clauses:
return_jump(return)    --> [return].
return_jump(jump(Lbl)) --> [jump(Lbl)].

% Insertion of environment instructions:
environment(no,  _, nop, nop).
environment(yes, N, allocate(N), deallocate(N)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Declarations for clause code generation:

acc_info(  sf, X,  In,  Out, includev(X,In,Out)).
acc_info(  vl, V, Out,   In, Out=[V|In]).
acc_info(code, I, Out,   In, Out=[I|In]).
acc_info(init, I, Out,   In, Out=[I|In]).
acc_info(form, F, InF, OutF, update_formula(F,InF,OutF)).
acc_info(fixp, _, In, Out, In=Out). % FixPerms for write-once permanents.
acc_info(repp, _, In, Out, In=Out). % ReplacePerms for them.
acc_info(fixi, Vs, In, Out, unionv(In,Vs,Out)). % FixInit for uninit mem vars.
acc_info(repi, Vs, In, Out, unionv(In,Vs,Out)). % Replacements for them.
acc_info(umem, X, Out, In, Out=[X|In]). % Vars previously uninit mem.
```

```
% Calculate global perms:
acc_info(vars,  Vs, In, Out, unionv(In,Vs,Out)). % Variables
acc_info(half,  Vs, In, Out, unionv(In,Vs,Out)). % Half-permanents
acc_info(perms, Vs, In, Out, unionv(In,Vs,Out)). % Permanents
% Calculate allowed uninit vars:
acc_info(varbag, X, Out, In, Out=[X|In]).
acc_info(uvars, Vs, In, Out, unionv(Vs,In,Out)).
acc_info(dlist_acc, _, In, Out, In=Out).

pass_info(dlist).
pass_info(gperms).
pass_info(aun). % Set of vars allowed to be created as uninitialized.

pred_info(  body,2, [gperms,aun,dlist,fixp,repp,fixi,repi,  sf,form,code]).

% For write-once permanents:
pred_info(perm_c,2, [gperms,          fixp,repp,fixi,repi,  sf,form,code]).
pred_info(  perm,2, [gperms,          fixp,repp,            sf,form,code]).
pred_info( update_mapping, 3, [fixp,repp]).
pred_info(init_extra_vars, 3, [                          vl,sf,form,code]).
pred_info( init_extra_var, 3, [                          vl,sf,form,code]).

% For goal compilation:
pred_info(         goal_c, 5, [aun,dlist,umem,fixi,repi,    sf,form,code]).
pred_info(           goal, 5, [aun,dlist,umem,fixi,         sf,form,code]).
pred_info(x_call_moves_post_c2, 2, [          fixi,repi,            code]).
pred_info(       the_call, 5, [    dlist,                      form,code]).
pred_info(call_instruction,2, [    dlist,                           code]).
pred_info(parameter_setup, 3, [                          vl,sf,form,code]).
pred_info(       args_pre, 3, [          umem,           vl,sf,form,code]).
pred_info(    one_arg_pre, 4, [          umem,           vl,sf,form,code]).
pred_info(   args_post_c2, 2, [               fixi,repi,    sf,form,code]).
pred_info(      args_post, 3, [                          vl,sf,form,code]).
pred_info(   one_arg_post, 4, [                          vl,sf,form,code]).
pred_info(    init_uninit, 1, [                          vl,sf,form,code]).
pred_info(init_uninit_set, 2, [                          vl,sf,form,code]).
pred_info(    init_uninit, 2, [                          vl,sf,     code,init]).

% Calculate global perms:
pred_info(gpermvars, 1, [vars,half,perms]).
pred_info(gpermstep, 1, [vars,half,perms]).
pred_info(gpermsurv, 1, [vars,half      ]).

% Calculate allowed uninit vars:
pred_info(uninit_allowbag, 1, [dlist_acc,varbag,uvars]).
pred_info(     unify_vars, 1, [                 uvars]).
pred_info(  unify_collect, 1, [          varbag,uvars]).
pred_info(     req_uninit, 1, [          varbag      ]).

% External: (defined in unify.pl)
pred_info(new_var_nf, 1, [    sf,vl,     code]).
pred_info(   new_var, 1, [    sf,vl,form,code]).
pred_info(   new_var, 2, [    sf,vl,form,code]).
pred_info(create_arg, 2, [    sf,vl,form,code]).
pred_info(   unify_g, 2, [aun,sf,   form,code,fixi]).
pred_info(unify_goal, 2, [aun,sf,   form,code]).
pred_info(pragma_tag, 2, [code]).
% External: (defined in conditions.pl)
pred_info(      remove_vars, 0, [form]).
pred_info(remove_all_uninit, 0, [form]).
pred_info(    remove_uninit, 1, [form]).
pred_info(    remove_uninit, 2, [form]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generate code for a clause:
% The clause must be in standard form.

clause((Head:-Body), DList, (Head:-NewBody), InF, OutF) -->
   {uninit_set_type(reg, InF, URegs)},
   head_moves_pre(Head, URegs, SF, HVl),
   {find_gperms((Head:-Body), Ps)},
   {uninit_allowset(Body, DList, Aun)}, % Allowed to be uninit.
   body(Body, NBody, Ps, Aun, DList,
        [], _, [], _, [], FixI, [], RepI, SF, OutSF, InF, OutF),
   head_moves_post_c(Head, URegs, RVl, FixI, RepI),
   {flat_conj(('$varlist'(HVl),NBody,'$varlist'(RVl)), NewBody)}.

% Calculate the set of variables that are allowed to be created
% as uninit.  These include the required uninit variables in:
% (1) goals in $body,
% (2) goals in strees in the dlist + required modes of the stree itself.
% Do NOT include vars in a unify goal's term that are also in a later goal.
uninit_allowset(Body, DList, Set) :-
   uninit_allowbag(Body, DList, _, Bag, [], [], _),
   sort(Bag, Set).

uninit_allowbag((A,B)) -->> !, uninit_allowbag(A), uninit_allowbag(B).
uninit_allowbag((A;B)) -->> !, uninit_allowbag(A), uninit_allowbag(B).
uninit_allowbag('$body'((_:-Body),_,_)) -->> !, uninit_allowbag(Body).
uninit_allowbag(Goal) -->>
   {functor(Goal, N, A)},
   DList/dlist_acc,
   {member(stree(N/A,(Head:-Disj),(H:-F),_,DL,SD), DList)},
   !,
   {copy((H:-F),(Goal:-Form))},
   uninit_bag(Form):varbag,
   {copy((Head:-Disj), (Goal:-Body))},
   % An example of a different use of an accumulator:
   insert(_, DL):dlist_acc,
   uninit_allowbag(Body),
   insert(_, DList):dlist_acc.
uninit_allowbag(Goal) -->>
   {require(Goal, Req)}, uninit_bag(Req):varbag.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Generate code for the body:
```

```
% Quad encapsulation with split_formula, efficient_entry, map_terms,
% and goal_c to ensure:
% (1) Independence of formula manipulation,
% (2) Implementation of efficient entry point choice (usu. for builtins).
% (3) Implementation of write-once permanents,
% (4) Reduction of new var dereference chain length (flow analysis consistency).
%     Later: for init. vars too?
body(_, fail) -->> InF/form, {InF=fail}, !,
   [fail]:code.
body(true, '$varlist'(Vl)) -->> !,
   init_uninit([]):vl(Vl,[]).
body((Goal,Body), (VGoal,NGoal,XGoal,AGoal,NBody)) -->>
   {stats(goal,Goal)},
   %********************
   % 1. Keep the mode formula for the goal as simple as possible, since
   % otherwise execution time goes up by a large factor.  Also, split off
   % irrelevant 'var' goals to keep them from being removed in goal.
   insert(InF, AdjInGoalF):form,
   SoFar/sf,
   {split_formula(_, SoFar, Goal, InF, InGoalF, RestF)},
   {stats(x,8)},
   {last_goal_adjust(Body, InGoalF, RestF, AdjInGoalF, AdjRestF)},
   {stats(x,9)},
       %****************
       % 2. Handle choice of more efficient entry point:
       efficient_entry(Goal, EGoal):[form,sf],
   {stats(x,10)},
       %****************
       % 3. Handle write-once permanent variables:
       % (Deref chain mapping done in perm_c/2, which calls perm/2.)
       perm_c(EGoal, VGoal),
   {stats(x,11)},
       F/fixp, R/repp,
       {map_terms(F, R, EGoal, PGoal)},
   {stats(x,12)},
       map_terms(F, R):form,
   {stats(x,13)},
       %************
       % 4. Reduce length of deref. chain for new vars:
       % This is necessary to be consistent with flow analysis.
       % (Deref chain mapping done in goal_c/5, which calls goal/5.)
       Form/form,
       {uninit_bag_type(mem, Form, U1)},
   {stats(x,14)},
           %********
           % 5. Compile the goal:
               goal_c(PGoal, NGoal, Data1, Data2, After):umem(U2,U1),
           %********
       {sort(U2, USet)},
       make_indirect(USet, Ind),
       append(USet):fixi,
       append(Ind):repi,
       x_call_moves_post_c2(Data1, XGoal),
       args_post_c2(Data2, AGoal),
        [After]:form, % Must be done after args_post.  WHY?
                  % Seems to lead to deref bug of C in test71.pl.
        % [USet]:fixi,
        % [Ind]:repi,
        %************
        map_terms(R, F):form,
        %****************
    insert(OutGoalF, OutF):form,
    {combine_formula(OutGoalF, AdjRestF, OutF)},
    %*******************
    body(Body, NBody).

% Add all remaining uninit goals to the last goal’s formula:
% This ensures all uninits are init before the last goal, for tail call opt.
last_goal_adjust(true, InGoalF, RestF, AdjInGoalF, AdjRestF) :- !,
    split_uninit(RestF, UnF, AdjRestF),
    combine_formula(UnF, InGoalF, AdjInGoalF).
last_goal_adjust(Body, InGoalF, RestF, InGoalF, RestF) :- \+Body=true, !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dereference chain transformation:

% Bypass the extra link in the dereference chain caused by the binding of
% new variables (incl. uninit mem).  Every occurrence X of a new variable
% that has been initialized is replaced by [X] (an extra indirection).
% This speeds up later dereferencing of that value & also guarantees
% that all variables considered dereferenced by flow analysis are really so.
% For correctness, ’pref’ allocation is not done for variables that get an
% extra indirection.

% For each goal in the clause:
goal_c(PGoal, NewNGoal, Data1, Data2, After) -->>
    I/fixi, {cons(I)}, !,
    R/repi,
    goal(PGoal, NGoal, Data1, Data2, After):[code(Code,[])],
    % Replace variable by an extra indirection:
    map_instr_list(Code, I, R):code,
    {map_varlist(NGoal, NewNGoal, I)}.
goal_c(PGoal, NGoal, Data1, Data2, After) -->>
    goal(PGoal, NGoal, Data1, Data2, After).

% head_moves_post generates code, so it too must do mapping:
head_moves_post_c(Head, URegs, NewVL, I, R) -->
    {cons(I)}, !,
    {head_moves_post(Head, URegs, VL, Code, [])},
    map_instr_list(Code, I, R),
    {map_varlist_list(VL, NewVL, I)}.
head_moves_post_c(Head, URegs, VL, _, _) -->
    head_moves_post(Head, URegs, VL).

% perm/2 generates code too, so it must also do indirection mapping:
perm_c(Goal, NewVGoal) -->>
    I/fixi, {cons(I)}, !,
   R/repi,
   perm(Goal, VGoal):[code(Code,[])],
   map_instr_list(Code, I, R):code,
   {map_varlist(VGoal, NewVGoal, I)}.
perm_c(Goal, VGoal) -->>
   perm(Goal, VGoal).

% x_call_moves_post/6 generates code after the call, so it must also do mapping:
% Must do this in two steps since it must use the UPDATED fixi and repi.
% 1. Save the arguments in goal/4:
x_call_moves_post_c1(Goal, URegs, data(Goal, URegs)).

% 2. Do x_call_moves_post/6 with updated fixi & repi in body/2:
x_call_moves_post_c2(data(Goal, URegs), VGoal) -->>
   I/fixi, {cons(I)}, !,
   R/repi,
   x_call_moves_post(Goal, URegs):[vl(Vl,[]),code(Code,[])],
   map_instr_list(Code, I, R):code,
   {map_varlist_goal(Vl, VGoal, I)}.
x_call_moves_post_c2(data(Goal, URegs), '$varlist'(Vl)) -->> !,
   x_call_moves_post(Goal, URegs):[vl(Vl,[]),code].
x_call_moves_post_c2(none, '$varlist'([])) -->> [].

% args_post/3 generates code after the call, so it must also do mapping:
% Must do this in two steps since it must use the UPDATED fixi and repi.
% 1. Save the arguments in goal/4:
args_post_c1(Flags, Args, NArgs, data(Flags,Args,NArgs)).

% 2. Do args_post with updated fixi & repi in body/2:
args_post_c2(data(Flags,Args,NArgs), VGoal) -->>
   I/fixi, {cons(I)}, !,
   R/repi,
   args_post(Flags, Args, NArgs):[vl(Vl,[]),code(Code,[])],
   map_instr_list(Code, I, R):code,
   {map_varlist_goal(Vl, VGoal, I)}.
args_post_c2(data(Flags,Args,NArgs), '$varlist'(Vl)) -->> !,
   args_post(Flags, Args, NArgs):vl(Vl,[]).
args_post_c2(none, '$varlist'([])) -->> [].

make_indirect([], []).
make_indirect([X|USet], [[X]|Ind]) :- make_indirect(USet, Ind).

% Remove 'pref' annotation if one of the preferred variables is an indirection:
% Create a new varlist to replace the old one.
map_varlist(Body, Varlist, I) :-
   bodylist(Body, VL, []),
   map_varlist_goal(VL, Varlist, I).

map_varlist_goal(VL, '$varlist'(NewVL,Link), I) :-
   map_varlist_list(VL, I, NewVL, Link).

map_varlist_list(VL, NewVL, I) :-
   map_varlist_list(VL, I, NewVL, []).
map_varlist_list([], _) --> [].
map_varlist_list([X,Y,Z|VL], I) -->
   {X==pref},
   {(memberv(Y, I) ; memberv(Z, I))},
   !,
   [Y,Z],
   map_varlist_list(VL, I).
map_varlist_list([X|VL], I) --> [X],
   map_varlist_list(VL, I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Generate code for goals where each argument has 3x3 possibilities:
% Method:
%  0. get required modes of the goal from tables.
%  1. call one_arg_pre for all Arg-NewArg pairs.
%  2. generate code for the call itself.
%  3. add pred's "after" modes to current formula.
%  4. call one_arg_post for all Arg-NewArg pairs.
% Wrapped calls ('$body' calls created by segment.pl)
% need minimal support (i.e. only passing arguments):
goal(G, '$varlist'(V1), Data, none, true) -->>
   {call_p(G)},
   {unwrap_goal(G, Goal, yes)}, !,
   {varbag(Goal, Varbag)},
   init_uninit(Varbag):vl(V1,V2),
   {functor(Goal, N, A)},
   {gensym("$e_", N/A, Np)},
   unwrap_form(G, Np/A):form,
   Form/form,
   {uninit_set_type(reg, Form, URegs)},
   x_call_moves_pre(Goal, URegs, V2, V3):code,
   {fence_if_die(Goal, V3, [])},
   [call(Np/A)]:code,
   {x_call_moves_post_c1(Goal, URegs, Data)},
   unionv(URegs):sf,
   remove_vars,
   remove_all_uninit.
% Standard calls need lots of support:
goal(G, VarList, none, Data, After) -->>
   {call_p(G)},
   {unwrap_goal(G, Goal, no)}, !,
   {functor(Goal, N, A)},
   {Goal=..[N|Args]},
   {VarList=('$varlist'(Vpre),VT,'$varlist'(Vm),VGoal)},
   parameter_setup(Goal, Flags, IReq):vl(Vpre,Vm1),
   {stats(x,15)},
   args_pre(Flags, Args, NArgs):vl(Vm1,[]),
   MyF/form,
   {stats(x,data16(ireq(IReq),form(MyF)))},
   {NGoal=..[N|NArgs]},
   rtests_m(IReq, VT):[form,code],
   {stats(x,17)},
   warn_fail(G):form,
   {append(Args, NArgs, AllArgs)},
   init_uninit(AllArgs):vl(Vm,[]),
   {stats(x,18)},
   the_call(Goal, NGoal, VGoal, Flags, NArgs),
   remove_all_uninit,
   {after(Goal, After)},
   {map_vars(Args, NArgs, After, NAfter)},
   % Use logical_simplify here?
   [NAfter]:form,
   {args_post_c1(Flags, Args, NArgs, Data)}.
   % args_post(Flags, Args, NArgs):vl(Vpost,[]).
% Unification goal:
goal(X=T, '$varlist'(Vlist), none, none, true) -->>
   % {unify_p(X=T)}, !,
   % unify handles all explicit variables in X=T:
   {varbag(X=T, Varbag)},
   init_uninit(Varbag):vl(Vlist,Vmid),
   unify_g(X=T, Vmid).

% Ensure that there are no double indirections in unify code:
% This ensures correctness; maybe more efficient code would result if
% some of the cases are handled in the write-once permanent transformation.
unify_g(X=T, Vm) -->> {var(X), compound(T)},
   I/fixi, {memberv(X, I)}, % X will have an extra indirection.
   F/form, {\+implies(F,unbound(X))}, % X needs read-mode unification.
   !,
   unify_goal((X=Z,Z=T), Vm).
unify_g(T=X, Vm) -->> {var(X), compound(T)},
   I/fixi, {memberv(X, I)}, % X will have an extra indirection.
   F/form, {\+implies(F,unbound(X))}, % X needs read-mode unification.
   !,
   unify_goal((X=Z,Z=T), Vm).
unify_g(X=T, Vm) -->> unify_goal(X=T, Vm).

% When the mode formula becomes fail:
warn_fail(G, InF, InF) :- InF=fail, !,
   warning(['Bad require modes--the goal ',G,' can never be reached.']).
warn_fail(_, InF, InF).

% Unwrap the possible wrapping done in segment.pl:
unwrap_goal('$body'((Goal:-_),_,_), Goal, yes) :- !.
unwrap_goal(Goal, Goal, no).

% Attach the mode formula to the end of the Forms list:
unwrap_form('$body'((Goal:-_),_,Entries), NpAr, F, F) :- !,
   end_of_list(Entries, E), E=[entry(NpAr,F)|_].
unwrap_form(_, _, F, F).

end_of_list(L, E) :- var(L), !, L=E.
end_of_list(L, E) :- nonvar(L), !, L=[_|T], end_of_list(T, E).

% Initialize all uninitialized variables in a formula which are not in
% the list L.  The variables in L are assumed to be taken care of elsewhere.
% For calls L is (Args U NArgs), which are handled by args_pre and args_post.
% For unifications L is all of the variables in the unification.
init_uninit(L) -->> InF/form,
   init_uninit(L, InF):init(Init,[]),
   remove_uninit(Init).

init_uninit(L, (A,B)) -->> !, init_uninit(L, A), init_uninit(L, B).
init_uninit(L,  Goal) -->> {an_uninit_mode(Goal,mem,V), \+memberv(V,L)}, !,
   pragma_tag(V, var),
   [V]:init, [V]:vl, [V]:sf, [move(V,[V])]:code.
init_uninit(L,  Goal) -->> {an_uninit_mode(Goal,reg,V), \+memberv(V,L)}, !,
   [V]:init, new_var_nf(V).
init_uninit(L,  Goal) -->> [].

% Initialize a set of uninit variables:
init_uninit_set([], _) -->> !.
init_uninit_set([V|Vars], UMSet) -->> {inv(V,UMSet)}, !,
   pragma_tag(V, var),
   [V]:vl, [V]:sf, [move(V,[V])]:code,
   init_uninit_set(Vars, UMSet).
init_uninit_set([V|Vars], UMSet) -->> {\+inv(V,UMSet)}, !,
   new_var_nf(V),
   init_uninit_set(Vars, UMSet).

macro_expand(Head, Code, Link, Varlist, Varlink, InSF, OutSF) :-
   macro(Head, Code, Link, Varlist, Varlink, Vars),
   sort(Vars, Vs),
   unionv(Vs, InSF, OutSF).

% *** The call itself:
% Compile the call itself & return the modified goal needed for regalloc.
% A call that is expanded: (usually a built-in)
% (It is assumed the expanded call will do no unification.)
the_call(Goal, NGoal, VGoal, _, NArgs) -->>
   expand(Goal, NGoal, VGoal):[form,code], !.
the_call(_, NGoal, '$varlist'(V1,V3), _, _) -->>
   macro_expand(NGoal):[code,vl(V1,V2),sf], !,
   {fence_if_die(NGoal, V2, V3)}.
% A call with fixed registers that is not expandable:
the_call(_, NGoal, '$varlist'(V1), Flags, NArgs) -->>
   {\+anyregs(NGoal)}, !,
   {low_reg(L)},
   call_moves_pre(L, Flags, NArgs, V1, V2):code,
   {fence_if_die(NGoal, V2, V3)},
   {functor(NGoal, N, A)},
   {stats(x,19)},
   call_instruction(N/A, NGoal),
   call_moves_post(L, Flags, NArgs, V3, []):code,
   remove_vars.
% A call with any registers that is not expandable:
the_call(_, NGoal, NGoal, _, _) -->>
   {anyregs(NGoal)}, !,
   [call(NGoal)]:code,
   {functor(NGoal, _, A)},
   remove_vars.
```

```
% Choose between standard call (needing allocate),
% one level simple_call (which doesn’t need allocate),
% and inline compilation of a dummy stree:
call_instruction(NaAr, _) -->> DList/dlist,
   {S=stree(NaAr,_,_,_,_,_)},
   {member(S, DList)}, !,
   compile_stree(S, nopeep, jump(_), noheader, nowrite):code.
call_instruction(NaAr, Goal) -->> {survive(Goal)}, !, [simple_call(NaAr)]:code.
call_instruction(NaAr, Goal) -->> {\+survive(Goal)}, !, [call(NaAr)]:code.

% *** Preamble to goal parameter passing:
% Calculate flags; initialize the uninit vars that need it;
% and return the required initialized modes.
parameter_setup(Goal, Flags, IReq) -->>
   {functor(Goal, N, A)},
   {Goal=..[N|Args]},

   % Initialize duplicate arguments that are uninitialized,
   % i.e. first occurrences and given uninitialized variables.
   Form1/form,
   SF1/sf,
   {uninit_set(Form1, USet)},
   {uninit_set_type(mem, Form1, UMSet)},
   {term_dupset_varset(Goal, DSet, VSet)},
   {diffv(VSet, SF1, New)},
   {unionv(New, USet, X)},
   {intersectv(DSet, X, InitSet)},
   {warn_dup_uninit(Goal, InitSet)},
   init_uninit_set(InitSet, UMSet),
   remove_uninit(InitSet),

   % Get flags of given arguments:
   Form2/form,
   SF2/sf,
   {uninit_set_type(reg, Form2, GRSet)},
   {uninit_set_type(mem, Form2, GMSet)},
   {get_given_flags(GRSet, GMSet, SF2, Args, Flags)},

   % Get flags of required arguments:
   {require(Goal, Req)},
   {split_uninit(Req, UReq, IReq)},
   {uninit_set_type(reg, UReq, RRSet)},
   {uninit_set_type(mem, UReq, RMSet)},
   {get_req_flags(RRSet, RMSet, Args, Flags)},

   % Warning if a required uninit is given an init:
   {warn_req_uninit(1, Goal, Flags, N/A)}.

warn_dup_uninit(_, []) :- !.
warn_dup_uninit(Goal, [X]) :- !,
   comment(Goal, [’Argument ’,X,’ of ’,Goal,’ is duplicated.’]).
warn_dup_uninit(Goal, S) :- cons(S), !,
   comment(Goal, [’Arguments ’,S,’ of ’,Goal,’ are duplicated.’]).
```

```
warn_req_uninit(_, _, [], _) :- !.
warn_req_uninit(I, Goal, [flag(ini,mem)|Flags], NaAr) :- !,
   arg(I, Goal, X),
   warning(Goal, ['Argument ',X,' of ',Goal,
              ' is given an init but requires an uninit.']),
   I1 is I+1,
   warn_req_uninit(I1, Goal, Flags, NaAr).
warn_req_uninit(I, Goal, [_|Flags], NaAr) :-
   I1 is I+1,
   warn_req_uninit(I1, Goal, Flags, NaAr).

% Type of argument a predicate is given:
get_given_flags(_, _, _, [], []) :- !.
get_given_flags(GRSet, GMSet, InSF, [A|Args], [flag(F,_)|Flags]) :-
   get_given_one(GRSet, GMSet, InSF, A, F),
   get_given_flags(GRSet, GMSet, InSF, Args, Flags).

% Order is important here:
% In particular, an uninit(mem) variable may or may not be an element of SF.
% The argument of reg(_) gives the origin of an uninit reg argument: either
% it comes from an uninit mode, or it is simply because the arg it notin SF.
% In the first case, need to pass result out from register.
% In the second case, allowed initialize the variable first.
% (Study this difference more.)
get_given_one(GR, GM, SF, A, ini) :- nonvar(A), !.
get_given_one(GR, GM, SF, A, mem) :- var(A), inv(A, GM), !.
get_given_one(GR, GM, SF, A, ini) :- var(A), inv(A, SF), !.
get_given_one(GR, GM, SF, A, reg(sf)) :- var(A), \+inv(A, SF), !.
get_given_one(GR, GM, SF, A, reg(uni)) :- var(A), inv(A, GR), !.

% Type of argument a predicate is required to have:
get_req_flags(_, _, [], []) :- !.
get_req_flags(RRSet, RMSet, [A|Args], [flag(_,F)|Flags]) :-
   get_req_one(RRSet, RMSet, A, F),
   get_req_flags(RRSet, RMSet, Args, Flags).

% Order is important here:
get_req_one(RR, RM, A, mem) :- inv(A, RM), !.
get_req_one(RR, RM, A, reg) :- inv(A, RR), !.
get_req_one(RR, RM, A, ini). % Defaulty case.

% *** Set up the arguments for the call:
args_pre([], [], []) -->> !.
args_pre([flag(F1,F2)|Flags], [A|Args], [B|NArgs]) -->>
   one_arg_pre(F1, F2, A, B),
   args_pre(Flags, Args, NArgs).

% *** Fix up arguments after the call:
args_post([], [], []) -->> !.
args_post([flag(F1,F2)|Flags], [A|Args], [B|NArgs]) -->>
   one_arg_post(F1, F2, A, B),
   args_post(Flags, Args, NArgs).
```

```
% Information about the arguments for one_arg_pre and one_arg_post:

% It is important to remember that the sf values are updated in
% args_pre and args_post; it is not done in call_moves_pre or call_moves_post.
% The given type reg(T) gives the source of the uninit-ness: T in {uni, sf}.
% Reg(uni) arguments are handled differently from reg(sf) arguments; both ways
% are correct, but sometimes one is more efficient than the other.
% The use of new_var(A,B) with two arguments reduces the number of references
% to the environment (e.g. in (main:-a(X),a(Y),b(X,Y))).

% First argument:
% reg(T) argument A is given uninit(reg,A) OR (var(A) & A notin InSF) (1st occ).
% mem    argument A is given uninit(mem,A).
% ini    argument A is given an initialized argument (default).

% Second argument:
% reg  predicate requires uninit(reg,A).
% mem  predicate requires uninit(mem,A).
% ini  predicate requires an initialized argument (default).

% Before the call operation:
% (Uninit modes are removed in goal, not here.)
% (Umem vars are those with an extra link.)
one_arg_pre(reg(_),reg, A,A) -->> !.
one_arg_pre(mem,reg, A,B) -->> !,
   [A]:umem.
one_arg_pre(ini,reg, A,B) -->> !.
one_arg_pre(reg(uni),mem, A,B) -->> !, [B]:sf,
   [B]:umem,
   [B,B]:vl,
        {tag(var, Tvar)},
        {align(K), K1 is K-1},
        [move(Tvar^r(h),B),adda(r(h),1,r(h)),pad(K1)]:code.
one_arg_pre(reg(sf), mem, A,B) -->> !, [A,B]:sf,
   [A,B]:umem,
   [A,pref,A,B,B]:vl,
        {tag(var, Tvar)},
        {align(K), K1 is K-1},
        [move(Tvar^r(h),A),move(Tvar^r(h),B),adda(r(h),1,r(h)),pad(K1)]:code.
one_arg_pre(mem,mem, A,A) -->> !,
   [A]:umem.
one_arg_pre(ini,mem, A,B) -->> !, [B]:sf,
   [B]:umem,
        [B,B]:vl,
        {tag(var, Tvar)},
        {align(K), K1 is K-1},
        [move(Tvar^r(h),B),adda(r(h),1,r(h)),pad(K1)]:code,
   combine_formula(uninit(B)):form.
one_arg_pre(reg(uni),ini, A,B) -->> !,
   [B]:umem,
   new_var(B).
one_arg_pre(reg(sf), ini, A,B) -->> !,
   [A,B]:umem,
   new_var(A, B).
```

```
one_arg_pre(mem,ini, A,B) -->> !, [B]:sf,
   [A,B]:umem,
   [A,pref,A,B]:vl,
   pragma_tag(A, var),
   [move(A,[A]),move(A,B)]:code,
   combine_formula((deref(A),var(A),A==B)):form.
one_arg_pre(ini,ini, A,B) -->> !,
   {varset(A, AVars)},
   SF/sf, {diffv(AVars, SF, New)},
   difflist(New):umem,
   create_arg(A, B).

% After the call operation:
one_arg_post(reg(_),reg, A,A) -->> !, [A]:sf.
% After the call A is still an uninit mem:
one_arg_post(mem,reg, A,B) -->> !, [B]:sf, [B,A]:vl, [move(B,A)]:code.
one_arg_post(ini,reg, A,B) -->> !, [B]:sf, unify(B=A):[sf,vl,form,code].
one_arg_post(reg(uni),mem, A,B) -->> !, [A]:sf, [B,A]:vl, [move(B,A)]:code.
one_arg_post(reg(sf), mem, _,_) -->> !.
one_arg_post(mem,mem, _,_) -->> !.
one_arg_post(ini,mem, A,B) -->> !, unify(B=A):[sf,vl,form,code].
one_arg_post(reg(uni),ini, A,B) -->> !, [A]:sf, [B,A]:vl, [move(B,A)]:code.
one_arg_post(reg(sf), ini, _,_) -->> !.
one_arg_post(mem,ini, _,_) -->> !.
one_arg_post(ini,ini, _,_) -->> !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Utilities for moving arguments to and from registers & variables
% for the head & for body goals.
% Note the special handling of uninitialized register arguments.

% Generate a sequence of moves to successive argument registers:
call_moves(_, []) --> !.
call_moves(I, [X|NArgs]) --> !,
   [move(X,r(I))],
   {I1 is I+1},
   call_moves(I1, NArgs).

% Generate code for head:
% Generate a sequence of moves from registers to non-uninit(reg) variables:
% Return OutSF and Varlist (uninit(reg) arguments are not in them).
head_moves_pre(Head, URegs, OutSF, VL) -->
   {Head=..[H|Args]},
        {low_reg(L)},
        head_moves_pre_2(Args, L, URegs, VL),
        {sort(Args, ArgVars)},
   {diffv(ArgVars, URegs, OutSF)}.

head_moves_pre_2([], _, _, []) --> !.
head_moves_pre_2([X|NArgs], I, URegs, VL) --> {inv(X, URegs)}, !,
   {I1 is I+1},
   head_moves_pre_2(NArgs, I1, URegs, VL).
head_moves_pre_2([X|NArgs], I, URegs, [pref,r(I),X|VL]) -->
```

```
   [move(r(I),X)],
   {I1 is I+1},
   head_moves_pre_2(NArgs, I1, URegs, VL).

% Generate a sequence of moves from uninit(reg) variables to the registers
% that must contain their values:
head_moves_post(Head, URegs, VL) -->
   head_moves_post_2(URegs, Head, VL).

head_moves_post_2([], _, []) --> [].
head_moves_post_2([X|URegs], Head, [pref,X,R|VL]) -->
   {head_reg(Head, X, R)},
   [move(X,R)],
   head_moves_post_2(URegs, Head, VL).

% Generate a sequence of moves to from vars to non-uninit(reg) arg. registers:
% This is done BEFORE the call instruction.
x_call_moves_pre(Head, URegs, VL, L) -->
   {varset(Head, HVars)},
   {diffv(HVars, URegs, CallSet)},
   x_call_moves_pre_2(CallSet, Head, VL, L).

x_call_moves_pre_2([], _, VL, VL) --> [].
x_call_moves_pre_2([X|Set], Head, [pref,X,R|VL], L) -->
   {head_reg(Head, X, R)},
   [move(X,R)],
   x_call_moves_pre_2(Set, Head, VL, L).

% Generate a sequence of moves from uninit(reg) argument registers to vars:
% This is done AFTER the call instruction.
x_call_moves_post(Head, URegs, VL, L) -->
   x_call_moves_post_2(URegs, Head, VL, L).

x_call_moves_post_2([], _, VL, VL) --> [].
x_call_moves_post_2([X|URegs], Head, [pref,R,X|VL], L) -->
   {head_reg(Head, X, R)},
   [move(R,X)],
   x_call_moves_post_2(URegs, Head, VL, L).

% Generate a sequence of moves to non-uninit(reg) argument registers:
% This is done BEFORE the call instruction.
call_moves_pre(_, [], [], VL, VL) --> !.
call_moves_pre(I, [flag(_,reg)|Flags], [_|NArgs], VL, L) --> !,
   {I1 is I+1},
   call_moves_pre(I1, Flags, NArgs, VL, L).
call_moves_pre(I, [_|Flags], [X|NArgs], [pref,X,r(I)|VL], L) -->
   [move(X,r(I))],
   {I1 is I+1},
   call_moves_pre(I1, Flags, NArgs, VL, L).

% Generate a sequence of moves from uninit(reg) argument registers:
% This is done AFTER the call instruction.
call_moves_post(_, [], [], VL, VL) --> !.
call_moves_post(I, [flag(_,reg)|Flags], [X|NArgs], [pref,r(I),X|VL], L) --> !,
   [move(r(I),X)],
   {I1 is I+1},
   call_moves_post(I1, Flags, NArgs, VL, L).
call_moves_post(I, [_|Flags], [_|NArgs], VL, L) -->
   {I1 is I+1},
   call_moves_post(I1, Flags, NArgs, VL, L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Extension of clause_code to handle write-once permanent variables.

% Two routines are defined in this section: perm/2 and find_gperms/2.
% Perm/2 ensures that global permanent variables in a goal are
% renamed if binding to them could take place.  Old permanent
% variables are dereferenced into the extra variables.

% Notes:
% 1. The permanents that should be renamed are the initialized global perms
%    that are in:
%    1. If   survive(Goal) then (vars to be dereffed) U (bindset)
%    2. If \+survive(Goal) then (vars to be dereffed)
%    where:
%       (vars to be dereffed) = (required dereffed) - (already dereffed)
%    Justification: binding of a survive goal arg changes the value in the
%    current environment, so it requires a new permanent.  Binding of a
%    \+survive goal arg doesn't change the value in the current environment,
%    so it doesn't need to be renamed in the current environment.
% 2. During execution of body/2, accumulators form and goal are kept with
%    their original variables & only mapped for goal/2.  Accumulators vl, sf,
%    and code always use extra variables.
% 3. The instruction deref(p(I)) should never be generated when the option
%    write_once exists.  If it is then there is a bug in the clause compiler.
% 4. Initialization of global permanent variables is not done here, which
%    should be ok, since this code only initializes the variables which it
%    creates, i.e. extra variables.

% Rename the permanent variables in a goal if necessary &
% generate a move(Pi,Ei) or deref(Pi,Ei) for them.
% The mapping Q->Qp from old permanents to new permanents
% is represented as a pair of lists Q and Qp.
% Mapping is done efficiently by continuing to use the original Perms
% set instead of continually updating it for each goal.
perm(Goal, '$varlist'(VList)) -->>
   {write_once}, !,
   Form/form,
   {vars_to_rename(Goal, Form, Vs)},
   Perms/gperms,
   {intersectv(Vs, Perms, Ws)},
   SoFar/sf,
   {intersectv(Ws, SoFar, Q)},
   {copy(Q, X), sort(X, Qp)},
   update_mapping(Q, Qp, Qold),
   init_extra_vars(Q, Qp, Qold):vl(VList,[]).
perm(Goal, '$varlist'([])) -->>
```

```
   {\+write_once}, !.

% The permanents that should be renamed are
% the initialized global perms that are in:
% 1. If   survive(Goal) then (vars to be dereffed) U (bindset)
% 2. If \+survive(Goal) then (vars to be dereffed)
% where:
% (vars to be dereffed) = (required dereffed) - (already dereffed)
vars_to_rename(Goal, Form, Vs) :-
   survive(Goal), !,
   vars_to_be_dereffed(Goal, Form, Ds),
   bindset(Goal, Form, Bs),
   unionv(Ds, Bs, Vs).
vars_to_rename(Goal, Form, Vs) :-
   \+survive(Goal), !,
   vars_to_be_dereffed(Goal, Form, Vs).

vars_to_be_dereffed(Goal, Form, Ds) :-
   require(Goal, Req),
   split_deref(Req, DReq),
   varset(DReq, Rs),
   not_deref(Rs, Form, Ds).

not_deref([], _, []) :- !.
not_deref([X|Xs], F, [X|Ns]) :- \+implies(F, deref(X)), !, not_deref(Xs, F, Ns).
not_deref([_|Xs], F, Ns) :- not_deref(Xs, F, Ns).

% Update the mapping f1:IF->IR to f2:OF->OR with a new mapping f3:Q->Qp.
% I.e. if X maps to Xold in f1, and f3 maps X to Xp,
% then f2 will also map X to Xp.
% Also returns the mapping f4:Q->Qold.
update_mapping(Q, Qp, Qold, IF, OF, IR, OR) :-
   map_terms(IF, IR, Q, Qold),
   map_terms(Qold, Qp, IR, MR),
   diffv(Q, IF, QOUT),
   map_terms(Q, Qp, QOUT, QOUTp),
   append(QOUTp, MR, OR),
   append(QOUT, IF, OF).

% Insert a move or deref instruction to initialize the extra vars:
init_extra_vars([], [], []) -->> !.
init_extra_vars([X|Q], [Xp|Qp], [Xold|Qold]) -->>
   init_extra_var(X, Xp, Xold),
   init_extra_vars(Q, Qp, Qold).

init_extra_var(X, Xp, Xold) -->>
   Form/form,
   {implies(Form, deref(X))}, !,
   [Xp]:sf,
   [pref,Xold,Xp]:vl,
   [move(Xold,Xp)]:code.
init_extra_var(X, Xp, Xold) -->>
   [Xp]:sf,
   [pref,Xold,Xp]:vl,
   [deref(Xold,Xp)]:code,
   [deref(X)]:form.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Find all the global permanent variables:
% (Note that this work is repeated in a more refined way by clause_allocate
% which will in addition find the local permanents and the extra vars which
% are permanent.)

find_gperms(Cl, Perms) :-
   split(Cl, Head, Body),
        varset(Head, HVars),
   gpermvars(Body, HVars, Vars, [], Half, [], Perms).

gpermvars(true) -->> !.
gpermvars((Goal,Body)) -->>
   gpermstep(Goal),
   gpermsurv(Goal),
        gpermvars(Body).

% Step over a single goal in the permanent variable calculation:
gpermstep(Goal) -->>
   {varset(Goal, GVars)},
   Half/half,
        {intersectv(GVars, Half, P)},
   [P]:perms,
   [GVars]:vars.

gpermsurv(Goal) -->> {survive(Goal)}, !.
gpermsurv(Goal) -->> {\+survive(Goal)}, !, Vars/vars, [Vars]:half.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Replace a call by a more efficient version of the call:
% The modal_entry/2 predicate is entered as a compiler directive.
% It defines a tree of specialized entry points for any goal.
% This is used to replace calls to builtins by faster versions,
% depending on the input mode.

% Used in clause_code:
% This entry point adds uninit modes to Form.
efficient_entry(Goal, EGoal, Form, Form, SF, SF) :-
   modal_entry(Goal, Tree),
   firstuse_uninit(Goal, SF, Form, NewF),
   tree_trav_entry(Tree, NewF, EGoal),
   !,
   comment(['Replaced ',Goal,' by ',EGoal]),
   comment(Goal, ['The mode formula at this point is ',NewF]).
efficient_entry(Goal, Goal, Form, Form, SF, SF).

% Used in flow analyzer:
% This entry point assumes that Form is as general as possible.
efficient_entry(Goal, EGoal, Form) :-
```

```
   modal_entry(Goal, Tree),
   tree_trav_entry(Tree, Form, EGoal),
   !,
   comment(['During analysis, replaced ',Goal,' by ',EGoal]),
   comment(Goal, ['The mode formula at this point is ',Form]).
efficient_entry(Goal, Goal, _).

tree_trav_entry(entry(EGoal), _, EGoal).
tree_trav_entry(mode(F,True,False), Form, EGoal) :-
   (implies(Form, F)
   -> tree_trav_entry(True, Form, EGoal)
   ;  tree_trav_entry(False, Form, EGoal)
   ).

% Convert the first-use singleton variables in a goal to uninit(either,X) modes
% and add them to the mode formula.  The existing uninit modes are unchanged.
% This gives better discrimination.  An uninit(either,X) mode implies both
% uninit(mem) and uninit(reg).
firstuse_uninit(Goal, SF, InF, OutF) :-
   var_args(Goal, Set),
   term_dupset(Goal, Dups),
   diffv(Set, Dups, U1),
   diffv(U1, SF, Uni1),
   uninit_set(InF, Uni2),
   diffv(Uni1, Uni2, Uni),
   make_uninit(Uni, UnF),
   union_formula(UnF, InF, MidF),
   squeeze_conj(MidF, OutF).

make_uninit([], true).
make_uninit([X|Uni], (uninit(either,X),UnF)) :- make_uninit(Uni, UnF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

conditions.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Medium level routines to do formula manipulations:
% Uses definitions from utility.pl and mutex.pl as primitives.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% General type checking:
% Routines for run-time and compile-time type checking.

% *** For testset selection:
% det_test(Test, Ra, InF, FalseLbl) --> rtest_in(Test, Ra, InF, _, FalseLbl).

% *** Compile-time checks:
% ctest(Test, InFormula, Link, Link)
%  Check that Test is satisfied at compile-time.
%  Fails if it is not.
% Arguments:
%       Test = a test that must be satisfied.
%       InFormula = the modes that hold just before the check.

ctest(Test, Formula) --> {ctest(Test, Formula)}.
ctest(Test, Formula, Formula) :- ctest(Test, Formula).
ctest(Test, Formula) :- prolog_implies(Formula, Test).

% Test that V will always be variable or nonvariable at run-time:
nonvartest(V, Formula) :- ctest(nonvar(V), Formula).
vartest(V,    Formula) :- ctest(var(V),    Formula).

% *** Run-time checks:
% rtest_in(Test, Ra, InF, InF, FailLbl, Code, Link)  No output formula
% rtest_1(Test, FailLbl, InF, OutF) One arg. tests only
% rtest(Test, Ra, InF, OutF, FailLbl, Code, Link)    With output formula
% rtest_in_deref(X, Ra, Rb, InF, InF, Code, Link)    No output formula
% rtest_deref(X, Ra, Rb, InF, OutF, Code, Link)      With output formula
% rtest_m(Test, Varlist, InF, OutF) Do multiple argument tests
%  Generate code to ensure that Test is satisfied at run-time.
%  The code will jump to FailLbl at run-time if it is not.
% Arguments:
%       Test    = a test that must be satisfied.
%       Ra      = the register the argument is in.
%       InF     = the modes that hold just before the check.
%       OutF    = the modes that hold after the check.
%  FailLbl = label to jump to if check fails.
%  Varlist = variable usage goal.

% Check that an argument satisfies a condition.
% If it does not, insert operations that will ensure it at run-time.
% The more mode information, the less that must be checked at run-time.
% The argument of the Test must be a variable for checking code to be inserted.
```

```
rtest_in(Test, Ra, InF, FailLbl) --> rtest_in(Test, Ra, InF, _, FailLbl).
rtest_in(Test, Ra, InF, InF, FailLbl) -->
   core_rtest(Test, Ra, FailLbl, InF, _, no_update).

rtests_in(T, Ra, InF, InF, FailLbl) --> {\+T=(_,_)}, !,
   rtest_in(T, Ra, InF, FailLbl).
rtests_in((T,Ts), Ra, InF, InF, FailLbl) --> !,
   rtest_in(T, Ra, InF, FailLbl),
   rtests_in(Ts, Ra, InF, _, FailLbl).

rtest(Test, Ra, InF, OutF, FailLbl) -->
   core_rtest(Test, Ra, FailLbl, InF, OutF, update).

rtest_in_deref(X, Y, [pref,X,Y|L], L, InF, InF) -->
   {write_once},
   {var(X), \+prolog_implies(InF, deref(X))}, !,
   [deref(X,Y)].
rtest_in_deref(X, X, [X|L], L, InF, InF) -->
   {\+write_once}, !,
   [deref(X)].
rtest_in_deref(X, X, L, L, InF, InF) --> [].

rtest_deref(X, Y, [pref,X,Y|L], L, InF, OutF) -->
   {write_once},
   {var(X), \+prolog_implies(InF, deref(X))}, !,
   [deref(X,Y)],
   {update_formula(deref(Y), InF, OutF)}.
rtest_deref(X, X, [X|L], L, InF, OutF) -->
   {\+write_once}, !,
   [deref(X)],
   {update_formula(deref(X), InF, OutF)}.
rtest_deref(X, X, L, L, InF, InF) --> [].

rtests(T, Ra, InF, OutF, FailLbl) --> {\+T=(_,_)}, !,
   rtest(T, Ra, InF, OutF, FailLbl).
rtests((T,Ts), Ra, InF, OutF, FailLbl) --> !,
   rtest(T, Ra, InF, MidF, FailLbl),
   rtests(Ts, Ra, MidF, OutF, FailLbl).

% One argument tests only (and 'true'):
rtest_1(true, _, InF, InF) --> !.
rtest_1(Test, FailLbl, InF, OutF) -->
   {arg(1, Test, A)},
   rtest(Test, A, InF, OutF, FailLbl).

rtests_1(T, FailLbl, InF, OutF) --> {\+T=(_,_)}, !,
   rtest_1(T, FailLbl, InF, OutF).
rtests_1((T,Ts), FailLbl, InF, OutF) --> !,
   rtest_1(T, FailLbl, InF, MidF),
   rtests_1(Ts, FailLbl, MidF, OutF).

% Note: this routine must insert NOTHING for uninitialized variables.
core_rtest(Test, Ra, FailLbl, InF, OutF, UFlag) -->
   {\+prolog_implies(InF, Test)},
   expand_test(Test, A, Ra, InF, FailLbl),
   {var(A)}, !,
   {update_choice(UFlag, Test, InF, OutF)}.
core_rtest(Test, Ra, FailLbl, F, F, _) --> [].

% This routine handles multiple argument tests, when the test's registers
% are the same as its arguments:
rtest_m(Test, Varlist, InF, OutF) -->
   rtest_m_nf(Test, Varlist, InF),
   {update_formula(Test, InF, OutF)}.

rtests_m(Tests, Varlist, InF, OutF) -->
   rtests_m_nf(Tests, Varlist, InF),
   {update_formula(Tests, InF, OutF)}.

% Same as above, but do not update the mode formula:
rtest_m_nf(Test, '$varlist'([]), fail) --> !, [fail].
rtest_m_nf(Test, Varlist,  InF) --> expand(Test, Test, Varlist, InF, _).

rtests_m_nf((T,Tests), (V,Varlist), InF) --> !,
   rtest_m_nf(T, V, InF),
   rtests_m_nf(Tests, Varlist, InF).
rtests_m_nf(T, Varlist, InF) -->
   rtest_m_nf(T, Varlist, InF).

update_choice(no_update, _, InF,  InF) :- !.
update_choice(   update, T, InF, OutF) :- update_formula(T, InF, OutF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Ensure validity of the test at run-time by inserting appropriate action:

% Generates no code if test is already implied:
expand_test(Test, Ra, InF) --> expand_test(Test, Ra, InF, fail).

expand_test(Test, Ra, InF, Lbl) --> {prolog_implies(InF, Test)}, !.
expand_test(Test, Ra, InF, Lbl) --> {prolog_implies(InF, not(Test))}, !,
   [jump(Lbl)].
expand_test(Test, Ra, InF, Lbl) --> expand_test(Test, A, Ra, InF, Lbl).

% expand_test(Test, Arg, Reg, InF, FailLbl, Code, Link)
%  Test is a test of one argument, Arg is the argument of the test,
%  and Reg is the register holding the arg.  InF is the formula that holds
%  on input.
%  Only expands tests that do no binding at run-time.
expand_test(deref(A), A, Ra, _, Lbl) --> !, [deref(Ra,Ra)].
expand_test(trail(A), A, Ra, _, Lbl) --> !, [trail(Ra)].
expand_test(trail_if_var(A), A, Ra, _, Lbl) -->
   {vlsi_plm}, !,
   [trail(Ra)].
expand_test(trail_if_var(A), A, Ra, _, Lbl) -->
   {\+vlsi_plm}, !,
   {tag(var, Tvar)},
   [test(ne,Tvar,Ra,L)],
   [trail(Ra)],
   [label(L)].
expand_test(nil(A), A, Ra, _, Lbl) --> !,
   [equal(Ra,Tatm^[],Lbl)], {tag(atom, Tatm)}.
expand_test(A==B, A, Ra, _, Lbl) --> {atomic(B)}, !,
   {atomic_word(B,W)},
   [equal(Ra,W,Lbl)].
expand_test(B==A, A, Ra, _, Lbl) --> {atomic(B)}, !,
   {atomic_word(B,W)},
   [equal(Ra,W,Lbl)].
% These are handled by the default case:
% expand_test(list(A), A, Ra, _, Lbl) --> !,
%  [test(eq,Tlst,Ra,L)], {tag(cons, Tlst)},
%  [equal(Ra,Tatm^[],Lbl)], {tag(atom, Tatm)},
%  [label(L)].
% expand_test(compound(A), A, Ra, _, Lbl) --> !,
%  [test(eq,Tstr,Ra,L)],   {tag(structure, Tstr)},
%  [test(ne,Tlst,Ra,Lbl)], {tag(cons, Tlst)},
%  [label(L)].
% expand_test(number(A), A, Ra, _, Lbl) --> {split_integer}, !,
%  [test(eq,Tpos,Ra,L)],   {tag(nonnegative, Tpos)},
%  [test(ne,Tneg,Ra,Lbl)], {tag(negative, Tneg)},
%  [label(L)].
% expand_test(integer(A), A, Ra, _, Lbl) --> {split_integer}, !,
%  [test(eq,Tpos,Ra,L)],   {tag(nonnegative, Tpos)},
%  [test(ne,Tneg,Ra,Lbl)], {tag(negative, Tneg)},
%  [label(L)].
expand_test(negative(A), A, Ra, _, Lbl) --> {\+split_integer}, !,
   {arith_test(X>=Y, Ges)},
   [jump(Ges,Ra,0,Lbl)].
   % [cmpi(Lts,Ra,0),jump(false,Lbl)].
expand_test(nonnegative(A), A, Ra, _, Lbl) --> {\+split_integer}, !,
   {arith_test(X<Y, Lts)},
   [jump(Lts,Ra,0,Lbl)].
   % [cmpi(Lts,Ra,0),jump(true,Lbl)].
expand_test(number(A), A, Ra, _, Lbl) --> {\+split_integer}, !,
   [test(ne,Tint,Ra,Lbl)], {tag(integer,Tint)}.
expand_test(Test, A, Ra, _, Lbl) --> {test_tag(Test,A,Tag)}, !,
   [test(ne,Tag,Ra,Lbl)].
expand_test(nonvar(A), A, Ra, _, Lbl) --> !,
   [test(eq,Tvar,Ra,Lbl)], {tag(var, Tvar)}.
expand_test(functor(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), F==’.’, N==2},
   ctest(nonvar(A), InF), !,
   [test(ne,Tlst,Ra,Lbl)], {tag(cons, Tlst)}.
expand_test(functor(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), N>0, (F\==’.’; N\==2)},
   ctest(nonvar(A), InF), !,
   tag(atom, Tatm),
   test_one_tag(structure(A), Ra, InF, Lbl),
   {tag(structure, Tstr)},
   [pragma(tag(Ra,Tstr))],
   {align(K)},
   [pragma(align(Ra,K))],
   [equal([Ra],Tatmˆ(F/N),Lbl)].
expand_test(functor(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), N==0},
   ctest(nonvar(A), InF), !,
   tag(atom, Tatm),
   [equal(Ra,TatmˆF,Lbl)].
expand_test(functor(A,F,N), A, Ra, InF, Lbl) -->
   {number(F), integer(N), N==0},
   ctest(nonvar(A), InF), !,
   [equal(Ra,F,Lbl)].
expand_test('$name_arity'(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), F=='.', N==2}, !,
   [test(ne,Tlst,Ra,Lbl)], {tag(cons, Tlst)}.
expand_test('$name_arity'(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), N>0, (F\=='.'; N\==2)}, !,
   tag(atom, Tatm),
   test_one_tag(structure(A), Ra, InF, Lbl),
   {tag(structure, Tstr)},
   [pragma(tag(Ra,Tstr))],
   {align(K)},
   [pragma(align(Ra,K))],
   [equal([Ra],Tatmˆ(F/N),Lbl)].
expand_test('$name_arity'(A,F,N), A, Ra, InF, Lbl) -->
   {atom(F), integer(N), N==0}, !,
   tag(atom, Tatm),
   [equal(Ra,TatmˆF,Lbl)].
expand_test('$name_arity'(A,F,N), A, Ra, InF, Lbl) -->
   {number(F), integer(N), N==0}, !,
   [equal(Ra,F,Lbl)].
expand_test(float(A), A, Ra, InF, Lbl) --> !,
   [jump(Lbl)]. % There are no floats--it always fails.
expand_test(Test, A, Ra, InF, Lbl) -->
   {test_to_disj(Test, A, Disj)},
   expand_test_disj(Disj, Ra, InF, Lbl).

% Expand a disjunction of tests:
expand_test_disj(Disj, Ra, InF, Lbl) --> expand_test_disj(Disj, Ra, InF,_, Lbl).

expand_test_disj(fail, _, _, End, Lbl) -->
   [jump(Lbl)],
   [label(End)].
expand_test_disj((T;Disj), Ra, InF, End, Lbl) -->
   expand_test(T, Ra, InF, L),
   [jump(End)],
   [label(L)],
   expand_test_disj(Disj, Ra, InF, End, Lbl).

test_one_tag(Test, Ra, InF, Lbl) --> test_one_tag(ne, Test, Ra, InF, Lbl).

test_one_tag(Eq, Test, Ra, InF, Lbl) -->
   tag(Test, Tag),
   {\+ctest(Test, InF)}, !,
   [test(Eq,Tag,Ra,Lbl)].
test_one_tag(Eq, Test, Ra, InF, Lbl) --> [].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Expansion of some calls to inline instructions:
% expand(SGoal, OGoal, VGoal, InF, OutF, Code, Link)
% +SGoal = source version of goal.
% +OGoal = object version of goal.  Has registers & atomic values as args.
% -VGoal = varlist version of goal. Used in register allocation.
% +InF   = mode formula used as input.
% -Code, +Link = code generated.

% Consistency of this predicate with info in tables.pl must be maintained.

acc_info(code, I, Out,   In, Out=[I|In]).
acc_info(form, F, InF, OutF, update_formula(F,InF,OutF)).

pred_info(expand, 3, [form,code]).

expand(true, true, true) -->> [].
% Arithmetic comparisons:
expand(STest, OTest, '$varlist'([A,B])) -->>
        {arith_test(OTest, A, B, Cond)}, !,
        arith_comparison(Cond, A, B, A, B, false, fail):code.
% Comparison of two atomic terms:
expand(G, '$equal'(A,B), '$equal'(A,B)) -->> [equal(A,B,fail)]:code.
% Type checking goal:
expand(STest, OTest, '$varlist'([Ra])) -->>
   {arg(1,OTest,Ra)},
   InF/form,
   expand_test(STest, Ra, InF):code, !.
expand(repeat, repeat, '$varlist'([])) -->> [choice(1/2,[],L),label(L)]:code.
% These don't need to dereference (because of the way cut is implemented).
expand(_,   '$cut_load'(X), '$varlist'([X])) -->> [move(r(b),X)]:code.
expand(_,   '$cut_deep'(X), '$varlist'([X])) -->> [cut(X)]:code.
expand(_,'$cut_shallow'(X), '$varlist'([X])) -->> [cut(X)]:code.
% Arithmetic instructions:
expand(G,    '$add'(A,B,C),     '$add'(A,B,C)) -->> [add(A,B,C)]:code.
expand(G,    '$sub'(A,B,C),     '$sub'(A,B,C)) -->> [sub(A,B,C)]:code.
expand(G,    '$mod'(A,B,C),     '$mod'(A,B,C)) -->> [mod(A,B,C)]:code.
expand(G,    '$mul'(A,B,C),     '$mul'(A,B,C)) -->> [mul(A,B,C)]:code.
expand(G,    '$div'(A,B,C),     '$div'(A,B,C)) -->> [div(A,B,C)]:code.
expand(G,    '$and'(A,B,C),     '$and'(A,B,C)) -->> [and(A,B,C)]:code.
expand(G,     '$or'(A,B,C),      '$or'(A,B,C)) -->>  [or(A,B,C)]:code.
expand(G,    '$xor'(A,B,C),     '$xor'(A,B,C)) -->> [xor(A,B,C)]:code.
expand(G,    '$sll'(A,B,C),     '$sll'(A,B,C)) -->> [sll(A,B,C)]:code.
expand(G,    '$sra'(A,B,C),     '$sra'(A,B,C)) -->> [sra(A,B,C)]:code.
expand(G,      '$not'(A,C),       '$not'(A,C)) -->> [not(A,C)]:code.
expand(G,             fail,              fail) -->> [fail]:code.
expand(A==B,        Ra==Rb, '$varlist'([Ra,Rb])) -->> InF/form,
   {implies(InF, (simple(A);simple(B)))}, !,
   [equal(Ra,Rb,fail)]:code.
expand(A\==B,      Ra\==Rb, '$varlist'([Ra,Rb])) -->> InF/form,
   {implies(InF, (simple(A);simple(B)))}, !,
   [equal(Ra,Rb,L),fail,label(L)]:code.
```

```
% *** Generating code for arithmetic comparisons:
arith_comparison(Cond, A, B, Ra, Rb, True) -->
        arith_comparison(Cond, A, B, Ra, Rb, true, True).

arith_comparison(Cond, A, B, Ra, Rb, true, TorFLbl) --> !,
   [jump(Cond,Ra,Rb,TorFLbl)].
arith_comparison(Cond, A, B, Ra, Rb, false, TorFLbl) --> !,
   {cond(Cond,Opp)},
   [jump(Opp,Ra,Rb,TorFLbl)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Goal subsumption:

% Two entry points are defined:
% subsume(P, G, SG)
%    This entry is used for subsumption of Prolog source code.
%    Given a logical formula P and a goal sequence G, simplify G assuming P is
%    true.  All goals implied by P and left mutex with P are removed.
%    Prolog semantics are followed: if G does any 'work' then it is not removed.
%    This predicate is currently too slow when P or G are long sequences, it
%    should be max O(n^2).
% logical_subsume(P, G, SG)
%    This entry is used for subsumption of a logical formula.
%    Given two logical formulas P and G, simplify G assuming P is true.
%    Logical semantics are followed: 'work' has no meaning here.
%    Just the logical meaning of a term is important.
%    All goals implied by P and after mutex with P are removed.

% Note:
% 0. This routine is slow.
% 1. Later this routine will be extended to handle user-defined goals and goals
%    with side effects.  These goals may only be removed if they don't do any
%    work that is not done already by P.
%    Currently, the only case that is handled is bindings with unify goals.
% 2. By 'work' I mean bindings and side-effects.  This definition must be made
%    more precise.

subsume(true, G, SG) :- !, simplify(G, SG).
subsume(   P, G, SG) :- sub(P, G, X, prolog), simplify(X, SG).

logical_subsume(true, G, SG) :- !, logical_simplify(G, SG).
logical_subsume(   P, G, SG) :- sub(P, G, X, logical), logical_simplify(X, SG).

subsume_list(_, [], []).
subsume_list(P, [Cl|Cls], [(H:-SB)|SCls]) :-
   split(Cl, H, B),
   subsume(P, B, SB),
   subsume_list(P, Cls, SCls).

% Subsume, but keep disjunction structure and don't simplify:
subdisj(P, G, DG) :-
   sub(P, G, X, prolog), simplify(X, Y), (disj_p(Y) -> DG=Y; DG=(Y;fail)).
sub(P, '$case'(N,I,B),     '$case'(N,I,S), L)        :- !, sub(P, B, S, L).
sub(P, '$test'(D,R,T,A,_), '$test'(D,R,T,A,fail), prolog) :-
   mutex(P, T, left, prolog), !.
sub(P, '$test'(D,R,T,A,_), '$test'(D,R,T,A,fail), logical) :-
   mutex(P, T, after, logical), !.
sub(P, '$test'(D,R,T,A,B), '$test'(D,R,T,A,S), L)    :- !, sub(P, B, S, L).
sub(P, '$else'(R,A,B),     '$else'(R,A,S), L)        :- !, sub(P, B, S, L).
sub(P,      (A,B),  (SA,SB), L) :- !, sub(P, A, SA, L), sub(P, B, SB, L).
sub(P,      (A;B),  (SA;SB), L) :- !, sub(P, A, SA, L), sub(P, B, SB, L).
sub(P,     (A->B), (SA->SB), L) :- !, sub(P, A, SA, L), sub(P, B, SB, L).
sub(P,     (A=>B), (SA=>SB), L) :- !, sub(P, A, SA, L), sub(P, B, SB, L).
sub(P,      \+(A),   \+(SA), L) :- !, sub(P, A, SA, L).
sub(P,     not(A),  not(SA), L) :- !, sub(P, A, SA, L).
sub(P, G, SG, L) :- sub_goal(P, G, SG, L).

% Subsumption of a single goal:
% Need the bindbag test because we don't want to remove G if it does work.
sub_goal(P, G, fail,  prolog) :- mutex(P, G, left, prolog), !.
sub_goal(P, G, fail, logical) :- mutex(P, G, after, logical), !.
sub_goal(P, G, true,  prolog) :- prolog_subsume_work(P, G, left),
   bindbag(G, P, []), !.
sub_goal(P, G, true, logical) :- subsume_work(P, G, after), !.
sub_goal(P, G,    G, _).

% Succeeds if the work P does subsumes the work the goal G does:
% (This predicate will be extended later.)
subsume_work(P, G, _) :- P==G, !.
subsume_work(P, G, Time) :- implies(P, G, Time).

prolog_subsume_work(P, G, _) :- P==G, !.
prolog_subsume_work(P, G, Time) :- prolog_implies(P, G, Time).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Else goals: else(P, G, EG)

% Keep the goals that are not mutex with not(P):
% Used in calculating the remaining goals for the else part of
% a deterministic case statement.

else(P, G, SG) :-
   simplify(not(P), NP),
   subsume(NP, G, SG).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Simplification of formulas:

% Two entry points are defined:
% simplify(In, Out)
%    Simplifies a Prolog formula, keeping with the semantics allowed by the
%    compiler options (default = standard sequential Prolog).  This takes
%    into account the number of solutions, binding, and side effects.
%    Keep predicate unchanged if to could lead to run-time error.
%    Number of solutions can be relaxed through a compiler option.
% logical_simplify(In, Out)
%    Simplifies a logical formula, to keep the same logical meaning.  Does not
%    follow Prolog semantics.
% Both predicates recognize built-ins that can be evaluated at compile-time.

% *** Main definitions:
simplify(A, SA)         :- simp_upv(A, SA, prolog), !.
logical_simplify(A, SA) :- simp_upv(A, SA, logical), !.
   % write('Simplifying '),write(A),write(' to '),write(SA), nl,!.

% *** Simplify and keep disjunction structure:
simpdisj(A, DSA) :- simplify(A, SA), (disj_p(SA) -> DSA=SA; DSA=(SA;fail)).

simp_upv(X,      SX, L) :- nonvar(X), !, simp_up(X, SX, L).
simp_upv(X, call(X), _) :-    var(X), !.

simp_up('$case'(N,I,C),    S,L):- !,
   simp_upv(C,B,L),simp_one('$case'(N,I,B),S,L).
simp_up('$test'(D,P,T,A,C),S,L):- !,
   simp_upv(C,B,L),simp_one('$test'(D,P,T,A,B),S,L).
simp_up('$else'(P,A,C),    S,L):- !,
   simp_upv(C,B,L),simp_one('$else'(P,A,B),S,L).
simp_up( (A,B), S, L) :- !,
   simp_upv(A,SA,L),simp_upv(B,SB,L),simp_one((SA,SB),S,L).
simp_up( (A;B), S, L) :- !,
   simp_upv(A,SA,L),simp_upv(B,SB,L),simp_one((SA;SB),S,L).
simp_up((A->B), S, L) :- !,
   simp_upv(A,SA,L),simp_upv(B,SB,L),simp_one((SA->SB),S,L).
simp_up((A=>B), S, L) :- !,
   simp_upv(A,SA,L),simp_upv(B,SB,L),simp_one((SA=>SB),S,L).
simp_up( \+(A), S, L) :- !,
   simp_upv(A,SA,L),simp_one( \+(SA),S,L).
simp_up(not(A), S, L) :- !,
   simp_upv(A,SA,L),simp_one(not(SA),S,L).
simp_up(     A, S, L) :- simp_one(A, S, L).

det_control('$case'(_,_,_)).
det_control('$test'(_,_,_,_,_)).
det_control('$else'(_,_,_)).

diff_sol :- \+same_number.

% Single step of simplification:
% Note how standard Prolog semantics is kept through the utilities
% diff_sol, deterministic, no_side_effects, no_bind, succeeds, fails, errs.
simp_one(not(not(G)), G, _).
simp_one(not(A=B), A\=B, logical).
simp_one( \+(A=B), A\=B, logical).
simp_one(not(G), NG, _) :- opposite(G, NG).
simp_one( \+(G), NG, _) :- opposite(G, NG).
simp_one((A;B), A, prolog) :- A==B, diff_sol, no_side_effects(A).
simp_one((A;B), A, logical) :- A==B.
simp_one((A,B), A, prolog) :- A==B, deterministic(A), no_side_effects(A).
simp_one((A,B), A, prolog) :- A==B, diff_sol, no_side_effects(A).
simp_one((A,B), A, logical) :- A==B.
simp_one((true,A), A, _).
simp_one((A,true), A, _).
simp_one((true;A), true, prolog) :- diff_sol, no_side_effects(A), no_bind(A).
simp_one((true;A), true, logical).
simp_one((A;true), true, prolog) :- diff_sol, no_side_effects(A), no_bind(A).
simp_one((A;true), true, logical).
simp_one((A,fail), fail, prolog) :- no_side_effects(A).
simp_one((A,fail), fail, logical).
simp_one((fail,_), fail, _).
simp_one((fail;A), A, _).
simp_one((A;fail), A, _).
simp_one(('$cut_shallow'(X),B;_), ('$cut_shallow'(X),B), prolog).
simp_one(   ('$cut_deep'(X),B;_),    ('$cut_deep'(X),B), prolog).
simp_one(('$cut_shallow'(X);_), ('$cut_shallow'(X)), prolog).
simp_one(   ('$cut_deep'(X);_),    ('$cut_deep'(X)), prolog).
% Must never confuse implies with prolog_implies for this to be correct:
simp_one('$cut_shallow'(_), true, logical).
simp_one(   '$cut_deep'(_), true, logical).
simp_one((true -> A;_), A, _).
simp_one((fail -> _;B), B, _).
simp_one((A -> true;B),  A, prolog) :- deterministic(A), succeeds(A).
simp_one((A -> true;B),  A, logical) :- succeeds(A).
simp_one((A -> fail;B), fail, prolog) :- no_side_effects(A), succeeds(A).
simp_one((A -> fail;B), fail, logical) :- succeeds(A).
simp_one((A -> fail;B), B, prolog) :- no_side_effects(A), fails(A).
simp_one((A -> fail;B), B, logical) :- fails(A).
simp_one((true => A;_), A, _).
simp_one((fail => _;B), B, _).
simp_one((true => A), A, _).
simp_one((fail => _), true, _).
simp_one((true => fail), fail, _).
simp_one(not((A;B)), (not(A),not(B)), _).
simp_one(Goal, fail, prolog) :- fails(Goal), no_side_effects(Goal).
simp_one(Goal, fail, logical) :- fails(Goal).
simp_one(A, true, prolog) :- diff_sol,succeeds(A),no_side_effects(A),no_bind(A).
simp_one(A, true, logical) :- succeeds(A).
simp_one(A, true, prolog) :- deterministic(A), succeeds(A), no_bind(A).
simp_one(A, true, logical) :- succeeds(A).
simp_one(deref(X), true, _) :- nonvar(X).
simp_one('$case'(_,_,true), true, _).
simp_one('$case'(_,_,fail), fail, _).
simp_one('$test'(_,_,_,_,fail), fail, _).
simp_one(A==Atm, '$name_arity'(A,Atm,0), prolog) :- var(A), atomic(Atm).
simp_one(Atm==A, '$name_arity'(A,Atm,0), prolog) :- var(A), atomic(Atm).
simp_one(functor(A,Na,Ar),  (A=Na), prolog) :- atomic(Na), integer(Ar), Ar=:=0.
simp_one(functor(A,Na,Ar), (A=Str), prolog) :- atom(Na), integer(Ar), Ar>=0,
        compile_option(functor_limit(N)), Ar=<N, !,
        functor(Str, Na, Ar).
simp_one(T1, T2, _) :- bitmap_simplify(T1, T2).
simp_one(A=B, true, _) :- A==B.
simp_one(G, G, _). % Need an 'else' notation for default case.
```

```
% Incorrect or ambiguous transformations:
% simp_one( \+((A;B)), (\+(A), \+(B))) :- Only if it gets simpler.
% Only true for the last choice:
% simp_one((true -> A), A).
% Not true; -> removes other choices:
% simp_one((A -> B), (A,B)) :- deterministic(A).
% Is ’true’ instead of ’fail’ for a purely logical implication:
% simp_one((A -> _), fail) :- fails(A).
% simp_one((not(A),not(B)), not((A;B))).

% Succeeds if the Prolog formula fails at run-time:
fails(fail).
fails((A,B)) :- fails(A).
fails((A,B)) :- no_side_effects(A), fails(B).
fails((A;B)) :- fails(A), fails(B).
fails(Test) :-
   encode_test(Test, F1, _, A),
   type_flags(A, F2, _),
   0 is F1/\F2,
   \+errs(Test).
fails(Rel) :-
   encode_relop(Rel, A, Ord, B),
   negate_relop(Ord, OppOrd),
   test_relop(A, OppOrd, B),
   \+errs(Rel).

% Succeeds if the predicate succeeds at run-time:
% Assumes NO bindings done by the tests.
succeeds(true).
succeeds((A,B)) :- succeeds(A), succeeds(B).
succeeds((A;B)) :- succeeds(A).
succeeds((A;B)) :- succeeds(B).
succeeds(functor(F,N,A)) :-
   nonvar(F), nonvar(N), nonvar(A),
   \+errs(functor(F,N,A)), functor(F,N,A).
succeeds(’$name_arity’(F,N,A)) :-
   nonvar(F), nonvar(N), nonvar(A),
   \+errs(functor(F,N,A)), functor(F,N,A).
succeeds(nonvar(A))      :- nonvar(A).
succeeds(ground(A))      :- ground(A).
succeeds(atom(A))        :- atom(A).
succeeds(nil(A))         :- nil(A).
succeeds(integer(A))     :- integer(A).
succeeds(negative(A))    :- negative(A).
succeeds(nonnegative(A)) :- nonnegative(A).
succeeds(number(A))      :- number(A).
succeeds(atomic(A))      :- atomic(A).
succeeds(list(A))        :- list(A).
succeeds(cons(A))        :- cons(A).
succeeds(structure(A))   :- structure(A).
succeeds(compound(A))    :- compound(A).
succeeds(composite(A))   :- compound(A).
succeeds(simple(A))      :- nonvar(A), simple(A).
succeeds(Rel)            :-
   encode_relop(Rel, A, Ord, B),
   test_relop(A, Ord, B),
   \+errs(Rel).

% Succeeds if the Prolog formula could give an error at run-time:
errs(Test) :-
   encode_relop(Test, A, _, B, arith),
   ( atom(A) ; atom(B) ; compound(A) ; compound(B) ).
errs(functor(_,_,X)) :-
   ( atom(X) ; compound(X) ).
errs(arg(X,_,_)) :-
   ( atom(X) ; compound(X) ).

opposite(X, Y) :- opp(X, Y), !.
opposite(X, Y) :- opp(Y, X), !.
opposite(X, Y) :- exact_bitmap(X), not_test(X, Y).

% If adding any tests that bind here then must change simp_one:
opp(true, fail).
opp(A<B, A>=B).
opp(A>B, A=<B).
opp(A=:=B, A=\=B).
opp(A@<B, A@>=B).
opp(A@>B, A@=<B).
opp(A==B, A\==B).
opp(var(A), nonvar(A)).

% Succeeds if the formula has no side-effects and has no cut:
% (Only for combinations of tests now, later for user-def preds
%  when flow analysis determines it.)
no_side_effects(true) :- !.
no_side_effects(fail) :- !.
no_side_effects((A,B)) :- !, no_side_effects(A), no_side_effects(B).
no_side_effects((A;B)) :- !, no_side_effects(A), no_side_effects(B).
no_side_effects(A) :- test(A).

% Succeeds if the formula is deterministic, i.e. it has zero or one
% solutions, no more:
% (Only for combinations of tests now, later for user-def preds
%  when flow analysis determines it.)
deterministic(true) :- !.
deterministic((A,B)) :- !, deterministic(A), deterministic(B).
deterministic(A) :- test(A).

% Succeeds if the formula does not bind any variables:
no_bind(true) :- !.
no_bind((A,B)) :- !, no_bind(A), no_bind(B).
no_bind((A;B)) :- !, no_bind(A), no_bind(B).
no_bind(A) :- \+binds(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Mode formula abstract data type:
```

```
% (See also the conjunction handling routines in utility.pl)

% *** Add a goal to the mode formula:
% Attempts to keep size of mode formulas small.
% Nonmonotonic reasoning: Removes var(X) if the added goal implies nonvar(X).
% (Somewhat defaulty.  Let’s see if the compiler can handle this too!)
% Note: if the goal being added is independent of the goals existing in the
% mode formula, then use combine_formula(G, InF, OutF) instead.  It is faster
% and it will not change the existing formula.
% Improvements for later:
% - When traversing a conjunction to be added, SoFar should be updated
%   if it exists.
% - updating with a mixture of var and uninit needs looking at.
update_formula(G, InF, OutF) --> {update_formula(G, InF, OutF)}.

% Generic update_formula:
% (Flag is connected with aliasing; see split_formula)
% For later: Also convert a test to bitmap representation if possible
% update_formula(G, InF, OutF) :-
%  exact_bitmap(G), !,
%  bitmap_test(G, GB),
%  xupdate_formula(GB, InF, yes, _, OutF).
update_formula(G, InF, OutF) :-
   xupdate_formula(G, prolog, InF, yes, _, OutF).

logical_update_formula(G, InF, OutF) :-
   xupdate_formula(G, logical, InF, yes, _, OutF).

% If SoFar is passed too:
update_formula(G, SoFar, InF, OutF) :-
   xupdate_formula(G, prolog, InF, _, SoFar, OutF).

xupdate_formula(fail, _, _, _, _, fail) :- !.
xupdate_formula(_, _, fail, _, _, fail) :- !.
xupdate_formula(G, _, true, _, _, G) :- !.
xupdate_formula(true, _, G, _, _, G) :- !.
xupdate_formula(var(X), _, InF, _, _, fail) :- implies(InF, nonvar(X)), !.
xupdate_formula(var(X), _, InF, _, _, (var(X),InF)) :- !.
xupdate_formula(G, _, InF, _, _, InF) :- memberv_conj(G, InF), !.
xupdate_formula(deref(X), _, InF, _, _, InF) :- memberv_conj(rderef(X), InF), !.
xupdate_formula(deref(X), _, InF, _, _, (deref(X),InF)) :- !.
xupdate_formula(rderef(X), _, InF, _, _, (rderef(X),MidF)) :- !,
   remove_formula(deref(X), InF, MidF).
xupdate_formula((A,B), T, InF, Fl, _, OutF) :-
   nonvar(Fl), !,
   xupdate_formula(A, T, InF, Fl, _, MidF),
   xupdate_formula(B, T, MidF, Fl, _, OutF).
xupdate_formula((A,B), T, InF, Fl, SF, OutF) :-
   var(Fl), !,
   xupdate_formula(A, T, InF, _, SF, MidF),
   xupdate_formula(B, T, MidF, _, SF, OutF).
xupdate_formula(G, T, InF, Flag, SoFar, OutF) :-
   ( T=logical ->
     ( mutex(G, InF, after) ->
       OutF=fail
     ; (bitmap_combine(G, InF, OutF) -> true ; OutF=(G,InF))
     )
   ; T=prolog ->
     ( var(Flag) -> extended_bindset(G, InF, SoFar, Bs)
     ; nonvar(Flag) -> extended_bindset(G, InF, Bs)
     ),
     ( Bs=[] ->
       ( mutex(G, InF, left, prolog) ->
         OutF=fail
       ; (bitmap_combine(G, InF, OutF) -> true ; OutF=(G,InF))
       )
     ; cons(Bs) ->
       split_formula(Flag, SoFar, Bs, InF, SplitF, RestF),
       remove_vars(SplitF, MidF),
       ( mutex(G, MidF, left, prolog) ->
         OutF=fail
       ; combine_formula((G,MidF), RestF, OutF)
       )
     ; Bs=all ->
       remove_vars(InF, MidF),
       ( mutex(G, MidF, left, prolog) ->
         OutF=fail
       ; OutF=(G,MidF)
       )
     )
   ).

% *** Handling of variable binding and aliasing:
% Because of possible aliasing, must remove ALL var(X) goals when variables
% are bound.  The design of split_formula, below, reduces the severity of
% this operation somewhat.
% In the future, anti-aliasing information from flow analysis can reduce
% the severity of this operation even more.

% Remove all var(X) goals and some deref(X) goals from a formula.
% 1. Rderef(X) is removed only if it’s not known that X is ground.
% 2. Deref(X) is removed only if it’s not known that X is nonvariable.
% 3. Rderef(X) is changed to deref(X) if X is known nonvar & not known ground.
% 4. Uninit modes are NOT affected by aliasing.  They are guaranteed not to be
%    aliased to anything.
remove_vars(InF, OutF) :-
   remove_all_vars(InF, MidF),
   notnonvar_ground_formula(InF, NotNV, NotGnd),
   remove_all_derefs(MidF, MidF2, NotNV, NotGnd),
   squeeze_conj(MidF2, OutF), !.

remove_all_vars((A,B), (RA,RB)) :- !,
   remove_all_vars(A, RA),
   remove_all_vars(B, RB).
remove_all_vars(var(V), true) :- var(V), !.
remove_all_vars(  Goal, Goal).

remove_all_derefs((A,B), (RA,RB), NNV, NGnd) :- !,
   remove_all_derefs(A, RA, NNV, NGnd),
   remove_all_derefs(B, RB, NNV, NGnd).
remove_all_derefs( deref(V), true,  NNV,    _) :- var(V), inv(V, NNV), !.
remove_all_derefs(rderef(V), Goal,  NNV, NGnd) :- var(V), inv(V, NGnd), !,
   ( inv(V, NNV) -> Goal=true ; Goal=deref(V) ).
remove_all_derefs(     Goal, Goal,    _,    _).

% *** Remove a goal from a mode formula:
remove_formula(U, InF, OutF) :-
   an_uninit_mode(U, Type, X), !,
   remove_uninit(Type, [X], InF, OutF).
remove_formula(T, InF, OutF) :-
   pred_exists(T, InF), !,
   remove_form(T, InF, MidF),
   flat_conj(MidF, OutF).
remove_formula(T, InF, InF).

remove_form(T, (A,B), (RA,RB)) :- !,remove_form(T, A, RA),remove_form(T, B, RB).
remove_form(T,     U,    true) :- T==U, !.
remove_form(T,     U,       U) :- !.

% *** Split all goals containing X from a mode formula:
split_from_formula(X, (A,B), (RA,RB), (SA,SB)) :- !,
   split_from_formula(X, A, RA, SA),
   split_from_formula(X, B, RB, SB).
split_from_formula(X, G, true,    G) :- varbag(G, Bag), memberv(X, Bag), !.
split_from_formula(X, G,    G, true) :- !.

% *** Succeeds if predicate T exists in a formula:
pred_exists(T, (A,B)) :- pred_exists(T, A), !.
pred_exists(T, (A,B)) :- pred_exists(T, B), !.
pred_exists(T, U) :- T==U, !.

% *** Combine two mode formulas:
% See the definition of unionv_conj:
combine_formula(F1, F2, OutF) :- unionv_conj(F1, F2, OutF).
% combine_formula(F1, F2, OutF) :- simplify((F1,F2), F), flat_conj(F, OutF).

% *** Intersecting two mode formulas:
% Note: this routine does not know that uninit modes have different forms.
intersect_formula(F1, F2, OutF) :- intersect_formula(F1, F2, OutF, true).

intersect_formula((C1,C2), Conj) --> !,
        intersect_formula(C1, Conj),
        intersect_formula(C2, Conj).
intersect_formula(A, Conj) --> {memberv_conj(A, Conj)}, !, co(A).
intersect_formula(A, Conj) --> [].

% *** Intersect a list of formulas:
intersect_formula_list([F], F) :- !.
intersect_formula_list([F1|Fs], F) :- cons(Fs), !,
        intersect_formula_list(Fs, F2),
        intersect_formula(F1, F2, F).
```

```
% *** Union of two mode formulas:
union_formula(F1, F2, OutF) :- unionv_conj(F1, F2, OutF).

% *** Split the deref(X) modes from a formula:
split_deref(    (A,B),   (AD,BD)) :- !, split_deref(A, AD), split_deref(B, BD).
split_deref( deref(X),  deref(X)) :- !.
split_deref(rderef(X), rderef(X)) :- !.
split_deref(        G,      true).

split_deref(    (A,B),   (AD,BD), (AG,BG)) :- !,
   split_deref(A, AD, AG),
   split_deref(B, BD, BG).
split_deref( deref(X),  deref(X), true) :- !.
split_deref(rderef(X), rderef(X), true) :- !.
split_deref(        G,      true,    G).

% *** Trim the mode formula according to a term: (NEEDS A FIX)
% Remove all modes which do not have at least one variable in common
% with a given term.  Exception: keep 'fail' if it's there.
% This should trim the required & before modes too, or else it should be
% done elsewhere, when those modes are stored in the database.
% For correctness, must keep only those uninits that are in the head
% and remove all others:
trim_mode((H:-F), (H:-NF)) :- keep_uninit(H, F, NF).
% This version is far too strong; it removes useful modes!
% trim_mode((H:-M), (H:-TM)) :- split_formula(yes, _, H, M, TM, _).

% *** Split a mode formula into two parts: One which contains all predicates
% relevant to the variables in Term, and the rest.
% Need to follow unification terms, hence the 'vars_in_unify' predicate
% and its implementation through graph closure.  The important variables
% are those which have been initialized (in SF) AND are related textually
% (through unification terms).

% Something like this for later?
% split_formula_goal(Flag, SoFar, Goal, InF, OutF, RestF) :-
%   varset(Goal, Vars),
%   ground_set(InF, Gnd),
%   uninit_set((ReqF,InF), Uni),
%   diffv(Vars, Gnd, VG),
%   diffv(VG, Uni, VGU),
%   split_formula(Flag, SoFar, VGU, InF, OutF, RestF).

split_formula(SoFar, Term, InF, OutF, RestF) :-
   split_formula(_, SoFar, Term, InF, OutF, RestF).

split_formula(Flag, SoFar, Term, InF, OutF, RestF) :-
   % varset(Term, Vars),
   % intersectv(Vars, SoFar, AVars),
   % vars_in_unify(AVars, InF, RelVars),
   vars_in_unify(Term, InF, RelVars),
   stats(x,data(Flag,Term)),
   ( var(Flag) ->
     uninit_set(InF, Uni),
     diffv(SoFar, Uni, AliVars),
     aliasing_flag(AliVars, RelVars, Flag)
   ; true
   ),
   notnonvar_ground_formula(InF, NotNV, NotGnd),
   stats(x,data(notnv(NotNV),notgnd(NotGnd))),
   split_form_vars(Flag, NotNV, NotGnd, RelVars, InF, F1, SF1),
   stats(x,data(f1(F1),sf1(SF1))),
   logical_simplify(F1, F2),
   stats(x,data(f2(F2))),
   flat_conj(F2, F3),
   stats(x,data(f3(F3))),
   squeeze_conj(F3, OutF),
   stats(x,6),
   flat_conj(SF1, RestF),
   stats(x,7),
   !.

split_form_vars(Flag, NN, NG, RelVars, (A1,A2), (B1,B2), (C1,C2)) :- !,
   split_form_vars(Flag, NN, NG, RelVars, A1, B1, C1),
   split_form_vars(Flag, NN, NG, RelVars, A2, B2, C2).
split_form_vars(Flag, NN, NG, RelVars, A, B, C) :-
   split_one(Flag, NN, NG, RelVars, A, B, C).

% Single-goal split:
% - Var(X) is only relevant if there is possible aliasing.
% - Rderef(X) is relevant if X is not implied to be ground.
% - Deref(X) is relevant if X is not implied to be nonvar.
%   If X is implied to be nonvar, then if it’s dereferenced
%   it stays that way, so it doesn’t have to be split off.
%   It is split off anyway, so that its truth will be known for
%   both output formulas.  This is useful in the use of split_formula
%   in flatten.
% - Else if a goal has no variables in common with Vs,
%   then it is not relevant (which is true independent of aliasing).
split_one(_,    _,  _,  _,      true,      true, true) :- !.
split_one(_,    _,  _,  _,      fail,      fail, true) :- !.
split_one(yes,  _,  _,  _,    var(X),    var(X), true) :- !.
split_one(yes, NN,  _,  _,  deref(X),  deref(X), true) :- inv(X, NN), !.
split_one(yes, NN,  _,  _,  deref(X),  deref(X), deref(X)) :- !.
split_one(yes,  _, NG,  _, rderef(X), rderef(X), true) :- inv(X, NG), !.
split_one(yes,  _, NG,  _, rderef(X), rderef(X), rderef(X)) :- !.
split_one(_,    _,  _, Vs,         A,      true,    A) :-
   varset(A, VA), disjointv(VA, Vs), !.
split_one(_,    _,  _,  _,         A,         A, true).

% Get set of all vars NOT implied nonvar:
notnonvar_formula(Form, Vars) :-
   varset(Form, VS),
   nonvar_set(Form, NVSet),
   diffv(VS, NVSet, Vars).

% Get set of all vars NOT implied nonvar & NOT implied ground:
notnonvar_ground_formula(Form, NotNV, NotGnd) :-
```

```
   varset(Form, VS),
   nonvar_set(Form, NV),
   ground_set(Form, Gnd),
   diffv(VS, NV, NotNV),
   diffv(VS, Gnd, NotGnd).

% Get all X in a formula that are implied to be nonvar:
% nonvar_formula((A,B)) --> !, nonvar_formula(A), nonvar_formula(B).
% nonvar_formula(Goal) -->{test_varbag(Goal,TV)},!,implies_nonvar_list(TV,Goal).
% nonvar_formula(Goal) --> [].

% implies_nonvar_list([], Goal) --> !.
% implies_nonvar_list([X|TV], Goal) --> {implies(Goal, nonvar(X))}, !, [X],
%  implies_nonvar_list(TV, Goal).
% implies_nonvar_list([_|TV], Goal) --> implies_nonvar_list(TV, Goal).

% Aliasing flag:
% Aliasing is possible if VarsInitSoFar and RelevantVars have at least
% one variable in common.
% aliasing_flag(SoFar, Rel, F) :- intersectv(SoFar, Rel, X), cons(X), !, F=yes.
aliasing_flag(AliVars, Rel, F) :- disjointv(AliVars, Rel), !, F=no.
aliasing_flag(AliVars, Rel, yes).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Utilities for uninitialized variables:

% Define the notation for uninitialized variables:
an_uninit_mode(U) :- an_uninit_mode(U, _, _).

an_uninit_mode(uninit(V),     mem, V).
an_uninit_mode(uninit_reg(V), reg, V).
an_uninit_mode(uninit(T,V),     T, V) :- mem_reg(T).
an_uninit_mode(uninit(T,V,_),   T, V) :- mem_reg(T).

mem_reg(mem).
mem_reg(reg).

% *** Split a formula into uninit modes & others:
split_uninit((A,B), (AU,BU), (AI,BI)) :- !,
   split_uninit(A, AU, AI),
   split_uninit(B, BU, BI).
split_uninit(G, G, true) :- an_uninit_mode(G), !.
split_uninit(G, true, G).

% *** Split a formula into uninit & deref modes & others:
% Uninit & deref are require modes for external mode input.
split_uninit_deref((A,B), (AU,BU), (AI,BI)) :- !,
   split_uninit_deref(A, AU, AI),
   split_uninit_deref(B, BU, BI).
split_uninit_deref(deref(X), deref(X), true) :- !.
split_uninit_deref(       G,        G, true) :- an_uninit_mode(G), !.
split_uninit_deref(       G,     true,    G).
```

```
% *** Split a formula into var + uninit modes, and others:
split_unbound((A,B), (AV,BV), (AO,BO)) :- !,
   split_unbound(A, AV, AO),
   split_unbound(B, BV, BO).
split_unbound(var(X), var(X), true) :- !.
split_unbound(     G,      G, true) :- an_uninit_mode(G), !.
split_unbound(     G,   true,    G).

% *** Gather uninitialized variables from a conjunction:
uninit_set(Conj, Set) :- uninit_bag(Conj, Bag), sort(Bag, Set).
uninit_bag(Conj, Bag) :- uninit_bag(Conj, Bag, []).

uninit_set_type(T, Conj, Set) :- uninit_bag_type(T, Conj, Bag), sort(Bag, Set).
uninit_bag_type(T, Conj, Bag) :- uninit_bag_type(T, Conj, Bag, []).

uninit_bag((A,B)) --> !, uninit_bag(A), uninit_bag(B).
uninit_bag(G) --> {an_uninit_mode(G, _, V)}, !, [V].
uninit_bag(G) --> [].

uninit_bag_type(T, (A,B)) --> !, uninit_bag_type(T, A), uninit_bag_type(T, B).
uninit_bag_type(T, G) --> {an_uninit_mode(G, T, V)}, !, [V].
uninit_bag_type(T, G) --> [].

% *** Gather unbound variables from a conjunction:
unbound_set(Conj, Set) :- unbound_bag(Conj, Bag), sort(Bag, Set).
unbound_bag(Conj, Bag) :- unbound_bag(Conj, Bag, []).

unbound_bag((A,B)) --> !, unbound_bag(A), unbound_bag(B).
unbound_bag(var(V)) --> !, [V].
unbound_bag(G) --> {an_uninit_mode(G, _, V)}, !, [V].
unbound_bag(G) --> [].

% *** Remove those uninitialized variables from a formula that are in
% the set Init (i.e. those that have been initialized):
remove_uninit(Init, InF, OutF) :-
   remove_uninit(Init, InF, MidF, true),
   flat_conj(MidF, OutF).

remove_uninit(Init, (A,B)) --> !,
        remove_uninit(Init, A),
        remove_uninit(Init, B).
remove_uninit(Init, G) -->
   {an_uninit_mode(G, _, V)},
        {var(V), memberv(V, Init)}, !.
remove_uninit(Init, G) -->
        co(G).

% Same as above, of a given type only:
remove_uninit(Type, Init, InF, OutF) :-
   remove_uninit(Type, Init, InF, MidF, true),
   flat_conj(MidF, OutF).

remove_uninit(Type, Init, (A,B)) --> !,
        remove_uninit(Type, Init, A),
        remove_uninit(Type, Init, B).
remove_uninit(Type, Init, G) -->
   {an_uninit_mode(G, Type, V), memberv(V, Init)}, !.
remove_uninit(Type, Init, G) -->
        co(G).

% *** Keep only the uninitialized variables from a formula
% that are in the term, i.e. remove the ones that are not
% relevant to the term.
keep_uninit(Term, InF, OutF) :-
   varbag(Term, Vars),
   keep_uninit(Vars, InF, MidF, true),
   squeeze_conj(MidF, OutF).

keep_uninit(Vars, (A,B)) --> !,
        keep_uninit(Vars, A),
        keep_uninit(Vars, B).
keep_uninit(Vars, G) -->
   {an_uninit_mode(G, _, V)},
        {var(V), \+memberv(V, Vars)}, !.
keep_uninit(Vars, G) -->
   co(G).

% *** Remove all uninitialized variables from a formula:
remove_all_uninit(InF, OutF) :-
        uninit_set(InF, UnVars),
        remove_uninit(UnVars, InF, OutF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Determine all variables that may be bound to a nonvariable by the
% predicate, given a mode formula.  This version knows about possible aliasing,
% and will return ’all’ instead of a set if other variables in the clause may
% be aliased to the bound variables in the goal.  This version also handles
% special cases of unify goals.
extended_bindset( X=Y, InF, SF, Set) :-
   uninit_set(InF, Uni),
   e_bindset(X, Y, Uni, SF, Set), !.
extended_bindset(Goal, InF, SF, Set) :-
   bindset(Goal, B),
   grounds_in_form(InF, Gnd), diffv(B, Gnd, Set),
   disjointv(Set, SF), !.
extended_bindset(Goal, InF, SF, all).

% *** Weaker version when SF set is not available:
extended_bindset( X=Y, InF, Set) :-
   uninit_set(InF, Uni),
   e_bindset(X, Y, Uni, Set), !.
extended_bindset(Goal, InF, Set) :-
   bindset(Goal, B),
   grounds_in_form(InF, Gnd), diffv(B, Gnd, Set),
   Set=[], !.
extended_bindset(Goal, InF, all).
```

```
e_bindset(X, Y, U, SF, BS) :- is_new(X, U, SF), is_new(Y, U, SF), !, BS=[].
e_bindset(X, Y, U, SF, BS) :- is_new(X, U, SF), !, BS=[X].
e_bindset(X, Y, U, SF, BS) :- is_new(Y, U, SF), !, BS=[Y].

e_bindset(X, Y, U, BS) :- inv(X, U), inv(Y, U), !, BS=[].
e_bindset(X, Y, U, BS) :- inv(X, U), !, BS=[X].
e_bindset(X, Y, U, BS) :- inv(Y, U), !, BS=[Y].

% Succeeds if variable is unbound & unaliased:
is_new(X,   _, SF) :- var(X), \+inv(X, SF), !.
is_new(X, Uni,  _) :- inv(X, Uni), !.

% *** Determine all variables in a predicate that may be bound to a nonvariable
% by the predicate, given a mode formula.
% This only returns the variables in the goal that may be bound.  It does not
% know about other variables (that may be aliased to the goal’s variables),
% that may also be bound.
% It uses the predicate bindbag(Goal, Bag) defined in tables.pl.
bindset( X=Y, InF, BS) :- uninit_set(InF, U),
   (inv(X, U), !, BS=[X] ; inv(Y, U), !, BS=[Y]).
bindset(Goal, InF, BS) :-
   bindset(Goal, B), nonvar_set(InF, NV), diffv(B, NV, BS).

bindbag(Goal, InF, BBag) :- bindset(Goal, InF, BBag).

% *** A more discriminating bindset that also uses the set of existing
% variables: Returns those variables in the goal that are not known not to be
% bound to a nonvariable.  Does not return any variables which are either not
% bound or remain unbound (i.e. being bound to another unbound variable is ok).
bindset( X=Y,   _, SF, BS) :- var(X), \+inv(X, SF), !, BS=[X].
bindset( X=Y,   _, SF, BS) :- var(Y), \+inv(Y, SF), !, BS=[Y].
bindset(Goal, InF,  _, BS) :- bindset(Goal, InF, BS).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

compiler.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

copyright(1,’Aquarius Prolog compiler’).
copyright(2,’Copyright (C) 1989 Peter Van Roy and Regents of the’).
copyright(3,’University of California.  All rights reserved.’).

% This program may be freely used and modified for non-commercial purposes
% provided this notice is kept unchanged and proper credit is given.
% Written by Peter Van Roy in the Aquarius Project.

compiler_version(’Mon Nov 12 14:58:53 PST 1990’).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Top level of the Aquarius compiler

% See the files _INFORMATION, _GLOBAL_CHANGES, and _LOCAL_CHANGES for
% information and details about the internal structure of the compiler.
% The file output_language.pl contains an executable Prolog specification
% of the compiler output’s syntax.

% Operator declarations:
:- op( 700, xfx, [’\=’]).
:- op(1050, xfx, [’=>’]).
:- op( 500, yfx, [and,or,xor]).

% The dynamic predicates: (needed for Quintus only)
% :- dynamic(compile_cputime/3).% See stats in ’utility’.
% :- dynamic(gensym_integer/1).     % See gensym in ’utility’.
:- dynamic(include_stack/1).
:- dynamic(compile_option/1).
:- dynamic(mode_option/5).
:- dynamic(modal_entry/2).
:- dynamic(save_clause/2).   % For piped compiler.
:- dynamic(macro/6).
:- dynamic(select_option/2).     % For evaluation of determinism selection.

% Default compiler options for piped compiler under Quintus Prolog:
% (These defaults are appropriate for the BAM processor)
default_option(flat).         % Flatten switch(unify) (for piped versions).
default_option(write).        % Write out the compiled code.
default_option(source).       % Write source (for examples & interactive).
default_option(comment).      % Write comments (just some information).
default_option(peep).         % Do peephole optimization.
default_option(low_reg(0)).   % Number of lowest register.
default_option(high_reg(1000)).    % Number of highest register (mapped to mem).
default_option(low_perm(0)).      % Number of lowest permanent variable.
default_option(align(2)).     % Align compound terms to this value.
default_option(compile).          % Compile the input.
default_option(factor).           % Do factoring source transformation.
default_option(uni).          % Generate unify_atomic instruction.
default_option(select_limit(1)).  % Depth limit on argument selection.
default_option(hash_size(5)).     % Minimum size of hash table.
default_option(functor_limit(5)). % Arity limit for translation of functor.
default_option(include_limit(8)). % Nesting limit for file inclusion.
default_option(depth_limit(2)).   % Nesting limit for unification goals.
default_option(short_block(6)).   % Threshold on basic block length for shuffle.
default_option(write_once).       % Use write-once permanent variables.
default_option(same_number_solutions). % Retain the same number of solutions.
default_option(same_order_solutions).  % Retain the same order of solutions.
default_option(split_integer).    % Separate tags for pos and neg (as in BAM).
default_option(system(quintus)).  % For Quintus Prolog.
default_option(user_test_size(1,2)). % For segmenting; see segment.pl.
default_option(analyze_uninit_reg).  % Analyze may generate uninit_reg modes.

% Other compiler options:
% mips      Output is for MIPS processor.  This changes testset selection
%               slightly.  This option is enabled by the ’mips’ directive.
% analyze   Perform simple flow analysis for the input file, giving
%       uninit, uninit_reg, ground, rderef & nonvar modes.
% debug     Write debugging messages.
% merge_add Peephole optimization: Merge adda and pad into push.
%       For best results this should be done in the reorderer.
% contiguous    Predicates must be contiguous (disabled).
% stats([t,c,p,s,d])    Stats in top,comp,peephole,selection,proc_code.
% system(cprolog)   Running under C-Prolog

% Ensure that the defaults hold if an option is deleted erroneously:
low_reg(N)      :- compile_option(low_reg(M)), !, N=M.
low_reg(N)      :- default_option(low_reg(M)), !, N=M.
high_reg(N)     :- compile_option(high_reg(M)), !, N=M.
high_reg(N)     :- default_option(high_reg(M)), !, N=M.
low_perm(N)     :- compile_option(low_perm(M)), !, N=M.
low_perm(N)     :- default_option(low_perm(M)), !, N=M.
align(N)        :- compile_option(align(M)), !, N=M.
align(N)        :- default_option(align(M)), !, N=M.
the_system(S)   :- compile_option(system(M)), !, S=M.
the_system(S)   :- default_option(system(M)), !, S=M.
select_limit(L) :- compile_option(select_limit(M)), !, L=M.
select_limit(L) :- default_option(select_limit(M)), !, L=M.
depth_limit(L)  :- compile_option(depth_limit(M)), !, L=M.
depth_limit(L)  :- default_option(depth_limit(M)), !, L=M.

split_integer   :- compile_option(split_integer), !.
merge_add       :- compile_option(merge_add), !.
write_once      :- compile_option(write_once), !.
same_number     :- compile_option(same_number_solutions), !.
same_order      :- compile_option(same_order_solutions), !.
mips            :- compile_option(mips), !.
vlsi_plm        :- compile_option(vlsi_plm), !.

% Write a pragma to inform the translator about the write_once option:
pragma_write_once(write_once)    :- !, w((:-(write_once))),    wn(’.’), nl.
pragma_write_once(_).
```

```
pragma_no_write_once(write_once) :- !, w((:-(no_write_once))), wn('.'), nl.
pragma_no_write_once(_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Piped compiler:

% Create saved state: (Quintus)
qmain :- qmain(compiler).

qmain(C) :- qdefault, creator(C).

% Create saved state: (C-Prolog)
% Note: this must be started up in a system equal in size or larger than
% the one which created it.
cmain(C) :- cdefault, creator(C).

creator(C) :-
   save(C, 1),
   ( pipe
   ; error('Sorry old chap, the compiler has failed.')
   ),
   halt.
creator(_) :-
   halt.

% Piped compiler working loop:
pipe :- prompt(_, ''), print_version, read_expand(Cl), pipe_loop(Cl).

pipe_loop(end_of_file) :-
   pop_incl_stack(F), !, seen, see(F),
   read_expand(Cl), pipe_loop(Cl).
pipe_loop(end_of_file) :- !.
pipe_loop(Cl) :- pipe_loop_one(Cl, NextCl), asserta(save_clause(NextCl)), fail.
pipe_loop(_) :- retract(save_clause(NextCl)), pipe_loop(NextCl).

pipe_loop_one(Cl, NextCl) :-
   collect_clauses(Cl, NextCl, Cls),
   internal_comp(Cls), !.

% Collect all the clauses to be compiled:
% All the way to the end of the file, that is.
collect_clauses(end_of_file, NextCl, Cls) :-
   pop_incl_stack(F), !, seen, see(F),
   read_expand(Cl),
   collect_clauses(Cl, NextCl, Cls).
collect_clauses(end_of_file, end_of_file, []) :- !.
collect_clauses(Cl, NextCl, Cls) :-
   one_step(Cl, MidCl, Cls, Mid),
   collect_clauses(MidCl, NextCl, Mid).

% Handle a single directive or predicate:
% Some directives are collected to be executed later.
one_step((:-D), NextCl, [(:-D)|Link], Link) :- after_collect(D), !,
   read_expand(NextCl).
one_step((:-D), NextCl, Link, Link) :- !,
   handle_dir(D),
   read_expand(NextCl).
one_step(Cl, NextCl, [Cl|Cls], Link) :-
   getname(Cl, NaAr),
   read_clauses(NaAr, NextCl, Cls, Link).

% Read a single predicate from input:
read_clauses(NaAr, NextCl, Cls, Link) :-
        read_expand(Clause),
   getname(Clause, NA),
        (NA=NaAr
   -> Cls=[Clause|Rest],
           read_clauses(NaAr, NextCl, Rest, Link)
   ;  Cls=Link,
           NextCl=Clause
   ), !.

% Handle grammar rules:
read_expand(Clause) :- read(Cl), expand_term(Cl, Clause).

getname(Clause, Name/Arity) :-
   split(Clause, Head, _),
   functor(Head, Name, Arity), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Handle a single directive:

handle_dir(D)     :- var(D), !, warning('Variable directive ignored').
handle_dir((A,B)) :- !, handle_dir(A), handle_dir(B).
handle_dir(D)     :- add_mode_option(D), !.
handle_dir(D)     :- add_entry_option(D), !.
handle_dir(D)     :- add_modal_entry(D), !.
handle_dir(D)     :- add_macro_def(D), !.
handle_dir(D)     :- D=..[DN|DA], handle_dir(DN, DA), !.
handle_dir(D)     :- warning(['Non-existent directive ''',D,''' ignored']).

handle_dir(         gc,  []) :- gc.
handle_dir(       nogc,  []) :- nogc.
handle_dir(    default,  []) :- default.
handle_dir(       mips,  []) :- mips_options.
handle_dir(   vlsi_plm,  []) :- vlsi_plm_options.
handle_dir(      clear,  []) :- clr_mode, clr_modal_entry, clr_entry.
handle_dir(    version,  []) :- print_version.
handle_dir(       help,  []) :- print_help.
handle_dir(printoption,  []) :- po.
handle_dir(         po,  []) :- po.
handle_dir(     option,   L) :- cox_list(L).
handle_dir(          o,   L) :- cox_list(L).
handle_dir(  notoption,   L) :- nox_list(L).
handle_dir(         no,   L) :- nox_list(L).
handle_dir( statistics,  []) :- wn('/*'), statistics, wn('*/').
handle_dir(         ex, [X]) :- ground(X), ex(X).
handle_dir(    include, [F]) :- atom(F), seeing(G), push_incl_stack(G), see(F).
handle_dir(    comment, [X]) :- w('% '), wn(X).
handle_dir(       pass, [X]) :- wq((:-pass(X))), wn('.').
handle_dir(     op, [A,B,C]) :- op(A,B,C), wq((:-op(A,B,C))), wn('.').
handle_dir(       call, [X]) :- nonvar(X), call(X), wn(X).

% All directives listed here are executed after all source transformation
% and dataflow analysis.  They are executed interleaved with compile_stree.
% All directives not listed here are executed immediately upon reading them.
% Note that 'include', 'op', 'entry', and 'mode' must be executed immediately,
% as well as 'option(analyze)', 'notoption(analyze)'.
after_collect((A,B)) :- !, after_collect(A), after_collect(B).
after_collect(D) :-
   nonvar(D), D=..[DN|DA],
   after_collect(DN, DA).

% after_collect(   vlsi_plm,  []).
after_collect(printoption,  []).
after_collect(         po,  []).
after_collect(     option,   L) :- \+ do_it_now(L).
after_collect(          o,   L) :- \+ do_it_now(L).
after_collect(  notoption,   L) :- \+ do_it_now(L).
after_collect(         no,   L) :- \+ do_it_now(L).
after_collect( statistics,  []).
after_collect(    comment, [_]).
after_collect(       pass, [_]).
after_collect(       call, [_]).

% Options in this list are done immediately:
do_it_now(L)   :- now_list(NL), member(X, NL), member(X, L).
do_it_now([L]) :- now_list(NL), member(X, NL), member(X, L).

now_list([nomem,stats(_),analyze,analyze_uninit_reg,compile,comment,uni,
     factor,test,test_unify,test_arith,test_typecheck,firstarg]).

% Set correct options for the MIPS processor:
mips_options :-
   default,
   cox_list([mips,align(1)]),
   nox_list([split_integer]).

% Set correct options for the VLSI-PLM:
vlsi_plm_options :-
   default,
   cox_list([vlsi_plm,high_reg(6),align(1)]),
   nox_list([split_integer]).

% Print version:
print_version :- nl, print_copyright, print_date, nl.

print_copyright :- copyright(_,C), w('% '), wn(C), fail.
print_copyright.
```

```
print_date :- compiler_version(V), !, w('% Creation date '), wn(V).
print_date.

% Print help information:
print_help :-
    nl,
    wn('% List of valid directives:'),
    wn('% default               Set default options and clear all else.'),
    wn('% mips                  Set default options for the MIPS.'),
    wn('% vlsi_plm              Set default options for the VLSI-PLM.'),
    wn('% clear                 Clear all modes, modal entries, and entries.'),
    wn('% option(OptList)       Add the listed options.'),
    wn('% notoption(OptList)    Remove the listed options.'),
    wn('% printoption           Print the current options.'),
    wn('% mode((Head:-Formula))  Mode information for a predicate.'),
    wn('% modal_entry(H,Tree)    Discrimination tree for efficient builtins.'),
    wn('% entry((Head:-Formula)) An entry point for flow analysis.'),
    wn('% include(F)            Insert the contents of file F.'),
    wn('% pass(X)               Pass :-pass(X). unaltered to the output.'),
    wn('% version               Print the creation date of this version.'),
    wn('% statistics            Print compiler execution statistics.'),
    wn('% op(A,B,C)             Operator declaration. Passed to the output.'),
    nl.

% Handle the include stack:
push_incl_stack(File) :-
   include_stack(Files),
   my_retractall(include_stack(_)),
   F = [File|Files],
   asserta(include_stack(F)),
   (length(F, N), compile_option(include_limit(M)), N>M
   -> warning(['The inclusion stack ',F,
           nl, 'has exceeded the limit of ',M,' nested includes.'])
   ;  true
   ).

pop_incl_stack(File) :-
   include_stack([File|Files]),
   my_retractall(include_stack(_)),
   asserta(include_stack(Files)).

include_stack([]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Interactive compiler:

% Compile to a file:
cf(F, RawCls) :- cf(F, RawCls, []).
cf(F, RawCls, M) :- tell(F), comp(RawCls, M), !, told.
cf(_, _, _) :- told.

% Compile to the screen:
% (Modes are added to the global database)
```

```
comp(RawCls) :- comp(RawCls, []).

comp(RawCls, Modes) :-
        make_list(Modes, ML),
        comp(RawCls, ML, Code),
        write_source(RawCls, ML),
        write_code(Code).

comp(RawCls, ModeList, Code) :-
        clr_mode,
        add_mode_options(ModeList),
        internal_comp(RawCls, Code).

% Compile an example predicate to the screen:
ex(Ex) :- ex(Ex, Code, Modes), !, comp(Code, Modes).

% Compile a list of example predicates to a file:
compex(ExampleList, File) :- tell(File), compex(ExampleList), told.

compex([]).
compex([X|Examples]) :- ex(X), fail.
compex([_|Examples]) :- compex(Examples).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Internal compiler:

% Compilation of a set of raw input clauses.
% The clauses may contain non-contiguous procedures.

% Write the compiled output.
internal_comp(RawCls) :- internal_comp(write, RawCls, _).

% Return the compiler output; don’t write it.
internal_comp(RawCls, Code) :- internal_comp(nowrite, RawCls, Code).

internal_comp(_, [], []) :- !.
internal_comp(WFlag, RawCls, Code) :- cons(RawCls), !,
   init_stats,
   cont_cls(RawCls, Cls),
   cls_to_ptrees(Cls, Ptrees),
   comp_steps(WFlag, Ptrees, Code).

comp_steps(WFlag) -->
   {stats(t,1)},
   transform_cut_ptrees,
   {stats(t,2)},
   factor_ptrees,
   {stats(t,3)},
   ptrees_to_strees,
   {stats(t,4)},
   flatten_strees,
   {stats(t,5)},
   inline_strees,
   {stats(t,6)},
   analyze_strees,
   {stats(t,7)},
   compile_strees(WFlag),
   {done_stats}, !.

% Compilation of a set of Strees.
compile_strees(WFlag, Ss, Code) :-
   compile_strees(Ss, peep, return, header, WFlag, Code, []).

compile_strees([], PF, RF, HF, WF) --> !.
compile_strees([S|Ss], PF, RF, HF, WF) --> {S=stree(NaAr,_,_,_,_,_)}, !,
   {compile_comment(NaAr)},
   {print_stree(S)},
   {init_gensym},
   compile_stree(S, PF, RF, HF, WF),
   compile_strees(Ss, PF, RF, HF, WF).
compile_strees([(:-D)|Ss], PF, RF, HF, WF) --> !,
   {handle_dir(D)},
   compile_strees(Ss, PF, RF, HF, WF).

print_stree(S) :- compile_option(print_stree), !,
   write('% '), q_inst_writeq(S), wn('.').
print_stree(_).

compile_comment(NaAr) :- compile_option(compile), !,
   nl, comment(['Compiling ',NaAr,' ...']).
compile_comment(_).

% Compilation of a single procedure in flattened Stree format.
% Flags governing the compilation:
% Peephole flag:
%   PFlag = peep    full peephole optimization is done.
%   PFlag = nopeep  no peephole optimization is done.
% Return flag:
%   RFlag = return  compile return instruction when code completes.
%   RFlag = jump(L) compile jump to end of code when code completes.
% Header flag:
%   HFlag = header  prepend a procedure header to the code.
%   HFlag = noheader    generate no header.
% Write flag:
%   WFlag = write   write the code; return nothing.
%   WFlag = nowrite return the code; don't write it.
compile_stree(Stree, PFlag, RFlag, HFlag, WFlag, Code, Link) :-
   compile_stree(Stree, PFlag, RFlag, HFlag, C),
   (WFlag = write
   -> write_code(C), Code=Link
   ;  difflist(C, Code, Link)
   ).

compile_stree(Stree, PFlag, RFlag, HFlag, Code) :-
   compile_option(compile), !,
        calc_select_depth(Stree),
   stats(x,20),
   builtin_warning(Stree),
   stats(c,1),
   trim_stree(Stree, TrimStree),
   stats(c,2),
   % Run-time selection without modes:
   select_stree(TrimStree, SelStree, BList),
   SelStree = stree(NaAr,(Head:-Disj),Mode,_,DList,_),
   proc_header(HFlag, NaAr, Code, PCode),

   % Procedure compilation:
   stats(c,3),
   segment_disj(Head, Disj, SegDisj, Mode, BodyList, BList),
   stats(c,4),
   selection_disj(Head, SegDisj, Mode, SelDisj),
   stats(c,5),
   proc_code(Head, SelDisj, Mode, DList, RFlag, UnoptCode, C2),

   % Clause compilation:
   stats(c,6),
   clause_code_list(BodyList, DList, C2, C3),

   % Handle RFlag’s destination:
   end_label(RFlag, C3, []),

   % Peephole optimization:
   stats(c,7),
   peephole(NaAr, UnoptCode, PCode, PFlag),
   stats(c,8),
   !.
compile_stree(_, _, _, _, []).

% Calculate selection depth:
% calc_select_depth(stree(_,(_:-D),_,_,_,SD1)) :-
%  var(SD1),
%       select_limit(SD), SD<2,
%  length_disj(D, N), N=:=2, !,
%       SD1 = 2.
calc_select_depth(stree(_,_,_,_,_,SD)) :-
   var(SD), !,
        select_limit(SD).
calc_select_depth(stree(_,_,_,_,_,SD)) :-
   nonvar(SD).

% Give a warning if the predicate being compiled is a builtin:
builtin_warning(stree(N/A,_,_,_,_,_)) :-
   functor(G, N, A),
   (builtin(G)
   -> warning([’The predicate ’,N/A,’ is a builtin.’])
   ;  true
   ).

% Trim the mode of an stree:
trim_stree(stree(NA,HD,M,OH,DL,SD),stree(NA,HD,TM,OH,DL,SD)) :- trim_mode(M,TM).
```

```
% Handle HFlag:
proc_header(noheader,     _) --> !.
proc_header(header,    NaAr) --> [procedure(NaAr)].

% Handle RFlag:
% Dummies have a jump to this label instead of a return.
% end_label(return)    --> [].
end_label(return)    --> [return].
end_label(jump(Lbl)) --> [label(Lbl)].

% 'Heuristic' run-time selection:
% When the mode is weak (e.g. flow analysis isn't always able to do a good job)
% this routine 'enriches' an stree by adding a choice between var(X) and
% nonvar(X).  The number of levels of enrichment is given by select_limit.
% A heuristic that generalizes the WAM's first argument selection is used to
% choose which argument will be enriched.  A select limit of 1 already improves
% on WAM selection.
% Compilation time of an stree goes up rapidly with the selection limit.
% Each choice is routed to a duplicate of the original code.  To do this
% without duplicating the whole predicate the stree's disjunction is first
% naively split into heads and bodies.  Only the heads are duplicated.
% Notes for improvement:
% - Only put in TestDisj those tests that can be used.
select_stree(Stree, SelStree, BodyList) :-
        Stree = stree(Na/Ar, (H:-Disj), (H:-F), OH, DL, SD),
   SD>0,                     % Select only if depth limit not reached
        Ar>0,                     % Select only if arity>0
   length_disj(Disj, N), N>1, % Select only if >1 clauses

   % Find the set of arguments that it's good to select with,
   % i.e. that are arguments of a unification not known to succeed,
   % i.e. in the unification neither argument is known to be unbound.
        segment_all_disj(Disj, H, F, TestDisj, BodyList, GoodVars),

   % Make sure that the number of good variables implied nonvar
   % is less than the depth limit & return the other good variables.
   % If analysis didn't create enough nonvar modes, then selection
   % will create more up to the selection limit.
   % If argument 1 is in NotNV, then it is used in selection, even
   % if that makes the total number of nonvars greater than SL.
   not_nonvar(GoodVars, F, NotNV, NV),
   select_limit(SL),
   ( NV<SL
   ; arg(1, H, A1), inv(A1, NotNV)
   ),

   % Use the lowest numbered good variable that
   % is not implied nonvar or unbound:
   range_option(1, I, Ar),
   arg(I, H, X),
   memberv(X, NotNV),
   !,
   comment(['Doing selection on argument ',I,' of ',Na/Ar]),
   asserta(select_option(H, I)),
        subsume(var(X),    TestDisj, VDisj), standard_disj(VDisj, VarDisj),
        subsume(nonvar(X), TestDisj, NDisj), standard_disj(NDisj, NonDisj),
        new_head("$v_",  H, [], H1), functor(H1, Na1, Ar1),
        new_head("$nv_", H, [], H2), functor(H2, Na2, Ar2),
   % DL is needed in three strees, since each can have calls to nested
   % strees.  This leads to some code duplication, which is reduced by
   % peep_uniq.  Alas, the increase in compilation time remains.
   SD1 is SD-1,
   after(var(X), VarF),
   after(nonvar(X), NonvarF),
   combine_formula(VarF, F, VF),
   combine_formula(NonvarF, F, NF),
        VarStr = stree(Na1/Ar1, (H1:-VarDisj), (H1:-VF), [], DL, SD1),
        NonStr = stree(Na2/Ar2, (H2:-NonDisj), (H2:-NF),[], DL, SD1),
   add_mode_option(H, H1, true, VarF, yes),
   add_mode_option(H, H2, true, NonvarF, yes),
        standard_disj((var(X),H1;nonvar(X),H2), NewDisj),
        SelStree = stree(Na/Ar,(H:-NewDisj),(H:-F),[],[VarStr,NonStr|DL],SD),
   !.
% Default if no selection done:
select_stree(Stree, Stree, []).

% Calculate the subset of the good variables not implied nonvar
% and the number of good variables implied nonvar:
not_nonvar(GoodVars, Form, NotNV, NV) :-
   not_nonvar(GoodVars, Form, NotNV, 0, NV).

not_nonvar([], _, [], N, N).
not_nonvar([X|Vars], Form, NotNV, I, N) :- implies(Form, nonvar(X)), !,
   I1 is I+1,
   not_nonvar(Vars, Form, NotNV, I1, N).
not_nonvar([X|Vars], Form, [X|NotNV], I, N) :-
   not_nonvar(Vars, Form, NotNV, I, N).

% For evaluation of determinism selection:
% Allow all arguments, or allow only first argument:
range_option(_, I, _) :- compile_option(firstarg), !, I=1.
range_option(L, I, H) :- range(L, I, H).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Handle compiler options:

% Add a modal_entry, which contains a discrimination tree of entry points for
% predicates (usually builtins).  This allows more efficient compilation of
% builtins by replacing the expensive general case by a cheaper alternative.
add_modal_entry(modal_entry(Head,Tree)) :- check_modal_entry(Head, Tree), !,
   my_retractall(modal_entry(Head,_)),
   asserta(modal_entry(Head,Tree)).
add_modal_entry(modal_entry(_,_)).

clr_modal_entry :- my_retractall(modal_entry(_,_)).

check_modal_entry(H, T) :-
```

```
   nonvar(H), nonvar(T), check_tree_entry(T), !.
check_modal_entry(H, _) :-
   error(['Modal entry ',H,' has incorrect syntax.']), fail.

check_tree_entry(entry(E)) :- nonvar(E).
check_tree_entry(mode(F,True,False)) :-
   nonvar(F), nonvar(True), nonvar(False),
   check_tree_entry(True),
   check_tree_entry(False).

% Add an external entry point and its mode to the global data base:
% An entry is a mode declaration with the additional task of being
% a starting point for flow analysis.
add_entry_option(entry(H,R,B,A,S)) :-
   add_mode_option(mode(H,R,B,A,S)),
   nox(entry(H,_)),
   translate((R,B), TB),
   flat_conj(TB, FB),
   logical_simplify(FB, SB),
   asserta(compile_option(entry(H,SB))).
add_entry_option(entry((H:-B))) :-
   add_mode_option(mode((H:-B))),
   nox(entry(H,_)),
   translate(B, TB),
   flat_conj(TB, FB),
   logical_simplify(FB, SB),
   asserta(compile_option(entry(H,SB))).
add_entry_option(entry(Name/Arity)) :-
   atom(Name), nonnegative(Arity),
   functor(H, Name, Arity),
   add_mode_option(mode((H:-true))),
   nox(entry(H,_)),
   asserta(compile_option(entry(H,true))).

clr_entry :- my_retractall(compile_option(entry(_,_))).

% Add/clear mode options in global data base:
% Parameters: mode(Head,Require,Before,After,Survive).
add_mode_option(mode(true)) :- !.
% Both uninit & deref are required modes for external input:
add_mode_option(mode((H:-F))) :- check_non_builtin(H), !,
   my_retractall(mode_option(H,_,_,_,_)),
   make_req_bef(F, Req, Bef),
   keep_uninit(H, Req, KReq),
   asserta(mode_option(H,KReq,Bef,true,n)).
add_mode_option(mode(_)).
% Enter modes generated by flow analysis:
% Only uninit modes are required modes for flow analysis.
add_mode_option(analyze_mode(H,R,B,A)) :- check_non_builtin(H), !,
   survive(H, S),
   my_retractall(mode_option(H,_,_,_,_)),
   keep_uninit(H, R, KR),
   keep_uninit(H, B, KB),
   translate(KR, TR), flat_conj(TR, FR), logical_simplify(FR, SReq),
   translate(KB, TB), flat_conj(TB, FB), logical_simplify(FB, SBef),
   translate(A, TA), flat_conj(TA, FA), logical_simplify(FA, SAft),
   asserta(mode_option(H,SReq,SBef,SAft,S)).
add_mode_option(analyze_mode(_,_,_,_)).
% Enter require & before modes generated for internal dummy predicates:
% Require dereference mode becomes a Before dereference mode.
% (This is done in flatten & factor)
add_mode_option(dummy_mode(H,R,B)) :- check_non_builtin(H), !,
   survive(H, S),
   after(H, SAft),
   my_retractall(mode_option(H,_,_,_,_)),
   keep_uninit(H, R, KR), split_deref(KR, D, R2),
   keep_uninit(H, B, KB),
   translate(R2,     TR), flat_conj(TR, FR), logical_simplify(FR, SReq),
   translate((D,KB), TB), flat_conj(TB, FB), logical_simplify(FB, SBef),
   asserta(mode_option(H,SReq,SBef,SAft,S)).
add_mode_option(dummy_mode(_,_,_)).
add_mode_option(mode(H,R,B,A,S)) :- check_non_builtin(H), !,
   my_retractall(mode_option(H,_,_,_,_)),
   keep_uninit(H, R, KR), translate(KR, TR), logical_simplify(TR, STR),
   keep_uninit(H, B, KB), translate(KB, TB), logical_simplify(TB, STB),
   translate(A, TA), logical_simplify(TA, STA),
   asserta(mode_option(H,STR,STB,STA,S)).
add_mode_option(mode(_,_,_,_,_)).

% Create a new mode by supplementing an old one with an extra formula:
% Needs to be used whenever a new stree is created by source transformation.
% Require dereference mode becomes a Before dereference mode.
% AFlag (yes/no) determines whether the After mode is copied too.
add_mode_option(OldHead, NewHead, Req, Bef, AFlag) :-
   mode_option(OldHead,R,B,A,S), !,
   split_deref(R, D, R2),
   split_deref(Req, Deref, Req2),
   flat_conj((Req2,R2), NewR),
   flat_conj((Deref,D,Bef,B), NewB),
   make_after(AFlag, A, NewA),
   add_mode_option(mode(NewHead,NewR,NewB,NewA,S)).
add_mode_option(_, NewHead, Req, Bef, _) :-
   split_deref(Req, Deref, NewR),
   flat_conj((Deref,Bef), NewB),
   add_mode_option(mode(NewHead,NewR,NewB,true,n)).

make_after(yes, After, After).
make_after( no,     _,  true).

make_req_bef(Form, SReq, SBef) :-
   translate(Form, TForm),
   flat_conj(TForm, FForm),
   split_uninit_deref(FForm, Req, Bef),
   logical_simplify(Req, SReq),
   logical_simplify(Bef, SBef).

add_mode_options([]) :- !.
add_mode_options([(H:-B)|ML]) :- !,
   add_mode_option(mode((H:-B))),
   add_mode_options(ML).
add_mode_options([true|ML]) :- !,
   add_mode_options(ML).
add_mode_options([M|ML]) :-
   add_mode_option(M),
   add_mode_options(ML).

clr_mode :- my_retractall(mode_option(_,_,_,_,_)).

% Make sure that no modes are added for builtins (they would override
% the modes in the tables):
check_non_builtin(Head) :-
   nonvar(Head),
   functor(Head, N, A),
   check_nb(Head,
        ['Attempt to give modes for builtin ',N/A,' is ignored.']).

check_nb(Head, L) :- builtin(Head), !, warning(L), fail.
check_nb(_, _).

% Add/remove a compiler option:
o(Option)  :- nonvar(Option), cox(Option, _), po, !.
no(Option) :- nonvar(Option), nox(Option, _), po, !.
co(Option) :- nonvar(Option), cox(Option, true), po, !.
po         :- pox(Opts), write('% Options = '), write(Opts), nl, pmodes, !.

pmodes :- mode_option(A,B,C,D,E), w('% '), inst_writeq(mode(A,B,C,D,E)),nl,fail.
pmodes :- modal_entry(A,B), w('% '), inst_writeq(modal_entry(A,B)), nl, fail.
pmodes.

% Add an option:
ox(Option) :- ox(Option, _).

ox(Option, true) :-
   compile_option(Option), !.
ox(Option, false) :-
   \+compile_option(Option),
   pragma_write_once(Option),
   asserta(compile_option(Option)).

% Remove an option:
nox(Option) :- nox(Option, _).

nox(Option, true) :-
   copy(Option, Op),
   compile_option(Op), !,
   pragma_no_write_once(Option),
   my_retractall(compile_option(Option)).
nox(Option, false) :-
   \+compile_option(Option).

nox_list([]) :- !.
nox_list([Opt|Opts]) :- !, nox_list(Opt), nox_list(Opts).
nox_list(Opt) :- \+list(Opt), !, nox(Opt).

my_retractall(C) :-
   copy(C, D),
   retract(D), !,
   my_retractall(C).
my_retractall(_).

% Change/add an option:
% Add if it doesn’t exist, change if it does.
cox(Option) :- cox(Option, true).
cox(Option, true) :-
   atomic(Option),
   compile_option(Option), !.
cox(Option, Exists) :-
   functor(Option, N, A),
   functor(Dummy, N, A),
   nox(Dummy, Exists), !,
   ox(Option, _).
cox(Option, _) :-
   ox(Option).

cox_list([]) :- !.
cox_list([Opt|Opts]) :- !, cox_list(Opt), cox_list(Opts).
cox_list(Opt) :- \+list(Opt), !, cox(Opt).

% Make a list of all options:
pox(Opts) :- setof(Opt, compile_option(Opt), Opts).

% Set default options:
% Does the correct thing for each system.
default :- compile_option(system(cprolog)), !, cdefault.
default :- compile_option(system(quintus)), !, qdefault.

% For piped use under Quintus Prolog:
qdefault :-
   my_retractall(compile_option(_)),
   my_retractall(mode_option(_,_,_,_,_)),
   my_retractall(modal_entry(_,_)),
   default_option(X),
   asserta(compile_option(X)),
   fail.
qdefault.

% For piped use under C-Prolog:
cdefault :-
   qdefault,
   cox(system(cprolog)).

% For interactive use:
interactive_default(cprolog) :- cdefault, nox(flat), po.
interactive_default(quintus) :- qdefault, nox(flat), po.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Handle macro definitions:
% Macros give the expansion of a call into BAM assembly instructions.
% An attempt to give a macro definition to a built-in predicate is ignored.

add_macro_def(macro((H:-B))) :- check_macro_def(H), !,
   my_retractall(macro(H,_,_,_,_,_)),
   conjlist(B, BL, []),
   remove_double_indirect(H, NewH, NewBL, BL),
   varset(NewBL, Vars),
   difflist(NewBL, Code, Link),
   macro_varlist(NewBL, Varlist, Varlink),
   asserta(macro(NewH,Code,Link,Varlist,Varlink,Vars)).
add_macro_def(macro(_)).

% Add move instructions for all arguments except outputs.
% These move instructions catch the indirections created by the dereference
% chain transformation, thus eliminating any possibility of double indirections
% in the resulting code.  If there are no indirections, then preferred register
% allocation removes the moves.
remove_double_indirect(OldH, NewH) -->
   {functor(OldH, Na, Ar)},
   {functor(NewH, Na, Ar)},
        {require(OldH, Req)},
        {uninit_set_type(reg, Req, Set)},
   match_heads(1, Ar, OldH, NewH, Set).

match_heads(I, Ar, _, _, _) --> {I>Ar}, !.
match_heads(I, Ar, OldH, NewH, Set) --> {I=<Ar},
   {arg(I, OldH, X)},
   {inv(X, Set)},
   !,
   {arg(I, NewH, X)},
   {I1 is I+1},
   match_heads(I1, Ar, OldH, NewH, Set).
match_heads(I, Ar, OldH, NewH, Set) --> {I=<Ar},
   {arg(I, NewH, X)},
   {arg(I, OldH, Y)},
   [move(X,Y)],
   {I1 is I+1},
   match_heads(I1, Ar, OldH, NewH, Set).

check_macro_def(Head) :-
   nonvar(Head),
   functor(Head, N, A),
   check_nb(Head,
     ['Attempt to give macro definition for builtin ',N/A,' is ignored.']).

% Correctness of this varlist generation depends on the fact that for all
% instructions the output is written rightmost.  This means that a varbag
% that keeps the order is correct.
macro_varlist([move(A,B)|List]) --> {var(A),var(B)}, !,
   [pref,A,B],
   macro_varlist(List).
macro_varlist([label(_)|List]) --> !,
   macro_varlist(List).
macro_varlist([I|List]) --> {branch(I,Lbls)}, !,
   {varbag(I,Vars)},
   {diffbag(Vars, Lbls, Ls)},
   difflist(Ls),
   macro_varlist(List).
macro_varlist([I|List]) -->
   varbag(I),
   macro_varlist(List).
macro_varlist([]) --> [].

% Difference of two bags, keeping the order of the first:
diffbag([], _, []).
diffbag([X|B1], B2, B) :- memberv(X, B2), !, diffbag(B1, B2, B).
diffbag([X|B1], B2, [X|B]) :- diffbag(B1, B2, B).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

expression.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Compiling arithmetic expressions into basic arithmetic:

% At some point, get rid of is/2 in compilation of functor & arg.

% Top level goal (X is Expr):
top_expr(Expr, X) --> {arith_eval(Expr, EExpr)}, xtop_expr(EExpr, X).

% Used for an expression in an argument:
expr(Expr, S)     --> {arith_eval(Expr, EExpr)}, xexpr(EExpr, S).

% Top level goal:
xtop_expr(Expr, X) --> {number(Expr)},   !, co((X = Expr)).
xtop_expr(Expr, X) --> {var(Expr)},      !, co((X is Expr)).
xtop_expr(Expr, X) --> {atom(Expr)},     !, co((X is Expr)).
xtop_expr(Expr, X) --> {compound(Expr)}, !, xexpr(Expr, X).

% Convert arithmetic expression into primitive operations:
xexpr(IExpr, IExpr) --> {var(IExpr)}, !.
xexpr(IExpr, IExpr) --> {number(IExpr)}, !.
xexpr([IExpr], OExpr) --> !, xexpr(IExpr, OExpr).
xexpr(IExpr, OExpr) -->
   {arith_operation(IExpr, Es, Ss, OExpr, Pred, Link)}, !,
   xexpr_list(Es, Ss), insert(Pred, Link).
xexpr(IExpr, OExpr) --> co((OExpr is IExpr)).

xexpr_list([], []) --> [].
xexpr_list([E|Es], [S|Ss]) --> xexpr(E, S), xexpr_list(Es, Ss).

arith_operation(      E1+E2, [E1,E2], [S1,S2], D) --> co('$add'(S1,S2,D)).
arith_operation(      E1-E2, [E1,E2], [S1,S2], D) --> co('$sub'(S1,S2,D)).
arith_operation(      E1*E2, [E1,E2], [S1,S2], D) --> co('$mul'(S1,S2,D)).
arith_operation(     E1//E2, [E1,E2], [S1,S2], D) --> co('$div'(S1,S2,D)).
arith_operation((E1 mod E2), [E1,E2], [S1,S2], D) --> co('$mod'(S1,S2,D)).
arith_operation( (E1 /\ E2), [E1,E2], [S1,S2], D) --> co('$and'(S1,S2,D)).
arith_operation((E1 and E2), [E1,E2], [S1,S2], D) --> co('$and'(S1,S2,D)).
arith_operation( (E1 \/ E2), [E1,E2], [S1,S2], D) --> co( '$or'(S1,S2,D)).
arith_operation( (E1 or E2), [E1,E2], [S1,S2], D) --> co( '$or'(S1,S2,D)).
arith_operation((E1 xor E2), [E1,E2], [S1,S2], D) --> co('$xor'(S1,S2,D)).
arith_operation(      -(E1),    [E1],    [S1], D) --> co( '$sub'(0,S1,D)).
arith_operation(      \(E1),    [E1],    [S1], D) --> co(   '$not'(S1,D)).
arith_operation( (E1 << E2), [E1,E2], [S1,S2], D) --> {\+vlsi_plm}, !,
   co('$sll'(S1,S2,D)).
arith_operation( (E1 >> E2), [E1,E2], [S1,S2], D) --> {\+vlsi_plm}, !,
   co('$sra'(S1,S2,D)).
arith_operation( (E1 << E2), [E1,E2], [S1,S2], S1) --> {vlsi_plm}, !,
   co('$sll'(S1,S2)).
arith_operation( (E1 >> E2), [E1,E2], [S1,S2], S1) --> {vlsi_plm}, !,
   co('$sra'(S1,S2)).
```

```
% Evaluate as much of an arithmetic expression as possible at compile-time:
arith_eval(Expr, Expr) :- var(Expr), !.
arith_eval(Expr,  Res) :- nonvar(Expr), !,
   functor(Expr, Op, Arity),
   functor(EExpr, Op, Arity),
   eval_args(1, Arity, Expr, EExpr),
   arith_one(EExpr, Res).

eval_args(I, N, _, _) :- I>N, !.
eval_args(I, N, Expr, EExpr) :- I=<N, !,
   arg(I, Expr, E),
   arg(I, EExpr, EE),
   arith_eval(E, EE),
   I1 is I+1,
   eval_args(I1, N, Expr, EExpr).

arith_one(Op, Res) :- arith_one_tab(Op, Res), !.
arith_one(Op, Op).

arith_one_tab(      [A], A).
arith_one_tab(      A+B, R) :- integer(A), integer(B), R is A+B.
arith_one_tab(      A-B, R) :- integer(A), integer(B), R is A-B.
arith_one_tab(      A*B, R) :- integer(A), integer(B), R is A*B.
arith_one_tab(     A//B, R) :- integer(A), integer(B), R is A//B.
arith_one_tab((A mod B), R) :- integer(A), integer(B), R is (A mod B).
arith_one_tab( (A /\ B), R) :- integer(A), integer(B), R is (A /\ B).
arith_one_tab((A and B), R) :- integer(A), integer(B), R is (A /\ B).
arith_one_tab( (A \/ B), R) :- integer(A), integer(B), R is (A \/ B).
arith_one_tab( (A or B), R) :- integer(A), integer(B), R is (A \/ B).
arith_one_tab((A xor B), R) :- integer(A), integer(B), xor(A, B, R).
   % R is (A /\ \(B)) \/ (B /\ \(A)).
arith_one_tab(     -(A), R) :- integer(A), R is -(A).
arith_one_tab(     \(A), R) :- integer(A), R is \(A).
arith_one_tab( (A << B), R) :- integer(A), integer(B), R is (A << B).
arith_one_tab( (A >> B), R) :- integer(A), integer(B), R is (A >> B).

xor(A, B, R) :- R is (A /\ \(B)) \/ (B /\ \(A)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

factor.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Clause head factoring:

% Collect groups of clauses whose heads can be combined in nontrivial
% fashion (with most-specific-generalization).  Then create dummy procedures
% for those groups and simplify the definition of the original predicate.

% For example, consider the predicate:
%  h([x|Y]).
%  h([y|Y]).
%  h([]).
% This factorizes to:
%  h([A|B]) :- h’(B, A).
%  h([]).
%
%  h’(B, x).
%  h’(B, y).

% This module implements the following heuristic:
% Only factor a given argument of a predicate if no non-adjacent clauses have
% a non-trivial MSG.  This ensures that no extra choice points will be created.

% The transformation will not cause solutions to be reordered on backtracking
% if the compiler option same_order_solutions is active.  If this option is
% disabled, then more aggressive factoring is done which will change the
% ordering of solutions on backtracking.  In that case, the above heuristic
% has no effect.

% This module uses the data type ’ptree(NameArity,Cls,Mode,DummyList)’ to
% represent a tree of dummy predicates, where Mode=(Head:-Formula).
% This algorithm executes in O(A N log N) time where A = arity and N = number
% of clauses.

% This module may need other heuristics to avoid superfluous choice point
% creation which may slow down execution in some cases.  For example, the
% savings of multiple structure creation (how many structures are saved)
% can be weighed against the cost of creating and using the extra choice point.
% What’s inside the MSG’s can also be taken into account.  If they’re all
% variables then deterministic selection may not be able to do much, but if
% they’re all different nonvariable terms then much can be done.

% Other heuristics for future consideration:
% 1. Atoms are not factored out because it gives no performance improvement.
% X. Factor only if there are modes.  If there are no modes, then default
%    selection has something to work on.  FAULTY HEURISTIC!  Factoring leaves
%    exactly what is required to work on.
% Possible: don’t factor if there is a nonvar mode & some nonvar terms in that
% argument.  See verb_form for an example where this might help.
% Possible heuristic: factor those args with structure variables which are in
% tests beyond the MSG.  Then some determinism selection will work
% for them.
% Possible heuristic: don’t factor if it’s not needed to get determinism.
% How to approximate this in practice?
% Possible heuristic: don’t factor if there are ’enough’ tests in the clauses.

% Notes:
% 1. The call graph of this module is highly cyclic, but clean.  Termination
%    is somewhat subtle to show.
% 2. The ’match_...’ predicates are also used elsewhere.  They may be better
%    off as utilities.
% 3. The transfer of modes from a predicate to its dummies is done in a naive
%    way: the same formula is used.  A later version should look inside
%    structures and lists and infer modes about them.
% 4. Atomic head arguments are NOT factored out, since this does NOT speed
%    up execution.
% 4. This module should be used before standardization is done.
% 5. Can do dereference factoring: know that arguments are dereferenced.
%    Convention that all args upon entry are always dereferenced.
% 6. This module increases the average arity of predicates.

% *** Entry Point:
% Factor a set of Ptrees:
factor_ptrees(Ps, Ps) :- \+compile_option(factor), !.
factor_ptrees([], []) :-   compile_option(factor), !.
factor_ptrees([D|Ps], [D|FPs]) :- compile_option(factor),
   directive(D), !,
   factor_ptrees(Ps, FPs).
factor_ptrees([P|Ps], [FP|FPs]) :- compile_option(factor), !,
   factor_dlist(P, DP),
   get_factor_args(P, Args),
   factor_args_ptree(Args, DP, FP),
   factor_ptrees(Ps, FPs).

factor_dlist(ptree(NaAr,Cls,M,D), ptree(NaAr,Cls,M,ND)) :-
   factor_ptrees(D, ND).

% Given the arguments, factor out all Args of a Ptree, and get a new Ptree:
% The DList is unchanged.
factor_args_ptree([], P, P).
factor_args_ptree([N|Args], ptree(NaAr,Cls,M,D), ptree(NaAr,NCls,M,ND)) :-
   factor_arg_cls(NaAr, N, Cls, NCls, M, Ptrees, []),
   factor_args_ptrees(Args, Ptrees, ND, D).

% Given the arguments, factor a set of Ptrees, and get a set of new Ptrees:
factor_args_ptrees(_, [], Link, Link).
factor_args_ptrees(Args, [P|Ptrees], [NP|NPtrees], Link) :-
   factor_args_ptree(Args, P, NP),
   factor_args_ptrees(Args, Ptrees, NPtrees, Link).

% Factor argument N of a procedure: (no recursive factoring)
factor_arg_cls(NaAr, N, Cls, NewCls, Mode, Ptrees, Link) :-
   check_arity(Cls, N),
   get_arg(Cls, N, Args),
   solution_order(Args, OrderArgs),
   collect_info(NaAr, OrderArgs, Info, MsgList),
   cons(Info),
   check_heuristic(MsgList),
   !,
   transform(Cls, 1, N, Info, NewCls, DummyProcs),
   keysort(DummyProcs, SortProcs),
   collect_ptrees(SortProcs, N, Mode, Ptrees, Link).
factor_arg_cls(NaAr, N, Cls, Cls, Mode, Link, Link).
   % \+check_arity(Cls, N), !.

solution_order(Args, Args) :- same_order, !.
solution_order(Args, Sort) :- \+same_order, !, keysort(Args, Sort).

% Succeed (& then factor the argument) only if no non-adjacent heads have
% a non-trivial MSG:
check_heuristic(MsgList) :-
   keysort(MsgList, Sort),
   \+adjacent_nontrivial(Sort).

adjacent_nontrivial([X-_,Y-_|_]) :- msg(X, Y, Z), compound(Z).
adjacent_nontrivial([_|L]) :- adjacent_nontrivial(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Get arguments for factoring:

% Succeeds with list of possible argument numbers for factoring:
get_factor_args(ptree(_,Cls,(_:-Form),_), Args) :-
   % \+Form=true,
   a_head(Cls, H), functor(H, _, Arity),
   bagof(N, (range(1,N,Arity),test_arg(N,Cls)), Args), !.
get_factor_args(_, []).

% Succeeds if argument N has two similar compound values:
% Can’t use setof/3 here because it combines identical elements.
test_arg(N, Cls) :-
   bagof(X-nop, H^(a_head(Cls, H),arg(N,H,X),compound(X)), SetArgs),
   keysort(SetArgs, SortArgs),
   similar_args(SortArgs), !.

a_head(Cls, H) :- member(Cl, Cls), split(Cl, H, _).

% Succeeds if there are two similar arguments:
similar_args([X-nop,Y-nop|Set]) :- similar(X,Y), !.
similar_args([_|Set]) :- similar_args(Set).

% Succeeds if A and B are similar in structure:
similar(A, B) :-
   functor(A, Name, Arity),
   functor(B, Name, Arity).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Check if argument number N =< arity of a procedure:
check_arity([Cl|Cls], N) :-
   split(Cl, Head, _),
   functor(Head, _, Ar),
   N=<Ar.

% Collect set of ArgN-I where
% ArgN = Nth argument and I = number of head containing the argument.
get_arg(Clauses, N, Args) :-
   get_arg(Clauses, N, Args, 1).

get_arg([], _, [], _).
get_arg([Cl|Cls], N, [Arg-I|Args], I) :-
   head_arg(N, Cl, Arg),
   I1 is I+1,
   get_arg(Cls, N, Args, I1).

head_arg(N, Cl, Arg) :-
   (Cl=(H:-B)
    -> arg(N, H, Arg)
     ; arg(N, Cl, Arg)
   ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This predicate collects the information necessary for the transformation:

% First collect adjacent arguments into a list Msgs with terms Msg-ClauseNums
% while the msg is not a variable or atomic.  Expand this list into clause order
% again and attach a dummy name to each clause index.  This name will be used
% for the dummy procedures to be generated.  Each element in Info is of the
% form:
%  ClauseNum-info(Msg,Dummy,Varnum,Rank)
% where:
%  Msg = the most-specific-generalization common to the ClauseNums
%  Dummy = the name to be used for the dummy predicate
%  Varnum = the number of variables in the Msg
%  Rank = 'first' for only one ClauseNum element of a given Msg

collect_info(NaAr, Args, Info, MsgList) :-
   collect_msg(Args, MsgList),
   remove_single(MsgList, Msgs),
   list_clauses(Msgs, NaAr, Unorder, []),
   keysort(Unorder, Info).

collect_msg([], []).
collect_msg([Arg-I|Args], Collect) :-
   collect_msg(Args, Arg, [I], Collect).

collect_msg([], CurMsg, CurIs, [CurMsg-CurIs]) :-   compound(CurMsg), !.
collect_msg([], CurMsg, CurIs,             []) :- \+compound(CurMsg), !.
collect_msg(Args,         CurMsg, CurIs, Collect) :-
   \+compound(CurMsg), !,
   collect_msg(Args, Collect).
collect_msg([Arg-I|Args], CurMsg, CurIs, Collect) :-
   compound(CurMsg),
   msg(Arg, CurMsg, Msg),
   ( nonvar(Msg)
   -> collect_msg(Args, Msg, [I|CurIs], Collect)
    ; Collect = [CurMsg-CurIs|NewColl],
      collect_msg(Args, Arg, [I], NewColl)
   ).

% Remove singletons from list of clause sets with common msg’s:
remove_single([], []).
remove_single([MI|Msgs], RMsgs) :-
   MI=(_-[_]), !,
   remove_single(Msgs, RMsgs).
remove_single([MI|Msgs], [MI|RMsgs]) :-
   \+(MI=(_-[_])), !,
   remove_single(Msgs, RMsgs).

% Go from Msg-indexed list to ClauseNum-indexed list:
list_clauses([], _) --> [].
list_clauses([Msg-Is|Msgs], NaAr) -->
   {gensym("$fac_", NaAr, Dummy)},
   {varset(Msg,VS), length(VS,V)},
   expand_clauses(Is, Msg, Dummy, V, first),
   list_clauses(Msgs, NaAr).

expand_clauses([], _, _, _, _) --> [].
expand_clauses([I|Is], Msg, Dummy, Varnum, Rank) -->
   {copy(Msg, Copy)},
   [I-info(Copy,Dummy,Varnum, Rank)],
   expand_clauses(Is, Msg, Dummy, Varnum, notfirst).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This predicate does the actual transformation:

% Picture of the transformation:
% Original clause:
%  h(--alpha--,[x|y],--beta--) :- Body
% Transformed clause:
%  nhead(--vars1--,[A|B],--vars2--) :- dhead1(--vars1--,B,--vars2--,A).
% (only one for all original clauses matching the pattern)
% One clause of dummy procedure:
%  dhead2(--alpha--,y,--beta--,x) :- Body.
% (one for each original clause matching the pattern)

transform(Cls, _, _, [], Cls, []) :- !.
transform([Cl|Cls], I, N, [J-Q|Info], NCls, [X|Procs]) :-
   I=:=J, !,
   split(Cl, Head, Body),
   Q=info(Msg,Dummy,Varnum,Rank),
   functor(Head, Name, Arity),
   functor(NHead, Name, Arity),
   calc_darity(Head, Varnum, DArity),
   functor(DHead1, Dummy, DArity),
   functor(DHead2, Dummy, DArity),
   minimum(Arity, DArity, MinArity),
   match_most(1, N, MinArity, Head, DHead2),
   match_most(1, N, MinArity, NHead, DHead1),
   match_rest(N, Arity, Head, NHead, DHead1, DHead2, Msg),
   X=(Dummy/DArity-info2(Msg,Varnum,(DHead2:-Body))),
   (Rank=first
    -> NCls=[(NHead:-DHead1)|NCls2]
     ; NCls=NCls2
   ),
   I1 is I+1,
   transform(Cls, I1, N, Info, NCls2, Procs).
transform([Cl|Cls], I, N, [J-Q|Info], [Cl|NCls], Procs) :-
   I<J, !,
   I1 is I+1,
   transform(Cls, I1, N, [J-Q|Info], NCls, Procs).
transform([Cl|Cls], I, N, [J-Q|Info], [Cl|NCls], Procs) :-
   I>J, !,
   I1 is I+1,
   transform(Cls, I1, N, Info, NCls, Procs).

% Calculate the arity of the dummy procedure:
calc_darity(Head, Varnum, DArity) :-
        functor(Head, _, Arity),
        DAr is (Arity+Varnum-1),
        maximum(DAr, Arity, DArity).    % Just in case MSG is an atom!

% Unify arguments 1 to A of two structures except argument N:
match_most(I, N, A, _, _) :- I>A, !.
match_most(I, N, A, Head, DHead) :- I=\=N, I=<A, !,
        arg(I, Head, Arg),
        arg(I, DHead, Arg),
        I1 is I+1,
        match_most(I1, N, A, Head, DHead).
match_most(I, N, A, Head, DHead) :- I=:=N, I=<A, !,
        I1 is I+1,
        match_most(I1, N, A, Head, DHead).

% Alternate definition:
% match_most(I, N, A, Head, DHead) :-
%   Nl is N-1,
%   Nr is N+1,
%   match_all(I, Nl, Head, DHead),
%   match_all(Nr, A, Head, DHead).

% Unify all arguments from L to H of two structures:
match_all(L, R, _, _) :- L>R, !.
match_all(L, R, Head, DHead) :- L=<R, !,
        arg(L, Head, Arg),
        arg(L, DHead, Arg),
        L1 is L+1,
        match_all(L1, R, Head, DHead).
```

```
% Unify all arguments from L to H of first structure with arguments
% starting at argument F of second structure:
match_offset(L, R, _, _, _) :- L>R, !.
match_offset(L, R, Head, F, DHead) :- L=<R, !,
   arg(L, Head, Arg),
   arg(F, DHead, Arg),
   L1 is L+1,
   F1 is F+1,
   match_offset(L1, R, Head, F1, DHead).

% Create the argument corresponding to the MSG
% in the right way.  This is slightly tricky because
% the MSG can be a nested structure.
match_rest(N, A, Head, NHead, DHead1, DHead2, Msg) :- N>A, !.
match_rest(N, A, Head, NHead, DHead1, DHead2, Msg) :- N=<A, !,
   arg(N, NHead, Msg),
   arg(N, Head, Term),
   corresponding_vars(Msg, Term, Msgvars, Termargs),
   (cons(Msgvars)
   -> NA is A+1,
      arg(N, DHead2, A2), Termargs=[A2|ET],
      end_fill_args(ET, DHead2, NA),
      arg(N, DHead1, A1), Msgvars=[A1|EM],
      end_fill_args(EM, DHead1, NA)
   ;  true
   ).

% Unify a list with the last arguments of a pred symbol:
end_fill_args([], _, _) :- !.
end_fill_args([X|L], Pred, I) :-
   arg(I, Pred, X),
   I1 is I+1,
   end_fill_args(L, Pred, I1).

% Traverse Msg and Term, for each variable encountered in Msg
% create an entry in Msgvars and put the corresponding term in Termargs.
% The first argument of Msgvars is the argument of the rightmost
% most-deeply-nested structure.  It replaces the argument in the head
% of the factorized predicate.
corresponding_vars(Msg, Term, Msgvars, Termargs) :-
   corr_vars(Msg, Term, [], Msgvars, [], Termargs).

corr_vars(Msg, Term, IV, OV, IT, OT) :- nonvar(Msg), !,
   functor(Msg, N, Arity),
   functor(Term, N, Arity),
   corr_args(1, Arity, Msg, Term, IV, OV, IT, OT).
corr_vars(Msg, Term, IV, [Msg|IV], IT, [Term|IT]) :- var(Msg).

corr_args(I, A, _, _, IV, IV, IT, IT) :- I>A, !.
corr_args(I, A, Msg, Term, IV, OV, IT, OT) :- I=<A, !,
   arg(I, Msg, MA),
   arg(I, Term, TA),
   corr_vars(MA, TA, IV, MV, IT, MT),
   I1 is I+1,
   corr_args(I1, A, Msg, Term, MV, OV, MT, OT).

% Identical to corr_vars, but only creates the list of variables of Msg:
msg_vars(Msg, Msgvars) :- msg_vars(Msg, [], Msgvars).

msg_vars(Msg, IV, OV) :- nonvar(Msg), !,
   functor(Msg, _, Arity),
   msg_args(1, Arity, Msg, IV, OV).
msg_vars(Msg, IV, [Msg|IV]) :- var(Msg).

msg_args(I, A, _, IV, IV) :- I>A, !.
msg_args(I, A, Msg, IV, OV) :- I=<A, !,
   arg(I, Msg, MA),
   msg_vars(MA, IV, MV),
   I1 is I+1,
   msg_args(I1, A, Msg, MV, OV).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Collect adjacent procedures:
collect_ptrees([], _, _) --> [].
collect_ptrees([NaAr-Info2|KeyCls], N, Mode) -->
   {Info2=info2(Msg,Varnum,Cl)},
   {Mode=(H:-_)},
   {require(H, Req)},
   {before(H, Bef)},
   {make_dformula((H:-Req), NaAr, Msg, N, Varnum, (DH:-DReq))},
   {make_dformula((H:-Bef), NaAr, Msg, N, Varnum, (DH:-DBef))},
   {flat_conj((DReq,DBef), DForm)},
   [ptree(NaAr,[Cl|Proc],(DH:-DForm),[])],
   {add_mode_option(dummy_mode(DH,DReq,DBef))},
   {collect_proc(KeyCls, NaAr, NextCls, Proc)},
   collect_ptrees(NextCls, N, Mode).

collect_proc([], _, [], []).
collect_proc([X|Cls], NaAr, [X|Cls], []) :-
   X=(NNaAr-info2(_,_,Cl)),
   Info2=info2(_,_,Cl),
   NaAr\==NNaAr, !.
collect_proc([X|Cls], NaAr, NewCls, [Cl|Proc]) :-
   X=(NaAr-info2(_,_,Cl)),
   collect_proc(Cls, NaAr, NewCls, Proc).

% Calculate modes of the dummy procedure:
% This propagates modes which can be propagated from the MSG and knowledge of
% the original predicate’s modes.  Currently the modes uninit(X), ground(X),
% rderef(X), and var(X) from the original predicate.
% (Later this can be extended to propagate recursively defined types as well.)
% Note that this predicate takes care to order the new arguments in exactly
% the same was as transform does it, by using msg_vars and end_fill_args.
make_dformula((Head:-Form), DN/DA, Msg, Argnum, Varnum, (DHead:-NewForm)) :-
   functor(Head, N, A),
   functor(DHead, DN, DA),
   minimum(A, DA, Min),
        match_most(1, Argnum, Min, Head, DHead),
   arg(Argnum, Head, X),
   msg_vars(Msg, Msgvars),
   (cons(Msgvars)
   -> Msgvars=[Hd|Tl],
      arg(Argnum, DHead, Hd),
      NA is A+1,
      end_fill_args(Tl, DHead, NA)
    ; true
   ),
   (type_propagate(Form, X, Type)
   -> mode_propagate(Msgvars, Type, NewForm, Form)
    ; NewForm=Form
   ).

% Calculate type to be propagated:
% Need to extend this to propagate uninit, var, rderef only for new variables
% (and rderef only if the argument is ground).
% type_propagate(Form, X, uninit) :- implies(Form, uninit(X)), !.
% type_propagate(Form, X,    var) :- implies(Form, var(X)), !.
type_propagate(Form, X, ground) :- implies(Form, ground(X)), !.
% type_propagate(Form, X, rderef) :- implies(Form, rderef(X)), !.

% Propagate the type to all variables in Msgvars:
mode_propagate([], Type) --> !.
mode_propagate([V|Msgvars], Type) -->
   one_propagate(V, Type),
   mode_propagate(Msgvars, Type).

one_propagate(V, uninit) --> co(uninit(V)), !.
one_propagate(V,    var) --> co(var(V)), !.
one_propagate(V, ground) --> co(ground(V)), !.
one_propagate(V, rderef) --> co(rderef(V)), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Most specific generalization:

% Useful for factorizing clause heads into common parts
% for selection into nested structures.
% For example, msg([x|Y], [y|Y], Z) gives Z = [_|Y].

msg(A, B, G) :-
   nonvar(A), nonvar(B), !,
   (functor(A, Na, Ar),
    functor(B, Na, Ar)
    -> A=..[Na|AArgs],
       B=..[Na|BArgs],
       msg_args(AArgs, BArgs, GArgs),
       G=..[Na|GArgs]
     ; true
   ).
msg(A, B, A) :- A==B, !.
msg(A, B, _) :- \+(A==B), !.
```

```
msg_args([], [], []).
msg_args([A|AArgs], [B|BArgs], [G|GArgs]) :-
   msg(A, B, G),
   msg_args(AArgs, BArgs, GArgs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

flatten.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Flattening transformation.
% Remove all disjunctions from a predicate.

% Notes:
% 1. Modify new head generation to minimize number of moves.
% 2. Flatten passes mode information into the dummy predicates.
% 3. Flatten provides additional Mode information for the dummy procedures
%    it creates.  This is implemented by
%    collecting all the goals traversed in the disjunction and adding them
%    to the modes formula.
%    Question: is this really needed?  In clause_code,
%    a formula will be given that subsumes this, right?
% 4. Depending on the style of transformation, a single clause could result in
%    multiple clauses.  This is an improvement for later.
%    a:-(b;c),d results in multiple clauses, a:-d,(b;c) does not.

% Convert an stree into a flattened stree, i.e. all disjunctions are removed
% from the component procedures and dummy strees are created for them.
% All output strees are in standard form with independent variables.
% The input must be an stree in standard form.

% Flatten an stree.  This version does not worry about cut.
flatten_stree(stree(NaAr,(Head:- Disj),Modes,OH, DList,SD),
         stree(NaAr,(Head:-FDisj),(Head:-Formula),OH,FDList,SD)) :- !,
   % Must do copy before varset, else subtle bug exists: (!)
   copy(Modes, (Head:-Formula)),
   varset(Head, HVars),
   require(Head, RF),
   before(Head, BF),
   flatten_disj(NaAr, Disj, FDisj, RF, BF, HVars, [], SD, FDList, L),
   flatten_strees(DList, L).
flatten_stree(D, D) :- directive(D), !.

flatten_strees([], []).
flatten_strees([S|Strees], [FS|FStrees]) :-
   flatten_stree(S, FS),
   flatten_strees(Strees, FStrees).

% Remove all disjunctions from inside a disjunction:
% This gives a list of strees.
flatten_disj(NaAr, (C;Cs), (NC;NCs), RF, BF, Bef, OutsideAft, SD) -->
   {disj_inside(C)}, !,
   flatten(NaAr, C, NC, RF, BF, Bef, _, OutsideAft, SD),
   flatten_disj(NaAr, Cs, NCs, RF, BF, Bef, OutsideAft, SD).
flatten_disj(NaAr, (C;Cs), (C;NCs), RF, BF, Bef, OutsideAft, SD) -->
   {\+disj_inside(C)}, !,
   flatten_disj(NaAr, Cs, NCs, RF, BF, Bef, OutsideAft, SD).
flatten_disj(NaAr, fail, fail, _, _, _, _, _) --> [].
```

```
% Succeed if there’s a disjunction inside:
disj_inside((A,B)) :- disj_inside(A), !.
disj_inside((A,B)) :- disj_inside(B), !.
disj_inside((A;B)).

% Bef = vars existing before D.
% Aft = vars existing after D; in Body and beyond it.
% OutsideAft = vars existing beyond Body.
flatten(NaAr, (D,Body), (Head,NewBody), RF, BF, Bef, NewAft, OutsideAft,SD) -->
   {strong_disj_p(D)}, !,
   {varset(D, VD)},
   {unionv(VD, Bef, NewBef)},
   % Keep only those required modes that aren’t used in D:
   {split_formula2(Bef, D, RF, BF, RF2)},
   {remove_vars(BF, BF2)},
   flatten(NaAr, Body, NewBody, RF2, BF2, NewBef, Aft, OutsideAft, SD),
   {unionv(VD, Aft, NewAft)},
   dummy_proc(NaAr, D, VD, RF, BF, Bef, Aft, Head, SD).
flatten(NaAr, (D,Body), (D,NewBody), RF, BF, Bef, NewAft, OutsideAft, SD) -->
   {\+strong_disj_p(D)}, !,
   {varset(D, VD)},
   {unionv(VD, Bef, NewBef)},
   {after(D, DAfter)},
   % Keep only those required modes that aren’t used in D:
   {split_formula2(Bef, D, RF, BF, RF2)},
   {update_formula(DAfter, Bef, BF, BF2)},
   flatten(NaAr, Body, NewBody, RF2, BF2, NewBef, Aft, OutsideAft, SD),
   {unionv(VD, Aft, NewAft)}.
flatten(NaAr, true, true, _, _, _, OutsideAft, OutsideAft, _) --> [].

% Convert a single disjunction into a new dummy procedure (represented as Stree)
% (Calls flatten recursively for nested disjunctions.)
dummy_proc(NaAr, D, VD, RF, BF, Bef, Aft, Head, SD, [Stree|Link], Link) :-
   % strong_disj_p(D), !,
   flatten_disj(NaAr, D, ND, RF, BF, Bef, Aft, SD, SL, []),
   unionv(Bef, Aft, Outside),
   intersectv(VD, Outside, DArgs),
   gensym("$fla_", NaAr, Name),
   Head=..[Name|DArgs],
   functor(Head,Name,Arity),
   % Selection depth is NOT copied along: (see calc_select_depth)
   flat_conj((RF,BF), Formula),
   trim_mode((Head:-Formula), HeadForm),
   copy(stree(Name/Arity,(Head:-ND),HeadForm,[],SL,_), Stree),
   add_mode_option(dummy_mode(Head,RF,BF)).

% Make sure that split_formula has all the information it needs
% so that it is not overly conservative:
split_formula2(SoFar, D, RF, BF, RF2) :-
   unionv_conj(RF, BF, AllF),
   split_formula(yes, SoFar, D, AllF, _, AllF2),
   intersectv_conj(AllF2, RF, RF2).
```

```
% Succeed if there is a disjunction in the goal list:
disj_exists((G,L)) :- (disj_p(G) -> true; disj_exists(L)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

inline.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Replace in-line those predicates whose definition is a single clause
% containing no cuts or dummy predicates.  This module must come after
% transform_cut, and should come before analyze for best results.

% Replacement is done if (1) the inline predicate’s definition consists
% of at most 2 calls, and (2) the caller does only tests
% before the inline predicate is called.

inline_strees(Ss, Rs) :-
   compile_option(inline), !,
   gather_single(Ss, SA),
   inline_replace_strees(Ss, Rs, SA).
inline_strees(Ss, Ss) :-
   \+compile_option(inline), !.

% *** Magic number:
% Gather into an array the non-builtin predicates defined by
% a single clause containing no cuts or dummy predicates.
% (Due to transform_cut, if the DList is empty then there are no cuts.)
gather_single([], SA) :- seal(SA).
gather_single([stree(NA,(H:-(C;fail)),_,_,[],_)|Ss], SA) :-
   length_test_user(C, NT, NU),
   % NT=<2, NU=<2,
   N is NT+NU, N=<2,
   \+builtin(H), !,
   get(SA, NA, X),
   enter_def(X, info((H:-C),NT,NU), NA),
   gather_single(Ss, SA).
gather_single([_|Ss], SA) :- gather_single(Ss, SA).

% Replace inline the one-clause predicates in the program:
% (Could do closure as long as an inline predicate contains another,
% but for conciseness will only do a single step now.)
inline_replace_strees([], [], _).
inline_replace_strees([stree(NA,(H:-D),M,OH,DL,SD)|Ss],
             [stree(NA,(H:-RD),M,OH,RDL,SD)|RSs], SA) :-
   inline_replace_disj(D, RD, SA),
   inline_replace_strees(DL, RDL, SA),
   inline_replace_strees(Ss, RSs, SA).

inline_replace_disj(fail, fail, _).
inline_replace_disj((C;D), (FC;RD), SA) :-
   inline_replace_conj(C, RC, SA),
   flat_conj(RC, FC),
   inline_replace_disj(D, RD, SA).

% Only replace in a conjunction if it has only tests before:
inline_replace_conj(true, true, _).
inline_replace_conj((G,C), (G,RC), SA) :-
   test(G), !,
   inline_replace_conj(C, RC, SA).
inline_replace_conj((G,C), (RG,RC), SA) :-
   functor(G, N, A),
   fget(SA, N/A, info(Def,NT,NU)), !,
   copy(Def, (G:-RG)),
   (NU=:=0
   -> inline_replace_conj(C, RC, SA)
    ; C=RC
   ).
inline_replace_conj((G,C), (G,C), SA).
   % inline_replace_conj(C, RC, SA).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

mutex.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Implication:

% Given goal sequences A and B, if implies(A,B) succeeds then
% A implies B is true.  But the predicate will fail in some cases
% where an implication does exist.

% Logical implication
implies(A, B) :- logical_simplify(not(B), NB), mutex(A, NB, left), !.
implies(A, B, Time) :- logical_simplify(not(B), NB), mutex(A, NB, Time), !.

% Prolog implication:
% (A and B are Prolog formulas)
prolog_implies(A, B) :- simplify(not(B), NB), prolog_mutex(A, NB, left), !.
prolog_implies(A, B, Time) :- simplify(not(B), NB), prolog_mutex(A, NB, Time),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Logical mutual exclusion: mutex(A, B, Time)
% Prolog mutual exclusion: prolog_mutex(A, B, Time)

% Determine whether two goal sequences are mutually exclusive:
% Success is sufficient but not necessary for mutual exclusion.

% (Test new definitions incorporating modes.)
% (Check backtracking behavior invariance.)
% not/1 and =>/2 are logically correct negation and implication
% \+/1 and ->/2 are the Prolog by-failure versions.

% If Time=before:
% If the goal mutex(S1, S2) succeeds then there does not exist an
% assignment of values to variables which causes both S1 and S2
% to succeed separately.  The values after S1 and after S2 do not have
% to be consistent.
% (problem with encode_relop and (A=B) & (A is B) which both bind)

% If Time=after:
% If the goal mutex(S1, S2) succeeds then there does not exist an
% assignment of values to variables which exists after both S1 and S2
% have succeeded.

% Time=left:
% There does not exist an assignment of values to variables which exists
% after (S1,S2) has successfully executed.

% Time=right:
% There does not exist an assignment of values to variables which exists
% after (S2,S1) has successfully executed.
% Because this predicate will be used many times in the compilation of a
% procedure I have sacrificed its completeness for execution efficiency.
% For example, (A==B,var(A)) and nonvar(B) are mutually exclusive, yet this
% predicate will not be able to determine it.

% The bit arrays used give a partitioning of all possible values of
% a variable into several disjoint sets.  This allows fast determination
% when two such sets are mutually exclusive.  If the arrays overlap
% then a more complex calculation could be done.

% Notes:
% 1. Two important definitions related to uninitialized variables which are
%    recognized here:
%  unbound(X)    == ( uninit(mem,X) ; uninit(reg,X) ; var(X) )
%  uninit(any,X) == ( uninit(mem,X) ; uninit(reg,X) )
%  uninit(either,X) --> uninit(mem,X)
%  uninit(either,X) --> uninit(reg,X)
% 2. Mutex is defined as a determinate version of mutex2, which does the work.
% 3. For best results the input to mutex should sometimes be simplified before
%    the call.  I.e. var(x(_)) will simplify to fail, which is mutex to
%    anything.
% 4. Relational operators as tests could be assumed to require
%    numeric operands, even if they are negated.  not(A<3) means A>=3, so
%    A must be a number for success. (This point is not completely clear:
%    not(A<3) could also succeed for nonnumeric A.  Also, A could be
%    evaluated before doing the test, which is what C-Prolog does.)
% 5. Mutual exclusion can exist even between predicates that do binding,
%    e.g. A=a and A=b are mutually exclusive.

% Possible extensions:
% 1. Incorporation of user-defined predicates by means of a predicate/2
%    clause, i.e. predicate(P, Def) succeeds if P is a predicate and Def
%    its definition (considered as a disjunction).  Watch out for infinite
%    loops and binding!
% 2. For tests, more info such as values of small ints or arities of strucs
%    can be kept in the bit map to make mutex more powerful.
%    A whole data structure can be attached to the bit map to keep
%    sets of possible values around for even more power.  Is this a useful
%    thing to do?  In general, a data structure can subsume the
%    present distinction between relational operators and tests.

mutex(A, B) :- compile_option(mutex_comment), !, comment([mutex(A,B)]),
   mutex2(A, B, before), !.
mutex(A, B) :- \+compile_option(mutex_comment), !,
   mutex2(A, B, before), !.

mutex(A, B, T)        :- mutex(A, B, T, logical).
prolog_mutex(A, B, T) :- mutex(A, B, T, prolog).

mutex(A, B, T, L) :- compile_option(mutex_comment), !,
   comment([mutex(A,B,T,L)]),
   mutex2(A, B, T, L), !.
mutex(A, B, T, L) :- \+compile_option(mutex_comment), !,
   mutex2(A, B, T, L), !.
```

```
% 1. Mutual exclusion with failure:
mutex2(fail, _, _, _).
mutex2(_, fail, _, _).
mutex2('$test'(_,0), _, _, _).
mutex2(_, '$test'(_,0), _, _).

% 2. Mutual exclusion of if-then-else (=> and ->):
mutex2((A->B;C), D, T, L) :- mutex2((A,B),D, T, L), mutex2((\+(A),C),D, T, L).
mutex2(A, (B->C;D), T, L) :- mutex2(A,(B,C), T, L), mutex2(A,(\+(B),D), T, L).
mutex2((A->_), C, T, L) :- mutex2(A, C, T, L).
mutex2((_->B), C, T, L) :- mutex2(B, C, T, L).
mutex2(A, (B->_), T, L) :- mutex2(A, B, T, L).
mutex2(A, (_->C), T, L) :- mutex2(A, C, T, L).

mutex2((A=>B;C), D, T, L) :- mutex2((A,B),D, T, L), mutex2((not(A),C),D, T, L).
mutex2(A, (B=>C;D), T, L) :- mutex2(A,(B,C), T, L), mutex2(A,(not(B),D), T, L).
mutex2((A=>_),   C, T, L) :- mutex2(A, C, T, L).
mutex2((_=>B),   C, T, L) :- mutex2(B, C, T, L).
mutex2(A,   (B=>_), T, L) :- mutex2(A, B, T, L).
mutex2(A,   (_=>C), T, L) :- mutex2(A, C, T, L).

% 3. Mutual exclusion of conjunctions and disjunctions:
% A mutex2 (B and C)  <=> (A mutex2 B) or (A mutex2 C)
% A mutex2 (B or C)  <=> (A mutex2 B) and (A mutex2 C)
mutex2((A,_), C, T, L) :- mutex2(A, C, T, L).
mutex2((_,B), C, T, L) :- mutex2(B, C, T, L).
mutex2(A, (B,_), T, L) :- mutex2(A, B, T, L).
mutex2(A, (_,C), T, L) :- mutex2(A, C, T, L).
mutex2((A;B), C, T, L) :- mutex2(A, C, T, L), mutex2(B, C, T, L).
mutex2(A, (B;C), T, L) :- mutex2(A, B, T, L), mutex2(A, C, T, L).

% 4. Mutual exclusion of negation (not and \+):
mutex2(A, B, T, logical) :-
   logical_simplify(A, SA), logical_simplify(B, SB),
   (SA==not(SB); SB==not(SA)).
mutex2(A, B, T, prolog) :-
   simplify(A, SA), simplify(B, SB),
   (SA==not(SB); SB==not(SA)).

mutex2(not((A;_)), C, T, L) :- mutex2(not(A), C, T, L).
mutex2(not((_;B)), C, T, L) :- mutex2(not(B), C, T, L).
mutex2(A, not((B;_)), T, L) :- mutex2(A, not(B), T, L).
mutex2(A, not((_;C)), T, L) :- mutex2(A, not(C), T, L).
mutex2(not((A,B)), C, T, L) :- mutex2(not(A), C, T, L), mutex2(not(B), C, T, L).
mutex2(A, not((B,C)), T, L) :- mutex2(A, not(B), T, L), mutex2(A, not(C), T, L).

mutex2(\+((A;_)), C, T, L) :- mutex2(\+(A), C, T, L).
mutex2(\+((_;B)), C, T, L) :- mutex2(\+(B), C, T, L).
mutex2(A, \+((B;_)), T, L) :- mutex2(A, \+(B), T, L).
mutex2(A, \+((_;C)), T, L) :- mutex2(A, \+(C), T, L).
mutex2(\+((A,B)), C, T, L) :- mutex2(\+(A), C, T, L), mutex2(\+(B), C, T, L).
mutex2(A, \+((B,C)), T, L) :- mutex2(A, \+(B), T, L), mutex2(A, \+(C), T, L).
```

```
% 5. Mutual exclusion of predicates related to uninitialized variables:
mutex2(A, not(B), _, _) :- u_match(A, B).
mutex2(A, not(B), _, _) :- u_match(B, A).
mutex2(A, not(B), _, _) :- u_implies(A, B).
mutex2(not(A), B, _, _) :- u_match(A, B).
mutex2(not(A), B, _, _) :- u_match(B, A).
mutex2(not(A), B, _, _) :- u_implies(B, A).

% 6. Mutual exclusion of relational operations:
mutex2(Rel1, Rel2, T, L) :-
   encode_relop(Rel1, A,F1,B),
   encode_relop(Rel2, C,F2,D),
   check_relops(A,F1,B, C,F2,D).

% 7. Mutual exclusion of tests:
mutex2(Test1, Test2, Time, _) :-
   encode_test(Test1, S1, F1, A, Binds1),
   encode_test(Test2, S2, F2, B, Binds2),
   A==B,
   \+can_overlap(Time, S1, F1, S2, F2, Binds1, Binds2).

% 8. Mutual exclusion of compound terms:
mutex2(Test1, Test2, T, _) :-
   encode_name_arity(Test1, A, Na, Aa, Fa),
   encode_name_arity(Test2, B, Nb, Ab, Fb),
   A==B,
   check_name_arity(Fa, Fb, Na, Aa, Nb, Ab).

% Equivalences & Implications related to uninitialized variables:
u_match(trail(X),      trail_if_var(Y)) :- X==Y.
u_match(uninit(V),       uninit(mem,W)) :- V==W.
u_match(uninit(V),     uninit(mem,W,_)) :- V==W.
u_match(uninit(mem,V), uninit(mem,W,_)) :- V==W.
u_match(uninit(reg,V),   uninit_reg(W)) :- V==W.

u_implies(var(V),       unbound(W)) :- V==W.
u_implies(rderef(V),      deref(W)) :- V==W.
u_implies(uninit(either,V), Uninit) :-
   an_uninit_mode(Uninit, _, W),
   V==W.
u_implies(Uninit, Goal) :-
   an_uninit_mode(Uninit, _, V),
   u_implies_goal(Goal, W),
   V==W.

u_implies_goal(unbound(W),      W).
u_implies_goal(uninit(any,W),   W).
u_implies_goal(deref(W),        W).
u_implies_goal(rderef(W),       W).
u_implies_goal(trail(W),        W).
u_implies_goal(trail_if_var(W), W).

% Check that a predicate is a (built-in?) test:
test(T) :- encode_relop(T, _, _, _), !.
test(T) :- encode_test(T, _, _, _), !.
test(T) :- encode_name_arity(T, _, _, _, _), !.

% From a test, get the variables that are tested:
test_varset(T, V) :- test_varbag(T, B), sort(B, V).
test_varbag(T, V) :- test_varbag(T, V, []).

% test_varbag(X=Y) --> {var(X), var(Y)}, !.
test_varbag(T) --> {encode_relop(T, X, _, Y, _)},      filter_vars([X,Y]), !.
test_varbag(T) --> {encode_test(T, _, _, X, _)},       filter_vars([X]), !.
test_varbag(T) --> {encode_name_arity(T, X, _, _, _)}, filter_vars([X]), !.

% Relational test with numbers:
relational_test(T, X, Y) :- encode_relop(T, X, _, Y, arith), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% If X is implied to be atomic in a formula,
% return its value if possible.  Fail otherwise.

atomic_value((A,B), X, Value) :- atomic_value(A, X, Value).
atomic_value((A,B), X, Value) :- atomic_value(B, X, Value).
atomic_value( Test, X, Value) :-
   encode_name_arity(Test, Y, Na, Ar, true),
   X==Y,
   Ar=0,
   atomic(Na),
   Value=Na.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Returns the operands and a three-bit array of flags telling
% what comparison is being done.  The three-bit array consists
% of flags <less,equal,greater>.

% Three kinds of comparison are distinguished:
% arith: arithmetic, numeric comparison.  The test traps if the arguments
%   are not numbers.
% stand: standard term ordering.
% unify: unification of terms.

standard_order(Test) :- encode_relop(Test, _, _, _, stand).

encode_relop(A,B,C,D) :- encode_relop(A,B,C,D,K).

encode_relop('$name_arity'(A,F,N), A, 2'010, F, unify) :- atomic(F), N==0.
% encode_relop('$name_arity'(A,F,N), A, 2'010, F, stand) :- atom(F), N==0.
encode_relop(functor(A,F,N), A, 2'010, F, unify) :- number(F), N==0.
encode_relop(functor(A,F,N), A, 2'010, F, unify) :- atom(F), N==0.
encode_relop(functor(A,F,N), A, 2'010, F, unify) :- var(F), N==0.
encode_relop(functor(A,F,N), N, 2'010, K, unify) :- nonvar(A),functor(A,_,K).
encode_relop(functor(A,F,N), F, 2'010, K, unify) :- nonvar(A),functor(A,K,_).
encode_relop(A>B,   A, 2'001, B, arith). % arithmetic comparison
encode_relop(A@>B,  A, 2'001, B, stand). % standard ordering
encode_relop(A=:=B, A, 2'010, B, arith). % arithmetic comparison
encode_relop(A==B,  A, 2'010, B, stand). % standard ordering
encode_relop(A=B,   A, 2'010, B, unify). % unification
encode_relop(A is B,A, 2'010, B, arith). % arithmetic evaluation
encode_relop(A>=B,  A, 2'011, B, arith). % arithmetic comparison
encode_relop(A@>=B, A, 2'011, B, stand). % standard ordering
encode_relop(A<B,   A, 2'100, B, arith). % arithmetic comparison
encode_relop(A@<B,  A, 2'100, B, stand). % standard ordering
encode_relop(A=\=B, A, 2'101, B, arith). % arithmetic comparison
encode_relop(A\==B, A, 2'101, B, stand). % standard ordering
encode_relop(A\=B,  A, 2'101, B, unify). % unification (never binds)
encode_relop(A=<B,  A, 2'110, B, arith). % arithmetic comparison
encode_relop(A@=<B, A, 2'110, B, stand). % standard ordering
encode_relop(not(RelOp), A, F, B, Kind) :-
   nonvar(RelOp),
   encode_relop(RelOp, A, Flags, B, Kind),
   negate_relop(Flags, F).
encode_relop(\+(RelOp), A, F, B, Kind) :-
   nonvar(RelOp),
   encode_relop(RelOp, A, Flags, B, Kind),
   negate_relop(Flags, F).

% Check that the two relations (A f1 B) and (C f2 D)
% are mutually exclusive.
check_relops(A,F1,B, C,F2,D) :-
   A==D,
   flip(F2, I2),
   get_relop(F1, I2, R),
   test_relop(B, R, C).
check_relops(A,F1,B, C,F2,D) :-
   A==C,
   get_relop(F1, F2, R),
   test_relop(B, R, D).
check_relops(A,F1,B, C,F2,D) :-
   B==D,
   flip(F1,I1),
   flip(F2,I2),
   get_relop(I1, I2, R),
   test_relop(A, R, C).
check_relops(A,F1,B, C,F2,D) :-
   B==C,
   flip(F1,I1),
   get_relop(I1, F2, R),
   test_relop(A, R, D).

% Bit reversal:
flip(N, I) :- I is (N/\2'001)<<2 \/ (N/\2'010) \/ (N/\2'100)>>2.

% Opposite condition:
negate_relop(Order, OppOrder) :- OppOrder is \(Order) /\ 2'111.

% get_relop(F1, F2, Rel)
% Get the relational operator Rel such that if (N1 Rel N2)
% holds then (A F1 N1) and (A F2 N2) are mutex.
% The table of 36 entries was obtained by iterating
% bagof_get_relop.  It is based on this table of
% nine entries:
%                f2
% OpTable |   <    =    >
% --------+----------------
%      <  | never  =<   =<
%  f1  =  |  >=    \=   =<
%      >  |  >=    >=  never

get_relop(2'001, 2'001, 2'000).
get_relop(2'001, 2'010, 2'011).
get_relop(2'001, 2'011, 2'000).
get_relop(2'001, 2'100, 2'011).
get_relop(2'001, 2'101, 2'000).
get_relop(2'001, 2'110, 2'011).
get_relop(2'010, 2'001, 2'110).
get_relop(2'010, 2'010, 2'101).
get_relop(2'010, 2'011, 2'100).
get_relop(2'010, 2'100, 2'011).
get_relop(2'010, 2'101, 2'010).
get_relop(2'010, 2'110, 2'001).
get_relop(2'011, 2'001, 2'000).
get_relop(2'011, 2'010, 2'001).
get_relop(2'011, 2'011, 2'000).
get_relop(2'011, 2'100, 2'011).
get_relop(2'011, 2'101, 2'000).
get_relop(2'011, 2'110, 2'001).
get_relop(2'100, 2'001, 2'110).
get_relop(2'100, 2'010, 2'110).
get_relop(2'100, 2'011, 2'110).
get_relop(2'100, 2'100, 2'000).
get_relop(2'100, 2'101, 2'000).
get_relop(2'100, 2'110, 2'000).
get_relop(2'101, 2'001, 2'000).
get_relop(2'101, 2'010, 2'010).
get_relop(2'101, 2'011, 2'000).
get_relop(2'101, 2'100, 2'000).
get_relop(2'101, 2'101, 2'000).
get_relop(2'101, 2'110, 2'000).
get_relop(2'110, 2'001, 2'110).
get_relop(2'110, 2'010, 2'100).
get_relop(2'110, 2'011, 2'100).
get_relop(2'110, 2'100, 2'000).
get_relop(2'110, 2'101, 2'000).
get_relop(2'110, 2'110, 2'000).

bagof_get_relop(F1, F2, Rel) :-
   % Get all possibilities:
   bagof(X, op_table(F1, F2, X), OpBag), !,
   % AND them together:
   and_list(OpBag, 2'111, Rel).
bagof_get_relop(F1, F2, 2'000).
```

```
op_table(Op1, Op2, 2'000) :- lt(Op1), lt(Op2).
op_table(Op1, Op2, 2'110) :- lt(Op1), eq(Op2).
op_table(Op1, Op2, 2'110) :- lt(Op1), gt(Op2).
op_table(Op1, Op2, 2'011) :- eq(Op1), lt(Op2).
op_table(Op1, Op2, 2'101) :- eq(Op1), eq(Op2).
op_table(Op1, Op2, 2'110) :- eq(Op1), gt(Op2).
op_table(Op1, Op2, 2'011) :- gt(Op1), lt(Op2).
op_table(Op1, Op2, 2'011) :- gt(Op1), eq(Op2).
op_table(Op1, Op2, 2'000) :- gt(Op1), gt(Op2).

lt(Op) :- 2'100 is 2'100 /\ Op.
eq(Op) :- 2'010 is 2'010 /\ Op.
gt(Op) :- 2'001 is 2'001 /\ Op.

and_list([], N, N).
and_list([X|L], I, N) :- I1 is X /\ I, and_list(L, I1, N).

% Test that two terms will satisfy a given order at run-time:
% Succeeds iff Order subsumes StrongOrder.
test_relop(X, Order, Y) :-
   strong_compare(X, StrongOrder, Y),
   0 is (\(Order) /\ 2'111 /\ StrongOrder).

% Calculate the Strong Comparison between two terms A and B:
% Return the set of possible orders that A and B can have at run-time,
% given their current (partial) instantiation.
strong_compare(A, 2'010, B) :- A==B, !.
strong_compare(A, 2'111, B) :- A\==B, var(A), !.
strong_compare(A, 2'111, B) :- A\==B, var(B), !.
strong_compare(A, Order, B) :-
   A\==B, nonvar(A), nonvar(B), !,
   functor(A, An, Aa),
   functor(B, Bn, Ba),
   weak_compare(Aa/An, Ord, Ba/Bn), % Note: Arity comparison done FIRST.
   (Ord=2'010
    -> strong_compare_args(1, Aa, A, Order, B)
     ; Order=Ord
   ).

strong_compare_args(I, Aa, A, 2'010, B) :- I>Aa, !.
strong_compare_args(I, Aa, A, Order, B) :- I=<Aa, !,
   arg(I, A, Ai),
   arg(I, B, Bi),
   strong_compare(Ai, Ord, Bi),
   (Ord=2'111  % Unordered
    -> Order=Ord
     ; (Ord=2'010   % Equal
        -> I1 is I+1,
           strong_compare_args(I1, Aa, A, Order, B)
         ; Order=Ord
       )
   ).
```

```
% Calculate the Weak Comparison between two terms A and B:
% Return the order that A and B presently have.
% If A\==B the order can change at run-time due to further instantiation.
weak_compare(A, 2'100, B) :- A@<B.
weak_compare(A, 2'010, B) :- A==B.
weak_compare(A, 2'001, B) :- A@>B.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Calculations using bitmap tests:

% Get the bitmap test or bitmap type of a variable implied by a mode formula:
get_type_test(InF, X, '$test'(X,B)) :- get_type(InF, X, 2'11111111, B).

% Get the tag of X implied by a mode formula.  Fail if there is none such.
get_tag(InF, X, Tag) :-
   get_type(InF, X, 2'11111111, B),
   bitmap_type(B, Type),
   tag(Type, Tag).

get_type((T1,T2), X) --> !, get_type(T1, X), get_type(T2, X).
get_type(T, X) --> {encode_test(T, Bits, _, Y, _), X==Y}, !, and_bits(Bits).
get_type(_, _) --> [].

and_bits(A, B, C) :- C is A /\ B.

% ! Watch out: often, converting from a regular test into a bitmap test
% broadens the set of values that can succeed.  The bitmap test is a necessary
% but not sufficient replacement for the regular test.
conj_test(T1, T2, '$test'(X,B)) :-
   encode_test(T1, B1, _, X, _),
   encode_test(T2, B2, _, Y, _),
   X==Y, !,
   B is B1 /\ B2.
disj_test(T1, T2, '$test'(X,B)) :-
   encode_test(T1, B1, _, X, _),
   encode_test(T2, B2, _, Y, _),
   X==Y, !,
   B is B1 \/ B2.
not_test(T1, '$test'(X,B)) :- encode_test(T1, _, B, X, _), !.
bitmap_test(T1, '$test'(X,B)) :- encode_test(T1, B, _, X, _), !.

% Create a single bitmap test from a sequence of conjunctions, disjunctions,
% and nots.  The bitmap test means exactly the same as this sequence.
merge_test(T, '$test'(X,B)) :- exact_bitmap(T, X, B), !.
merge_test((A,B), CT) :- merge_test(A, AT), merge_test(B, BT), !,
   conj_test(AT, BT, CT).
merge_test((A;B), DT) :- merge_test(A, AT), merge_test(B, BT), !,
   disj_test(AT, BT, DT).
merge_test(\+(A), BT) :- merge_test(A, AT), !, not_test(AT, BT).

% Simplify a bitmap test:
bitmap_simplify('$test'(X,2'00000000), fail).
bitmap_simplify('$test'(X,2'11111111), true).
```

```
% Merge bitmap test into a formula:
% Fail if it is not possible.
bitmap_combine('$test'(X,B), InF, OutF) :- bitmap_combine(InF, X, B, OutF).

bitmap_combine('$test'(Y,B1), X, B2, '$test'(Y,B)) :- X==Y, !, B is B1/\B2.
bitmap_combine((T1,T2), X, B, (TB1,T2)) :- bitmap_combine(T1, X, B, TB1), !.
bitmap_combine((T1,T2), X, B, (T1,TB2)) :- bitmap_combine(T2, X, B, TB2), !.

bitmap_name_arity(2'00000010, _, '.', 2) :- !.
bitmap_name_arity(2'00100000, _,  [], 0) :- !.
bitmap_name_arity(B,          X,   X, 0) :- B=\=0, B /\ 2'10000111 =:= 0.

% Convert a test to a disjunction of tests that are easy
% to implement.  This for bitmap tests and others as well.
test_to_disj('$test'(X,B), X, Disj) :- !, bitmap_to_disj(X, B, Disj).
test_to_disj(Test, X, Disj) :-
   exact_bitmap(Test, X, B),
   bitmap_to_disj(X, B, Disj).

bitmap_to_disj(X, B, Disj) :- bitmap_to_disj(B, B, X, Disj, fail).

bitmap_to_disj(0, _, _) --> !.
bitmap_to_disj(B, AllB, X) --> {B=\=0},
   {exact_bitmap(Test, X, B1)},
   {\+complex_bitmap(Test)}, % Check that Test is easy to implement.
   {0 =:= (B1 /\ \(AllB))}, % Check that B1 is a subset of the original B.
   {NewB is B /\ \(B1)},    % Subtract the bits B1 from the current B.
   {NewB =\= B}, % Check that it had some effect.
   !,
   di(Test),
   bitmap_to_disj(NewB, AllB, X).
bitmap_to_disj(B, _, X) -->
   {error(['Could not convert bitmap ''',B,''' to a disjunction.',nl,
        'Replacing by fail.'])}.

% Tests that mean exactly the same as their bitmap:
% (In particular, they also do no binding)
exact_bitmap(Test)    :- exact_bitmap(Test, _, _).
exact_bitmap(Test, B) :- exact_bitmap(Test, _, B).

exact_bitmap(nonvar(A),      A, 2'01111111).
exact_bitmap(atom(A),        A, 2'01100000).
exact_bitmap(var(A),         A, 2'10000000).
exact_bitmap(cons(A),        A, 2'00000010).
exact_bitmap(structure(A),   A, 2'00000001).
exact_bitmap(nil(A),         A, 2'00100000).
exact_bitmap(A==B,           A, 2'00100000) :- B==[].
exact_bitmap(B==A,           A, 2'00100000) :- B==[].
exact_bitmap(negative(A),    A, 2'00001000).
exact_bitmap(nonnegative(A), A, 2'00010000).
exact_bitmap(float(A),       A, 2'00000100).
exact_bitmap(simple(A),      A, 2'11111100).
exact_bitmap(compound(A),    A, 2'00000011).
exact_bitmap(list(A),         A, 2’00100010).
exact_bitmap(atomic(A),       A, 2’01111100).
exact_bitmap(number(A),       A, 2’00011100).
exact_bitmap(integer(A),      A, 2’00011000).

% These require multiple tests to implement in BAM code:
complex_bitmap(simple(_)).
complex_bitmap(compound(_)).
complex_bitmap(list(_)).
complex_bitmap(atomic(_)).
complex_bitmap(number(_)) :- split_integer.
complex_bitmap(integer(_)) :- split_integer.
complex_bitmap(negative(_)) :- \+split_integer.
complex_bitmap(nonnegative(_)) :- \+split_integer.

bitmap_type(2’10000000, var).
bitmap_type(2’00000010, cons).
bitmap_type(2’00000001, structure).
bitmap_type(2’01100000, atom).
bitmap_type(2’00100000, atom).
bitmap_type(2’01000000, atom).
bitmap_type(2’00001000, negative) :- split_integer.
bitmap_type(2’00010000, nonnegative) :- split_integer.
bitmap_type(2’00011000, integer) :- \+split_integer.
bitmap_type(2’00001000, integer) :- \+split_integer.
bitmap_type(2’00010000, integer) :- \+split_integer.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Gathering information from a mode formula:

% Get the set of ground arguments implied by a mode formula:
ground_set(Form, GndSet) :-
   ground_bag(Form, Bag, []),
   sort(Bag, GSet),
   filter_vars(GSet, GndSet).

ground_bag((A;B)) --> !,
   {ground_set(A, NA)},
   {ground_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
ground_bag((A,B)) --> !, ground_bag(A), ground_bag(B).
ground_bag(Test) --> {encode_relop(Test, X, _, Y, arith)}, !, [X,Y].
ground_bag(Test) --> {encode_test(Test, F, _, X, _), 0=:=F/\2’10000011}, !, [X].
ground_bag(ground(X)) --> !, [X].
ground_bag(_) --> [].

% Get the set of nonvar arguments implied by a mode formula:
nonvar_set(Form, NonSet) :-
   nonvar_bag(Form, Bag, []),
   sort(Bag, NSet),
   filter_vars(NSet, NonSet).
```

```
nonvar_bag((A;B)) --> !,
   {nonvar_set(A, NA)},
   {nonvar_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
nonvar_bag((A,B)) --> !, nonvar_bag(A), nonvar_bag(B).
nonvar_bag(Test) --> {encode_test(Test, F, _, X, _), 0=:=F/\2'10000000}, !, [X].
nonvar_bag(_) --> [].

% Get the set of dereferenced arguments implied by a mode formula:
% (Only top level is dereferenced)
deref_set(Form, DrfSet) :-
   deref_bag(Form, Bag, []),
   sort(Bag, DSet),
   filter_vars(DSet, DrfSet).

deref_bag((A;B)) --> !,
   {deref_set(A, NA)},
   {deref_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
deref_bag((A,B)) --> !, deref_bag(A), deref_bag(B).
deref_bag(rderef(X)) --> !, [X].
deref_bag(deref(X)) --> !, [X].
deref_bag(_) --> [].

% Get the set of full recursively dereferenced arguments implied by a mode
% formula.  This includes rderef modes & simple modes which are dereferenced.
rderef_set(Form, RDrfSet) :-
   rderef_bag(Form, Bag, []),
   sort(Bag, RSet),
   filter_vars(RSet, RFSet),
   simple_set(Form, ASet),
   deref_set(Form, DSet),
   intersectv(ASet, DSet, ADSet),
   unionv(ADSet, RFSet, RDrfSet).

rderef_bag((A;B)) --> !,
   {rderef_set(A, NA)},
   {rderef_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
rderef_bag((A,B)) --> !, rderef_bag(A), rderef_bag(B).
rderef_bag(rderef(X)) --> !, [X].
rderef_bag(_) --> [].

% Get the set of atomic arguments implied by a mode formula:
atomic_set(Form, AtmSet) :-
   atomic_bag(Form, Bag, []),
   sort(Bag, ASet),
   filter_vars(ASet, AtmSet).

atomic_bag((A;B)) --> !,
   {atomic_set(A, NA)},
   {atomic_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
atomic_bag((A,B)) --> !, atomic_bag(A), atomic_bag(B).
atomic_bag(T) --> {encode_relop(T, X, _, Y, arith)}, !, [X,Y].
atomic_bag(T) --> {encode_test(T, F, _, X, _), 0 =:= F /\ 2'10000011}, !, [X].
atomic_bag(_) --> [].

% Get the set of simple arguments implied by a mode formula:
simple_set(Form, SmpSet) :-
   simple_bag(Form, Bag, []),
   sort(Bag, SSet),
   filter_vars(SSet, SmpSet).

simple_bag((A;B)) --> !,
   {simple_set(A, NA)},
   {simple_set(B, NB)},
   {intersectv(NA, NB, N)},
   difflist(N).
simple_bag((A,B)) --> !, simple_bag(A), simple_bag(B).
simple_bag(T) --> {encode_relop(T, X, _, Y, arith)}, !, [X,Y].
simple_bag(T) --> {encode_test(T, F, _, X, _), 0 =:= F /\ 2'00000011}, !, [X].
simple_bag(_) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Returns the subset of all possible terms that a term has AFTER
%     a successful test, and the subset of terms that lead to failure.

% Returns a flag (y or n) indicating whether the test binds its argument
% if it's a variable.

% The subset is encoded as an array of flags:
% <var,nonnil_const,nil,nonneg_int,neg_int,float,list,struc> flags.
% This array of flags represents a partition of all possible terms.
% Each flag represents a set of terms.
% The success and failure arrays do not have to be complements
% of each other.

% Notes:
% 1. The _freeze versions delay execution until A is nonvariable.
%    They have a logical meaning.  The others have no logical
%    meaning since they depend on the var/nonvar instantiation
%    of a term.
% 2. Should a float(A) test be distinguished from noninteger_real?
%    Currently it is not.
% 3. This predicate assumes that the success of arithmetic relational
%    operations implies that the operands are integers.

encode_test(A,B,C,D)   :- encode_test(A,B,C,D,_).

% Goal implies Test: (Is this correct?)
% (This predicate uses backtracking to try the tests
%  defined due to the tag definitions)
```

```
subsuming_test(Goal, Test, Type, true) :-
   encode_test(Goal, F1, _, A, n),
   type_test(Type, A, Test),
   encode_test(Test, F2, _, A, n),
   0 =:= \(F2) /\ F1.
subsuming_test(Goal, not(Test), Type, false) :-
   encode_test(Goal, F1, _, A, n),
   type_test(Type, A, Test),
   encode_test(Test, _, F2, A, n),
   0 =:= \(F2) /\ F1.

encode_test('$test'(X,Bitmap),        Bitmap,        Neg, X, n) :-
   Neg is \(Bitmap) /\ 2'11111111.
encode_test('$name_arity'(A,F,N), 2'01000000, 2'11111111, A, n) :-
   atom(F), F\==[], N==0.
encode_test('$name_arity'(A,F,N), 2'00100000, 2'11011111, A, n) :-
   F==[], N==0.
encode_test('$name_arity'(A,F,N), 2'00010000, 2'11111111, A, n) :-
   nonnegative(F),N==0.
encode_test('$name_arity'(A,F,N), 2'00001000, 2'11111111, A, n) :-
   negative(F),N==0.
encode_test('$name_arity'(A,F,N), 2'00000100, 2'11111111, A, n) :-
   number(F), \+integer(F), N==0.
encode_test('$name_arity'(A,F,N), 2'00000010, 2'11111101, A, n) :-
   atom(F), integer(N), F='.', N=:=2.
encode_test('$name_arity'(A,F,N), 2'00000001, 2'11111111, A, n) :-
   atom(F), integer(N), N>0, (F\=='.'; N=\=2).
encode_test(functor(A,F,N),       2'01000000, 2'10111111, A, y) :-
   atom(F), F\==[], N==0.
encode_test(functor(A,F,N),       2'00100000, 2'11011111, A, y) :-
   F==[], N==0.
encode_test(functor(A,F,N),       2'00010000, 2'11101111, A, y) :-
   nonnegative(F),N==0.
encode_test(functor(A,F,N),       2'00001000, 2'11110111, A, y) :-
   negative(F), N==0.
encode_test(functor(A,F,N),       2'00000100, 2'11111011, A, y) :-
   number(F), \+integer(F), N==0.
encode_test(functor(A,F,N),       2'00000010, 2'11111101, A, y) :-
   atom(F), integer(N), F='.', N=:=2.
encode_test(functor(A,F,N),       2'00000001, 2'11111111, A, y) :-
   atom(F), integer(N), N>0, (F\=='.'; N=\=2).
encode_test(functor(A,F,N),       2'00010000, 2'11101111, N, y) :-
   atomic(A).
encode_test(functor(A,F,N),       2'00010000, 2'11101111, N, y) :-
   compound(A).
encode_test(functor(A,F,N),       2'01000000, 2'10111111, F, y) :-
   atom(A), A\==[].
encode_test(functor(A,F,N),       2'00100000, 2'11011111, F, y) :-
   A==[].
encode_test(functor(A,F,N),       2'00010000, 2'11101111, F, y) :-
   nonnegative(A).
encode_test(functor(A,F,N),       2'00001000, 2'11110111, F, y) :-
   negative(A).
encode_test(functor(A,F,N),       2'01100000, 2'10011111, F, y) :-
```

```
   compound(A).
encode_test(functor(_,A,_),       2'01111100, 2'11111111, A, y).
encode_test(functor(_,_,A),       2'00010000, 2'11111111, A, y).
encode_test(A=..X,                2'01111111, 2'11111111, A, y).
encode_test(X=..A,                2'00000010, 2'11111111, A, y).
encode_test(var(A),               2'10000000, 2'01111111, A, n).
encode_test(nonvar(A),            2'01111111, 2'10000000, A, n).
encode_test(ground(A),            2'01111111, 2'10000011, A, n).
encode_test(atom(A),              2'01100000, 2'10011111, A, n).
encode_test(nil(A),               2'00100000, 2'11011111, A, n).
encode_test(integer(A),           2'00011000, 2'11100111, A, n).
encode_test(negative(A),          2'00001000, 2'11110111, A, n).
encode_test(nonnegative(A),       2'00010000, 2'11101111, A, n).
encode_test(number(A),            2'00011100, 2'11100011, A, n).
encode_test(float(A),             2'00000100, 2'11111011, A, n).
encode_test(atomic(A),            2'01111100, 2'10000011, A, n).
encode_test(list(A),              2'00100010, 2'11011101, A, n).
encode_test(cons(A),              2'00000010, 2'11111101, A, n).
encode_test(structure(A),         2'00000001, 2'11111110, A, n).
encode_test(compound(A),          2'00000011, 2'11111100, A, n).
% encode_test(composite(A),         2'00000011, 2'11111100, A, n).
encode_test(simple(A),            2'11111100, 2'00000011, A, n).
encode_test(var_freeze(A),        2'00000000, 2'01111111, A, n).
encode_test(nonvar_freeze(A),     2'01111111, 2'00000000, A, n).
encode_test(ground_freeze(A),     2'01111111, 2'00000011, A, n).
encode_test(atom_freeze(A),       2'01100000, 2'00011111, A, n).
encode_test(nil_freeze(A),        2'00100000, 2'01011111, A, n).
encode_test(integer_freeze(A),    2'00011000, 2'01100111, A, n).
encode_test(negative_freeze(A),   2'00001000, 2'01110111, A, n).
encode_test(nonnegative_freeze(A), 2'00010000, 2'01101111, A, n).
encode_test(number_freeze(A),     2'00011100, 2'01100011, A, n).
encode_test(atomic_freeze(A),     2'01111100, 2'00000011, A, n).
encode_test(list_freeze(A),       2'00100010, 2'01011101, A, n).
encode_test(cons_freeze(A),       2'00000010, 2'01111101, A, n).
encode_test(structure_freeze(A),  2'00000001, 2'01111110, A, n).
encode_test(compound_freeze(A),   2'00000011, 2'01111100, A, n).
encode_test(composite_freeze(A),  2'00000011, 2'01111100, A, n).
encode_test(simple_freeze(A),     2'11111100, 2'00000011, A, n).
encode_test(A is _, 2'00011100, 2'01111111, A, y).
encode_test(Rel,   S, I, A, n) :- sign_flags(Rel, S, I, A).
encode_test(Rel,    2'00011000, 2'00011000, A, n) :-
   \+sign_flags(Rel, _, _, _), arith_test(Rel, A, _).
encode_test(Rel,    2'00011000, 2'00011000, A, n) :-
   \+sign_flags(Rel, _, _, _), arith_test(Rel, _, A).
encode_test(A=X,   T, I, A, y) :- type_flags(X, S, I), T is 2'00000000\/S.
encode_test(X=A,   T, I, A, y) :- type_flags(X, S, I), T is 2'00000000\/S.
encode_test(A\=X,  S, F, A, n) :- type_flags(X, I, S), F is 2'10000000\/I.
encode_test(X\=A,  S, F, A, n) :- type_flags(X, I, S), F is 2'10000000\/I.
encode_test(A==X,  S, F, A, n) :- type_flags(X, S, I), F is 2'10000000\/I.
encode_test(X==A,  S, F, A, n) :- type_flags(X, S, I), F is 2'10000000\/I.
encode_test(A\==X, T, I, A, n) :- type_flags(X, I, S), T is 2'10000000\/S.
encode_test(X\==A, T, I, A, n) :- type_flags(X, I, S), T is 2'10000000\/S.
encode_test(not(G), S, I, A, T) :- nonvar(G), encode_test(G, I, S, A, T).
encode_test( \+(G), S, I, A, n) :- nonvar(G), encode_test(G, I, S, A, _).
```

```
% Flags and inverses from equality tests:
% Fails if X is a variable, since X can be any type at run-time.
% Does not take variable flag into account.
type_flags(X, 2'01000000, 2'01111111) :- atom(X), X\==[].
type_flags(X, 2'00100000, 2'01011111) :- nil(X).
type_flags(X, 2'00010000, 2'01111111) :- nonnegative(X).
type_flags(X, 2'00001000, 2'01111111) :- negative(X).
type_flags(X, 2'00000100, 2'01111111) :- number(X), \+integer(X).
type_flags(X, 2'00000010, 2'01111111) :- cons(X).
type_flags(X, 2'00000001, 2'01111111) :- structure(X).

% When the sign of a relational operator can be determined:
sign_flags( A>N, 2'00010000, 2'00001000, A) :- nonnegative(N).
sign_flags(A>=N, 2'00010000, 2'00001000, A) :- nonnegative(N).
sign_flags( N<A, 2'00010000, 2'00001000, A) :- nonnegative(N).
sign_flags(N=<A, 2'00010000, 2'00001000, A) :- nonnegative(N).
sign_flags( N>A, 2'00001000, 2'00010000, A) :- nonpositive(N).
sign_flags(N>=A, 2'00001000, 2'00010000, A) :- negative(N).
sign_flags( A<N, 2'00001000, 2'00010000, A) :- nonpositive(N).
sign_flags(A=<N, 2'00001000, 2'00010000, A) :- negative(N).

% can_overlap(Time, S1, F1, S2, F2, Binds1, Binds2)
% Succeeds if Test1 and Test2 can succeed together:
% This takes into account the possibility of binding by one of the tests.
% It does NOT take into account the trapping behavior of relational tests.
can_overlap(     _, S1,  _, S2,  _, _, _) :- 0 =\= S1 /\ S2.
can_overlap(  left, S1, F1, S2, F2, _, y) :- 0 =\= 2'10000000 /\ S1.
can_overlap( right, S1, F1, S2, F2, y, _) :- 0 =\= 2'10000000 /\ S2.
can_overlap(before, S1, F1, S2, F2, _, y) :- 0 =\= 2'10000000 /\ S1.
can_overlap(before, S1, F1, S2, F2, y, _) :- 0 =\= 2'10000000 /\ S2.

% Complicated version that's not right:
% This happens if after1 /\ before2 =\= 0 or before1 /\ after2 =\= 0.
% This takes into account the execution order and the possibility
% of binding happening by either Test1 or Test2 or both.
% can_overlap(     _, S1,  _, S2,  _, _, _) :- 0 =\= S1 /\ S2.
% can_overlap(  left, S1, F1, S2, F2, _, _) :- left_overlap(S1, F1, S2, F2).
% can_overlap(  left, S1, F1, S2, F2, _, y) :- 0 =\= 2'10000000 /\ F1.
% can_overlap( right, S1, F1, S2, F2, _, _) :- right_overlap(S1, F1, S2, F2).
% can_overlap( right, S1, F1, S2, F2, y, _) :- 0 =\= 2'10000000 /\ F2.
% % can_overlap( after, S1, F1, S2, F2, _, _) :- left_overlap(S1, F1, S2, F2).
% % can_overlap( after, S1, F1, S2, F2, _, _) :- right_overlap(S1, F1, S2, F2).
% % can_overlap(before, S1, F1, S2, F2, _, _) :- left_overlap(S1, F1, S2, F2).
% % can_overlap(before, S1, F1, S2, F2, _, _) :- right_overlap(S1, F1, S2, F2).
% can_overlap(before, S1, F1, S2, F2, _, y) :- 0 =\= 2'10000000 /\ F1.
% can_overlap(before, S1, F1, S2, F2, y, _) :- 0 =\= 2'10000000 /\ F2.

% These formulas take into account what happens when the predicates
% can abort on a type not listed in the success & fail flags:
% (This occurs for relational operations)
left_overlap(S1, F1, S2, F2)  :- 0 =\= S1 /\ (S2 \/ \(S2 \/ F2)).
right_overlap(S1, F1, S2, F2) :- 0 =\= S2 /\ (S1 \/ \(S1 \/ F1)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Returns the functor and arity of a variable at run-time, if it can be found.
% This predicate gives more precise information about compound terms
% than encode_test’s flag bits.  It tells whether the name and arity are
% equal or not equal to a given variable or atom.

% What is represented by F & A:
% Flag = true: F AND A.
% Flag = false: F OR A (F & A being negative terms here).

encode_name_arity(’$test’(X,B), Y, F, A, true) :-
   bitmap_name_arity(B,          Y,   F, A).
encode_name_arity(X=..Y, Y, ’.’, 2, true).
encode_name_arity(  X=Y, X, F, A, true) :- nonvar(Y), functor(Y, F, A).
encode_name_arity(  X=Y, Y, F, A, true) :- nonvar(X), functor(X, F, A).
encode_name_arity( X==Y, X, F, A, true) :- nonvar(Y), functor(Y, F, A).
encode_name_arity( X==Y, Y, F, A, true) :- nonvar(X), functor(X, F, A).
encode_name_arity(X=:=Y, X, F, A, true) :- integer(Y), functor(Y, F, A).
encode_name_arity(X=:=Y, Y, F, A, true) :- integer(X), functor(X, F, A).
% These are wrong:
% encode_name_arity( X\=Y, X, F, A, false) :- nonvar(Y), functor(Y, F, A).
% encode_name_arity( X\=Y, Y, F, A, false) :- nonvar(X), functor(X, F, A).
encode_name_arity(X=\=Y, X, not(F), not(A), false) :- integer(Y),functor(Y,F,A).
encode_name_arity(X=\=Y, Y, not(F), not(A), false) :- integer(X),functor(X,F,A).
encode_name_arity(functor(X,F,A), X, F, A, true).
encode_name_arity(functor(X,F,A), F, F, 0, true) :- var(X).
encode_name_arity(functor(X,F,A), F, G, 0, true) :- nonvar(X), functor(X, G, _).
encode_name_arity(functor(X,F,A), A, A, 0, true) :- var(X).
encode_name_arity(functor(X,F,A), A, B, 0, true) :- nonvar(X), functor(X, _, B).
encode_name_arity(’$name_arity’(X,F,A), X, F, A, true).
encode_name_arity(atom(X), X, X, 0, true).
encode_name_arity(nil(X), X, [], 0, true).
encode_name_arity(integer(X), X, X, 0, true).
encode_name_arity(negative(X), X, X, 0, true).
encode_name_arity(nonnegative(X), X, X, 0, true).
encode_name_arity(number(X), X, X, 0, true).
encode_name_arity(atomic(X), X, X, 0, true).
encode_name_arity(cons(X), X, ’.’, 2, true).
encode_name_arity(not(P), X, NotF, NotA, NotJ) :-
   nonvar(P),
   \+invalid_negation(P),
   encode_name_arity(P, X, F, A, J),
   negate(F, NotF),
   negate(A, NotA),
   negate_boolean(J, NotJ).

% Negating these predicates does not imply anything about the main functor
% since the failure could be due to an argument.
invalid_negation(X=..Y).
invalid_negation(X=Y).
invalid_negation(X\=Y).
invalid_negation(X==Y).
invalid_negation(X\==Y).
```

```
negate(N,      X) :- nonvar(N), N=not(X), !.
negate(N, not(N)) :- var(N), !.
negate(N, not(N)) :- \+N=not(_), !.

% Given the name & arity information of two predicates,
% and whether the known values are logically positive or negative,
% succeeds for two tests that are mutually exclusive:
check_name_arity(true,  true, Na, Aa, Nb, Ab) :-
   ( check_different(Na, Nb)
   ; check_different(Aa, Ab)
   ), !.
check_name_arity(true, false, Na, Aa, Nb, Ab) :-
   check_different(Na, Nb),
   check_different(Aa, Ab).
check_name_arity(false, true, Na, Aa, Nb, Ab) :-
   check_different(Na, Nb),
   check_different(Aa, Ab).

% Succeeds for two atoms or variables that are guaranteed to be different at
% run-time.
% (i.e. variables don’t match against atoms since they could be the same,
%  but X and not(X) are different for all X.)
check_different(X, Y) :- atomic(X), atomic(Y), X\==Y.
check_different(X, Y) :- nonvar(X), X=not(Z), atomic(Z), Z==Y.
check_different(X, Y) :- nonvar(Y), Y=not(Z), atomic(Z), X==Z.

% Get opposite logical combiner:
negate_boolean(true, false).
negate_boolean(false, true).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

peephole.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Low-level optimizations:

% Notes:
% 1. A transitive closure is done on all the intermediate peephole steps.
% 2. The sequence move(r(1),r(7)) ... move(r(7),r(2)) can be optimized in two
%    steps: (1) replace second r(7) by r(1) as above, (2) see that r(7) is
%    never read, so remove it.  This second part requires a backward pass.
% 3. Order of the peephole optimizations is important.  For example,
%    peep_inst must be done before instantiation of variables.
% 4. Don’t repeat dereferences, i.e. remember when something has been derefed.
%    This is similar to synonym.
% 5. Propagate instruction sequences that are common between all destinations
%    of a branch to before the branch if they don’t affect the branch outcome.
%    Do this BEFORE dead code elimination.
%    For a choice-block must look at ALL choices.
% 6. Propagate instruction sequences that are only used in one direction of a
%    branch to inside the branch.  This occurs fairly often in if-then-elses.
% 7. Coalesce identical basic blocks.  How often are basic blocks slightly
%    different in form, but identical in semantics?
% 8. Even if the gain is very small, patterns that achieve it should be put in.
%    Philosophy: add patterns that I’ve seen in real code.

peephole(NaAr, Code, Code, nopeep) :- !.
peephole(NaAr, Code, ICode, peep) :-
   \+compile_option(peep), !,
   peep_flat(Code, FCode),
   inst_labl(NaAr, FCode, ICode).
peephole(NaAr, Code, PCode, peep) :-
   compile_option(peep), !,
        write_debug(’Before peephole optimization:’), nl_debug,
        xpeephole(NaAr, Code, PCode),
        (compile_option(debug) -> write_code(Code), nl; true),
        write_debug(’After peephole optimization:’).

xpeephole(NaAr) -->
   write_debug(’peephole started’),
   stats(p,1),
   peep_flat,       % Remove nesting of switch(unify)
   stats(p,2),
   inst_labl(NaAr),    % Instantiate labels.
   stats(p,3),      % At this point the code is ground.
   peep_simp,       % First optimization (to reduce code size).
   % Doing this reduces the number of duplicate basic blocks in prover.pl:
   % peep_inst(nocall),
   stats(p,4),
   peep_closure(NaAr), % Closure on all peephole optimizations.
   stats(p,5),
   !.
```

```
peep_closure(NaAr, Code, OutCode) :-
   peep_seq(NaAr, Code, SCode),
   peep_end_closure(NaAr, Code, SCode, OutCode).

% Repeat as long as there is a change:
peep_end_closure(NaAr, Code, SCode, PCode) :- Code=SCode, !, PCode=Code.
peep_end_closure(NaAr, _, SCode, PCode) :- peep_closure(NaAr, SCode, PCode).

% Is there an optimal ordering among these steps?
% For best results:
% 1. No peep_inst before peep_uniq (otherwise fewer duplicate blocks).
% 2. No peep_inst before peep_jump_closure (fewer inline calls possible).
% 3. Peep_dead just after peep_uniq to remove basic blocks that aren’t
%    jumped to any more--this avoids Re-inserting short duplicate basic blocks
%    that have just been replaced by jumps.  This gives a significant codesize
%    reduction (>10% for chat).
peep_seq(NaAr) -->
   stats(p1,1),
   peep_uniq,
   stats(p1,2),
   peep_dead,
   stats(p1,3),
   peep_jump_closure(NaAr),
   stats(p1,4),
   peep_labl,
   stats(p1,5),
   synonym,
   stats(p1,6),
   peep_inst(call),
   stats(p1,7),
   !.

% Instantiate all labels in the code to numbered structures:
inst_labl(NaAr, Code, Code) :- varset(Code, Vars), inst_labl_list(Vars, NaAr).

inst_labl_list([], _).
inst_labl_list([l(NA,I)|Vars], NA) :- gennum(I), inst_labl_list(Vars, NA).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Remove duplicate basic blocks: an optimization to reduce code size.

% Replace all duplicate basic blocks except the last one by a jump
% to the last one.  This respects the rule that all jumps go forward.

% Could improve this by improving the comparison of two basic blocks.
% If they both contain local labels, then the local labels don’t have
% to be the same, as long as they correspond.

peep_uniq(Code, UCode) :-
   make_large_blocks(Code, Blocks),
   reverse(Blocks, Rev),
   make_unique_array(Rev, UA, 0, F),
   % Only map blocks if there exists a duplicate:
   (F>0
   -> comment(['Number of duplicate blocks = ',F]),
      map_blocks(Code, UA, UCode)
   ;  Code=UCode
   ).

% Make an array indexed by basic block that contains one entry per
% unique block.  The last blocks are inserted first.  Also count the
% number of duplicated basic blocks.
make_unique_array([], UA, InF, InF) :- seal(UA).
make_unique_array([L-B|Blocks], UA, InF, OutF) :-
   get(UA, B, X), !,
   (var(X) -> X=L, MidF=InF; MidF is InF+1),
   make_unique_array(Blocks, UA, MidF, OutF).
make_unique_array([_|Blocks], UA, InF, OutF) :-
   make_unique_array(Blocks, UA, InF, OutF).

% Replace all duplicated blocks in the code by jumps to the last occurrence.
% No other change is done.
map_blocks([], _, []).
map_blocks([label(L1)|Code], UA, UCode) :-
   first_block(Code, B, Rest),
   fget(UA, B, L2),
   L1\==L2, !,
   UCode=[label(L1),jump(L2)|UC],
   map_blocks(Rest, UA, UC).
map_blocks([Branch|Code], UA, UCode) :-
   branch(Branch), \+distant_branch(Branch),
   \+Code=[label(_)|_],
   first_block(Code, B, Rest),
   fget(UA, B, L), !,
   comment(['Able to remove a block after ',Branch]),
   UCode=[Branch,jump(L)|UC],
   map_blocks(Rest, UA, UC).
map_blocks([I|Code], UA, [I|UCode]) :-
   map_blocks(Code, UA, UCode).

% Collect all blocks that are large enough to make it worthwhile to remove
% the duplicates:
% Don't collect any blocks that have labels in them; they will almost surely
% not have any duplicates worth collecting.  Also, collecting these blocks
% will make the size of stuff that bagof collects go up as O(codesizeˆ2) for
% long stretches of nonbranching code with many labels.  This causes bagof
% to choke if there are 100's of instructions.
make_large_blocks([], []).
make_large_blocks([label(L)|Code], [L-Block|Blocks]) :-
   first_block(Code, Block),
   \+has_label(Block),
   \+small_block(Block),
   !,
   make_large_blocks(Code, Blocks).
make_large_blocks([_|Code], Blocks) :- make_large_blocks(Code, Blocks).
```

```
% Bagof uses O(codesizeˆ2) memory in this version:
% make_large_blocks(Code, Blocks) :-
%  bagof(Lbl-Block,
%        BlockEntryˆ
%      (basic_block(Code,[label(Lbl)|BlockEntry]),
%       first_block(BlockEntry, Block),
%       \+has_label(Block),
%       \+small_block(Block)
%      ),
%        Blocks), !.
% make_large_blocks(Code, []).

small_block([jump(_)]).
small_block([return]).
small_block([fail]).

has_label([label(_)|_]).
has_label([_|Code]) :- has_label(Code).

% count_label([label(_)|C], M, N) :- !, M1 is M+1, count_label(C, M1, N).
% count_label([_|Code], M, N) :- count_label(Code, M, N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Jump & basic block optimization:

% Rearrange basic blocks so that the number of jumps is minimized.
% When basic blocks are copied, the labels in them are renamed so
% there are no duplicate labels.
% Current implementation takes O(n log n) time.

% Peep_jump must be done before peep_inst(call), and it must do a closure.
% This is because peep_inst(call) does the transformation
% (call,deallocate,return -> deallocate,jump) which makes basic block
% rearrangement more difficult, in fact the current implementation of
% peep_jump would be incorrect because of it. (is this true?)

% Definition of basic block: any block of code starting with a label and
% ending with a distant branch and that contains no distant branches except
% for the last instruction.

% Notes:
% 1. What about jumps that go nowhere (as in unify optimization)?

peep_jump_closure(NA, Code, OutCode) :-
   peep_jump(NA, Code, SCode),
   peep_jump_end_closure(NA, Code, SCode, OutCode).

% Repeat as long as there is a change:
peep_jump_end_closure(NA, Code, SCode, PCode) :- Code=SCode, !, PCode=Code.
peep_jump_end_closure(NA,_,SCode,PCode) :- peep_jump_closure(NA, SCode, PCode).

peep_jump(NA, Code, JCode) :-
   make_block_array(Code, BArray),
   non_empty_array(BArray), !,
   get_uniq_labels(Code, UArray),
   rearr_jump(Code, NA, UArray, BArray, JCode).
peep_jump(_, Code, Code).

% *** Get all labels that are only jumped to once:
get_uniq_labels(Code, UArray) :-
   get_label_bag(Code, LblBag, []),
   filter_uniq(LblBag, UniqSet),
   length(UniqSet, N),
   comment(['Number of unique labels = ',N]),
   create_array(UniqSet, UArray).

get_label_bag([]) --> !.
get_label_bag([B|Code]) --> branch(B), !, get_label_bag(Code).
get_label_bag([_|Code]) --> get_label_bag(Code).

get_label_set(Code, LSet) :- get_label_bag(Code, LBag, []), sort(LBag, LSet).

% *** The heart of peep_jump:
% 1. replace a jump by its basic block if the block is only jumped to once.
% 2. replace a jump by its basic block if the block is short.
% 3. replace a call by the predicate called if that's straightline code.
% 4. replace a cond. branch to a cond. branch by a new cond. branch.
% 5. replace the label of any branch to a jump or to fail by the destination.
rearr_jump([], _, _, _, []).
rearr_jump([jump(J)|Code], NA, UArray, BArray, RCode) :-
   \+Code=[label(J)|_],
   fget(BArray, J, BlockEntry), \+no_opt_block(BlockEntry),
   first_block(BlockEntry, Block),
   (fget(UArray, J, _) ; short_block(Block)),
   % Block should not end in the jump itself, otherwise an endless series
   % of loop unrolling may result:
   \+last(Block,jump(J)),
   !,
   insert_block(Block, NA, RCode, RC),
   rearr_jump(Code, NA, UArray, BArray, RC).
rearr_jump([call(J)|Code], NA, UArray, BArray, RCode) :-
   \+Code=[return,label(J)|_],
   fget(BArray, J, BlockEntry),
   first_block(BlockEntry, Block),
   returns_or_fails(Block, Bl),
   !,
   insert_block(Bl, NA, RCode, RC),
   rearr_jump(Code, NA, UArray, BArray, RC).
% Replacing next two clauses by this causes drop into query mode on verb_form/4.
%% rearr_jump([Branch|Code], NA, UArray, BArray, RCode) :-
%% map_branch(Branch, L1, NewBranch, L2),
%% fget(BArray, L1, Block),
%% ( Block=[jump(L2)], RCode=[NewBranch|RC]
%% ; Block=[fail], L2=fail, RCode=[NewBranch|RC]
%% ; merge_branch(Branch, Block, Lbls, RCode, RC),
%%   inst_labl_list(Lbls, NA)
%% ), !,
%% rearr_jump(Code, NA, UArray, BArray, RC).
rearr_jump([Branch|Code], NA, UArray, BArray, RCode) :-
   branch(Branch, [L]), fget(BArray, L, BlockEntry),
   merge_branch(Branch, BlockEntry, Lbls, RCode, RC), !,
   inst_labl_list(Lbls, NA),
   rearr_jump(Code, NA, UArray, BArray, RC).
rearr_jump([Branch|Code], NA, UArray, BArray, RCode) :-
   map_branch(Branch, L1, NewBranch, L2),
   fget(BArray, L1, BlockEntry),
   ( BlockEntry=[jump(L2)|_]
   ; BlockEntry=[fail|_], L2=fail
   ), !,
   RCode=[NewBranch|RC],
   rearr_jump(Code, NA, UArray, BArray, RC).
rearr_jump([I|Code], NA, UArray, BArray, [I|RC]) :-
   rearr_jump(Code, NA, UArray, BArray, RC).

% *** Optimize some branches to branches:
% Replace a branch to a block beginning with a branch by another branch.
% merge_branch(Branch, DestBlock, NewLabels, ReplaceBranch, Link)
merge_branch(test(ne,Tvar,R,L), [switch(T,R,fail,L2,L3)|_], [L1]) -->
   [switch(T,R,L1,L2,L3),label(L1)], {tag(var,Tvar)}.
% merge_branch(B1, [B2|_], _) -->
%  {branch(B1), branch(B2)},
%  {comment(['Branch ',B1,' jumps to ',B2])},
%  {fail}.

% *** The block is part of writemode structure creation:
no_opt_block([pragma(push(cons))|_]).
no_opt_block([pragma(push(structure(_)))|_]).
% The block is part of unification with an atom:
no_opt_block([unify_atomic(_,_,fail)|_]).

% *** Create an array of basic blocks:
make_block_array(Code, BArray) :-
   make_blocks(Code, Blocks),
   create_array(Blocks, BArray).

% *** Make a list of all basic blocks' entry points:
make_blocks([], []).
make_blocks([label(L)|Code], [L-Code|Blocks]) :- !,
   make_blocks(Code, Blocks).
make_blocks([_|Code], Blocks) :- make_blocks(Code, Blocks).

% Bagof uses O(codesize^2) memory in this version:
% make_blocks(Code, Blocks) :-
%  bagof(Lbl-BlockEntry,
%        Lbl^(basic_block(Code,[label(Lbl)|BlockEntry])),
%        Blocks), !.
% make_blocks(Code, []).

% *** Return all basic blocks' entry points on backtracking:
% This routine returns the basic block's entry point in the code,
% not the basic block itself.  This is important for the bagofs
% which call this routine because otherwise the size of the data
% that is collected is O(codesizeˆ2) which blows up for long stretches
% of nonbranching code (100’s of instructions) with many labels.
% basic_block([label(L)|Code], [label(L)|Code]).
basic_block(Block,    Block) :- Block=[label(_)|_].
basic_block([_|Code], Block) :- basic_block(Code, Block).

% Return the basic block starting the code:
first_block(Code, Block) :- first_block(Code, Block, _).

first_block([I], [I], []) :- !.
first_block([I|Code], [I|Block], Rest) :- \+end_block(I), !,
   first_block(Code, Block, Rest).
first_block([I|Rest], [I], Rest) :- end_block(I), !.

% Mark the last instruction of a basic block:
end_block(End) :- distant_branch(End).

% *** Copy a basic block and rename the local labels:
insert_block(Block, NA, Code, Link) :-
   get_labels(Block, Lbls, []),
   assoc_labels(Lbls, ALbls),
   inst_labl_list(ALbls, NA),
   replace_labels(Block, Lbls, ALbls, Code, Link).

get_labels([]) --> !.
get_labels([label(L)|Block]) --> !, [L], get_labels(Block).
get_labels([_|Block]) --> get_labels(Block).

assoc_labels([], []) :- !.
assoc_labels([_|L], [_|A]) :- assoc_labels(L, A).

replace_labels([], _, _) --> !.
replace_labels([I|Block], Lbls, ALbls) -->
   replace_labels_one(I, Lbls, ALbls),
   replace_labels(Block, Lbls, ALbls).

replace_labels_one(label(L), Lbls, ALbls, [label(A)|Link], Link) :-
   memberv2(L, Lbls, A, ALbls), !.
replace_labels_one(I, Lbls, ALbls, [J|Link], Link) :-
   map_branches(I, ILbls, J, JLbls), !,
   map_terms(Lbls, ALbls, ILbls, JLbls).
replace_labels_one(I, _, _, [I|Link], Link).

% *** Succeeds if the block returns or fails and remains straight-line code.
% (It removes the return, but keeps the fail.)
returns_or_fails(Block, Bl) :- returns_or_fails(Block, Bl, []).

returns_or_fails([fail]) --> !, [fail].
returns_or_fails([return]) --> !.
returns_or_fails([call(A)|Block]) --> !, [call(A)], returns_or_fails(Block).
returns_or_fails([I|Block]) --> {\+branch(I)}, !, [I], returns_or_fails(Block).
returns_or_fails([I|Block]) --> {pure_branch(I,Lbls), all_fails(Lbls)}, !,
   [I],
   returns_or_fails(Block).

% Succeeds if the block is short enough:
short_block(List) :- compile_option(short_block(N)), shorter_than(List, N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dead code elimination:

% Notes:
% 1. This code needs to know all the instructions that can branch.
% 2. This version does not remove any code inside of a switch instruction
%    generated by unification code.  This is OK since such a switch is
%    self-contained.
% 3. Speed = O( (number of labels) * (code size) ).  This can be reduced
%    to log(number of labels) if profiling shows it to be necessary.

peep_dead(Code, DCode) :-
   gather_closure(Code, Lbls),
   peep_dead(Code, Lbls, DCode).

peep_dead([], _, []).
peep_dead([Instr|Code], Lbls, [Instr|DCode]) :- distant_branch(Instr), !,
   to_next_label(Code, Lbls, MCode),
   peep_dead(MCode, Lbls, DCode).
peep_dead([Instr|Code], Lbls, [Instr|DCode]) :-
   peep_dead(Code, Lbls, DCode).

% Do closure calculation to find all labels that are reached through any
% number of jumps in succession:
gather_closure(Code, Lbls) :- gather_closure(Code, [], Lbls).

gather_closure(Code, InL, OutL) :-
   gather_labels(Code, InL, MidL),
   sort(MidL, SetL),
        gather_end_closure(Code, InL, SetL, OutL).

% Repeat as long as there is a change:
gather_end_closure(Code, InL, MidL, OutL) :- InL=MidL, !, OutL=MidL.
gather_end_closure(Code, InL, MidL, OutL) :- gather_closure(Code, MidL, OutL).

% Follow labels in one pass to find reached jumps:
gather_labels([], L, L).
gather_labels([Instr|Code], InL, OutL) :- branch(Instr, MidL, InL), !,
   next_code(Instr, Code, NCode, MidL),
   gather_labels(NCode, MidL, OutL).
gather_labels([label(L)|Code], InL, OutL) :- member(L, InL), !,
   gather_labels(Code, InL, OutL).
gather_labels([_|Code], InL, OutL) :-
   gather_labels(Code, InL, OutL).

next_code(Instr, Code, NCode, LblSF) :- distant_branch(Instr), !,
   to_next_label(Code, LblSF, NCode).
next_code(_, Code, Code, _).
```

```
% Remove all instructions up to a label that is a jump destination:
to_next_label([], _, []).
to_next_label([Instr|Code], LblSF, DCode) :-
   (Instr=label(L), member(L, LblSF)
   -> DCode=[Instr|Code]
   ;  to_next_label(Code, LblSF, DCode)
   ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Remove unused labels:

% This algorithm is O( code_size*log(num_labels) ).

peep_labl(Code, LCode) :-
   branch_dest_bag(Code, Labels),
   list_to_key(Labels, KeyLabels),
   create_array(KeyLabels, LArray),
   remove_label_inst(Code, LArray, LCode).

remove_label_inst([], _, []).
remove_label_inst([Instr|Code], LArray, LCode) :-
   ((Instr=label(Lbl), \+fget(LArray, Lbl, _))
    -> LCode=MCode
    ;  LCode=[Instr|MCode]
   ),
   remove_label_inst(Code, LArray, MCode).

branch_dest_bag([], []).
branch_dest_bag([Instr|Code], Labels) :-
   (branch(Instr, Labels, L)
    -> branch_dest_bag(Code, L)
    ;  branch_dest_bag(Code, Labels)
   ).

branch_dest_set(Code, Set) :- branch_dest_bag(Code, Bag), sort(Bag, Set).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Simple instruction level peephole optimization:

% Optimized to do transitive closure in one pass.

% Possible improvement: look at ’relevant’ instructions instead of
% adjacent instructions, skipping ’irrelevant’ instructions.
% This increases the power of the optimization.

% Added basic block array to make it possible to follow jumps & calls.
% This is a first step towards more complex propagation optimizations.

% CallFlag = call: convert call instructions to jumps wherever possible.
% CallFlag = nocall: do not convert call instructions to jumps.
```

```
peep_inst(CallFlag, Code, PCode) :-
   make_block_array(Code, BArray),
   peep_inst(Code, BArray, CallFlag, PCode, []).

% Simple peephole optimization:
peep_simp([], []).
peep_simp([I|Code], PCode) :- peep_simp(I, Code, PCode).

peep_simp(move(R,R), Code, PCode) :- !, peep_simp(Code, PCode).
peep_simp(nop, Code, PCode) :- !, peep_simp(Code, PCode).
peep_simp(I, Code, [I|PCode]) :- peep_simp(Code, PCode).

peep_inst([], _, _) --> [].
% Remove choice(1/N) if is followed by a cut:
peep_inst([choice(1/N,_,_)|Code], BA, F) -->
   {success_to_cut(Code, BA)}, !,
   peep_inst(Code, BA, F).
% Replace choice(I/N) (1<I<N) by choice(N/N) if is followed by a cut:
% (A more complete opt. would go back & redo all register sets)
peep_inst([choice(I/N,R,_)|Code], BA, F) --> {1<I, I<N},
   {success_to_cut(Code, BA)}, !,
   peep_inst([choice(N/N,R,fail)|Code], BA, F).
peep_inst([call(X)|Code], BA, call) --> {rearr_dealloc(Code, X, C)}, !,
   peep_inst(C, BA, call).
peep_inst([jump(L)|Code], BA, F) --> {lbl(L), contains_label(Code, L)}, !,
   peep_inst(Code, BA, F).
peep_inst([I,J,K|Code], BA, call) --> {cpeep3(I, J, K, C, Code)}, !,
   peep_inst(C, BA, call).
peep_inst([H,I,J,K|Code], BA, F) --> {peep3r(I, J, K, C, Code)}, !,
   peep_inst([H|C], BA, F).
peep_inst([I,J,K|Code], BA, F) --> {peep3r(I, J, K, C, Code)}, !,
   peep_inst(C, BA, F).
peep_inst([I,J,K|Code], BA, F) --> {peep3(I, J, K, C, Code)}, !,
   peep_inst(C, BA, F).
peep_inst([H,I,J|Code], BA, F) --> {peep2r(I, J, C, Code)}, !,
   peep_inst([H|C], BA, F).
peep_inst([I,J|Code], BA, F) --> {peep2r(I, J, C, Code)}, !,
   peep_inst(C, BA, F).
peep_inst([I,J|Code], BA, F) --> {peep2(I, J, C, Code)}, !,
   peep_inst(C, BA, F).
peep_inst([H,I|Code], BA, F) --> peep1r(I), !,
   peep_inst([H|Code], BA, F).
peep_inst([I|Code], BA, F) --> peep1r(I), !,
   peep_inst(Code, BA, F).
peep_inst([I|Code], BA, F) --> peep1(I), !,
   peep_inst(Code, BA, F).
peep_inst([I|Code], BA, F) --> % Default case.
   [I],
   peep_inst(Code, BA, F).

% Code starts with a sequence of pseudo-instructions and the label L:
contains_label([label(L)|_], L) :- !.
contains_label([label(_)|Code], L) :- contains_label(Code, L).
contains_label([pragma(_)|Code], L) :- contains_label(Code, L).
```

```
% Optimizations which convert calls to jumps:
% All such are done only with 'call' flag.

% Tail-call optimization of a call followed by deallocate(s) and return:
% (call+deallocate(s)+return) --> (deallocate(s)+jump)
rearr_dealloc([deallocate(N)|Code], X, [deallocate(N)|RCode]) :- !,
   rearr_dealloc(Code, X, RCode).
rearr_dealloc([return|Code], X, [jump(X)|Code]).

% cpeep2r(call(A), return) --> [jump(A)].

% cpeep3r(call(A), nop, return) --> [jump(A)].
% cpeep3r(call(A), deallocate(N), return) --> [deallocate(N),jump(A)].

cpeep3(call(A), label(B), return) --> [jump(A),label(B),return].

% Single instruction & reduce:
peep1r(               nop) --> [].
peep1r(         cut(r(b))) --> [].
peep1r(         move(X,X)) --> [].
peep1r(  unify(X,X,_,_,_)) --> [].
peep1r(      equal(X,X,_)) --> [].
% Can't do this because pointer offsets may depend on the pad:
% peep1r(          pad(M)) --> [], {align(A), 0 is M mod A}.
peep1r(          pad(0)) --> [].
peep1r(pragma(align(_,1))) --> [].
peep1r(    jump(ne,R,R,_)) --> [].
peep1r( jump_nt(ne,R,R,_)) --> [].
peep1r(   move(_,r(void))) --> [].
peep1r(  deref(_,r(void))) --> [].

% Single instruction:
peep1(simple_call('$sll'/2)) --> {vlsi_plm}, [sll].
peep1(simple_call('$sra'/2)) --> {vlsi_plm}, [sra].
peep1(deref(X,X)) --> {\+write_once}, [deref(X)].
peep1(move(X,Y)) -->
   {X\==Y},
   {s_oper(X, PX), s_oper(Y, PY)},
   [move(PX,PY)].
peep1(equal(X,Y,L)) -->
   {s_oper(X, PX), s_oper(Y, PY)},
   [equal(PX,PY,L)].
peep1(unify(X,Y,Tx,Ty,L)) -->
   {s_oper(X, PX), s_oper(Y, PY)},
   [unify(PX,PY,Tx,Ty,L)].
peep1(switch(unify,T,A,L1,L2,F)) -->
   {peep_inst(L1, P1), peep_inst(L2, P2)},
   [switch(unify,T,A,P1,P2,F)].
peep1(jump(fail)) -->
   [fail].
peep1(switch(_,_,fail,fail,fail)) -->
   [fail].
peep1(switch(_,_,L,L,L)) --> {L\==fail},
    [jump(L)].
peep1(switch(_,R,fail,L,L)) -->
    [test(ne,Tvar,R,L),fail], {tag(var,Tvar)}.
peep1(switch(_,R,L,fail,fail)) -->
    [test(eq,Tvar,R,L),fail], {tag(var,Tvar)}.
peep1(switch(T,R,fail,L,fail)) -->
    [test(eq,T,R,L),fail].
peep1(switch(T,R,L,fail,L)) -->
    [test(ne,T,R,L),fail].
peep1(pad(M)) --> {align(A), K is M mod A, K<M, K=\=0}, [pad(K)].
peep1(jump(eq,R,R,fail)) --> [fail].
peep1(jump_nt(eq,R,R,fail)) --> [fail].
% Need to extend this eventually to handle overflows:
peep1(add(I,J,R)) --> {integer(I), integer(J), K is I+J}, [move(K,R)].
peep1(sub(I,J,R)) --> {integer(I), integer(J), K is I-J}, [move(K,R)].
peep1(mul(I,J,R)) --> {integer(I), integer(J), K is I*J}, [move(K,R)].
peep1(div(I,J,R)) --> {integer(I), integer(J), K is I//J}, [move(K,R)].
peep1(and(I,J,R)) --> {integer(I), integer(J), K is I /\ J}, [move(K,R)].
peep1( or(I,J,R)) --> {integer(I), integer(J), K is I \/ J}, [move(K,R)].
peep1(xor(I,J,R)) --> {integer(I), integer(J), xor(I, J, K)}, [move(K,R)].
% peep1(  not(I,R)) --> {integer(I), integer(J), K is \(I)}, [move(K,R)].
peep1(sll(I,J,R)) --> {integer(I), integer(J), K is I<<J}, [move(K,R)].
peep1(sra(I,J,R)) --> {integer(I), integer(J), K is I>>J}, [move(K,R)].
% User instructions: (for macro definitions)
peep1(add_nt(I,J,R)) --> {integer(I), integer(J), K is I+J}, [move(K,R)].
peep1(sub_nt(I,J,R)) --> {integer(I), integer(J), K is I-J}, [move(K,R)].
peep1(and_nt(I,J,R)) --> {integer(I), integer(J), K is I /\ J}, [move(K,R)].
peep1( or_nt(I,J,R)) --> {integer(I), integer(J), K is I \/ J}, [move(K,R)].
peep1(xor_nt(I,J,R)) --> {integer(I), integer(J), xor(I, J, K)}, [move(K,R)].
peep1(sll_nt(I,J,R)) --> {integer(I), integer(J), K is I<<J}, [move(K,R)].
peep1(sra_nt(I,J,R)) --> {integer(I), integer(J), K is I>>J}, [move(K,R)].

% Two instructions & reduce:
peep2r(pragma(X), pragma(X)) --> [pragma(X)].
peep2r(pragma(tag(R,T)),  move(X,Y)) --> {\+ind(X),\+ind(Y)},  [move(X,Y)].
peep2r(pragma(tag(R,T)), deref(X,Y)) --> {\+ind(X),\+ind(Y)}, [deref(X,Y)].
peep2r(allocate(N), deallocate(N)) --> [].
peep2r(deallocate(N), allocate(N)) --> [].
% peep2r(  jump(L),  label(L)) --> [label(L)].
% peep2r(jump(_,L),  label(L)) --> [label(L)].
% What about trapping behavior?
% peep2r(jump(_,_,_,L), label(L)) --> [label(L)].
peep2r(jump_nt(_,_,_,L), label(L)) --> [label(L)].
% peep2r( label(L),   jump(L)) --> [jump(L)].
peep2r(       Br,         I) --> {distant_branch(Br), \+I=label(_)}, [Br].
% peep2r(  jump(L),         I) --> {\+I=label(_)}, [jump(L)].
peep2r( label(L),  label(L)) --> [label(L)].
peep2r(move(X,Y), move(Y,X)) --> {\+is_in(Y,X)}, [move(X,Y)].
peep2r(move(_,Y), move(Z,Y)) --> {\+is_in(Y,Z)}, [move(Z,Y)].
peep2r(move(X,Y), move(X,Y)) --> {\+is_in(Y,X)}, [move(X,Y)].
peep2r(   return,    return) --> [return].
% Can’t do this unless R is not an uninit_reg to be returned:
% peep2r(move(_,R),    return) --> {reg(R)}, [return].
peep2r(move(_,R), jump(N/A)) --> {reg(R,I), low_reg(L), I>=A+L},
   {functor(F, N, A), \+survive(F)},
   [jump(N/A)].
peep2r(move(_,R), call(N/A)) --> {reg(R,I), low_reg(L), I>=A+L},
   {functor(F, N, A), \+survive(F)},
   [call(N/A)].
peep2r(        I,      fail) --> {local_instr(I), \+I=cut(_)}, [fail].
peep2r(        I,      fail) --> {pure_branch(I,Lbls), all_fails(Lbls)}, [fail].
peep2r(pragma(_),      fail) --> [fail].
peep2r(     fail,      fail) --> [fail].
peep2r(     fail,         I) --> {\+I=label(_)}, [fail].
peep2r(pair(A,B),         I) --> {\+I=pair(_,_), \+I=label(_)}, [pair(A,B)].
% peep2r(cmp(ne,R,R), jump(false,fail)) --> [fail].
% peep2r(cmp(eq,R,R), jump(true, fail)) --> [fail].
% peep2r(cmp(eq,R,R), jump(false,_)) --> [cmp(eq,R,R)].
% peep2r(cmp(ne,R,R), jump(true, _)) --> [cmp(ne,R,R)].
peep2r(choice(1/N,_,_),    cut(R)) --> [cut(R)].
peep2r(choice(I/N,_,L),      fail) --> [jump(L)], {I>1, I<N}.
peep2r(test(ne,T,R,L), equal(R,T^A,L)) --> {tag(atom, T)}, [equal(R,T^A,L)].
peep2r(test(_,_,_,L), label(L)) --> [label(L)].
peep2r(equal(_,_,L),  label(L)) --> [label(L)].
peep2r(push(X,R,N),   adda(R,M,R)) --> {merge_add}, [push(X,R,K)], {K is N+M}.
peep2r(push(X,r(h),N),     pad(M)) --> {merge_add}, [push(X,r(h),K)],{K is N+M}.
peep2r(adda(R,N,R),   adda(R,M,R)) --> {merge_add}, [adda(R,K,R)], {K is N+M}.
peep2r(     pad(N),        pad(M)) -->                   [pad(K)], {K is N+M}.
% Shows up in ex(n5):
peep2r(test(eq,T,R,L), switch(U,R,_,A,B)) --> {tag(var,T)},
   [switch(U,R,L,A,B)].
peep2r(test(eq,T,R,L), switch(T,R,A,_,B)) -->
   [switch(T,R,A,L,B)].
peep2r(test(eq,T,R,L), switch(U,R,A,B,L)) --> {tag(var,V), T\==V, T\==U},
   [switch(U,R,A,B,L)].
peep2r(test(E,T,R,L),     label(L)) --> [label(L)], {L\==fail}.
peep2r(pragma(tag(R,_)),  move(X,r(void))) --> {is_in(R,X)}.
peep2r(pragma(tag(R,_)), deref(X,r(void))) --> {is_in(R,X)}.
peep2r(move(C,R), deref(R,R)) -->
   {reg(R), (integer(C) ; C=(T^_),tag(var,V),T\==V)},
   [move(C,R)].

% Two instructions:
peep2(test(Eq,T,R,fail), jump(L)) --> {lbl(L)},
   [test(Ne,T,R,L),fail], {eq_ne(Eq,Ne)}.
peep2(move(A,B), move(B,C)) --> {perm(B), reg(C)},
   [move(A,C),move(C,B)].
peep2(deref(A,B), move(B,C)) --> {perm(B), reg(C)},
   [deref(A,C),move(C,B)].
peep2(deref(A,B), deref(B,C)) -->
   [deref(A,B),move(B,C)].
peep2(move(C,P), move(C,R)) --> {reg(R), perm(P), \+reg(C), \+is_in(P,C)},
   [move(C,R),move(R,P)].
peep2(move(C,I), move(C,R)) --> {reg(R), ind(I), \+reg(C), \+is_in(R,I)},
   [move(C,R),move(R,I)].
peep2(move(R,I), move(I,C)) --> {reg(R), \+reg(C), ind(I)},
   [move(R,I),move(R,C)].
peep2(jump(Cond,R,S,fail), jump(L)) --> {cond(Cond,Inv), lbl(L)},
   [jump(Inv,R,S,L),fail].
peep2(jump_nt(Cond,R,S,fail), jump(L)) --> {cond(Cond,Inv), lbl(L)},
   [jump_nt(Inv,R,S,L),fail].
% peep2(jump(TF,fail), jump(L)) --> {tf_ft(TF,FT)},
%  [jump(FT,L),fail].
peep2(switch(T,R,fail,L,L),   label(L)) -->
   [test(eq,Tvar,R,fail),label(L)], {tag(var,Tvar)}.
peep2(switch(T,R,L,fail,fail),label(L)) -->
   [test(ne,Tvar,R,fail),label(L)], {tag(var,Tvar)}.
peep2(switch(T,R,fail,L,fail),label(L)) -->
   [test(ne,T,R,fail),label(L)].
peep2(switch(T,R,L,fail,L),   label(L)) -->
   [test(eq,T,R,fail),label(L)].
peep2(switch(T,R,L1,L2,L1),  label(L1)) -->
   [test(eq,T,R,L2),label(L1)].
peep2(switch(T,R,L1,L2,L1),  label(L2)) -->
   [test(ne,T,R,L1),label(L2)].
peep2(switch(T,R,L1,L2,L2), label(L1)) -->
   [test(ne,Tvar,R,L2),label(L1)], {tag(var,Tvar)}.
peep2(switch(T,R,L1,L2,L2), label(L2)) -->
   [test(eq,Tvar,R,L1),label(L2)], {tag(var,Tvar)}.
peep2(I, adda(R,M,R)) --> {merge_add}, [adda(R,M,R),J], {inc_reg(I,R,M,J)}.
peep2(I,      pad(M)) -->                   [pad(M),J], {inc_reg(I,r(h),M,J)}.

% Three instructions & reduce:
peep3r(Br, label(L), Br) --> {distant_branch(Br)}, [label(L),Br].
peep3r(jump(Cond,R,S,L1), jump(L2), label(L1)) --> {cond(Cond,Inv), lbl(L2)},
   [jump(Inv,R,S,L2),label(L1)].
peep3r(jump_nt(Cond,R,S,L1), jump(L2), label(L1)) --> {cond(Cond,Inv), lbl(L2)},
   [jump_nt(Inv,R,S,L2),label(L1)].
peep3r(jump(Cond,R,S,L1), fail, label(L1)) --> {cond(Cond,Inv)},
   [jump(Inv,R,S,fail),label(L1)].
peep3r(jump_nt(Cond,R,S,L1), fail, label(L1)) --> {cond(Cond,Inv)},
   [jump_nt(Inv,R,S,fail),label(L1)].
peep3r(test(EqNe,A,B,L),      fail, label(L)) --> {eq_ne(EqNe, NeEq)},
   [test(NeEq,A,B,fail),label(L)].
peep3r(test(EqNe,A,B,L1), jump(L2), label(L1)) --> {eq_ne(EqNe, NeEq), lbl(L2)},
   [test(NeEq,A,B,L2),label(L1)].
% peep3r(jump(T,L),     fail,label(L)) --> [jump(F,fail),label(L)],{tf_ft(T,F)}.
% peep3r(jump(T,L1),jump(L2),label(L1)) --> [jump(F,L2),label(L1)],{tf_ft(T,F)}.
peep3r(move(T^R1,R), adda(R1,N,R1), move(R,[R])) -->
   [move(T^R1,R),push(R,R1,N)],
   {tag(T),integer(N),reg(R),reg(R1),R\==R1}.
peep3r(move(C,R), move(R,P), move(X,R)) --> {reg(R),\+is_in(R,P),\+is_in(R,X)},
   [move(C,P),move(X,R)].
peep3r(pragma(tag(R,_)), pragma(align(R,_)),  move(X,r(void))) --> {is_in(R,X)}.
peep3r(pragma(tag(R,_)), pragma(align(R,_)), deref(X,r(void))) --> {is_in(R,X)}.

% Three instructions:
peep3(I, label(L), fail) --> {local_instr(I), \+I=cut(_), \+I=fail},
   [fail,label(L),fail].
peep3(pragma(tag(IR,T)), move(C,I), move(C,R)) -->
   {reg(R), ind(I, IR), a_var(IR), \+reg(C), \+is_in(R,I)},
   [move(C,R),pragma(tag(IR,T)),move(R,I)].
```

```
peep3(move(P,R), pragma(tag(P,T)), move(R,[R])) -->
   {perm(P), reg(R)},
   [move(P,R), pragma(tag(R,T)), move(R,[R])].
peep3(move(Tˆr(h),R), push(R,r(h),N), deref(R,S)) -->
   {reg(R), a_var(S), tag(T)},
   [move(Tˆr(h),R), push(R,r(h),N), move(R,S)].
% Gives error in writemode unification:
% peep3(move(Tˆr(h),R), adda(r(h),N,r(h)), pad(M)) -->
%  [pad(M),move(Tˆr(h),R),adda(r(h),N,r(h))],
%  {tag(T),integer(N),integer(M),a_var(R),R\==r(h)}.

% *** Up to a cut there are only instructions that always succeed &
% there is no move to the b register (this move marks the point to cut to).
% This routine follows the execution path as far as it can.  It follows
% jumps and calls.  It could be extended to follow all branches.
% Proc_code has a higher-level version of this predicate.
success_to_cut([cut(_)|_], _) :- !.
success_to_cut([move(_,r(b))|_], _) :- !, fail.
success_to_cut([jump(L)|_], BA) :- fget(BA, L, Blk), !, success_to_cut(Blk, BA).
success_to_cut([call(L)|_], BA) :- fget(BA, L, Blk), !, success_to_cut(Blk, BA).
success_to_cut([I|Code], BA) :- local_instr(I), !, success_to_cut(Code, BA).

all_fails([]).
all_fails([X|L]) :- nonvar(X), X=fail, all_fails(L).

eq_ne(eq, ne).
eq_ne(ne, eq).

tf_ft(true, false).
tf_ft(false, true).

% Update an instruction when an adda moves across it:
% (Fails when it shouldn’t be done)
inc_reg(move(S,D), R, I, move(Sn,Dn)) :- add_ea(S,R,I,Sn), add_ea(D,R,I,Dn).
inc_reg(pragma(P), _, _, pragma(P)).

add_ea(TagˆX, Reg, Inc, TagˆY) :- pointer_tag(Tag), add_reg(X, Reg, Inc, Y).
add_ea(  [X], Reg, Inc,   [Y]) :- add_reg(X, Reg, Inc, Y).
add_ea(    X, Reg, Inc,     Y) :- add_reg(X, Reg, Inc, Y).
add_ea( Else,   _,   _,  Else). % An addressing mode not containing Reg.

add_reg(R+N, R, M, RK) :- reg(R), integer(N), K is  N-M, make_reg(R, K, RK).
add_reg(R-N, R, M, RK) :- reg(R), integer(N), K is -N-M, make_reg(R, K, RK).
add_reg(R,   R, M, RK) :- reg(R),             K is   -M, make_reg(R, K, RK).

make_reg(R, K, R+K) :- K>0, !.
make_reg(R, K, R)   :- K=:=0, !.
make_reg(R, K, R-N) :- K<0, !, N is -K.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Remove nesting of switch:

peep_flat(Code, FCode) :- compile_option(flat), !, peep_flat(Code, FCode, []).
peep_flat(Code,  Code) :- \+compile_option(flat).

peep_flat([]) --> [].
peep_flat([I|Code]) --> peep_flat(I, Code).

peep_flat(switch(unify,T,R,VC,SC,F), Code) --> !,
   [switch(T,R,VL,SL,F)],
   [label(VL)],
   peep_flat(VC),
   [jump(End)],
   [label(SL)],
   peep_flat(SC),
   [jump(End)],  % !! For best code, this must be kept as is.
             % This is because jumps delimit basic blocks.
   [label(End)],
   peep_flat(Code).
peep_flat(Instr, Code) -->
   {\+Instr=switch(unify,_,_,_,_,_)},
   [Instr],
   peep_flat(Code).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Addressing mode utilities:
% These utilities are derived from the addressing modes given in bam_syntax.pl.

% Label:
lbl(fail).
lbl(N/A)      :- atom(N), integer(A), A>=0.
lbl(l(N/A,I)) :- atom(N), integer(A), A>=0, integer(I).

% Register:
% (This includes r(h) and r(b))
reg(r(I)) :- atomic(I).

reg(r(I), I) :- integer(I).

perm(p(I)) :- integer(I).

a_var(R) :- reg(R).
a_var(R) :- perm(R).

an_atom(Int)     :- integer(Int).
an_atom(T^A)     :- atom(A), tag(atom, T).
an_atom(T^(F/N)) :- atom(F), positive(N), tag(atom, T).

ind([_]).

ind([I], I).

complex(_^_).
complex([_]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Utilities:

% Simplify an operand:
s_oper(X, S) :- (s_o(X, S) -> true ; X=S).

% Table of operand simplifications:
s_o(    X+0,       X) :- !.
s_o(    X-0,       X) :- !.
s_o(  [X+0],     [X]) :- !.
s_o(  [X-0],     [X]) :- !.
s_o(    X+N,     X-I) :- neg_abs(N, I), !.
s_o(    X-N,     X+I) :- neg_abs(N, I), !.
s_o(  [X+N],   [X-I]) :- neg_abs(N, I), !.
s_o(  [X-N],   [X+I]) :- neg_abs(N, I), !.
s_o(T^(r(h)+0),     T^r(h)) :- !.
s_o(T^(r(h)-0),     T^r(h)) :- !.
s_o(T^(r(h)+N), T^(r(h)-I)) :- neg_abs(N, I), !.
s_o(T^(r(h)-N), T^(r(h)+I)) :- neg_abs(N, I), !.

neg_abs(N, I) :- integer(N), N<0, I is -N.

% A term is inside another.
is_in(A, A).
is_in(A, A+_).
is_in(A, A-_).
is_in(A, [A]).
is_in(A, [A+_]).
is_in(A, [A-_]).
is_in(A, Tag^A) :- pointer_tag(Tag).
is_in(A, Tag^(A+_)) :- pointer_tag(Tag).
is_in(A, Tag^(A-_)) :- pointer_tag(Tag).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

preamble.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Convert a raw list of input Prolog clauses into a list of ptrees ready for
% further compilation.

% This file contains routines for two conversions:
% 1. Bringing together non-contiguous clauses.
% 2. Converting contiguous clauses into ptrees.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Bring together non-contiguous clauses so that they are contiguous.
% This is done while enforcing two constraints:
% 1. Directives stay in front of the predicate they are originally in front of.
% 2. The first predicate remains first.  This is for compatibility with
%    the simulator, which executes the first predicate.
% Also randomizes the order of predicates (except the first).  This ensures
% logarithmic time for the array handling done in the analysis module.

% *** Accumulator declarations:
acc_info( cls, V,   Out,     In, Out=[V|In]).
acc_info(dirs, V,   Out,     In, Out=[V|In]).

pred_info(extract_merge, 3, [dirs,cls]).

% *** Main entry point:
cont_cls(Cls, Cls) :-
   compile_option(contiguous), !.
cont_cls(Cls, CCls) :-
   stats(cc,1),
   keylist_cls(Cls, Keylist),
   stats(cc,2),
   keysort(Keylist, Sort),
   stats(cc,3),
   merge_cls(Sort, CCls),
   stats(cc,4).

% Convert clauses to keylist suitable for keysort.
% Key is of the form key(R,H,NA) where
% 1. R is the rank:
%    R=1 for directives up to & including the first clause
%        and all clauses with the same name/arity.
%    R=2 for directives & clauses up to & including the last clause
%        except those with the same name/arity as the first clause.
%    R=3 for directives beyond the last clause.
% 2. H is a hash of the name & arity.  This ensures that the predicates are
%    arranged pseudo-randomly, so that array handling has no problems later.
% 3. NA is the name & arity of the next clause in the list.
keylist_cls(Cls, KCls) :-
   extract_directives(Cls, Cls2, D, L),
   first_namearity(Cls2, NA1),
   keylist_cls(Cls2, KCls, D, L, NA1).

first_namearity([Cl|_], N/A) :- !, split(Cl, Head, _), functor(Head, N, A).
first_namearity(_, _).

keylist_cls([], Ks, D, L, _) :-
   (D==L -> Ks=[] ; Ks=[key(3,_,end)-pair(D,L,L1,L1)]).
keylist_cls([Cl|Cls], [K-pair(D,L,[Cl|L1],L1)|KCls], D, L, NA1) :-
   split(Cl, Head, _),
   functor(Head, N, A),
   hash_name(N, A, H),
   (N/A=NA1 -> R=1 ; R=2),
   K = key(R,H,N/A),
   extract_directives(Cls, Cls2, D2, L2),
   keylist_cls(Cls2, KCls, D2, L2, NA1).

hash_name(N, A, H) :-
   X is A*10,
   name(N, NL),
   diff_sum(NL, X, Y),
   random_step(Y, Z),
   random_step(Z, H).

diff_sum([]) --> [].
diff_sum([A]) --> add(A).
diff_sum([A,B|L]) --> add(A), sub(B), diff_sum(L).

merge_cls([], []).
merge_cls(KCls, OutCls) :-
   extract_merge(KCls, _, OutKCls, OutCls, MidCls, MidCls, EndCls),
   merge_cls(OutKCls, EndCls).

extract_merge([], _, []) -->> !.
extract_merge([key(_,_,NA)-pair(D,L,C,L2)|KCls], NA, OutKCls) -->> !,
   insert(D, L):dirs, insert(C, L2):cls,
   extract_merge(KCls, NA, OutKCls).
extract_merge([K-Pair|KCls], NA, [K-Pair|KCls]) -->> !.

extract_directives([], []) --> [].
extract_directives([D|Cls], Es) --> {directive(D)}, !, [D],
   extract_directives(Cls, Es).
extract_directives([Cl|Cls], [Cl|Cls]) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Separate the procedures into lists of clauses and convert to Ptrees:
cls_to_ptrees(Cls, Ptrees) :-
   expand_clauses(Cls, ECls),
   translate_clauses(ECls, TCls),
   cls_to_ptrees_loop(TCls, Ptrees),
   add_modes(Ptrees), !.

cls_to_ptrees_loop(Cls, Ptrees) :-
```

```
    (next_name(Cls, C2, NaAr, Ptrees, P2),
     cls_to_proc(C2, C3, NaAr, P)
     -> P2=[ptree(NaAr,P,_,[])|P3],
        cls_to_ptrees_loop(C3, P3)
      ; Ptrees=Cls
    ).

next_name([D|Cls], RestCls, NaAr) --> {directive(D)}, !,
   [D], next_name(Cls, RestCls, NaAr).
next_name([Cl|Cls], [Cl|Cls], N/A) -->
   {split(Cl, Head, _), functor(Head, N, A)}.

% Handle DCG expansion:
expand_clauses([], []).
expand_clauses([C|Cls], [E|ECls]) :-
   expand_term(C, E),
   expand_clauses(Cls, ECls).

% Add modes to the ptrees from the global database:
add_modes([]).
add_modes([D|Ptrees]) :- directive(D), !,
   add_modes(Ptrees).
add_modes([ptree(Na/Ar,_,(Head:-Formula),_)|Ptrees]) :-
   functor(Head, Na, Ar),
   ( require(Head,Req), before(Head,Bef),
     combine_formula(Req, Bef, Formula)
   ; Formula=true
   ), !,
   add_modes(Ptrees).

% Split one procedure from a list of clauses, and return the remainder:
cls_to_proc([], [], _, []).
cls_to_proc([Cl|Cls], RestCls, NaAr, Proc) :-
   split(Cl, Head, _),
   functor(Head, N, A), NaAr=N/A, !,
   Proc=[Cl|P],
   cls_to_proc(Cls, RestCls, NaAr, P).
cls_to_proc([Cl|Cls], [Cl|Cls], NaAr, []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Table of generic translations done before the main compilation:
% Notes:
% 1. This translation must be done after the DCG expansion.
% 2. Macro expansion could be done here.
% 3. This can become a cog in a more general program transformation scheme
%    for later.
% 4. This distinguishes between Quintus and C-Prolog for the c/3 predicate.
% 5. This adds a body to ALL clauses so that later uses of split can be
%    removed, which should be done later in a cleanup phase.

translate_clauses([], []).
translate_clauses([D|Cls], [D|TCls]) :- directive(D), !,
   translate_clauses(Cls, TCls).
translate_clauses([C|Cls], [(H:-SB)|TCls]) :-
   split(C, H, B),
   translate(B, TB),
   simplify(TB, SB),
   translate_clauses(Cls, TCls).

% *** The gist of the translation:
% These two clauses for DCG’s:
translate(   c(A,B,C), A=[B|C]) :- compile_option(system(cprolog)), !.
translate( ’C’(A,B,C), A=[B|C]) :- compile_option(system(quintus)), !.
% Arithmetic expressions:
translate((A is Expr),    Conj) :- number(A), !,
   top_expr(Expr, X, Conj, X=:=A).
translate((A is Expr),    Conj) :- var(A), !,
   top_expr(Expr, X, Conj, X=A).
translate(\+(A is Expr),  Conj) :- number(A), !,
   top_expr(Expr, X, Conj, X=\=A).
% translate(\+(A is Expr),  Conj) :- var(A), !,
   % top_expr(Expr, X, Conj, (integer(A),X=\=A)).
% Built-ins containing arithmetic expressions at compile-time:
translate(     put(E),    Conj) :- !,
   expr(E, NE, Conj, put(NE)).
translate(  ArithTest,    Conj) :-
   arith_test(ArithTest, A, B, Cond), !,
   expr(A, NA, Conj, C2),
   expr(B, NB, C2, NewTest),
   arith_test(NewTest, NA, NB, Cond).
% Can’t do this until the inverse transformation is there too:
% translate(ComplexTest,    Test) :- merge_test(ComplexTest, Test), !.
% Don’t support expressions in functor and arg:
% translate(functor(A,B,C), Conj) :- !,
%  expr(C, NC, Conj, functor(A,B,NC)).
% translate( arg(A,B,C),    Conj) :- !,
%  expr(A, NA, Conj, arg(NA,B,C)).
% Unification:
translate(X=Y,            Conj) :- !, translate_unify(X, Y, Conj).

% *** Control predicates are unchanged:
translate( (A,B),  (TA,TB)) :- !, translate(A, TA), translate(B, TB).
translate( (A;B),  (TA;TB)) :- !, translate(A, TA), translate(B, TB).
translate((A->B), (TA->TB)) :- !, translate(A, TA), translate(B, TB).
translate(not(A),  not(TA)) :- !, translate(A, TA).
translate( \+(A),   \+(TA)) :- !, translate(A, TA).
translate((A=>B), (TA=>TB)) :- !, translate(A, TA), translate(B, TB).

% *** Default:
translate(G, G).

% *** Unification of two nonvariables:
% Unravel it into a conjunction of unifications of the form X=Y
% where X is a variable.
translate_unify(X, Y, Conj) :- nonvar(X), nonvar(Y),
   functor(X, Na, Ar), functor(Y, Na, Ar), !,
   translate_unify(X, Y, 1, Ar, Conj).
translate_unify(X, Y, fail) :- nonvar(X), nonvar(Y), !.
translate_unify(X, Y, X=Y) :- var(X), !.
translate_unify(X, Y, Y=X) :- var(Y), !.

translate_unify(X, Y, I, Ar, true) :- I>Ar, !.
translate_unify(X, Y, I, Ar, (C,Conj)) :- I=<Ar, !,
   arg(I, X, Xi),
   arg(I, Y, Yi),
   translate_unify(Xi, Yi, C),
   I1 is I+1,
   translate_unify(X, Y, I1, Ar, Conj).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

proc_code.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Procedure code generation:

% Generates compiled code for procedure selection.
% This module uses clause compilation as a primitive operation.

% (Assumes that all head vars exist--this may not remain true; it will depend
%  on the Modes).
proc_code(Head, Disj, Modes, DList, RFlag) -->
   {copy(Modes, (Head:-InF))},
   code(Head, InF, DList, RFlag, Disj), !.

% Three cases:
% 1. Generate deterministic selection:
code(Head, InF, DList, RFlag, '$case'(Name,Ident,Case)) --> !,
   {stats(d, det_code)},
   det_code(Head, InF, DList, RFlag, Name, Ident, Case).
% 2. Generate a choice-block:
code(Head, InF, DList, RFlag, Disj) -->
   {disj_p(Disj)}, !,
   {stats(d, choice_block)},
   choice_block(Head, InF, DList, RFlag, Disj).
% 3. A single clause:
code(Head, InF, DList, RFlag, Goal) -->
   {\+disj_p(Goal)},
   {\+Goal='$case'(_,_,_)}, !,
   {flat_conj(Goal, SGoal)},
   {stats(d, clause_code)},
   clause_code((Head:-SGoal), DList, RFlag, InF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Management of choice points:

% This code calculates the minimal sets of registers that must be saved by the
% first choice and restored by the following choices.  The minimal set to be
% saved is all argument variables used in choices after the first
% and all argument variables that are uninit mem vars, minus all argument
% vars that are uninit reg vars.
% The minimal set to be restored by a clause is all argument variables
% used in the clause and all argument variables that are uninit mems, minus all
% argument vars that are uninit reg vars.
% Also inserts dereferencing instructions for those predicates in Disj which
% need them (i.e. those requiring deref arguments).  This avoids
% superfluous dereferencing on backtracking.
choice_block(Head, InF, DList, RFlag, Disj) -->
   {require_derefs(Disj, InF, DVars)},
   insert_derefs(Head, DVars, InF, MidF),
   {split_disj(Disj, _, Rest)},
   {varset_disj(Rest, DisjVars)},
   {uninit_set_type(mem, InF, UMSet)},
   {uninit_set_type(reg, InF, URSet)},
   {unionv(UMSet, DisjVars, SV)},
   {diffv(SV, URSet, SaveVars)},
   {varset_numset(Head, SaveVars, All)},
   choice_block(Head, MidF, All, UMSet,URSet, DList, RFlag, Disj, _, 0, N).

choice_block(Head, F, A, U,R, DL, RFlag, fail,    _, N, N) --> {N=0}, !, [fail].
choice_block(Head, F, A, U,R, DL, RFlag, fail, fail, N, N) --> {N>0}, !, [].
choice_block(Head, F, A, U,R, DL, RFlag, (D;Disj), EntryLbl, I1, N) --> !,
   {I is I1+1},
   [label(EntryLbl)],
   {functor(Head, _, Arity)},
   cp_instr(A, U, R, D, Head, I, N, Disj, RetryLbl),
   {stats(d, I)},
   code(Head, F, DL, RFlag, D),
   choice_block(Head, F, A, U,R, DL, RFlag, Disj, RetryLbl, I, N).
choice_block(Head, F, A, U,R, DL, RFlag, Goal, EntryLbl, I, N) -->
   {\+disj_p(Goal)}, !,
   choice_block(Head, F, A, U,R, DL, RFlag, (Goal;fail), EntryLbl, I, N).

cp_instr(  _, _, _, _,    _, 1, _, fail,   _) --> !.
cp_instr(All, _, _, _,    _, I, N,    _, Lbl) --> {I=:=1}, !,
   [choice(I/N,All,Lbl)].
cp_instr(All, U, R, D, Head, I, N,    _, Lbl) --> {I>1}, !,
   {varset_conj(D, DVars)},
   {unionv(U, DVars, RV)},
   {diffv(RV, R, RestoreVars)},
   {varset_numset(Head, RestoreVars, DNums)},
   {format_numset(All, DNums, Nums)},
   [choice(I/N,Nums,Lbl)].

% Given a head containing variable arguments and a set of variables this
% predicate constructs a set of registers numbers corresponding to the
% variables in the head that are part of the variable set.
varset_numset(Head, VarSet, NumSet) :-
   functor(Head, _, Arity),
   low_reg(Low),
   varset_numset(1, Arity, Head, VarSet, Low, NumSet).

varset_numset(I, Arity, Head, VarSet, Low, []) :- I>Arity, !.
varset_numset(I, Arity, Head, VarSet, Low, NumSet) :- I=<Arity, !,
   arg(I, Head, A),
   N is Low+(I-1),
   incl_if_inv(A, VarSet, N, NumSet, NumSet2),
   I1 is I+1,
   varset_numset(I1, Arity, Head, VarSet, Low, NumSet2).

incl_if_inv(A, VarSet, N) -->   {inv(A, VarSet)}, !, [N].
incl_if_inv(A, VarSet, N) --> {\+inv(A, VarSet)}, !.

% Calculate the set of variables needed by a disjunction:
varset_disj(Disj, Set) :- varbag_disj(Disj, Bag, []), sort(Bag, Set).
```

```
varbag_disj((A;B)) --> !, varbag_disj(A), varbag_disj(B).
varbag_disj(Conj) --> varbag_conj(Conj).

% Calculate the set of variables needed by a conjunction:
% Recognize that '$body' goals need different handling.
varset_conj(Conj, Set) :- varbag_conj(Conj, Bag, []), sort(Bag, Set).

varbag_conj((A,B)) --> !, varbag_conj(A), varbag_conj(B).
varbag_conj('$body'(_,Vars,_)) --> !, difflist(Vars).
varbag_conj(Goal) --> varbag(Goal).

% Format the set of register numbers to be restored so that
% they correspond positionally to the set that was saved.  This is used to tell
% choices after the first what part of the choice point they need to load
% in.  Otherwise they don't know what to restore from a choice point.
format_numset([N|Try], [N|Nums], [N|Out]) :- !,
   format_numset(Try, Nums, Out).
format_numset([N|Try], [M|Nums], [no|Out]) :- N<M, !,
   format_numset(Try, [M|Nums], Out).
format_numset([N|Try], [], [no|Out]) :- !, format_numset(Try, [], Out).
format_numset([], [], []).

% *** Could be a little sharper by keeping track of SoFar variables too ***
% Gather all required dereferencing instructions in a formula:
% Stop gathering at a unification goal that may be aliased, since any
% dereferencing done before that has to be repeated anyway.  Continue
% gathering when encountering other goals, including unification goals
% that are not aliased (i.e. which have an uninit argument).
require_derefs(Disj, Form, ReqDeref) :-
   get_kind_d(Disj, Form, deref, DBag, []),
   sort(DBag, ReqDeref).

% Traverse the disjunction:
get_kind_d((A;B), F, K) --> !, get_kind_c(A, F, K), get_kind_d_stop(A, B, F, K).
get_kind_d(A, F, K) --> get_kind_c(A, F, K).

% This predicate can be removed if register lifetime analysis is done in peep:
% Don't continue to later conj's if this one stops with an always-executed cut.
get_kind_d_stop(A, _, _, _) --> {success_to_cut(A)}, !.
get_kind_d_stop(_, B, F, K) -->
   get_kind_d(B, F, K).

% Peephole has a lower-level version of this:
success_to_cut('$cut_shallow'(_)).
success_to_cut('$cut_deep'(_)).
success_to_cut(('$cut_shallow'(_),_)).
success_to_cut(('$cut_deep'(_),_)).
success_to_cut((G,Conj)) :- succeeds(G), success_to_cut(Conj).

% Traverse a conjunction:
get_kind_c((A,B), F, K) --> !, get_kind_g(A, K), get_kind_c_stop(A, B, F, K).
get_kind_c(A, _, K) --> get_kind_g(A, K).
```

```
% Don’t continue to later goals if the unification goal is a non-trivial one.
get_kind_c_stop(X=Y, _, F, _) -->
   {\+implies(F, (uninit(any,X);uninit(any,Y)))}, !.
get_kind_c_stop(_, B, F, K) -->
   get_kind_c(B, F, K).

get_kind_g(Goal, K) --> {require(Goal, Req)}, filter_kind(Req, K).

% Extract all arguments of dereference modes:
filter_kind((A,B), K) --> !, filter_kind(A, K), filter_kind(B, K).
filter_kind((A;B), K) --> !, filter_kind(A, K), filter_kind(B, K).
filter_kind(deref(X),  deref) --> {var(X)}, !, [X].
filter_kind(rderef(X), deref) --> {var(X)}, !, [X].
% filter_kind(Uninit, uninit) --> {an_uninit_mode(Uninit,mem,X), var(X)}, !,[X].
filter_kind(_, _) --> [].

% Insert dereferencing instructions:
insert_derefs(Head, DVars, InF, OutF) -->
   {varset(Head, HVars)},
   {intersectv(DVars, HVars, HDVars)},
   insert_derefs_loop(Head, HDVars, InF, OutF).

insert_derefs_loop(Head, [], InF, InF) --> !.
insert_derefs_loop(Head, [X|Vars], InF, OutF) -->
   {head_reg(Head, X, Rx)},
   rtest(deref(X), Rx, InF, MidF, fail),
   insert_derefs_loop(Head, Vars, MidF, OutF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generate deterministic code for a test-set:

% This predicate is testset-dependent.
% Given: Name, Ident of test-set, and Case into test+body.
det_code(Head, InF, DList, RFlag, Type, v(A), Case) -->
   {type(Type)}, !,
   {stats(d, Type)},
   {head_reg(Head, A, Ra)},
   rtest(deref(A), Ra, InF, MidF, fail),
   {type_test(Type, A, Test)},
   rtest_in(Test, Ra, MidF, _, False),
   {simplify(not(Test), NotT)},
   choice(true, Head, Test, MidF, DList, RFlag, Case,True),
   choice(false,Head, NotT, MidF, DList, RFlag, Case,False).
det_code(Head, InF, DList, RFlag, comparison(Cond,arith), v(A,B), Case) --> !,
   {stats(d, comparison(Cond,arith))},
   {head_reg(Head, A, Ra)},
   {head_reg(Head, B, Rb)},
   rtest(deref(A), Ra,  InF, MidF, fail),
   rtest(deref(B), Rb, MidF, OutF, fail),
   arith_comparison(Cond, A, B, Ra, Rb, True),
   {arith_test(Test, A, B, Cond)},
   {simplify(not(Test), NotT)},
   choice(false,Head, NotT, OutF, DList, RFlag, Case,False),
   choice(true, Head, Test, OutF, DList, RFlag, Case,True).
det_code(Head, InF, DList, RFlag, hash(Type), v(A), Case) --> !,
   {stats(d, hash(Type))},
   {head_reg(Head, A, Ra)},
   rtest(deref(A), Ra, InF, MidF, fail),
   {type_test(Type, A, Test)},
   rtest_if_structure(Type, Test, Ra, MidF, OutF, SkipLbl),
   [hash(Type,Ra,Len,Lbl)],
   [label(SkipLbl)],
   insert(ElseCode, ELink),
   [label(Lbl)],
   [pragma(hash_length(Len))],
   insert(HTable, HLink),
   hash_table(Head, OutF, DList, RFlag, Case, Else, 0, Len, HTable, HLink),
   % The test does not hold beyond the hash table:
   {code(Head, MidF, DList, RFlag, Else, ElseCode, ELink)}.
det_code(Head, InF, DList, RFlag, switch(Type), v(A), Case) -->
   {tag(Type, Tag)}, !,
   {stats(d, switch(Type))},
   {head_reg(Head, A, Ra)},
   rtest(deref(A), Ra, InF, OutF, fail),
   [switch(Tag,Ra,VarLbl,TypeLbl,Other)],
   {type_test(Type, A, Test)},
   {disj_test(var(A), Test, VorT), not_test(VorT, NVorT)},
   % This order avoids an extraneous jump in VLSI-BAM translation:
   choice(other,Head, NVorT,  OutF, DList, RFlag, Case,Other),
   choice(var,  Head, var(A), OutF, DList, RFlag, Case,VarLbl),
   choice(Type, Head, Test,   OutF, DList, RFlag, Case,TypeLbl).
det_code(Head, InF, DList, RFlag, equal, v(A,B), Case) --> !,
   {stats(d, equal)},
   {head_reg(Head, A, Ra)},
   {head_reg(Head, B, Rb)},
   rtest(deref(A), Ra,  InF,  MidF, fail),
   rtest(deref(B), Rb, MidF, MidF2, fail),
   [equal(Ra,Rb,False)],
   choice(true, Head, A==B,  MidF2, DList, RFlag, Case, True),
   choice(false,Head, A\==B, MidF2, DList, RFlag, Case, False).
det_code(Head, InF, DList, RFlag, equal(atomic,Atomic), v(A), Case) --> !,
   {stats(d, equal(atomic,Atomic))},
   {head_reg(Head, A, Ra)},
   rtest(deref(A), Ra, InF, MidF, fail),
   {atomic_word(Atomic, W)},
   [equal(Ra,W,False)],
   choice(true, Head, A=Atomic,  MidF, DList, RFlag, Case, True),
   choice(false,Head, A\=Atomic, MidF, DList, RFlag, Case, False).
det_code(Head, InF, DList, RFlag, equal(structure,Na/Ar), v(A), Case) --> !,
   {stats(d, equal(structure,Na/Ar))},
   {head_reg(Head, A, Ra)},
   rtests((deref(A),structure(A)), Ra, InF, MidF, fail),
   {tag(atom, Tatm)},
   pragma_tag(Ra, structure),
   {align(K)},
   [pragma(align(Ra,K))],
        [equal([Ra],Tatm^(Na/Ar),False)],
   choice(true, Head,     '$name_arity'(A,Na,Ar),  MidF, DList, RFlag,
          Case, True),
   choice(false,Head, not('$name_arity'(A,Na,Ar)), MidF, DList, RFlag,
          Case, False).

% The hash table type test is only needed for structures:
rtest_if_structure(structure, Test, Ra, InF, OutF, Lbl) --> !,
   rtest(Test, Ra, InF, OutF, Lbl).
rtest_if_structure(    Other, Test, Ra, InF, InF, Lbl) --> !.

% add_align --> {align(K), K>1}, !, [align(K)].
% add_align --> [].

atomic_word(N,      N) :- number(N), !.
atomic_word(A, Tatm^A) :- atom(A), !, term_tag(atom, Tatm).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Insert the code for one choice of a multiway branch.
% Label = branch label (= 'fail' if choice and else case don't exist)

% First clause used in compilation of arg builtin:
choice(Direc, Head, Test, InF, DList, RFlag, Case, fail) -->
   {mutex(Test, InF, left)}, !,
   {comment(Head, ['Mutex of ',Test,' and ',InF,' in choice'])},
   {stats(d, Direc/Test-mutex_fail)},
   [fail].
% Direc exists--use T instead of Test:
choice(Direc, Head, Test, InF, DList, RFlag, Case, Label) -->
   {choice('$test'(Direc,Preamble,T,Bindvars,Code), Case)}, !,
   [label(Label)],
   update_head(Bindvars, T, InF, OutF, DList, RFlag,
           Head, NewHead),
   {subsume(T, Code, SCode)},
   {stats(d, Direc/Test)},
   code(NewHead, OutF, DList, RFlag, SCode).
choice(Direc, Head, Test, InF, DList, RFlag, Case, fail) -->
   {\+choice('$test'(Direc,Preamble,T,Bindvars,_), Case)},
   {choice('$else'(_,_,fail), Case)}, !,
   {stats(d, Direc/Test-else_fail)},
   [fail].
% In this case T doesn't exist so Test is used instead:
choice(Direc, Head, Test, InF, DList, RFlag, Case, Label) -->
   {\+choice('$test'(Direc,Preamble,T,Bindvars,_), Case)},
   {choice('$else'(EPreamble,EBindvars,Code), Case)},
   {\+(Code=fail)}, !,
   [label(Label)],
   update_head(EBindvars, Test, InF, OutF, DList, RFlag,
           Head, NewHead),
   {subsume(Test, Code, SCode)},
   {stats(d, Direc/Test)},
   code(NewHead, OutF, DList, RFlag, SCode).

choice(Choice, Choice) :- \+Choice=(_;_).
choice(Choice, Choice) :- \+Choice=(_;_).
choice(Choice, (_;Disj)) :- choice(Choice, Disj).
choice(Choice, (Disj;_)) :- choice(Choice, Disj).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Hash table is a collection of pairs:
hash_table(Head, InF, DList, RFlag,
      '$else'(_,_,Else), Else, Len, Len,
      HLink, HLink) --> !.
hash_table(Head, InF, DList, RFlag,
      ('$test'(X,_,Test,Bindvars,Code);Case), Else, In, Out,
      [pair(E,Lbl)|HT], HLink) --> !,
   {make_hash_entry(X,E)},
   {In1 is In+1},
   [label(Lbl)],
   update_head(Bindvars, Test, InF, OutF, DList, RFlag, Head, NewHead),
   % {subsume(InF, Code, SCode)},
   code(NewHead, OutF, DList, RFlag, Code),
   hash_table(Head, InF, DList, RFlag, Case, Else, In1, Out, HT, HLink).

make_hash_entry(  Int,        Int) :- integer(Int), !.
make_hash_entry(    A,     Tatm^A) :- atom(A), !, tag(atom, Tatm).
make_hash_entry((N/A), Tatm^(N/A)) :- tag(atom, Tatm).

% Handle tests that bind:
% This code is unfinished, as the scheme for binding tests has not yet been
% designed satisfactorily.  Currently it just updates the modes.
% For tests that bind, insert code that does the binding necessary.
% Binding may create new variables.  These are added to the clause head
% that is passed to further code generation.
% For tests that do not bind, do nothing.
update_head([], Test, InF, OutF, DList, RFlag, Head, Head) --> !,
   {update_formula(Test, InF, OutF)}.
update_head(Bindvars, Test, InF, OutF, DList, RFlag, Head, Head) -->
   {\+Bindvars=[]}, !,
   {update_formula(Test, InF, OutF)}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Determine the register corresponding to the given variable's position
% in the head:
head_reg(Head, Arg, Reg) :- var(Arg), !,
   functor(Head, _, Arity),
   head_reg(1, Arity, Head, Arg, WhichReg),
   low_reg(L),
   R is WhichReg+L-1,
   Reg=r(R).
head_reg(Head, Arg, (Tatm^Arg)) :- atom(Arg), !,
   term_tag(Arg, Tatm).
head_reg(Head, Arg, Arg) :- number(Arg), !.
head_reg(Head, Arg, r(complex_type_error)) :- compound(Arg), !.

head_reg(I, Max, Head, Var, WhichReg) :- I>Max, !, WhichReg=r(arity_error).
head_reg(I, Max, Head, Var, WhichReg) :- I=<Max,
   arg(I, Head, Arg),
   Arg==Var, !,
   WhichReg=I.
head_reg(I, Max, Head, Var, WhichReg) :- I=<Max,
   arg(I, Head, Arg),
   Arg\==Var, !,
   I1 is I+1,
   head_reg(I1, Max, Head, Var, WhichReg).

% Determine the set of registers in a head corresponding to a set of arguments:
head_regs(Head, Args, Regs) :-
   head_regs(Head, Args, Regs, []).

head_regs(Head, Args, Regs, Link) :-
   functor(Head, _, Arity),
   low_reg(L),
   head_regs(Arity, Head, L, Args, Regs, Link).

head_regs(I, _, _, _) --> {I=<0}, !.
head_regs(I, Head, L, Args) --> {I>0, arg(I,Head,X), inv(X,Args)}, !,
   [r(R)],
   {I1 is I-1},
   {R is I+L-1},
   head_regs(I1, Head, L, Args).
head_regs(I, Head, L, Args) --> {I>0}, !,
   {I1 is I-1},
   head_regs(I1, Head, L, Args).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

regalloc.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Register allocation:

% Notes:
% 1. Previously, nested usages had to be done from the inside out for
%    correctness. (Partially overlapping usages are not a problem.)
%    Presently, alloc_temps/2 has been changed to correct this problem.
%    Otherwise the registers allocated in the outer usages are not seen by
%    the inner usages.  The arrangement of arguments in a usage structure
%    together with the call to sort in usage solve this.  This is yet another
%    manifestation of the variable-instantiation-order problem that could be
%    solved by a variant of constraint programming.
% 2. This module is an example of general duality: to create structures
%    implicitly using backtracking or to create structures explicitly on the
%    heap.  The same time order in linear algorithms, but first method uses no
%    heap space. What are the true differences?  Execution time order?
% 3. Overflows temporaries into permanents if there are not enough registers
%    available.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Temporary and permanent register allocation in a clause:
% Returns number of permanents and a flag (yes/no) telling whether
% an environment is necessary.
% The varlist is the key to the operation of the whole caboodle.

% Needs 'AllocCl', a modified clause generated by clause_code
% which gives the variable use information necessary for allocation:
% 1. It contains terms of the form '$varlist'(List) where List is a
%    list of variables used in the order in which they are used.
% 2. The head and body goals do not contain structures in their arguments.
%    These structures have been unraveled by clause_code.
clause_allocate(AllocCl) :- clause_allocate(AllocCl, _, _).

clause_allocate(AllocCl, EnvNeeded, NumPerms) :-
   stats(reg,1),
   varlist(AllocCl, Varlist),
   stats(reg,2),
   usage(Varlist, Usage),
   stats(reg,3),
   alloc_voids(Usage),
   stats(reg,4),
   find_perms(Varlist, Perms),
   stats(reg,5),
   split_temps(Usage, Perms, TempUsage),
   alloc_temps(TempUsage, Varlist),
   stats(reg,6),
   split_perms(Usage, PermUsage),
   alloc_perms(PermUsage, Varlist),
   stats(reg,7),
   num_perms(PermUsage, NumPerms),
   env_needed(Varlist, NumPerms, EnvNeeded),
   stats(reg,8),
   !.
clause_allocate(AllocCl, _, _) :-
   error(['Register allocation could not be done for the clause:', nl,
          AllocCl, nl,
          'because of a bug in the register allocation routine.']).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Variable list of a clause:
% Very useful for register allocation.
% Includes argument registers for head and body goals.
% The varlist is calculated from the modified clause given by clause_code
% which contains goals tailored to give variable lists.

% The varlist contains the following possible entries:
%  V       (unallocated) variable
%  r(I)    temporary register
%  pref, V, W  preferably allocate variables V and W to the same reg.
%  fence       temporary variables don't survive beyond this.
% Note: pref and fence are created both here and in unify.pl.
% Note: bodylist is used also in clause_code (in map_varlist).
varlist(Cl, Varlist) :- varlist(Cl, Varlist, []).

varlist((Head:-Body)) --> /* headlist(Head), */ bodylist(Body), !.

% headlist(Head) --> {Head=..[H|Args], low_reg(L)}, headlist(L, Args).
% headlist(I, []) --> [].
% headlist(I, [A|Args]) --> [pref,r(I),A], {I1 is I+1}, headlist(I1, Args).

bodylist((Goal,Body)) --> !, bodylist(Goal), bodylist(Body).
bodylist(Goal) --> goallist(Goal).

% Unify goal has been replaced by a list of its variables:
goallist('$varlist'(VarList)) --> !, difflist(VarList).
goallist('$varlist'(VarList,Link)) --> !, insert(VarList,Link).
% For later: this clause needs modification for uninit_reg modes.
goallist(Goal) --> {  anyregs(Goal)}, !,
   varbag(Goal), fence_if_die(Goal).
goallist(Goal) --> {\+anyregs(Goal)}, !,
   {Goal=..[G|Args], low_reg(L)}, goallist(L, Args), fence_if_die(Goal).

goallist(I, []) --> [].
goallist(I, [A|Args]) --> {var(A)}, !,
   [pref,A,r(I)], {I1 is I+1}, goallist(I1, Args).
goallist(I, [A|Args]) --> {nonvar(A)}, !,
   [r(I)], {I1 is I+1}, goallist(I1, Args).

fence_if_die(Goal) --> {  survive(Goal)}, !.
fence_if_die(Goal) --> {\+survive(Goal)}, !, [fence].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Return flag (yes/no) if an environment is needed :

% An environment is needed if there are permanent variables or
% if there is a fence that is followed by non-trivial code.
env_needed(_,  NPerms, yes) :- NPerms>0, !.
env_needed(Varlist, _, yes) :- env_needed(Varlist), !.
env_needed(Varlist, _, no).

env_needed([V|Vs]) :- V==fence, \+trivial_moves(Vs).
env_needed([_|Vs]) :- env_needed(Vs).

trivial_moves([]).
trivial_moves([pref,R,R|Vs]) :- trivial_moves(Vs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Extract the usages of temporary/permanent variables:

split_temps([], _, []) :- !.
split_temps([U|Usage], Perms, [U|TempUsage]) :-
   get_usage(U, V, _, _), \+inv(V, Perms), !,
   split_temps(Usage, Perms, TempUsage).
split_temps([_|Usage], Perms, TempUsage) :-
   split_temps(Usage, Perms, TempUsage).

% All variables not yet allocated are permanents.
% (For correctness, must be done after temporary variable allocation.)
split_perms([], []).
split_perms([U|Usage], [U|PermUsage]) :-
   get_usage(U, V, _, _), var(V), !,
   split_perms(Usage, PermUsage).
split_perms([_|Usage], PermUsage) :-
   split_perms(Usage, PermUsage).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Get usage ranges of all variables in a varlist:
% Use of reverse so stack space needed stays proport. to len(Varlist).
usage(Varlist, Usage) :-
   varset(Varlist, Varset),
   init_usage(Varset, Usage),
   first_usage(Varlist, 1, Usage),
   length(Varlist, N),
   reverse(Varlist, Reverse),
   last_usage(Reverse, N, Usage).

init_usage([], []) :- !.
init_usage([V|Varset], [var(L,F,V)|Usage]) :-
   init_usage(Varset, Usage).

% These two routines are somewhat of a hack, alas:
first_usage([], _, _) :- !.
first_usage([X|Varlist], I, Usage) :-
   var(X), !,
   I1 is I+1,
   find_usage(X, Usage, First, _),
   fix_it(I, First),
   first_usage(Varlist, I1, Usage).
first_usage([X|Varlist], I, Usage) :-
   nonvar(X), !,
   I1 is I+1,
   first_usage(Varlist, I1, Usage).

last_usage([], _, _) :- !.
last_usage([X|Reverse], I, Usage) :-
   var(X), !,
   I1 is I-1,
   find_usage(X, Usage, _, Last),
   fix_it(I, Last),
   last_usage(Reverse, I1, Usage).
last_usage([X|Reverse], I, Usage) :-
   nonvar(X), !,
   I1 is I-1,
   last_usage(Reverse, I1, Usage).

find_usage(X, [var(L,F,V)|Set], First, Last) :-
   (X==V
    -> First=F, Last=L
     ; find_usage(X, Set, First, Last)
   ).

fix_it(I, X) :- (var(X) -> I=X; true).

get_usage(var(L,F,V), V, F, L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Find permanent variables:
% (This is an example of the use of several accumulators)

acc_info(vars,  V,  In, Out, includev(V,In,Out)). % Variables
acc_info(half,  Vs, In, Out, unionv(In,Vs,Out)).  % Half-permanents
acc_info(perms, V,  In, Out, includev(V,In,Out)). % Permanents

pred_info(permvars, 1, [vars,half,perms]).
pred_info(permstep, 1, [     half,perms]).

find_perms(Varlist, Perms) :- permvars(Varlist, [], Vars, [], Half, [], Perms).

% Handle the list of variables used in unification:
% Encounter of ’fence’ doesn’t allow registers to survive.
permvars([]) -->> !.
permvars([V|Vs]) -->> {var(V)},   !,  [V]:vars, permstep(V), permvars(Vs).
permvars([V|Vs]) -->> {V==fence}, !, Vars/vars, [Vars]:half, permvars(Vs).
permvars([V|Vs]) -->> {nonvar(V), V\==fence}, !, permvars(Vs).
```

```
% Step over a single variable in the permanent variable calculation:
permstep(V) -->> Half/half, {inv(V, Half)}, !, [V]:perms.
permstep(V) -->> !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Allocate permanent variables:

% Use a naive allocation for write-once permanents, and a more sophisticated
% allocation for standard permanents.
alloc_perms(_,   Varlist) :-   write_once, !, naive_alloc_perms(Varlist).
alloc_perms(PermUsage, _) :- \+write_once, !, soph_alloc_perms(PermUsage).

% *** Naive allocation that’s needed for write-once permanent variables:
naive_alloc_perms(Varlist) :-
   reverse(Varlist, Reverse),
   low_perm(L),
   naive_alloc_perms(Reverse, L, _).

naive_alloc_perms([]) --> !.
naive_alloc_perms([V|Vs]) --> alloc_if_var(V), naive_alloc_perms(Vs).

alloc_if_var(V, I, I1) :- (var(V) -> I1 is I+1, V=p(I); I1=I).

% *** More sophisticated allocation that can’t be used with write-once perms:
% Variables not yet allocated are allocated to permanents.
% Non-overlapping variables are allocated to the same register.
% Sort+reverse ensures that permanents are allocated last-first.
soph_alloc_perms(PermUsage) :-
   sort(PermUsage, SUsage),
   reverse(SUsage, RUsage),
   soph_alloc_perms(RUsage, RUsage).

soph_alloc_perms([], _) :- !.
soph_alloc_perms([X|Usage], AllUsage) :-
   get_usage(X, V, First, Last),
   low_perm(L), max_int(H), range(L, I, H), R=p(I),
   \+usage_overlap(First, Last, AllUsage, R),
   V=R,
   soph_alloc_perms(Usage, AllUsage).

% *** Calculate number of permanent variables:
num_perms(PermUsage, NumPerms) :-
   low_perm(P), P1 is P-1,
   num_perms(PermUsage, P1, NP),
   NumPerms is NP-P1.

num_perms([], N, N).
num_perms([U|PermUsage], I, N) :-
   get_usage(U, p(J), _, _),
   max(I, J, K),
   num_perms(PermUsage, K, N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Allocate temporary variables:
% This algorithm is O(v^2 * n) where v = number of variables in clause
% and n = number of occurrences of variables in the clause.
% Later clean up & make more efficient.

% Note: using generators and backtracking to allocate instead of creating
% structures on the heap and doing deterministic manipulations has the
% same time order, but uses much less heap space.

% Future speed improvement:
% Precalculate for each variable the list of preferred variables.
% When variables get allocated, the preferred list becomes more instantiated.
% Precalculate also for each variable the list of variables in whose rhs
% it occurs (the "inversion" of the above list).  This gives the dependency
% necessary for propagating variable allocations.

alloc_temps(Usage, Varlist) :-
   pref_closure(Usage, Varlist), % Allocate preferred registers.
   rest_temps(Usage, Varlist, Usage). % Allocate other registers.

% Allocate preferred registers & repeat until no more are allocated:
pref_closure(Usage, Varlist) :-
   pref_temps(Usage, Varlist, Usage, 0, NewN),
   pref_closure(Usage, Varlist, NewN).

pref_closure(Usage, Varlist, NewN) :- NewN>0, !, pref_closure(Usage, Varlist).
pref_closure(Usage, Varlist, NewN) :- NewN=0, !.

% (This routine counts the number of newly allocated variables.)
pref_temps([], _, _, InN, InN) :- !.
pref_temps([X|Usage], Varlist, AllUsage, InN, OutN) :-
   get_usage(X, V, First, Last),
   ( var(V), pref_allocate(V, Varlist, AllUsage, First, Last)
   -> MidN is InN+1
    ; MidN = InN
   ),
   pref_temps(Usage, Varlist, AllUsage, MidN, OutN).

% Allocate a preferred register without conflicts, if it exists.
pref_allocate(V, Varlist, AllUsage, First, Last) :-
   First=:=Last,
   prefer(V, Varlist, R), V=R.
pref_allocate(V, Varlist, AllUsage, First, Last) :-
   First=\=Last,
   prefer(V, Varlist, R),
   \+already_used(V, R, First, Last, Varlist, AllUsage),
   V=R.

rest_temps([], _, _) :- !.
rest_temps([X|Usage], Varlist, AllUsage) :-
   get_usage(X, V, First, Last),
   ( var(V), rest_allocate(V, Varlist, AllUsage, First, Last)
   -> % Optimization: interleave rest & pref calculation:
      pref_closure(AllUsage, Varlist)
     ; true
    ),
    rest_temps(Usage, Varlist, AllUsage).

% Allocate a register starting from the lowest register up:
% (It would be just as correct to do it in the other direction)
rest_allocate(V, Varlist, AllUsage, First, Last) :-
    low_reg(L), high_reg(H), range(L, I, H), R=r(I),
    \+already_used(V, R, First, Last, Varlist, AllUsage),
    V=R.

already_used(V, R, First, Last, Varlist, Usage) :-
    usage_overlap(First, Last, Usage, R), !.
already_used(V, R, First, Last, Varlist, Usage) :-
    reg_overlap(V, R, 1, First, Last, Varlist), !.

% Succeeds if register R is already used for another variable
% in between First and Last in Varlist.
% (Try compiling this one!)
reg_overlap(V, R, I, First, Last, [_|Varlist]) :- I=<First,
    I1 is I+1,
    reg_overlap(V, R, I1, First, Last, Varlist).
reg_overlap(V, R, I, First, Last, Varlist) :- First<I, I<Last,
    no_conflict(V, R, Varlist, NewVL),
    I1 is I+1,
    reg_overlap(V, R, I1, First, Last, NewVL).
reg_overlap(V, R, I, First, Last, [R2|Varlist]) :- First<I, I<Last,
    R==R2.

% It's ok if R is used in a pref-pair with V:
no_conflict(V, R, [P,X,Y|VL], VL) :- pref(P), (V==X, R==Y; V==Y, R==X), !.
no_conflict(V, R,     [P|VL], VL) :- R\==P, !.

% Succeeds if register R is already used by another variable in Usage:
% (i.e. there exists an allocated X such that R and X's usages overlap.)
% (This routine is also used for permanent variable allocation.)
usage_overlap(First, Last, [X|Usage], R) :-
    get_usage(X, V, F, L), nonvar(V),
    % The usages of R and X overlap:
    overlap(First, Last, F, L),
    % V and R use the same register:
    V=R, !.
usage_overlap(First, Last, [_|Usage], R) :-
    usage_overlap(First, Last, Usage, R).

% The intervals [A,B] and [C,D] overlap:
% (! Try compiling this one)
overlap(A, B, C, D) :- A<C, C<B. % Left edge overlaps.
overlap(A, B, C, D) :- A<D, D<B. % Right edge overlaps.
overlap(A, B, C, D) :- A<C, D<B. % Inside.
overlap(A, B, C, D) :- C<A, B<D. % Outside.

% R is a preferred register for V:
```

```
prefer(V, [P,X,R|Varlist], R) :-
   pref(P),
   V==X, temp_register(R).
prefer(V, [P,R,X|Varlist], R) :-
   pref(P),
   V==X, temp_register(R).
prefer(V, [_|Varlist], R) :- prefer(V, Varlist, R).

pref(P) :- P==pref.
temp_register(R) :- nonvar(R), R=r(I), integer(I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Allocate void variables:

% This method is weak because the varlist is merged for the read and write
% branches of unification.  However, for pure read or write mode it works
% quite well.

% This allocates all registers that occur only once to r(void).
% Peephole optimization removes the instructions loading r(void).

alloc_voids([]).
alloc_voids([U|Usage]) :-
   get_usage(U, V, F, L),
   (F=L -> V=r(void) ; true),
   alloc_voids(Usage).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

segment.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Split a procedure in standard form into segments which are used in
% determinism extraction.  Changes the order of goals when that better
% exposes the determinism.  Care is taken not to change the semantics
% when goals are reordered.

% Returns a list of bodies and a disjunction which incorporates the goals
% which are useful for procedural clause selection.
% Format as follows:
%  A segment head = Tests,$body(($b_name(Args..NewArgs):-Body),BodyVars,Entries)
%  A segment body = body(($b_name(Args..NewArgs):-Body),Entries)
% The $body structure is compiled as a call.  Useful information is wrapped
% into the $body structure for other parts of the compiler:
% 1. The body call $b_name(...), which is compiled as a call.
%    Currently, this call imposes an order on the arguments.  Later, better
%    interaction between the caller & callee (using preferred registers)
%    could get around this problem.
% 2. The Body itself, which is very useful for gathering information about
%    what comes AFTER this call (see choice_block in proc_code.pl and
%    req_uninit_varset in clause_code.pl).  It is used to look BEYOND this
%    call, even though the $body goal is compiled as a single call.
% 3. BodyVars - The set of all variables in the body.
% 4. Entries - a list of entry(NaAr,Form), i.e. entry labels and formulas with
%    which $body is called.  In clause_code.pl the correct code is generated
%    for each entry.

% Notes:
% 1. Only those tests all of whose variables are in the head and are not
%    bound are split off.
%    This makes point 2 below superfluous since the heads don't change.
%    I'll keep 2's code around for now.
% 2. For each choice in the disjunction, add arguments to make a $body head.
%    The extra arguments are those that are common between the tests and
%    the body of the choice:
%    BodyHead = Head U ((TestVars /\ BodyVars) - HeadVars)
%    The arguments already in the head are not duplicated.
%    The extra arguments link together the selection code and the
%    clause body code so that register allocation will be consistent.
%    Existing head arguments are kept so that there won't be any superfluous
%    move operations.
% 3. Create_bodies does a lot of effort to simplify the bodies and to give them
%    as many modes as possible.  This effort speeds up test set selection.
%    Should check to see that this effort isn't duplicated elsewhere.
% 4. Later improvements will also segment user-defined tests.
% 5. An effort is done to arrange goals as follows: non-binding tests,
%    var modes, others (incl. uninit modes).  Var modes are affected by
%    aliasing, uninit modes are not.  Compare compilation of ex(ncon) and
%    ex(l1).
% Entry point: a single procedure in standard form.
segment_disj(Head, Disj, SegDisj, Modes) -->
   {Modes=(Head:-Formula)},
   {varset(Head, HVar)},
   create_bodies(Disj, SDisj, LinkHeads, TVars, BVars, HVar, Formula),
   {create_heads(LinkHeads, Head, HVar, TVars, BVars)},
   {standard_disj(SDisj, SegDisj)}.

% CommonV is the set of variables in common between the Head+Test and
% the Body.  If CommonV is [] then no arguments need be passed to the
% clause body code.
create_heads([], _, _, [], []).
create_heads([LH|LinkHeads], Head, HV, [TV|TVars], [BV|BVars]) :-
   intersectv(TV, BV, AllExtraArgs),
   diffv(AllExtraArgs, HV, ExtraArgs),
   unionv(TV, HV, AllV),
   intersectv(AllV, BV, CommonV),
   new_head(CommonV, "$b_", Head, ExtraArgs, LH),
   create_heads(LinkHeads, Head, HV, TVars, BVars).

new_head(CommonV, Prefix, Head, ExtraArgs, NewHead) :-
   cons(CommonV), !,
   new_head(Prefix, Head, ExtraArgs, NewHead).
new_head(CommonV, Prefix, Head, ExtraArgs, NewHead) :-
   nil(CommonV), !,
   functor(Head, Name, Arity),
   gensym(Prefix, Name/Arity, NewHead).

new_head(Prefix, Head, ExtraArgs, NewHead) :-
   functor(Head,Name,Arity),
   Head=..[Name|Args],
   append(Args, ExtraArgs, NewArgs),
   gensym(Prefix, Name/Arity, NewName),
   NewHead=..[NewName|NewArgs].

create_bodies(fail, fail, [], [], [], _, _, Link, Link) :- !.
create_bodies((D;Disj), (SD;SegDisj), LHs, TVs, BVs, HVar, InF, Bs, Link) :-
   segment_test(D, SBody, Tests, HVar, InF),
   % Use InF to simplify Tests or vice-versa?
   (  length_test_user(SBody, Nt, Nu),
      compile_option(user_test_size(Su,St)),
      Nu=<Su, Nt=<St
   -> flat_conj((Tests,SBody), SD),
      Bs=Bs2,
      LHs=LHs2,
      TVs=TVs2,
      BVs=BVs2
    ; flat_conj(SBody, FBody),
      varset(Tests, TV),
      varset(FBody, BV),
      SD=(Tests,'$body'((LH:-FBody),BV,Entries)),
      Bs=[body((LH:-FBody),Entries)|Bs2],
      LHs=[LH|LHs2],
      TVs=[TV|TVs2],
      BVs=[BV|BVs2]
   ),
   create_bodies(Disj, SegDisj, LHs2, TVs2, BVs2, HVar, InF, Bs2, Link).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% New implementation of segmentation:
% Notes:
% 1. Big disadvantage of current selection scheme: A test may not create
%    new variables.
% 2. The mode formula is only updated for goals that remain in the ’test’
%    accumulator.
% 3. The ’testvars’ accumulator is only updated for var and uninit modes
%    that are pushed to the end.
% 4. Eventually could look beyond ’$body’, but what about the body in the list?
%    Will it just not be called, and therefore not compiled?  Will there be
%    code duplication?
% 5. For later: A block of unify goals may be rearranged in any way
%    without changing the semantics.  Would it be useful to have a separate
%    segment routine just for unify goals?  Problem: HOW to rearrange?
% 6. Related question: in (A=T1,B=T2) the mode var(B) is known.  Should the
%    var mode unification come first or last?  How to decide?

% testvars = variables that may not occur in a test, since their value
% is given by a var or uninit mode unification which has been reordered.
% sf = variables with a value.

acc_info(uninit, T, Out, In, (Out=(T,In)), _, _).
acc_info(test, T, Out, In, (Out=(T,In))).
acc_info(var, T, Out, In, (Out=(T,In))).
acc_info(testvars, X, In, Out, includev(X,In,Out)).
acc_info(form, F, InF, OutF, update_formula(F,InF,OutF)).
acc_info(sf, Vs, In, Out, unionv(Vs,In,Out)).

pred_info(segment_1, 2, [test,var,uninit,testvars,sf,form]).
pred_info(add_name_arity, 1, [test,form]).

% For debugging:
st(InBody, Head, InF) :-
   varset(Head, HVar),
   segment_test(InBody, OutB, Ts, HVar, InF),
   wn(Ts),
   wn(OutB).

% Keep all tests that bind only first-occurrence variables.
% Collect all var-mode and uninit-mode unifications & put
% them afterwards.
% Stop at a standard order test, since it depends on the result of
% unification.
segment_test(InBody, OutBody, Ts, HVar, InF) :-
   Te=true,
   OutBody=Vs, Ve=Us, Ue=Rest,
   segment_1(InBody, Rest, Ts,Te, Vs,Ve, Us,Ue, [], _, HVar, _, InF, _).
   % This is one way to calculate output mode, if it’s needed:
```

```
   % update_formula(Ts, InF, OutF).

segment_1(true, true) -->> !.
segment_1(B, Rest) -->> {next_goal(B, Goal, Body), split_unify(Goal, X, Term)},
   {var(X),atomic(Term)},
   F/form, {implies(F, nonvar(X))}, !,
   ['$name_arity'(X,Term,0)]:test,
   ['$name_arity'(X,Term,0)]:form,
   segment_1(Body, Rest).
segment_1(B, Rest) -->> {next_goal(B, Goal, Body), test(Goal)},
   {\+standard_order(Goal)},
   {varset(Goal, Vs)},
   TV/testvars,
   {intersectv(Vs, TV, [])},
   SF/sf,
   {subsetv(Vs, SF)},
   F/form,
   {bindset(Goal, F, [])}, !,
   add_name_arity(Goal),
   [Goal]:test,
   [Goal]:form,
   segment_1(Body, Rest).
segment_1(B, Rest) -->> {next_goal(B, Goal, Body), split_unify(Goal, X, Term)},
   {var(X),compound(Term)},
   F/form, {implies(F, nonvar(X))},
   SF/sf, {varset(Term, Vs)}, {intersectv(Vs, SF, [])}, !,
   {functor(Term,Na,Ar)},
   ['$name_arity'(X,Na,Ar)]:test,
   [Goal]:test,
   insert(InF, MidF):form, {update_formula(Goal, SF, InF, MidF)},
   [[X],Vs]:sf,
   segment_1(Body, Rest).
segment_1(B,    B) -->> {next_goal(B, Goal, Body), split_unify(Goal, X, Term)},
   {var(X),compound(Term)},
   F/form, {implies(F, nonvar(X))}, !,
   {functor(Term,Na,Ar)},
   ['$name_arity'(X,Na,Ar)]:test,
   ['$name_arity'(X,Na,Ar)]:form.
segment_1(B, Rest) -->> {next_goal(B, Goal, Body), split_unify(Goal, X, Term)},
   {var(X)},
   F/form, {implies(F, var(X))}, !,
   [Goal]:var,
   [X]:testvars,
   segment_1(Body, Rest).
segment_1(B, Rest) -->> {next_goal(B, Goal, Body), split_unify(Goal, X, Term)},
   {var(X)},
   F/form, {implies(F, uninit(any,X))}, !,
   [Goal]:uninit,
   [X]:testvars,
   segment_1(Body, Rest).
segment_1(Rest, Rest) -->> !.

% This could be extended to look inside '$body' goals:
next_goal((Goal,Body), Goal, Body) :- !.
next_goal(Goal, Goal, true).

% Add '$name_arity' for a unify goal when the argument is implied nonvar:
add_name_arity(Goal) -->>
   {split_unify(Goal, X, Term)},
   {var(X), nonvar(Term)},
   Form/form, {implies(Form, nonvar(X))}, !,
   {functor(Term, Na, Ar)},
   ['$name_arity'(X,Na,Ar)]:test,
   ['$name_arity'(X,Na,Ar)]:form.
add_name_arity(Goal) -->> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 'Naive' segmenting for argument selection without modes:
% All tests are collected, whether they bind or not.
% Also gets the set of args that are args of some unification that is
% not known to succeed (i.e. neither arg is known to be unbound).
% (The format of body-linkage is compatible with that done in the more
% discriminating segmenting done above)
% Note that the more discriminating segmenting must be compiled before
% the naive one, to instantiate the naive one's input formula, or else
% the compiler will think the naive one is not being called.
segment_all_disj(Disj, Head, Form, TestDisj, BodyList, GoodVars) :-
   varset(Head, HVs),
   unbound_set(Form, UVs),
   segment_all_disj(Disj, Head, HVs, UVs, TestDisj, VBag, [], BodyList),
   % Make sure that variable X is an argument of a unification
   % which does not have an arg known to be unbound.
   % (Being an argument of another test is not useful, since other
   % tests don't need the var/nonvar distinction)
   sort(VBag, VSet),
   intersectv(VSet, HVs, GoodVars).

segment_all_disj(fail, H, _, _, fail, Link, Link, []).
segment_all_disj((D;Disj), H, HVs, UVs, (T;TDisj), Vs, Link, BodyList) :-
        segment_all_conj(D, Tests, Body, UVs, Vs, Mid),
   varset(Tests, TVs),
   (  length_test_user(Body, Nt, Nu),
      compile_option(user_test_size(Su,St)),
      Nu=<Su, Nt=<St
   -> T = (Tests,Body),
      BodyList = NewList
   ;  varset(Body, BVs),
      intersectv(TVs, BVs, X),
      diffv(X, HVs, ExtraArgs),
           new_head("$s_", H, ExtraArgs, LH),
           T = (Tests,'$body'((LH:-Body),BVs,Entries)),
           BodyList = [body((LH:-Body),Entries)|NewList]
   ),
        segment_all_disj(Disj, H, HVs, UVs, TDisj, Mid, Link, NewList).

segment_all_conj(true, true, true, UVs) --> !.
segment_all_conj((T,B), (T,Ts), Body, UVs) --> {T=(X=Y)},
   {filter_vars([X,Y], UBag)},
   {sort(UBag, USet)},
   {disjointv(USet, UVs)}, !,
   insert(USet),
        segment_all_conj(B, Ts, Body, UVs).
segment_all_conj((T,B), (T,Ts), Body, UVs) --> {test(T)}, !,
        segment_all_conj(B, Ts, Body, UVs).
segment_all_conj((T,B), true, (T,B), UVs) --> {\+test(T)}, !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

selection.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Determinism extraction from a predicate.
% Convert a predicate in standard form into a form that exposes the
% inherent determinism.

% Files needed: conditions, standardize, utility

% Notes:
% 1. It is necessary to retain ordering of tests because of data dependencies,
%    e.g. in (h(A) :- A=[X], X<3) the unification must be done before the test.
%    Factoring takes care of some cases of this.  In other cases a fundamental
%    problem remains: (A=[X],X<10) and (A=[Y],Y>=10) are not recognized as
%    mutex because X and Y are different variables!  The algorithm
%    given here is too simplistic.  The written notes contain one possible
%    solution to the problem using the notion of the 'origin' of a variable.
%    The currently implemented solution is to do a head factorization first,
%    and to allow segmenting to take place only for tests all of whose vars
%    are in the head.  Variables with the same origin are considered identical.
% 2. Even for rather simple predicates and a simple collection of test-sets,
%    a large number of mostly useless test-sets is generated in find_testset.
%    This can lead to memory problems with large predicates.  One solution is
%    to allow the measure of goodness parameter of each test-set to influence
%    find_testset to gather only test-sets that are sufficiently good.
% 4. Nested testing in code_testset: new modes are (Test,Formula), new prev.vars
%    are (TVars U HVars).  Selection is called recursively.
% 5. The output has the form $case(Name,Ident,Case)
%    where Name & Ident identify the particular test-set,
%    Case = ($test(Direc,Pre,Test,BV,Code);$test(..);...;$else(Pre,BV,Code))
%    Direc = the direction of each test.
%    This form allows straightforward code generation.
% 6. Giving lots of modes will slow down selection significantly.  Need to
%    optimize the speed of mutex?

% Possible extensions:
% 1. instead of disj only as data struc, better is disj+prev vars
%    where prev vars = set of vars existing before the disj.
%    Flattening takes care of this at pred. level, but need to consider
%    nested case selection.
% 2. internal_testset predicate: user-defined tests can have different
%    arg. variables in different clauses - the actual testset depends
%    on modes, possibly on def. of the predicate.  Unraveling removes
%    some of the correspondence between clauses - can worsen results.
%    This will be solved later using idea of variable origins.
%    At the same time, can remove the use of Prolog variables to represent
%    variables in the source code.  These are really meta-variables, and
%    it may be better to represent them as constants or compound terms.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% *** Debug hook

% Assume predicate is a single clause, not in standard form:
s((Head:-Body), Modes, (Head:-Modes,Code)) :-
   standard_disj(Body, StdDisj),
   selection((Head:-Modes,StdDisj), (Head:-Modes,Code)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generate deterministic code from a predicate:

% Entry point 1: Assume predicate is in standard form.
%    The result is not simplified.
selection_disj(Head, Disj, Modes, CaseDisj) :-
   copy(Modes, (Head:-Formula)),
   varset(Head, HVars),
   subsume(Formula, Disj, SDisj),
   selection_3(SDisj, Head, [], HVars, Formula, CaseDisj), !.

% Entry point 2: Assume source is a conjunction,
%    or a disjunction of conjunctions.
%    The result is not simplified.
selection(Source, Head, PrevKeys, PrevVars, Formula, Code) :-
   simplify(Source, SimpSource),
   selection_3(SimpSource, Head, PrevKeys, PrevVars, Formula, Code), !.

% Entry point 3: Assume predicate is a simplified disjunction and the
%    previously existing variables are known.
%    The result is not simplified.
%    There must be more than one choice for any selection to be done.
selection_3(Disj, Head, PrevKeys, PrevVars, Formula, Code) :-
   length_disj(Disj, N), N>1,
   stats(s,1),
   find_testset(Disj, Head, PrevKeys, PrevVars, Formula, Tests, TestSets),
   stats(s,2),
   pick_testset(TestSets, Tests, TestSet), !,
   stats(s,3),
   thin_testset(TestSet, ThinTestSet),
   stats(s,4),
   code_testset(ThinTestSet, Head, PrevKeys, PrevVars, Formula, Disj,Code),
   stats(s,5),
   (compile_option(debug)
   -> write('Selection code:'), nl,
      write(Code), nl
   ;  true
   ), !.
selection_3(Disj, _, PrevKeys, PrevVars, Formula, Disj).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Collect the test-sets of all tests in a predicate.
% A predicate is represented as (Head:-Disj), and only the disjunction
% is given as an argument here.
% A test may be in several test-sets.  This code recognizes that fact.
% An test-set member is represented here as the structure
% (key/2 is used only to gather like test-sets efficiently with keysort)
%  key(Name,Ident)-val(Goal,Test,ClauseNum,GoalNum,Direc,ClauseBody)
% with meaning:
%  key(Name,Ident)  Uniquely identifies the testset. (Name is ground and
%          Ident contains the variables needed by the test-set.)
%  Goal     The source goal that caused selection of this testset.
%  Test     The test that can be done by the testset.  Test does
%          part of the work of Goal, i.e. Goal implies Test.
%  ClauseNum   Number of the clause containing Goal (starting from 1).
%  GoalNum     Number of the goal in the clause (starting from 1).
%  Direc       Direction of the branch.
%  ClauseBody  The body of the clause containing Goal.
% A returned testset is a list of val(Goal,Test,ClauseNum,GoalNum,Direc).
% Execution time of this module is O(c + t log t) where c = number of clauses,
% t = total number of tests in all clauses.
% Needed: testset/6 predicate which depends on the ISP.

find_testset(Disj, Head, PrevKeys, PrevVars, Formula, UnorderTests, TestSets) :-
   unbound_set(Formula, US),
   pred_gather(Disj, Head, PrevKeys, PrevVars, US, Formula, 1,
           UnorderTests, []),
   keysort(UnorderTests, Ordered),
   part_testset(Ordered, TestSets).

% *** Gather all test-set members in a predicate:
pred_gather((Conj;Disj), Head, PrevKeys, PrevVars, US, Formula, Nc) --> !,
   {flat_conj(Conj, Flat)},
   clause_gather(Flat, Head, Conj, PrevKeys, PrevVars, US, Formula,Nc,1,_),
   {Nc1 is Nc+1},
   pred_gather(Disj, Head, PrevKeys, PrevVars, US, Formula, Nc1).
pred_gather(Conj, Head, PrevKeys, PrevVars, US, Formula, Nc) -->
   {flat_conj(Conj, Flat)},
   clause_gather(Flat, Head, Conj, PrevKeys, PrevVars, US, Formula,Nc,1,_).

% Gather all test-set members in a clause.
% Constraints:
% Don’t pick a test-set when:
% 1. It has been used already, or
% 2. Formula => Test, i.e. no extra information is obtained from
%    picking it since it is already known.
% 3. For an arithmetic relation, only pick testsets with
%    the same trapping behavior.
% 4. It needs an environment (\+survive).
% 5. It has arguments known from Formula to be unbound.
% 6. Stop traversing the clause when encountering a cut.
% Still to do: compound terms as arguments of tests.
clause_gather(true, _, _, _, _, _, _, _, Nb, Nb) --> [].
clause_gather((Goal,Body), _, _, _, _, _, _, _, Nb, Ob) -->
   {cut_p(Goal)}, !, {Nb=Ob}.
clause_gather((Goal,Body), Head, Cl, PrevKeys, PrevVars, US, Form, Nc,Nb,Ob) -->
   one_clause_gather(Goal, Head, Cl, PrevKeys, PrevVars, US,Form,Nc,Nb,Mb),
   !,
   clause_gather(Body, Head, Cl, PrevKeys, PrevVars, US, Form, Nc, Mb, Ob).
```

```
% Default clause:
clause_gather((_,Body), Head, Cl, PrevKeys, PrevVars, US, Form, Nc, Nb, Ob) -->
   {Mb is Nb+1},
   clause_gather(Body, Head, Cl, PrevKeys, PrevVars, US, Form, Nc, Mb, Ob).

one_clause_gather(Goal, Head, Cl, PrevKeys, PrevVars, US, Form, Nc, Nb, Ob) -->
   {Ob is Nb+1},
   {bagof(key(Name,Ident)-val(Goal,Test,Nc,Nb,Direc,_),
          valid_testset(Goal, Head, Form, Name, Ident, Direc, Test,
                 PrevKeys, PrevVars, US),
          Bag)}, !,
   {make_correct(Goal, Cl, Bag)},
   difflist(Bag).

% The conditions upon which it is valid to pick a test-set:
valid_testset(Goal, Head, Form, Name, Ident, Direc, Test,
         PrevKeys, PrevVars, US) :-
   internal_testset(Goal,Form,Name,Ident,Direc,T),
   logical_simplify(T, Test),
   relational_check(Goal,Form,Name),
   \+memberv(key(Name,Ident),PrevKeys),
   \+implies(Form,Test),
   survive(Test),
   varset(Goal,Vs), subsetv(Vs,PrevVars),
   varset(Ident,Is), disjointv(Is,US),
   valid_option(Goal, Head, Vs).

% For evaluation of determinism selection:
valid_option(Goal, Head, Vs) :-
   firstarg_option(Head, Goal, Vs),
   goal_type(Head, Goal, Type),
   test_option(Type), !.

% For evaluation of determinism selection:
% Allow only tests containing the first argument
firstarg_option(Head, Goal, Vs) :-
   compile_option(firstarg),
   \+unify_p(Goal),
   !,
   arg(1, Head, X),
   memberv(X, Vs).
firstarg_option(Head, Goal, Vs) :-
   compile_option(firstarg),
   unify_p(Goal),
   !,
   split_unify(Goal, X, Y),
   arg(1, Head, Z),
   X==Z.
firstarg_option(_, _, _).

% For evaluation of determinism selection:
% Allow only tests of a given type:
test_option(unify)     :- compile_option(test), compile_option(test_unify), !.
test_option(arith)     :- compile_option(test), compile_option(test_arith), !.
test_option(typecheck) :- compile_option(test),compile_option(test_typecheck),!.
test_option(_) :- \+ compile_option(test).

% For evaluation of determinism selection:
% Get type of test:
goal_type(_,      Goal, unify) :- split_unify_v_nv(Goal, _, _), !.
goal_type(H,    var(_), unify) :- select_option(H, _), !.
goal_type(H, nonvar(_), unify) :- select_option(H, _), !.
goal_type(_, '$name_arity'(_,_,_), unify) :- !.
goal_type(_,      Goal, arith) :- encode_relop(Goal, _, _, _, arith), !.
goal_type(_,      Goal, typecheck).

% Link to external testset/4 predicate:
% Note: When unification binds then it's usually not a test, because it
%  usually succeeds.  Therefore unification is only accepted here when it
%  binds nothing.  This is a heuristic incorporated to take care of
%  predicates such as max/3 when mode analysis doesn't say that one of
%  the arguments is a variable.
% (Generalize later to: a goal that always succeeds is not a test.)
% Note: The predicate valid after the testset is returned in Test.  For unify
%  goals, only the top level is returned (as a 'functor' goal),  since the
%  tests of a testset do not currently look
%  inside of compound terms.
internal_testset(Goal, Formula, Name, Ident, Direc, Test) :-
   bindbag(Goal, Formula, []),
   testset(Goal, Test, Name, Ident, Direc).
internal_testset(A=B, Formula, equal, v(A,B), true, '$equal'(A,B)) :-
   implies(Formula, (atomic(A),atomic(B))), !.
% Take only tests that don't bind:
internal_testset(Goal, Formula, Name, Ident, Direc, Test) :-
   \+unify_p(Goal),
   bindbag(Goal, Formula, []),
   testset(Goal, Test, Name, Ident, Direc).

% Check whether a particular testset is allowed for a particular goal:
% This is used to guarantee that a relational operator that needs
% integers, but has some trapping behavior on non-integers, will choose
% a testset that has the same behavior on non-integers.
relational_check(Goal, Formula, Name) :-
   \+relational_test(Goal, X, Y), !.
relational_check(Goal, Formula, Name) :-
   relational_test(Goal, X, Y),
   relational_testset(Name), !.
relational_check(Goal, Formula, Name) :-
   relational_test(Goal, X, Y),
   \+relational_testset(Name),
   logical_simplify((integer(X),integer(Y)), Ints),
   implies(Formula, Ints), !.

% Ensure that variables in Bag are instantiated correctly:
% Note that logically this predicate is superfluous.
% It is needed because bagof generates new sets of variables.
make_correct(G, Cl, [_-val(G,_,_,_,_,Cl)|Bag]) :- !, make_correct(G, Cl, Bag).
make_correct(_, _, []).
```

```
% *** Partition all contiguous matching keys into separate test-sets:
part_testset(List, [Key-[Val|C]|TestSets]) :-
   List=[Key-Val|Rest], !,
   one_testset(Key, Val, Rest, C, NewList),
   part_testset(NewList, TestSets).
part_testset([], []).

% Gather all contiguous keys matching Key into a single test-set:
% Take only one value of a series of contiguous identical ones.
one_testset(Key, Val, [K-V|List], TestSet, NewList) :-
   Key==K, Val==V, !,
   one_testset(Key, V, List, TestSet, NewList).
one_testset(Key, Val, [K-V|List], [V|TestSet], NewList) :-
   Key==K, Val\==V, !,
   one_testset(Key, V, List, TestSet, NewList).
one_testset(Key, Val, [K-V|List], [], [K-V|List]) :-
   Key\==K, !.
one_testset(_, _, [], [], []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Pick the best test-set.
% Methods:
% For later:
%    Calculate information (in #bits) for each test-set, pick maximum.
%    This is done by seeing how many clauses are mutex with each member
%    of the test-set, and using the info theoretic formula sum(Pi log Pi)
%    to get a bit value.  This may be slow.
% Now:
% 1. Pick the test-set with largest value of goodness function.  This is simple,
%    but it doesn't always give good results.  Speed O(t) where t = total number
%    of tests.  The goodness function is architecture-dependent and is therefore
%    defined in testset.pl.
% 2. If the first test in the predicate is a relational operator then pick the
%    best of the relational testsets.  If the first test is not a relational
%    operator, then pick the best of the non-relational testsets.  This is for
%    correct handling of the trapping behavior of relational tests.
%    However, it's not complete yet.

% Method 2:
pick_testset(TestSets, Tests, KCBest) :-
   first_test(Tests, FK-FV),
   comment(['First test is ',FK-FV]),
   rel_flag(FK, Flag),
   pick_eq(none-[], 0, TestSets, KCBest, GBest, Flag),
   % Check that one was chosen:
   GBest>0, !.

pick_eq(KC, G, [K2-C2|TestSets], KCBest, GBest, Flag) :- !,
   ((rel_flag(K2, F), F=Flag,
     goodness_testset(K2,C2,Goodness), Goodness>G)
   -> pick_eq(K2-C2,Goodness,TestSets, KCBest, GBest, Flag)
    ; pick_eq(KC, G, TestSets, KCBest, GBest, Flag)
   ).
pick_eq(KC, G, [], KC, G, _).

% *** Find the first goal in the list of tests:
first_test(Tests, First) :-
   Tests=[T|Ts],
   T=_-val(_,_,Nc,Nb,_,_),
   first_test(Ts, Nc, Nb, T, First).

first_test([], _, _, InT, InT).
first_test([T|Ts], Ic, Ib, InT, OutT) :-
   T=_-val(_,_,Nc,Nb,_,_),
   (Nc<Ic; Nc=:=Ic, Nb<Ib), !,
   first_test(Ts, Nc, Nb, T, OutT).
first_test([T|Ts], Ic, Ib, InT, OutT) :-
   first_test(Ts, Ic, Ib, InT, OutT).

rel_flag(key(Name,_), yes) :-   relational_testset(Name), !.
rel_flag(key(Name,_),  no) :- \+relational_testset(Name), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Make sure a test-set contains no redundant information.
% Such information results in redundant code and increased compilation time.

thin_testset(Key-TestSet, Key-Thin) :-
   show_tests(TestSet, ShowTs),
   keysort(ShowTs, SortTs),
   make_thin(SortTs, Thin), !. % Remove duplicate tests.

% Expose the clause number for keysort:
show_clnum([], []).
show_clnum([val(A,T,B,C,D,E)|Ts], [B-val(A,T,B,C,D,E)|Ks]) :-
   show_clnum(Ts, Ks).

% Expose the Test+Direc for keysort:
show_tests([], []).
show_tests([val(A,T,B,C,D,E)|Ts], [[D|T]-val(A,T,B,C,D,E)|Ks]):-
   show_tests(Ts, Ks).

% Keep only one unique copy of each [D|T]:
make_thin([T|Ts], Thin) :- make_thin(T, Ts, Thin).

make_thin(K-V, [], [V]).
make_thin(K1-V1, [K2-V2|Ts],      Thin) :-  K1==K2,!,make_thin(K2-V2, Ts, Thin).
make_thin(K1-V1, [K2-V2|Ts], [V1|Thin]) :- K1\==K2,!,make_thin(K2-V2, Ts, Thin).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generate case selection code using an test-set:

% Bugs:
% 1. Must keep order of tests unchanged, i.e. variables in a test may get
%    their values from a previous unification, and that unification must
%    preceed the test.

% Notes:
% 1. This version handles tests that bind, although such tests are not presently
%    used.
% 2. For each test, collect the clauses that are not mutually exclusive with it.
%    Format: $test(Direc,Preamble,Test,BindVars,Disj)
%    where:
%  Direc = direction of branch
%  Preamble = saves and restores needed for tests that bind vars.
%  Test = the test
%  BindVars = the vars that may be bound in the test
%  Disj = the code to be executed if the test succeeds
%    Also collect an '$else' case of clauses selectable if none of the tests
%    succeed.
%    Format: $else(Preamble,BindVars,Disj)
% 3. Needs Formula parameter to do correct binding code generation.  Note that
%    present alg. does not look at body for determining which vars are to be
%    saved & restored.  A more accurate alg. would set
%    BTVars = BTVars /\ (Vars in SDisj).
% 4. The predicates extract_else_disj and merge_disj_list are needed to reduce
%    execution time from O(test_clauses ^ 2) to O(test_clauses * other_clauses).
%    This shows up in the compilation of large databases of facts.
% 5. Note that the '$else' case is what needs to be executed if all others fail.
%    It may be that the others cannot all fail at the same time.  In that
%    situation, the '$else' case may be ignored.

code_testset(Key-TestSet, Head, PrevKeys, HVars, Formula, Disj,
        '$case'(Name,Ident,Code)) :-
   Key=key(Name,Ident),
   % skim_testset(TestSet, SkimSet),
   get_clause_nums(TestSet, Nums, []),
   sort(Nums, SNums),
   extract_else_disj(SNums, 1, Disj, EDisj, EList),
   % comment(['Clnums: ',SNums,' EDisj: ',EDisj,' EList: ',EList]),
   full_testset(Name, Ident, TestSet, NewTestSet),
   code_testset(NewTestSet, Head, [Key|PrevKeys], HVars, Formula, [],
            EList, EDisj, EDisj, Code).

code_testset([val(_,Test,Nc,_,Direc,Cl)|TestSet],
        Head, PK, HV, F, BVars, EList, EDisj, Else,
        ('$test'(Direc,Preamble,Test,BTVars,Code);Case)) :-
   needset(Test, HV, TVars),
   bindset(Test, F, BTVars),
   restore_block(TVars, BVars, Preamble, T),
   save_block(BTVars, T, true),
   unionv(BVars, BTVars, BVars2),
   merge_disj_list(EList, Nc, EDisj, Cl, MDisj),
   % comment(['Merge: ',EList, Nc, EDisj, Cl, MDisj]),
   subsume(Test, MDisj, SDisj),
   % comment(['Subsume: ',Test, SDisj]),
   else(Test, Else, NewElse),
   % comment(['Else: ',Else, NewElse]),
   % Nested det. extraction of SDisj is done here:
```

```
   % If SDisj is not simplified then do not continue recursion.
   % (commented out since for ex(a13) this is not the right condition)
   ( fail /* Disj==SDisj */ ->
     Code=SDisj
   ; unionv(TVars, HV, PrevVars),
   % \/ The following should be replaced by a non-monotonic update_formula
     logical_subsume(Test, F, SF),
     logical_simplify((Test,SF), NewF),
          selection(SDisj, Head, PK, PrevVars, NewF, Code)
   ),
   code_testset(TestSet, Head, PK, HV, F, BVars2, EList, EDisj, NewElse,
            Case),!.
code_testset([], Head, PK, HV, F, BVars, EList, EDisj, Else,
             '$else'(Preamble,BVars,Code)) :-
   restore_block(BVars, BVars, Preamble, true),
   selection(Else, Head, PK, HV, F, Code).

get_clause_nums([]) --> [].
get_clause_nums([val(_,_,Nc,_,_,_)|TS]) --> [Nc], get_clause_nums(TS).

% *** Remove all clauses whose number is in the set Nums.
% Retain the other clauses & their numbers.
extract_else_disj([N|Nums], I, (Conj;Disj), (Conj;EDisj), [I|EList]) :-
   I<N, !,
   I1 is I+1,
   extract_else_disj([N|Nums], I1, Disj, EDisj, EList).
extract_else_disj([I|Nums], I, (_;Disj), EDisj, EList) :- !,
   I1 is I+1,
   extract_else_disj(Nums, I1, Disj, EDisj, EList).
extract_else_disj([I],      I,        _,  fail, []).
extract_else_disj([], I, Disj, Disj, EList) :-
   number_disj(Disj, I, EList).

number_disj(fail, _, []) :- !.
number_disj((_;Disj), I, [I|List]) :- !, I1 is I+1, number_disj(Disj, I1, List).
number_disj(_, I, [I]).

% *** Merge the current clause into the other (non-testset) clauses:
merge_disj_list(_, none, EDisj, _, EDisj) :- !.
merge_disj_list([I|EList], Nc, (Conj;EDisj), Cl, (Conj;MDisj)) :-
   I<Nc, !,
   merge_disj_list(EList, Nc, EDisj, Cl, MDisj).
merge_disj_list(_, _, EDisj, Cl, (Cl;EDisj)).

% *** Determine the variables that Test needs:
needset(Test, HVars, TVars) :- varset(Test, TV), intersectv(HVars, TV, TVars).

% *** Generate blocks of save and restore goals.
% Do save(BT) for all BT in BTs.
% Assumes ordered lists as sets.
save_block([BT|BTs]) --> co(save(BT)), save_block(BTs).
save_block([]) --> [].

% Do restore(T) if T in Ts/\Bs.
% Assumes ordered lists as sets.
restore_block([T|Ts], [B|Bs]) --> {T@<B}, restore_block(Ts, [B|Bs]).
restore_block([T|Ts], [B|Bs]) --> {T==B}, co(restore(T)), restore_block(Ts, Bs).
restore_block([T|Ts], [B|Bs]) --> {T@>B}, restore_block([T|Ts], Bs).
restore_block(_, []) --> [].
restore_block([], _) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

standard.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Standard form:
% Utilities for converting Prolog clauses into kernel Prolog.

% Notes:
% 1. In an stree, Head in mode formula and Head of disjunction are the same.
% 2. Goals that are variables must be surrounded by call() and reported.
% 3. The standard form allows easy ’cdring down’ conjunctions and disjunctions.
% 4. Standard form of a procedure is (Head:-Disj) where Disj is a disjunction
%    whose choices are all in standard form and all the variables of Head
%    are distinct.

% Top level call:
ptrees_to_strees([], []).
ptrees_to_strees([P|Ps], [S|Ss]) :-
   ptree_to_stree(P, S),
   ptrees_to_strees(Ps, Ss).

% Convert a ptree to an stree:
% (Selection limit is calculated in compile_strees)
ptree_to_stree(ptree(Name,Cls,Mode,PDummy),
          stree(Name,(Head:-Disj),(Head:-Form),OldHeads,SDummy,_)) :- !,
   standard_cls(Cls, Mode, Head, OldHeads, Disj),
   copy(Mode, (Head:-Form)),
   ptrees_to_strees(PDummy, SDummy).
ptree_to_stree(D, D) :- directive(D), !.

% Convert a procedure Cls (a list of raw input clauses)
% into a Head and a single disjunction in standard form.
% (Also returns the old clause heads so that they can be unraveled
% again in analyze with the information gleaned there.)
standard_cls([], _, _, [], fail).
standard_cls([C|Cls], Mode, Head, [OldHead|OHs], (Body;Disj)) :-
   expand_term(C, E),
   copy(E, Cl),
   standard_form(Cl, Mode, OldHead, Head, Body),
   standard_cls(Cls, Mode, Head, OHs, Disj).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Convert a clause to standard form:

% Standard form of a clause =
% 1. Conjunctions and disjunctions are reorganized to be right associative.
% 2. Conjunctions have no internal true and are terminated by true.
% 3. Disjunctions have no internal fail and are terminated by fail.
% 4. Single goals inside disjunctions are considered as conjunctions.
% 5. Single goals inside conjunctions are just single goals (except ->, see 9)
% 6. The head is unraveled, body goals are unchanged.
% 7. \+ is converted to ->.
% 8. Arguments of -> are in standard conjunctive form, i.e. terminated by true.
% 9. (A->B) as a goal in a conjunction is terminated by fail like a disjunction.
% 10. Unify goals are written as X=T where if T is var then X is var.

% Note:
% 1. The single asymmetry between conjunctions and disjunctions exists
%    to make removing disjunctions from clauses easier.

% Assumes that clause head has distinct variables:
standard_form(Clause, Head, SBody) :-
   split(Clause, Head, Body),
   simplify(Body, SimpBody),
   standard_conj(SimpBody, SBody).

% Any input clause is acceptable:
% This version will also unravel the clause’s head, using the mode information
% to create the resulting unifications to aid determinism extraction.
standard_form(Clause, Mode, Head, SHead, SBody) :-
   split(Clause, Head, Body),
   simplify(Body, SimpBody),
   unr_head(Head, Mode, SHead, SBody, B),
   standard_conj(SimpBody, B).

standard_conj(Body, SBody) :- conj(Body, SBody, true).
standard_disj(Body, SBody) :- disj(Body, SBody, fail).

conj((A,B)) --> !, conj(A), conj(B).
conj(true)  --> !.
conj(G) --> {\+conj_p(G)}, inside_conj(G).

disj((A;B)) --> !, disj(A), disj(B).
disj(fail)  --> !.
disj(G) --> {\+disj_p(G)}, inside_disj(G).

% Note: C-Prolog does not recognize if-then-else inside DCG’s.
inside_disj((A->B)) --> {!, conj(A,C,true), conj(B,D,true)}, di((C->D)).
inside_disj(G) --> {conj_p(G), !, conj(G,GC,true)}, di(GC).
inside_disj(G) --> {\+conj_p(G), !, co(G,GC,true)}, di(GC).

inside_conj(\+(A)) --> {!, simplify(\+(A),B)}, negation_as_failure(B).
inside_conj((A->B)) --> {!, conj(A,C,true), conj(B,D,true)}, co(((C->D);fail)).
inside_conj(G) --> {disj_p(G), !, disj(G,GD,fail)}, co(GD).
inside_conj(G) --> {\+disj_p(G), \+anyregs(G)}, !, co(G).
inside_conj(G) --> {\+disj_p(G),   anyregs(G)}, !, unr_goal(G).

negation_as_failure(G) --> {\+(G=(\+(_)))}, !, conj(G).
negation_as_failure(\+(G)) --> {conj(G,GC,true)}, co((GC->fail,true;true;fail)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Utilities for unraveling a goal into a new goal and a series of unifications:

% Unraveling of the head is done in two steps.  First, those arguments that
% are known to be nonvar or unbound at run-time are split off from the others.
% The nonvars are put last & the unbounds are put first in the argument list.
% Second, this list is traversed & explicit unify goals are created for args
% that are identical.  Since the nonvars are last, they are unified together
% (if they are identical).  Since the unbounds are first, their arguments are
% replaced before the others, which means that any other tests in the clause
% DON'T get unbound arguments.
% The nonvar unifications are put first, followed by the other unifications
% and finally the unbound unifications, which always succeed.
% The nonvar unify goals are put before the other unify goals, and the
% unbound unify goals are put last.
% Goals which are not unbound (i.e. nonvar & other) are replaced first at the
% highest argument number, and the result is reversed so that unifications
% of low argument numbers come first.
% This transformation encourages better determinism extraction.

acc_info(unbound, X, Out, In, (Out=[X|In])).
acc_info(  other, X, Out, In, (Out=[X|In])).
acc_info( nonvar, X, Out, In, (Out=[X|In])).
acc_info( nonvar_goal, G, Out, In, (Out=(G,In))).
acc_info(  other_goal, G, Out, In, (Out=(G,In))).
acc_info(unbound_goal, G, Out, In, (Out=(G,In))).

pred_info(split_nonvars, 5, [unbound,other,nonvar]).
pred_info(    match_unr, 5, [nonvar_goal,other_goal,unbound_goal]).
pred_info(     conj_unr, 4, [nonvar_goal,other_goal,unbound_goal]).

unr_head(Head, (_:-true), NewHead, Conj, Link) :- !,
   functor(Head, _, A),
   downrange_list(A, 1, ArgNums),
   match_unr(ArgNums, [], [], Head, NewHead, Conj, Link).
unr_head(Head, (FHead:-Formula), NewHead, Conj, Link) :-
   functor(Head, _, A),
   split_nonvars(1, Formula, FHead, Head, A, Unb,[], Other,[], Nonvar,[]),
   reverse(Other, ROther),
   reverse(Nonvar, RNonvar),
   append(Unb, ROther, RNonvar, ArgNums),
   match_unr(ArgNums, Nonvar, Unb, Head, NewHead, Conj, Link).

match_unr(ArgNums, Nonvar, Unb, Head, NewHead, Conj, Link) :-
   functor(Head, N, A),
   functor(NewHead, N, A),
   match_unr(ArgNums, Nonvar, Unb, Head, NewHead, NC,true,OC,true,UC,Link),
   reverse_conj(NC, RNC),
   reverse_conj(OC, ROC),
   append_conj(RNC, ROC, UC, Conj).

% Split arguments into those implied to be nonvar, those implied to be unbound,
% and others.  Note that FHead and Head must be kept separate, because Head's
% arguments can be anything--unifying them together ruins the form of FHead.
split_nonvars(I, Formula, FHead, Head, A) -->> {I>A}, !.
split_nonvars(I, Formula, FHead, Head, A) -->> {I=<A},
   {arg(I, FHead, Y), implies(Formula, nonvar(Y))}, !,
   [I]:nonvar,
   {I1 is I+1},
   split_nonvars(I1, Formula, FHead, Head, A).
split_nonvars(I, Formula, FHead, Head, A) -->> {I=<A},
   {arg(I, FHead, Y), implies(Formula, unbound(Y))}, !,
   [I]:unbound,
   {I1 is I+1},
   split_nonvars(I1, Formula, FHead, Head, A).
split_nonvars(I, Formula, FHead, Head, A) -->> {I=<A}, !,
   [I]:other,
   {I1 is I+1},
   split_nonvars(I1, Formula, FHead, Head, A).

% Unravel head so that all its arguments become different variables:
% Nonvar & unbound are handled correctly due to the order of ArgNums.
match_unr([], _, _, _, _) -->> !.
match_unr([I|ArgNums], Nonvar, Unb, Head, NewHead) -->>
   {arg(I, Head, X), var(X)},
   {member(J, ArgNums), arg(J, Head, X2), X==X2}, !,
   {arg(I, NewHead, Var)},
   conj_unr(Var=X, I, Nonvar, Unb),
   match_unr(ArgNums, Nonvar, Unb, Head, NewHead).
match_unr([I|ArgNums], Nonvar, Unb, Head, NewHead) -->>
   {arg(I, Head, X), var(X)}, !,
   {arg(I, NewHead, X)},
   match_unr(ArgNums, Nonvar, Unb, Head, NewHead).
match_unr([I|ArgNums], Nonvar, Unb, Head, NewHead) -->>
   {arg(I, Head, X), nonvar(X)}, !,
   {arg(I, NewHead, Var)},
   conj_unr(Var=X, I, Nonvar, Unb),
   match_unr(ArgNums, Nonvar, Unb, Head, NewHead).

conj_unr(Goal, I, Nonvar, _) -->> {member(I,Nonvar)}, !, [Goal]:nonvar_goal.
conj_unr(Goal, I, _,    Unb) -->> {member(I,Unb)}, !, [Goal]:unbound_goal.
conj_unr(Goal, I, _,      _) -->> [Goal]:other_goal.

% *** Unraveling a body goal:
% Unifications are put in the right order.
% Non-unification goals are unraveled.
unr_goal(Goal) --> unr_goal(Goal, NGoal), co(NGoal).

unr_goal(A=B, fail) --> {\+(A=B)}, !.
unr_goal(A=B, A=B)  --> {var(A)}, {\+(\+(A=B))}, !.
unr_goal(A=B, B=A)  --> {var(B)}, {\+(\+(A=B))}, !.
unr_goal(A=B, A=B)  --> {nonvar(A), nonvar(B)}, !.
unr_goal(Goal, NGoal) --> {\+unify_p(Goal)}, !,
   unr_str(Goal, NGoal).

% Unravel term into single-word item:
unr_sng(Sng, Sng) --> {single_word(Sng)}, !.
unr_sng(Mul, Var) --> {\+single_word(Mul)}, !, co(Var=Mul).

% True iff X can be stored in a single word:
single_word(X) :- var(X), !.
single_word(X) :- atomic(X), !.
```

```
% Unravel term into structure:
% (Possible later optimization: use arg instead of univ.)
unr_str(Str, NStr) -->
   {Str=..[Name|Args]},
   unr_args(Args, NArgs),
   {NStr=..[Name|NArgs]}.

unr_args([], []) --> [].
unr_args([A|Args], [NA|NArgs]) --> unr_sng(A, NA), unr_args(Args, NArgs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Unravel the old heads with new mode information:
% This is an optimization pass done after flow analysis, to increase the
% opportunities for determinism extraction.  This pass is not needed for
% correctness.

% This code assumes that running unr_head again with the new mode information
% gives the same number of unification goals that it did before.
re_unr_strees(Ss, NSs) :-
   compile_option(compile), !,
   re_unr_strees_2(Ss, NSs).
re_unr_strees(Ss, Ss) :-
   \+compile_option(compile), !.

re_unr_strees_2([], []).
re_unr_strees_2([S|Ss], [NS|NSs]) :-
   re_unr_stree(S, NS),
   re_unr_strees_2(Ss, NSs).

re_unr_stree(stree(NaAr,   (Head:-Disj),Mode,OldHeads,DL,SD),
        stree(NaAr,(Head:-NewDisj),Mode,OldHeads,NDL,SD)) :-
   cons(OldHeads),
   non_trivial(Mode), !,
   re_unr_disj(OldHeads, Head, Disj, NewDisj, Mode),
   re_unr_strees(DL, NDL).
% There are no old heads to unravel, or the mode is trivial:
re_unr_stree(stree(NaAr,HeadDisj,Mode,OldHeads,DL,SD),
        stree(NaAr,HeadDisj,Mode,OldHeads,NDL,SD)) :- !,
   re_unr_strees(DL, NDL).
re_unr_stree(D, D) :- directive(D), !.

re_unr_disj([], _, fail, fail, _).
re_unr_disj([OH|OHs], Head, (Conj;Disj), (NewConj;NewDisj), Mode) :-
   % Copy the old head & the conj to retain variable structure:
   copy((OH:-Conj), (COH:-CConj)),
   % Unravel the old head with the new mode information:
   unr_head(COH, Mode, Head, RepConj, true),
   replace_start_conj(RepConj, CConj, NewConj),
   re_unr_disj(OHs, Head, Disj, NewDisj, Mode).

non_trivial((_:-Formula)) :- \+Formula=true.
```

%%

synonym.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Synonym Optimization:

% This module performs a strength reduction of instruction arguments,
% by remembering which addressing modes have the same value.

% Notes:
% 1. This module implements an abstract data type representing a partitioning
%    of all possible arguments of the move instruction.  The internal
%    representation is as a set of equivalence classes, denoted by a list of
%    lists. An equivalence class contains all arguments with identical contents.
%    Classes of size 1 are not stored.  This implementation is sufficient
%    because there will almost always be a small number of small equivalence
%    classes.
% 2. For best results, this module should be done AFTER superfluous label
%    elimination.  The reason is that the synonym set is emptied when a
%    label is encountered.
% 3. Small improvement possible with call: only remove the registers from the
%    synonym set, and keep the environment variables.
% 4. Improvement with all conditional branches: continue down the fall-
%    through case.  Don't have to change the SynSet in any way, but doing it
%    would be an extra optimization.  For example, jump(ne,A,B,L)
%    is an equal_maker.

% Algorithm in procedural form:
% Perform the following actions while traversing the instructions:
% move(A,B) or equal(A,B,_) or unify(A,B,_,_,_) =>
%  if A and B in same equivalence class then remove the instruction.
%  else
%      if A's equivalence class has a cheaper entry than A
%      then replace A by the cheaper entry.
%      else
%      if move(_,_) then remove B from its equivalence class.
%      add an equivalence class [A,B].
% a distant branch, a call with non-survive argument or a label =>
%  empty the partition.
% a switch_unify =>
%  call recursively on its arguments.
%  then empty the partition.
% destructive instruction =>
%  remove destroyed elements from all equivalence classes.
% other instruction =>
%  replace arguments by their cheapest equivalents.
%  keep the partition.

% Worst case speed is O((code_size)^2) because the abstract data type
% operations can take O(code_size).  Usually, however, these operations
% will be constant time, since the number and size of equivalence classes
% is limited.
```

```
synonym(Code, NCode) :- synonym(Code, NCode, []), !.

synonym([], [], _).
synonym([I|Code], NCode, SynSet) :- synonym(I, Code, NCode, SynSet).

synonym(Oper, Code, NCode, SynSet) :-
   equal_maker(Oper, A, B), !,
   make_set(A, B, S),
   equal_synonym(Oper, A, B, S, SynSet, OutSyn, NCode, L),
   synonym(Code, L, OutSyn).
synonym(label(Lbl), Code, [label(Lbl)|NCode], SynSet) :- !,
   synonym(Code, NCode, []).
% Add later:
% synonym(call(N/A), Code, [call(N/A)|NCode], SynSet) :-
   % functor(G, N, A),
   % survive(G), !,
   % Remove args r(L)...r(L+A-1).
   % remove_non_permanents(SynSet, NewSynSet),
   % synonym(Code, NCode, NewSynSet).
% This is only right if L is a unique label:
% synonym(Branch, [label(L)|Code], NCode, SynSet) :-
%  branch(Branch, Lbls),
%  member(L, Lbls), !,
%  synonym_step(Branch, SynSet, NewSynSet, NCode, [label(L)|NLink]),
%  synonym(Code, NLink, NewSynSet).
synonym(Oper, Code, NCode, SynSet) :-
   synonym_step(Oper, SynSet, NewSynSet, NCode, NLink),
   synonym(Code, NLink, NewSynSet).

synonym_step(Oper, SynSet, NewSynSet, Code, Link) :-
   synonym_step1(Oper, SynSet, Code, Link),
   synonym_step2(Oper, SynSet, NewSynSet).

synonym_step1(Oper, SynSet, [DOper|L], L) :-
   map_instruction(Oper, List, DOper, DList), !,
   cheapest_list(List, SynSet, DList).
synonym_step1(Oper, _, [Oper|L], L).

synonym_step2(Oper, SynSet, NewSynSet) :-
   destr_instruction(Oper, Els), !,
   remove_all(SynSet, Els, NewSynSet).
synonym_step2(Oper, SynSet, SynSet).

% Handle the case that A and B are synonyms:
% Also update the synonym set if A & B are not known yet as synonyms.
equal_synonym(Oper, A, B, S, InSyn, OutSyn, [DOper|Link], Link) :-
   not_independent(Oper), !,
   map_equal_maker(Oper, List, DOper, DList),
   cheapest_list(List, InSyn, DList),
   remove_if_move(Oper, InSyn, B, OutSyn).
equal_synonym(Oper, A, B, S, InSyn, OutSyn, [DOper|Link], Link) :-
   \+is_syn(S, InSyn), !,
   map_equal_maker(Oper, List, DOper, DList),
   cheapest_list(List, InSyn, DList),
   remove_if_move(Oper, InSyn, B, MidSyn),
   add_set(MidSyn, S, OutSyn).
equal_synonym(Oper, A, B, S, InSyn, InSyn, Link, Link).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Instructions that result in their arguments becoming equal:
equal_maker(       move(A,B), A, B).
equal_maker(    equal(A,B,_), A, B).
equal_maker(unify(A,B,_,_,_), A, B).

% The instruction’s arguments are not independent, i.e.
% doing the instruction does not make them equal:
not_independent(move(A,B)) :- ind(A), is_in(B, A).

% Map NON-destructive parts of the equal-makers:
% Note that the destination of a move can partake of the advantages of
% replacement by the cheapest element only if it is an indirection.
map_equal_maker(     move(A,[B]), [A,B],      move(S,[T]), [S,T]) :- !.
map_equal_maker(       move(A,X), [A],          move(S,X), [S]) :- \+X=[_], !.
map_equal_maker(    equal(A,B,L), [A,B],     equal(S,T,L), [S,T]) :- !.
map_equal_maker(unify(A,B,X,Y,Z), [A,B], unify(S,T,X,Y,Z), [S,T]) :- !.

% Map the NON-destructive arguments of an instruction to new arguments:
map_instruction(       pragma(tag(R,T)), [R],        pragma(tag(S,T)), [S]).
map_instruction(     pragma(align(R,N)), [R],      pragma(align(S,N)), [S]).
map_instruction(                 cut(R), [R],                  cut(S), [S]).
map_instruction(               trail(R), [R],                trail(S), [S]).
map_instruction(        trail_if_var(R), [R],         trail_if_var(S), [S]).
map_instruction(    unify_atomic(A,I,L), [A],     unify_atomic(S,I,L), [S]).
map_instruction(   test(Cond,Tag,R,Lbl), [R],    test(Cond,Tag,S,Lbl), [S]).
map_instruction(     jump(Cond,R1,R2,L), [R1,R2],  jump(Cond,S1,S2,L), [S1,S2]).
map_instruction(hash(Type,R,Length,Lbl), [R], hash(Type,S,Length,Lbl), [S]).
map_instruction( switch(Tag,R,V1,S1,F1), [R],  switch(Tag,S,V1,S1,F1), [S]).
map_instruction(             add(A,B,C), [A,B],           add(S,T,C), [S,T]).
map_instruction(             sub(A,B,C), [A,B],           sub(S,T,C), [S,T]).
map_instruction(             mul(A,B,C), [A,B],           mul(S,T,C), [S,T]).
map_instruction(             div(A,B,C), [A,B],           div(S,T,C), [S,T]).
map_instruction(             and(A,B,C), [A,B],           and(S,T,C), [S,T]).
map_instruction(              or(A,B,C), [A,B],            or(S,T,C), [S,T]).
map_instruction(             xor(A,B,C), [A,B],           xor(S,T,C), [S,T]).
map_instruction(               not(A,C), [A],               not(S,C), [S]).
map_instruction(             sll(A,B,C), [A,B],           sll(S,T,C), [S,T]).
map_instruction(             sra(A,B,C), [A,B],           sra(S,T,C), [S,T]).
map_instruction(            push(A,B,N), [A,B],          push(S,T,N), [S,T]).
% User instructions: (for macro definitions)
map_instruction(               ord(A,C), [A],               ord(S,C), [S]).
map_instruction(             val(T,A,C), [A],             val(T,S,C), [S]).
map_instruction(     jump_nt(C,R1,R2,L), [R1,R2],  jump_nt(C,S1,S2,L), [S1,S2]).
map_instruction(          add_nt(A,B,C), [A,B],        add_nt(S,T,C), [S,T]).
map_instruction(          sub_nt(A,B,C), [A,B],        sub_nt(S,T,C), [S,T]).
map_instruction(          and_nt(A,B,C), [A,B],        and_nt(S,T,C), [S,T]).
map_instruction(           or_nt(A,B,C), [A,B],         or_nt(S,T,C), [S,T]).
map_instruction(          xor_nt(A,B,C), [A,B],         xor_nt(S,T,C), [S,T]).
map_instruction(            not_nt(A,C), [A],             not_nt(S,C), [S]).
map_instruction(          sll_nt(A,B,C), [A,B],         sll_nt(S,T,C), [S,T]).
map_instruction(          sra_nt(A,B,C), [A,B],         sra_nt(S,T,C), [S,T]).
map_instruction(           trail_bda(R), [R],            trail_bda(S), [S]).

% Give list of arguments destroyed by an instruction.
% If there are none, then fail.
% Simple_call destroys exactly those arguments which are declared as
% uninit(reg,X).
destr_instruction(simple_call(N/A), Rs) :-
   functor(Head, N, A),
   require(Head, Req),
   uninit_set_type(reg, Req, Set),
   head_regs(Head, Set, Rs),
   cons(Rs).
destr_instruction(         cut(C), [r(b)]) :- \+C=r(b).
destr_instruction(       deref(C), [C]).
destr_instruction(     deref(_,C), [C]).
destr_instruction(     add(_,_,C), [C]).
destr_instruction(    adda(_,_,C), [C]).
destr_instruction(     sub(_,_,C), [C]).
destr_instruction(     mul(_,_,C), [C]).
destr_instruction(     div(_,_,C), [C]).
destr_instruction(     and(_,_,C), [C]).
destr_instruction(      or(_,_,C), [C]).
destr_instruction(     xor(_,_,C), [C]).
destr_instruction(       not(_,C), [C]).
destr_instruction(     sll(_,_,C), [C]).
destr_instruction(     sra(_,_,C), [C]).
destr_instruction(    push(_,R,N), [R]) :- N=\=0. % Increments R.
destr_instruction(         pad(N), [r(h)]) :- N=\=0. % Increments r(h).
destr_instruction(choice(1/N,_,_), [r(b)]).
destr_instruction(choice(I/N,S,_), Rs) :-
   1<I, I<N, regset_to_regs(S, Rs), cons(Rs).
destr_instruction(choice(N/N,S,_), [r(b)|Rs]) :- regset_to_regs(S, Rs).
destr_instruction(        call(_), [r(b),r(h),r(_)]). % Changes r(b),r(h),temps.
destr_instruction(    allocate(_), [r(e),p(_)]). % Changes r(e) and all perms.
destr_instruction(  deallocate(_), [r(e),p(_)]). % Changes r(e) and all perms.
% User instructions: (for macro definitions)
destr_instruction(       ord(_,C), [C]).
destr_instruction(     val(_,_,C), [C]).
destr_instruction(  add_nt(_,_,C), [C]).
destr_instruction(  sub_nt(_,_,C), [C]).
destr_instruction(  and_nt(_,_,C), [C]).
destr_instruction(   or_nt(_,_,C), [C]).
destr_instruction(  xor_nt(_,_,C), [C]).
destr_instruction(    not_nt(_,C), [C]).
destr_instruction(  sll_nt(_,_,C), [C]).
destr_instruction(  sra_nt(_,_,C), [C]).

regset_to_regs([], []).
regset_to_regs([N|Set], [r(N)|Rs]) :- integer(N), !, regset_to_regs(Set, Rs).
regset_to_regs([no|Set], Rs) :- regset_to_regs(Set, Rs).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Synonym list abstract data type:
% Used to implement the synonym optimizer.

% This source code is written to take full advantage of red cuts.
% The compiler should still be able to generate efficient object
% code for it.

% Data structure: SynSet, a list of lists which represents a set of
% sets of registers which are synonyms. The sets can be considered
% to be equivalence classes.

% Operations:
% make_set, add_set, is_syn, intersect_syn, remove_all, less_p, cheapest_list

% *** Make a pair of elements into a set:
make_set(A, B, S) :- A@<B,  !, S=[A,B].
make_set(A, B, S) :- A@>=B, !, S=[B,A].

% *** Add an equivalence class to the synonym set:
add_set(SynSet, [], SynSet) :- !.
add_set(SynSet, [_], SynSet) :- !.
add_set([S1|SynSet], S, [NS|SynSet]) :-
   not_disjoint(S, S1), !,
   % intersect(S, S1, S2), \+(S2=[]), !,
   union(S, S1, NS).
add_set([S1|SynSet], S, [S1|NewSyn]) :-
   add_set(SynSet, S, NewSyn).
add_set([], S, [S]).

% *** Intersect two synonym sets:
intersect_syn([S|Syn1], Syn2, [ST|Result]) :-
   member(T, Syn2),
   intersect(S, T, ST),
   \+(ST=[]), !,
   diff(Syn2, [T], NSyn),
   intersect_syn(Syn1, Syn2, Result).
intersect_syn([_|Syn1], Syn2, Result) :-
   intersect_syn(Syn1, Syn2, Result).
intersect_syn([], _, []).

% *** Test if a set is composed of synonyms:
is_syn(S, [S1|SynSet]) :- subset(S, S1), !.
is_syn(S,  [_|SynSet]) :- is_syn(S, SynSet).

% Remove an element from a synonym set if the instruction is a move:
% This is needed because among equal, unify, and move, only the move is
% destructive.  The others do not alter existing synonyms.
remove_if_move(move(_,_), S, El, NS) :- !, remove_all(S, [El], NS).
remove_if_move(Oper, S, El, S) :- \+Oper=move(_,_).

% *** Remove all synonyms that use an element in the set Els
% from a synonym set (i.e. [h+2] uses h):
remove_all([], Els, []).
remove_all([S|SynSet], Els, NewSyn) :-
   remove_one(S, Els, NS),
   (NS=[_]
   -> NewSyn=NSyn
   ;  NewSyn=[NS|NSyn]
   ),
   remove_all(SynSet, Els, NSyn).

% Remove all synonyms that use an element in the set Els
% from a single set:
remove_one([], Els, []).
remove_one([X|S], Els,     NS) :- uses_el(X, Els), !, remove_one(S, Els, NS).
remove_one([X|S], Els, [X|NS]) :- remove_one(S, Els, NS).

% Succeeds for an addressing mode that uses an element of Els:
% Assumes p(V) (with var(V)) represents ALL permanent variables,
% and similarly r(V) (with var(V)) represents ALL temporary variables.
% Assumes [Q] (with Q anything) represents ALL indirections (because of
% possible aliasing, when a memory location is changed then all memory
% locations must be assumed changed also).
uses_el([X],       Els) :- uses_el(X, Els).
uses_el((X+N),     Els) :- uses_el(X, Els).
uses_el(Tag^X,     Els) :- pointer_tag(Tag), uses_el(X, Els).
uses_el(p(I),      Els) :- member(p(X), Els), var(X), !.
uses_el(r(_),      Els) :- member(r(X), Els), var(X), !.
uses_el([_],       Els) :- !, member([_], Els).
uses_el(El,        Els) :- memberv(El, Els).

% *** Order relation for elements:
less_p(E1, E2) :-
   cost_syn(E1, N1),
   cost_syn(E2, N2),
   N1@<N2.

% *** Replace all elements of a list by their cheapest equivalents:
cheapest_list([], _, []).
cheapest_list([El|Els], SynSet, [Ch|Chs]) :-
   cheapest(El, SynSet, Ch),
   cheapest_list(Els, SynSet, Chs).

% Replace an element by its cheapest equivalent:
cheapest(El, SynSet, Ch) :-
   map_ea(El, R, Ch, S),
   member(Syn, SynSet),
   member(R, Syn), !,
   minimum_el(Syn, S).
cheapest(El, SynSet, El).

map_ea(      R, R,       S, S).
map_ea(    [R], R,     [S], S).
map_ea(  [R+N], R,   [S+N], S).
map_ea(    T^R, R,     T^S, S) :- pointer_tag(T).
map_ea(Tˆ(R+D), R, Tˆ(S+D), S) :- pointer_tag(T).

minimum_el([E|Els], M) :- minimum_el(Els, E, M).

minimum_el([], M, M).
minimum_el([El|Els], M1, M) :- less_p(El, M1), !, minimum_el(Els, El, M).
minimum_el([El|Els], M1, M) :- minimum_el(Els, M1, M).

% Cost function of addressing modes.
% The cost is represented as a compound term.  Relative cost is given by
% position in the standard order of terms.
% This cost function is designed for the VLSI-BAM processor.
% Approximate VLSI-BAM cost = (2*cycles + delay slots).
cost_syn(    r(b), a(0)) :- !. % Allows create & remove of cut(b).
cost_syn(    r(h), b(0)) :- !.                  % Directly usable.
cost_syn(    r(I), b(I)) :- integer(I), !.     % Directly usable (0 cycles).
cost_syn(       A, c(0)) :- an_atom(A), !.     % Needs ldi (1 cycle).
cost_syn(     TˆX, c(0)) :- pointer_tag(T), !. % Needs lea (1 cycle).
cost_syn(    r(A), d(0)) :- atom(A), !.    % Needs ld (1 cycle + 1 slot)
cost_syn(    p(I), d(I)) :- integer(I), !. % Needs ld (1 cycle + 1 slot).
cost_syn([r(I)],   d(I)) :- integer(I), !. % Needs ld (1 cycle + 1 slot).
cost_syn([r(I)+N], d(I)) :- integer(I), !. % Needs ld (1 cycle + 1 slot).
cost_syn([r(A)],   e(0)) :- atom(A), !. % Needs 2 ld’s (2 cycles + 2 slots).
cost_syn([r(A)+N], e(0)) :- atom(A), !. % Needs 2 ld’s (2 cycles + 2 slots).
cost_syn([p(I)],   e(I)) :- !. % Needs 2 ld’s (2 cycles + 2 slots).
cost_syn([p(I)+N], e(I)) :- !. % Needs 2 ld’s (2 cycles + 2 slots).
cost_syn( r(void), f(0)) :- !. % NOT changed here, but in peep_inst.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

tables.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Tables of information about predicates:

% The information that this table provides is given through predicates
% that make user-defined procedures and builtins indistinguishable.  Modes
% of user-defined predicates are used to obtain the information where this
% is possible.

% This table contains four flags for each predicate:
% 1. Whether the predicate lets registers survive across it;
% 2. Whether the predicate can use any registers as arguments
%    (as opposed to only specific ones);
% 3. Whether the predicate is a built-in, i.e. it is not defined by the
%    programmer.
% 4. Whether the predicate may be expanded into simple instructions
%    instead of being compiled as a call.

% Later improvements to this table:
% 1. Use sets of registers instead of yes-no flags.

% It is presently assumed that all non-builtin predicates (1) do not let
% registers survive and (2) have to be called with fixed argument registers.
% All builtins currently have deref required modes for all their arguments.

% The unify goal A=B is handled specially in the compiler; the entry given here
% only scratches the surface of what is done with it.  The special handling
% is necessary because A=B kills its temporary registers only when a call to the
% general unifier is necessary.  Look at clause_code.pl, unify.pl, and
% regalloc.pl to see how unify goals are handled.

survive(Goal,  y) :-   survive(Goal), !.
survive(Goal,  n) :- \+survive(Goal), !.

anyregs(Goal,  y) :-   anyregs(Goal), !.
anyregs(Goal,  n) :- \+anyregs(Goal), !.

builtin(Goal,  y) :-   builtin(Goal), !.
builtin(Goal,  n) :- \+builtin(Goal), !.

expanded(Goal, y) :-   expanded(Goal), !.
expanded(Goal, n) :- \+expanded(Goal), !.

fixregs(Goal) :- \+anyregs(Goal).
fixregs(Goal) :- \+builtin(Goal).

survive(Goal)  :- info(Goal, y, _, _, _), !.
survive(not(Goal))  :- info(Goal, y, _, _, _), !.

anyregs(Goal)  :- info(Goal, _, y, _, _), !.
```

```
builtin(Goal)  :- info(Goal, _, _, y, _), !.
expanded(Goal) :- info(Goal, _, _, _, y), !.
compmodes(Goal,F) :- info(Goal, _, _, _, _, _, _, F), !.

% This routine does not take the mode formula into account, so it
% gives a conservative answer.
% In conditions.pl is a generalization of this routine which does take
% the mode formula into account.
bindset(Goal, S) :- bindbag(Goal, B), sort(B, S).

bindbag(Goal, B) :- info(Goal, _, _, _, _, BArgs, _, _), !, varbag(BArgs, B).
bindbag(Goal, B) :- varbag(Goal, B), !. % No mode information known.

binds(Goal) :- bindbag(Goal, S), cons(S).

require(Goal,    S) :- info(Goal,_,_,_,_,_,Req,_), !, logical_simplify(Req, S).
require(Goal, true) :- \+info(Goal), !.

after(Goal,    S) :-   info(Goal,_,_,_,_,_,_,Aft), !, logical_simplify(Aft, S).
after(Goal,    S) :- \+info(Goal),   test(Goal), !, logical_simplify(Goal, S).
after(Goal, true) :- \+info(Goal), \+test(Goal), !.

% Only for user-defined predicates:
before(Goal, S) :- mode_option(Goal,_,Bef,_,_), !, logical_simplify(Bef, S).
before(Goal, true) :- \+mode_option(Goal,_,_,_,_), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The master table:

% Predicate  Survive?  AnyRegs?  Builtin?  Expanded?  Bindbag  Require  After
% Survive  = y iff temporaries live across it.
% AnyRegs  = y iff any registers can be arguments.
% Builtin  = y iff not user-defined, i.e. part of the standard system.
% Expanded = y iff it may be expanded instead of compiled as a call.
% Bindbag  = list of arguments that may be bound by the goal.
%            Must use varset, not sort, to convert this into a set.
% Require  = mode formula required for correct execution.
% After    = mode formula true after execution (usually most of Require remains
%            true as well; After only gives the additional truths.)
info(P) :- info(P, _, _, _, _).
info(P, S, A, B, E) :- info(P, S, A, B, E, _, _, _).

% User-defined predicates:
info(UserPred, Surv, n, n, n, Bindbag, Require, After) :-
   mode_option(UserPred,Require,Before,After,Surv), !,
   UserPred=..[_|Bindbag].

% Built-in predicates handled specially:
info(   '$cut_load'(X), y, y, y, y, [X], uninit_reg(X), true).
info(   '$cut_deep'(_), y, y, y, y, [], true, true).
info('$cut_shallow'(_), y, y, y, y, [], true, true).
info(    '$add'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$sub'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$mul'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$div'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$mod'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$and'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(     '$or'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$xor'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$sll'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(    '$sra'(A,B,C), y, y, y, y, [C], Re, Af) :- ar_m(A, B, C, Re, Af).
info(      '$sll'(A,B), y, n, y, y,  [], Re, Af) :- sh_m(A, B, Re, Af).
info(      '$sra'(A,B), y, n, y, y,  [], Re, Af) :- sh_m(A, B, Re, Af).
info(      '$not'(A,C), y, y, y, y, [C], Re, Af) :- ar_m(A, C, Re, Af).
info(            A@<B, n, n, y, n, [], Re,  A@<B) :- d_m([A,B], Re).
info(            A@>B, n, n, y, n, [], Re,  A@>B) :- d_m([A,B], Re).
info(           A@=<B, n, n, y, n, [], Re, A@=<B) :- d_m([A,B], Re).
info(           A@>=B, n, n, y, n, [], Re, A@>=B) :- d_m([A,B], Re).
info(            A==B, n, n, y, y, [], Re,  A==B) :- d_m([A,B], Re).
info(           A\==B, n, n, y, y, [], Re, A\==B) :- d_m([A,B], Re).
info(             A<B, y, y, y, y, [], Re,   (Af,A<B)) :- rel_m(A, B, Re, Af).
info(             A>B, y, y, y, y, [], Re,   (Af,A>B)) :- rel_m(A, B, Re, Af).
info(            A=<B, y, y, y, y, [], Re,  (Af,A=<B)) :- rel_m(A, B, Re, Af).
info(            A>=B, y, y, y, y, [], Re,  (Af,A>=B)) :- rel_m(A, B, Re, Af).
info(           A=\=B, y, y, y, y, [], Re, (Af,A=\=B)) :- rel_m(A, B, Re, Af).
info(           A=:=B, y, y, y, y, [], Re, (Af,A=:=B)) :- rel_m(A, B, Re, Af).
% The following two goals' survive flags must agree:
% This entry's Re entry used in choice_block:
info(             A=B, y, y, y, y, [A,B], Re, (A==B)) :- d_m([A,B], Re).
% This entry's Survive entry used in general unify for register allocation:
info('$unify'(A,B),y, y, y, n, [A,B], Re, (A==B)) :- d_m([A,B], Re).
info(        A\=B, n, n, y, y, [], Re,         A\=B) :- d_m([A,B], Re).
info('$test'(A,B), y, y, y, y, [], Re, ('$test'(A,B),Re)) :- d_m([A], Re).
info('$equal'(A,B), y, y, y, y, [], Re, (A==B,Af)) :- rd_m([A,B], Re, Af).
info(      var(A), y, y, y, y, [], Re,       (var(A),Af)) :- rd_m([A], Re, Af).
info(   nonvar(A), y, y, y, y, [], Re,    (nonvar(A),Re)) :- d_m([A], Re).
info(     atom(A), y, y, y, y, [], Re,      (atom(A),Af)) :- rd_m([A], Re, Af).
info(   atomic(A), y, y, y, y, [], Re,    (atomic(A),Af)) :- rd_m([A], Re, Af).
info(      nil(A), y, y, y, y, [], Re,       (nil(A),Af)) :- rd_m([A], Re, Af).
info(     cons(A), y, y, y, y, [], Re,      (cons(A),Re)) :- d_m([A], Re).
info(     list(A), y, y, y, y, [], Re,      (list(A),Re)) :- d_m([A], Re).
info(structure(A), y, y, y, y, [], Re, (structure(A),Re)) :- d_m([A], Re).
info( compound(A), y, y, y, y, [], Re,  (compound(A),Re)) :- d_m([A], Re).
info(   simple(A), y, y, y, y, [], Re,    (simple(A),Af)) :- rd_m([A], Re, Af).
info(   ground(A), n, n, y, y, [], Re,    (ground(A),Re)) :- d_m([A], Re).
info(   number(A), y, y, y, y, [], Re,    (number(A),Af)) :- rd_m([A], Re, Af).
info(  integer(A), y, y, y, y, [], Re,   (integer(A),Af)) :- rd_m([A], Re, Af).
info( negative(A), y, y, y, y, [], Re,  (negative(A),Af)) :- rd_m([A], Re, Af).
info(nonnegative(A), y, y, y, y,[],Re,(nonnegative(A),Af)):- rd_m([A], Re, Af).
info(functor(A,B,C), n, n, y, y, [A,B,C], Re,
    ('$name_arity'(A,B,C),atomic(B),integer(C))) :- d_m([A,B,C], Re).
info('$name_arity'(A,B,C), y, y, y, y, [], Re,
    ('$name_arity'(A,B,C),Re)) :- d_m([A], Re).
info(  arg(A,B,C), n, n, y, y, [A,B,C], Re,
    (integer(A),nonvar(B))) :- d_m([A,B,C], Re).
% info(    c(A,B,C), y, y, y, true). % <- Handled in preamble.
info(      A is B, n, n, y, y, [A], (deref(A),deref(B)), integer(A)).
info(       A=..B, n, n, y, n, [A,B], (deref(A),deref(B)), (nonvar(A),cons(B))).
info(           !, y, y, y, y, [], true, true).
info(        true, y, y, y, y, [], true, true).
info(        fail, y, y, y, y, [], true, fail).

% Standard builtins:
info(copy_term(A,B), n, n, y, n, [B], (deref(A),deref(B)), true).
info(     call(P), n, n, y, n, [P], deref(P), nonvar(P)).
info(length(L,N), n, n, y, n, [L,N], (deref(L),deref(N)), (list(L),integer(N))).
info(compare(C,X,Y), n, n, y, n, [C], (deref(C),deref(X),deref(Y)), atom(C)).
info(expand_term(T,X), n, n, y, n, [X], (deref(T),deref(X)),
    (nonvar(T),nonvar(X))).
info(   sort(L,S), n, n, y, n, [S], (deref(L),deref(S)), (list(L),list(S))).
info(keysort(L,S), n, n, y, n, [S], (deref(L),deref(S)), (list(L),list(S))).
info(   name(A,L), n, n, y, n, [A,L], (deref(A),deref(L)),
    (atomic(A),list(L))).
info(atom_chars(A,L), n, n, y, n, [A,L], (deref(A),deref(L)),
    (atom(A),list(L))).
info(number_chars(A,L), n, n, y, n, [A,L], (deref(A),deref(L)),
    (number(A),list(L))).
info(      repeat, n, n, y, n, [], true, true).
% expanded_exprs

% Recursive list predicates:
info(        is_list(A), n, n, y, n, [], deref(A), (var(A);list(A))).
info( is_proper_list(A), n, n, y, n, [], deref(A), list(A)).
info(is_partial_list(A), n, n, y, n, [], deref(A), (var(A);cons(A))).

% Program database builtins:
info(abolish(F,N), n, n, y, n, [], (deref(F),deref(N)), (atomic(F),integer(N))).
info(   assert(C), n, n, y, n, [], deref(C), nonvar(C)).
info(  asserta(C), n, n, y, n, [], deref(C), nonvar(C)).
info(  assertz(C), n, n, y, n, [], deref(C), nonvar(C)).
info(  retract(C), n, n, y, n, [C], deref(C), nonvar(C)).
info(bagof(X,P,B), n, n, y, n, [B], (deref(X),deref(P),deref(B)), true).
info(setof(X,P,B), n, n, y, n, [B], (deref(X),deref(P),deref(B)), true).
info( clause(P,Q), n, n, y, n, [P,Q], (deref(P),deref(Q)),
    (nonvar(P),nonvar(Q))).

% System & debugging support:
info(       abort, n, n, y, n, [], true, true).
info(       break, n, n, y, n, [], true, true).
info(        halt, n, n, y, n, [], true, true).
info(       trace, n, n, y, n, [], true, true).
info(     nodebug, n, n, y, n, [], true, true).
info(       debug, n, n, y, n, [], true, true).
info(    leash(M), n, n, y, n, [], deref(M), true).
info(   debugging, n, n, y, n, [], true, true).
info(nofileerrors, n, n, y, n, [], true, true).
info(  fileerrors, n, n, y, n, [], true, true).
info(    nospy(P), n, n, y, n, [], deref(P), nonvar(P)).
info(      spy(P), n, n, y, n, [], deref(P), nonvar(P)).
info(   op(P,T,A), n, n, y, n, [], (deref(P),deref(T),deref(A)), true).
info( prompt(A,B), n, n, y, n, [A], (deref(A),deref(B)), atom(B)).
info(  statistics, n, n, y, n, [], true, true).
info(          sh, n, n, y, n, [], true, true).
info(   system(S), n, n, y, n, [], deref(S), ground(S)).

info(     listing, n, n, y, n, [], true, true).
info(  listing(P), n, n, y, n, [], deref(P), ground(P)).
info(current_atom(A), n, n, y, n, [A], deref(A), atom(A)).
info(current_functor(A,T), n, n, y, n, [A,T], (deref(A),deref(T)),
    (atomic(A),nonvar(T))).
info(current_predicate(A,P), n, n, y, n, [A,P], (deref(A),deref(P)),
    (atomic(A),nonvar(P))).
info(current_op(A,B,C), n, n, y, n, [A,B,C], (deref(A),deref(B),deref(C)),
    (integer(A),atom(B),atom(C))).

% Program files:
info(       [F|R], n, n, y, y, [], (deref(F),deref(R)), (ground(F),ground(R))).
info(  consult(F), n, n, y, n, [], deref(F), atomic(F)).
info(    close(F), n, n, y, n, [], deref(F), atomic(F)).
info(   exists(F), n, n, y, n, [], deref(F), atomic(F)).
info(reconsult(F), n, n, y, n, [], deref(F), atomic(F)).
info( rename(F,G), n, n, y, n, [], (deref(F),deref(G)), (atomic(F),atomic(G))).
info(     save(F), n, n, y, n, [], deref(F), atomic(F)).
info(      see(A), n, n, y, n, [], deref(A), atomic(A)).
info(   seeing(A), n, n, y, n, [A], deref(A), atomic(A)).
info(        seen, n, n, y, n, [], true, true).
info(     tell(A), n, n, y, n, [], deref(A), atomic(A)).
info(  telling(A), n, n, y, n, [A], deref(A), atomic(A)).
info(        told, n, n, y, n, [], true, true).

% Character I/O:
info(      get(C), n, n, y, n, [C], deref(C), integer(C)).
info(     get0(C), n, n, y, n, [C], deref(C), integer(C)).
info(     skip(C), n, n, y, n,  [], deref(C), integer(C)).
info(  display(T), n, n, y, n, [], deref(T), true).
info(    print(T), n, n, y, n, [], deref(T), true).
info(      tab(N), n, n, y, n, [], deref(N), integer(N)).
info(          nl, n, n, y, n, [], true, true).
info(      put(C), n, n, y, n, [], deref(C), integer(C)).
info(     read(A), n, n, y, n, [A], deref(A), true).
info(    write(A), n, n, y, n, [], deref(A), true).
info(   writeq(A), n, n, y, n, [], deref(A), true).

% Internal database:
info(recorda(K,T,R), n, n, y, n, [T,R], (deref(K),deref(T),deref(R)),nonvar(R)).
info(recordz(K,T,R), n, n, y, n, [T,R], (deref(K),deref(T),deref(R)),nonvar(R)).
info(recorded(K,T,R),n, n, y, n,[K,T,R],(deref(K),deref(T),deref(R)),nonvar(R)).
info(erase(R), n, n, y, n, [], deref(R), nonvar(R)).
info(current_key(N,T), n, n, y, n, [N,T], (deref(N),deref(T)),
    (atomic(N),nonvar(T))).

% The following two predicates for the BAM:

% Modes for arithmetic predicates:
ar_m(A, B, C, Require, After) :-
```

```
   Require=(deref(A),deref(B),uninit_reg(C)),
   After=(integer(A),integer(B),integer(C),rderef(A),rderef(B),rderef(C)).
ar_m(A, C, Require, After) :-
   Require=(deref(A),uninit_reg(C)),
   After=(integer(A),integer(C),rderef(A),rderef(C)).

% Modes for relational predicates:
rel_m(A, B, Require, After) :-
   Require=(deref(A),deref(B)),
   After=(integer(A),integer(B),rderef(A),rderef(B)).

% Modes for vlsi_plm shifts (which have two arguments):
sh_m(A, B, Require, After) :-
   Require=(deref(A),deref(B)),
   After=(integer(A),integer(B),rderef(A),rderef(B)).

% Variables in the list are all dereferenced:
d_m([A], deref(A)).
d_m([A|L], (deref(A),DL)) :- d_m(L, DL).

% Variables in the list are all recursively dereferenced:
rd_m([A], deref(A), rderef(A)).
rd_m([A|L], (deref(A),DL), (rderef(A),RDL)) :- rd_m(L, DL, RDL).

/*
% The same two predicates for a processor that doesn't have
% type-checking built into its arithmetic:
% (But: Must TRAP if not integer...)
ar_m(A, B, C, Require, After) :-
   Require=(deref(A),deref(B),integer(A),integer(B),uninit_reg(C)),
   After=(integer(C),rderef(C)).
ar_m(A, C, Require, After) :-
   Require=(deref(A),integer(A),uninit_reg(C)),
   After=(integer(C),rderef(C)).

rel_m(A, B, Require, After) :-
   Require=(deref(A),integer(A),deref(B),integer(B)),
   After=true.
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Control flow:

control( (_,_)).
control( (_;_)).
control((_->_)).
control((_=>_)).
control( \+(_)).
control(not(_)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Branch operation utilities:
```

```
% Used in peephole optimization by dead code elimination and
% superfluous branch elimination.

% Succeeds for branches:
branch(Branch) :- branch(Branch, _, _).
branch(Branch, Lbls) :- branch(Branch, Lbls, []).

% Give the destinations of any branch:
branch(                    fail) --> [].
branch(                  return) --> [].
branch(               jump(Lbl)) --> [Lbl].
branch(         jump(_,_,_,Lbl)) --> [Lbl].
branch(      jump_nt(_,_,_,Lbl)) --> [Lbl].
branch(               call(Lbl)) --> [Lbl].
branch(         test(_,_,_,Lbl)) --> [Lbl].
branch(          equal(_,_,Lbl)) --> [Lbl].
branch(      unify(_,_,_,_,Lbl)) --> [Lbl].
branch(   unify_atomic(_,_,Lbl)) --> [Lbl].
branch(         choice(_,_,Lbl)) --> [Lbl].
branch(         hash(_,_,_,Lbl)) --> [Lbl].
branch(             pair(_,Lbl)) --> [Lbl].
branch(    switch(_,_,L1,L2,L3)) --> [L1,L2,L3].
branch(switch(unify,_,_,_,_,L3)) --> [L3].

% For basic block rearrangement in peephole:
% Duplicate a branch instruction with another destination.
% (later do it for switch also)
map_branch(            jump(L1), L1,             jump(L2), L2).
map_branch(      jump(X,Y,Z,L1), L1,       jump(X,Y,Z,L2), L2).
map_branch(   jump_nt(X,Y,Z,L1), L1,    jump_nt(X,Y,Z,L2), L2).
map_branch(            call(L1), L1,             call(L2), L2).
map_branch(      test(A,B,C,L1), L1,       test(A,B,C,L2), L2).
map_branch(       equal(A,B,L1), L1,        equal(A,B,L2), L2).
map_branch(   unify(A,B,C,D,L1), L1,    unify(A,B,C,D,L2), L2).
map_branch(unify_atomic(A,B,L1), L1, unify_atomic(A,B,L2), L2).
map_branch(      choice(A,B,L1), L1,       choice(A,B,L2), L2).
map_branch(      hash(A,B,C,L1), L1,       hash(A,B,C,L2), L2).
map_branch(          pair(A,L1), L1,           pair(A,L2), L2).
map_branch(  switch(A,B,L1,C,D), L1,   switch(A,B,L2,C,D), L2).
map_branch(  switch(A,B,C,L1,D), L1,   switch(A,B,C,L2,D), L2).
map_branch(  switch(A,B,C,D,L1), L1,   switch(A,B,C,D,L2), L2).

map_branches(            jump(L), [L],             jump(R), [R]).
map_branches(      jump(X,Y,Z,L), [L],       jump(X,Y,Z,R), [R]).
map_branches(   jump_nt(X,Y,Z,L), [L],    jump_nt(X,Y,Z,R), [R]).
map_branches(            call(L), [L],             call(R), [R]).
map_branches(      test(A,B,C,L), [L],       test(A,B,C,R), [R]).
map_branches(       equal(A,B,L), [L],        equal(A,B,R), [R]).
map_branches(   unify(A,B,C,D,L), [L],    unify(A,B,C,D,R), [R]).
map_branches(unify_atomic(A,B,L), [L], unify_atomic(A,B,R), [R]).
map_branches(      choice(A,B,L), [L],       choice(A,B,R), [R]).
map_branches(      hash(A,B,C,L), [L],       hash(A,B,C,R), [R]).
map_branches(          pair(A,L), [L],           pair(A,R), [R]).
map_branches(  switch(A,B,L,M,N), [L,M,N], switch(A,B,R,S,T), [R,S,T]).
```

```
% Succeeds for pure branches:
% (i.e. simple branches with no call or choice point complexity)
pure_branch(Branch) :- pure_branch(Branch, _, _).
pure_branch(Branch, Lbls) :- pure_branch(Branch, Lbls, []).

pure_branch(               jump(Lbl)) --> [Lbl].
pure_branch(         jump(_,_,_,Lbl)) --> [Lbl].
pure_branch(      jump_nt(_,_,_,Lbl)) --> [Lbl].
pure_branch(         test(_,_,_,Lbl)) --> [Lbl].
pure_branch(          equal(_,_,Lbl)) --> [Lbl].
pure_branch(      unify(_,_,_,_,Lbl)) --> [Lbl].
pure_branch(    switch(_,_,L1,L2,L3)) --> [L1,L2,L3].
pure_branch(switch(unify,_,_,_,_,L3)) --> [L3].
pure_branch(   unify_atomic(_,_,Lbl)) --> [Lbl].

% Branches that don't fall through to the next operation:
% (unless a label happens to be there)
distant_branch(fail).
distant_branch(return).
distant_branch(jump(_)).
distant_branch(switch(_,_,_,_,_)).

% An instruction that always succeeds and has a purely local effect:
% No choice point creation done.
local_instr(I) :- local_instr(I, _, _).

% Gives source operands (This list is currently NOT USED):
local_instr(    pragma(_)) --> [].
local_instr(     deref(A)) --> [A].
local_instr(   deref(A,B)) --> [A].
local_instr(     trail(A)) --> [A].
local_instr(    move(A,_)) --> [A].
local_instr(  push(A,B,_)) --> [A,B].
local_instr(  adda(A,B,_)) --> [A,B].
local_instr(       pad(_)) --> [r(h)].
local_instr(  allocate(_)) --> [].
local_instr(deallocate(_)) --> [].
local_instr(   add(A,B,_)) --> [A,B].
local_instr(   sub(A,B,_)) --> [A,B].
local_instr(   mul(A,B,_)) --> [A,B].
local_instr(   div(A,B,_)) --> [A,B].
local_instr(   and(A,B,_)) --> [A,B].
local_instr(    or(A,B,_)) --> [A,B].
local_instr(   xor(A,B,_)) --> [A,B].
local_instr(   sll(A,B,_)) --> [A,B].
local_instr(   sra(A,B,_)) --> [A,B].
local_instr(     not(A,_)) --> [A].
% User instructions: (for macro definitions)
local_instr(     ord(A,_)) --> [A].
local_instr(   val(_,A,_)) --> [A].
local_instr(add_nt(A,B,_)) --> [A,B].
local_instr(sub_nt(A,B,_)) --> [A,B].
local_instr(and_nt(A,B,_)) --> [A,B].
local_instr( or_nt(A,B,_)) --> [A,B].
local_instr(xor_nt(A,B,_)) --> [A,B].
local_instr(sll_nt(A,B,_)) --> [A,B].
local_instr(sra_nt(A,B,_)) --> [A,B].
local_instr(  not_nt(A,_)) --> [A].
local_instr( trail_bda(A)) --> [A].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tag database and utilities:

% The tag names and tests for the basic types are given here.
% Program transformation should be able to remove all of the overhead
% of using these predicates instead of using inline constants.

% Correspondence between tags, types, and tests:
tag_type_test(tint,     integer, X,     integer(X)). /* :- \+split_integer. */
tag_type_test(tpos, nonnegative, X, nonnegative(X)) :-   split_integer.
tag_type_test(tneg,    negative, X,    negative(X)) :-   split_integer.
tag_type_test(tlst,        cons, X,        cons(X)).
tag_type_test(tstr,   structure, X,   structure(X)).
tag_type_test(tatm,        atom, X,        atom(X)).
tag_type_test(tvar,         var, X,         var(X)).

% The tag corresponding to a term’s type:
term_tag(Term, tint) :- integer(Term),   \+split_integer.
term_tag(Term, tpos) :- nonnegative(Term), split_integer.
term_tag(Term, tneg) :- negative(Term),    split_integer.
term_tag(Term, tlst) :- cons(Term).
term_tag(Term, tstr) :- structure(Term).
term_tag(Term, tatm) :- atom(Term).

% Tags that are pointers:
pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).

% Utilities built with the above tag database:

tag(Tag) --> {tag(Tag)}.

tag(Tag) :- tag_type_test(Tag, _, _, _), !.

tag(Type, Tag) --> {tag_type_test(Tag, Type, _, _)}.
tag(Test, Tag) --> {tag_type_test(Tag, _, _, Test)}.

tag(Type, Tag) :- tag_type_test(Tag, Type, _, _).
tag(Test, Tag) :- tag_type_test(Tag, _, _, Test).

test_tag(Test, Tag) :- tag_type_test(Tag, _, _, Test).

test_tag(Test, X, Tag) :- tag_type_test(Tag, _, X, Test).

term_tag(Term, Tag) --> {term_tag(Term, Tag)}.
```

```
simple_type(Type) :- tag_type_test(Tag, Type, _, _), \+pointer_tag(Tag), !.
simple_type(atomic).

tag_type(Type) :- tag_type_test(_, Type, _, _).

type(Type) :- type_test(Type, _, _).

type_test(Type, X, Test) :- tag_type_test(_, Type, X, Test), !.
type_test(atomic, X, atomic(X)).
type_test(simple, X, simple(X)).
type_test(compound, X, compound(X)).

% Get the type of a term:
term_type(Term, Type) :- term_tag(Term, Tag), tag_type_test(Tag, Type, _, _).

% Get a test (with argument X) that checks for the term's type:
term_test(Term, X, Test) --> {term_test(Term, X, Test)}.
term_test(Term, X, Test) :- term_tag(Term, Tag), tag_type_test(Tag, _, X, Test).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Arithmetic comparisons:

arith_test(Test) :- arith_test(Test, _).
arith_test(Test, Cond) :- arith_test(Test, _, _, Cond).
arith_test(Test, X, Y) :- arith_test(Test, X, Y, _).

arith_test(A=:=B, A, B, eq).
arith_test(A=\=B, A, B, ne).
% Signed arithmetic:
arith_test(A<B,   A, B, lts).
arith_test(A=<B,  A, B, les).
arith_test(A>=B,  A, B, ges).
arith_test(A>B,   A, B, gts).

% To break the symmetry when finding testsets:
special_cond(eq).
special_cond(lts).
special_cond(gts).

cond(X) :- cond(X, _).

% Conditions and their opposites:
cond(lts, ges). /* Signed less than */
cond(les, gts). /* Signed less than or equal */
cond(gts, les). /* Signed greater than */
cond(ges, lts). /* Signed greater than or equal */
cond( eq,  ne).  /* Equal */
cond( ne,  eq).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

testset.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Architecture-dependent part of deterministic code generation:
% Determining the test-set a test belongs to.

% Succeed if Goal generates some testset:
testset(Goal) :-
   testset(Goal, _, _, _, _, _).

testset(Goal, Test, Name, Ident, Direc) :-
   testset(Goal, Test, Name, Ident, Direc, _).

% testset(Goal, Test, Name, Ident, Direc, OppDirec)
% with: Goal  = any goal.
%       Test  = the predicate that is true if this direction is taken.
%               The goal implies(Goal,Test) is always true, i.e. Test does
%      part of the work of Goal.
%          Test may be different from Goal; this allows test-sets to be
%          recognized that are only implicitly defined by a clause.
%       Name  = name for the particular kind test-set.
%       Ident = a structure containing all the variables in the test,
%          which serves as a further identification of the test-set.
%               Name and Ident together uniquely identify the test-set.
%  Direc = name of the direction the branch will take.  Code generation
%          uses this to put the conditional code in the correct sense.

% To calculate all test-sets in less than O((c*t)^2) time it is necessary
% for single decisions to affect large groups of tests.  This is not possible
% if the selection predicate can only do an equality test; it must be able
% to do an (ordered) relational test.  Therefore instead of a
% predicate with two parameters which succeeds/fails depending on whether they
% are equivalent it is better to map test-sets onto
% an ordered set.  A test-set is uniquely identified by its name and
% by the set of arguments occurring in a test of the test-set.

% Notes:
% 1. A test may be in zero to several test-sets.  It is up to the
%    higher level selection code to take advantage of this.
% 2. What about user-defined goals?  Give them all their own number?
%    Only take goals from a clause up to the first one that fails same/1?
% 3. Is it reasonable to generalize test sets to the non-mutual exclusive case?
%    Probably not, since the mutual exclusivity allows a straightforward code
%    generation.
% 4. Modes are used elsewhere to see which variables in the tests may be bound.
%    They are not needed here.
% 5. Negation is handled through the Direc output.  A special output is given
%    to distinguish positive and negative tests.  I’m not sure how to
%    distinguish not(..) from \+(..) here.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% *** BAM-specific goodness function ***

% Goodness function for testsets.
% The number of directions is weighted highest; for same number the goodness
% of the key is the tie-breaker.
% Two arguments: key(Name,Ident) and list of val(Goal,Test,ClNum,GoalNum,Dir,Cl)
% Returns: a goodness measure of the testset.
goodness_testset(key(Key,_), Values, Goodness) :-
   number_of_direcs(Values, N),
   goodness_key(Key, C),
   (  Key=hash(_), compile_option(hash_size(H)), N<H
   -> Goodness = 0
   ;  Goodness is 1000*N+C
   ), !,
   comment(['Goodness of ',Key,' is ',Goodness]).
   % w('% Goodness of '),w(Key),w(' is '),wn(Goodness).

% Ranking according to efficiency of implementation:
% This ranking is optimized for the BAM architecture, but it can be modified
% for other architectures (e.g. see the 'mips' option).
% 0. Equal(atomic,[]) is very fast in the MIPS architecture, using the bne and
%    beq instructions and because [] is in a register.
% 1. Switch(\=atom) (i.e. swt) first because it gives more information for
%    its cost (i.e. it is three-way; it tells when something is a variable too.
% 2. Switch(atom) is slightly worse because it penalizes the recursive branch
%    (the 'other' branch) of a procedure--that branch needs an extra type check.
% 3. switch(integer) is penalized slightly because it's a bit awkward in
%    the BAM due to its separate tpos and tneg tags.
% 3. Fast two way branches (btgeq, btgne).
% 4. Two-instruction two-way tests (cmp + branch).
% 5. Negative and nonnegative are fast in the BAM, but maybe not so fast in
%    other architectures.  Integer is slower than them for the BAM, but faster
%    for other architectures.
% 6. Atomic and compound need two tag checks, with speed similar to integer.
% 7. Equality comparison of a structure's name & arity needs a memory reference,
%    so it is slow.
% 8. Simple is pretty slow, it needs lots of tag checks.
% 9. Hashing is the slowest.  Hashing of atomic terms is slightly faster
%    because hashing of structures needs to fetch the main functor from memory.

% ? Should switch(integer) be worse than equal(atomic,Int)?  Try the
%   program (a(A):-A=:=23,d).
% ? Is this a total order, or only a pairwise partial order?

goodness_key(equal(atomic,[]),      132) :- mips.
goodness_key(switch(negative),      130).
goodness_key(switch(nonnegative),   130).
goodness_key(switch(atom),          130).
goodness_key(switch(integer),       129).
goodness_key(switch(X),             131).
goodness_key(var,                   120).
goodness_key(atom,                  120).
goodness_key(cons,                  120).
goodness_key(structure,            120).
goodness_key(negative,             120) :- split_integer.
goodness_key(nonnegative,          120) :- split_integer.
goodness_key(equal,                 85).
goodness_key(equal(atomic,_),       80).
goodness_key(comparison(_,_),       80).
goodness_key(integer,               79).
goodness_key(atomic,                79).
goodness_key(compound,              79).
goodness_key(negative,              78) :- \+split_integer.
goodness_key(nonnegative,           78) :- \+split_integer.
goodness_key(equal(structure,_),    60).
goodness_key(simple,                50).
goodness_key(hash(atomic),          41).
goodness_key(hash(structure),       40).

% The number of different directions in the list of values:
number_of_direcs(Values, N) :-
   get_direcs(Values, Direcs),
   sort(Direcs, Sort),
   length(Sort, N).

get_direcs([], []).
get_direcs([val(_,_,_,_,D,_)|Values], [D|Direcs]) :- get_direcs(Values, Direcs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** The testsets which may be used when the test is a relational predicate.
% This is necessary to maintain proper trapping behavior for arithmetic
% comparisons.

relational_testset(comparison(_,arith)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Expand a testset to have a choice for each possible direction.
% Currently only switch(_) is expanded since it really needs **three**
% directions, a condition which is not often satisfied in selection.pl.
% Call: full_testset(Name, Ident, OldTestSet, NewTestSet)

full_testset(switch(Kind), v(X), TS, NewTS) :- !,
   switch_testset_list(Kind, X, TL),
   update_testset(TS, TL, TS, NewTS).
full_testset(_, _, TS, TS).

update_testset([val(_,_,_,_,Direc,_)|TS], TL, OldTS, NewTS) :-
   delete(TL, val(_,_,_,_,Direc,_), NewTL),
   update_testset(TS, NewTL, OldTS, NewTS).
update_testset([], TL, OldTS, NewTS) :- append(OldTS, TL, NewTS).

switch_testset_list(Kind, X,
   [val(_,var(X),none,_,  var,ErrCl),
    val(_,  Test,none,_, Kind,ErrCl),
    val(_, OTest,none,_,other,ErrCl)]) :-
   ErrCl = ('$sel_error',true),
   type_test(Kind, X, Test),
   disj_test(var(X), Test, NT),
   not_test(NT, OTest).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** BAM-specific testsets ***

% *** Equality to a given atom:
testset(Goal, '$name_arity'(A,Na,0), equal(atomic,Na), v(A), true, false) :-
   test_type_name_arity(Goal, A, Type, Na, 0),
   simple_type(Type).
testset(Goal, not('$name_arity'(A,Na,0)), equal(atomic,Na), v(A), false, true):-
   logical_simplify(not(Goal), NotG),
   test_type_name_arity(NotG, A, Type, Na, 0),
   simple_type(Type).

% *** Equality to a given structure:
testset(Goal, '$name_arity'(A,Na,Ar), equal(structure,Na/Ar), v(A),true,false):-
   test_type_name_arity(Goal, A, structure, Na, Ar).
testset(Goal, not('$name_arity'(A,Na,Ar)), equal(structure,Na/Ar), v(A),false,true):-
   logical_simplify(not(Goal), NotG),
   test_type_name_arity(NotG, A, structure, Na, Ar).

% *** Hashing:
% Assumes the existence of a hashing instruction (similar to WAM)
testset(Goal, '$name_arity'(A,Na,Ar), hash(Type), v(A), Direc, not(Direc)) :-
   test_type_name_arity(Goal, A, T, Na, Ar),
   T\==cons,
   % Integers are hashed the same as atoms:
   (simple_type(T) -> Type=atomic; Type=structure),
   (Ar>0 -> Direc=(Na/Ar); Direc=Na).

% *** Two-way relational branches:
% Assumes the existence of conditional branches, i.e.
% A<B, A>=B, B>A, and B=<A are in a single test-set
% since a single branch can distinguish between them.
testset(Goal, Goal, comparison(Class,Kind), v(A,B), Direc, NotD) :-
   Kind=arith,
   encode_relop(Goal, A, J, B, Kind),
   A@=<B,
   map_relop(Goal, Class, Direc, NotD).
testset(Goal, Goal, comparison(Class,Kind), v(A,B), Direc, NotD) :-
   Kind=arith,
   encode_relop(Goal, B, J, A, Kind),
   A@=<B,
   flip(J, I),
   encode_relop(Flip, A, I, B, Kind),
   map_relop(Flip, Class, Direc, NotD).

% *** Two-way Type/NotType branches:
testset(Goal, Test, Type, v(A), True, False) :-
   many_way(Goal, Test, A, Type, True, False).
```

```
% *** Three-way branches (also used in unify):
testset(Goal, Test, switch(T1), v(A), Direc, not(Direc)) :-
   many_way(Goal, Test, A, T2, true, false),
   tag_type(T1), T1\==var,
   map_three(T1, T2, Direc).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Local utilities:

% Goal implies that A has structure Na/Ar:
test_type_name_arity(Goal, A, Type, Na, Ar) :-
   encode_name_arity(Goal, A, Na, Ar, true),
   name_arity_type(Na, Ar, Type).

name_arity_type(Na, Ar, atom) :-
   atom(Na), integer(Ar), Ar=:=0.
name_arity_type(Na, Ar, integer) :-
   integer(Na), integer(Ar), Ar=:=0.
name_arity_type(Na, Ar, negative) :- split_integer,
   negative(Na), integer(Ar), Ar=:=0.
name_arity_type(Na, Ar, nonnegative) :- split_integer,
   negative(Na), integer(Ar), Ar=:=0.
name_arity_type(Na, Ar, structure) :-
   nonvar(Na), integer(Ar), Ar>0, \+((Na='.', Ar=:=2)).
name_arity_type(Na, Ar, cons) :-
   Na=='.', integer(Ar), Ar=:=2.

% Map a relational test into a branch condition, a branch direction
% and the opposite direction.
map_relop(Test, Cond, true, false) :-
   arith_test(Test, Cond),
   special_cond(Cond).
map_relop(Test, Cond, false, true) :-
   opposite(Test, NotT),
   arith_test(NotT, Cond),
   special_cond(Cond).

% Find & return a test that a goal satisfies:
% (use mutex to make this more powerful and slower?)
many_way(Test, Test, X, Type, true, false) :- type_test(Type, X, Test).
many_way(Goal, Test, A, Type, true, false) :-
   test_type_name_arity(Goal, A, Type, Na, Ar),
   type_test(Type, A, Test).
many_way(Not,  NotT, A, Type, True, False) :-
   negation(Not, Goal),
   many_way(Goal, Test, A, Type, False, True),
   out_negation(Test, NotT).
many_way(Goal, nonvar(X), X, var, false, true) :-
   test_varbag(Goal, Vs),
   member(X, Vs),
   implies(Goal, nonvar(X)).
```

```
% Negation:
negation(\+(Goal), Goal) :- !.
negation(not(Goal), Goal) :- !.
% negation(nonvar(A), var(A)) :- !.

out_negation(not(Goal), Goal) :- !.
out_negation(Goal, not(Goal)) :- !.

% Map multiple choices onto the ones possible with a three-way branch:
map_three(Type,    var,   var).
map_three(Type,   Type,  Type).
% map_three(Type, InType, other) :- InType\==var, InType\==Type.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

transform_cut.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Transform away cut and if-then-else.

% Source transformation of a predicate containing cuts and/or if-then-elses
% into an internal representation based on specialized builtin predicates.
% These predicates are expanded into move instructions to and from B,
% thereby implementing the removal of choice points.

% This module handles input in standard form.  It handles if-then-else
% as well as cut by appropriately loading and restoring the B register.
% Semantics of cut inside an if-then-else:  the cut propagates all the way
% to the top level.  Note that this is different from C-Prolog.

% The three special builtins:
% $cut_load(X)      expands to move(b,X)
% $cut_shallow(X)   expands to move(X,b)
% $cut_deep(X)      expands to move(X,b)
% All cuts are replaced by $cut_shallow or $cut_deep.  The value must have
% been loaded into X by $cut_load.  This means that an extra argument is
% needed to pass X down.

% Notes:
% 1. This module increases the average arity of predicates.

cut_p('$cut_deep'(_)).
cut_p('$cut_shallow'(_)).

transform_cut_ptrees([], []).
transform_cut_ptrees([P|Ps], [TP|TPs]) :-
   transform_cut_ptree(P, TP),
   transform_cut_ptrees(Ps, TPs).

transform_cut_ptree(P, TP) :-
   P=ptree(NaAr,Cls,M,L), !,
   transform_cut_cls(Cls, CCls, DCL, DF, AF),
   CP=ptree(NaAr,CCls,M,TL),
   transform_cut_ptrees(L, TL),
   transform_2(AF, DCL, CP, TP).
transform_cut_ptree(D, D) :-
   directive(D), !.

transform_2(false, _, P, P) :- !.
transform_2(true, DCL, P, TP) :-
   P=ptree(Name/Arity,Cls,(H:-Formula),DL),
   gensym("$cut_", Name/Arity, NewName),
   NewArity is Arity+1,
   dup_head(H, NewName, NewArity, X, MH),
   TP=ptree(Name/Arity,[(H:-'$cut_load'(X),MH)],(H:-Formula),[MP]),
   dup_head(H, NewName, NewArity, DCL, NH),
   replace_heads(Cls, NCls, NewName, NewArity, DCL),
   MP=ptree(NewName/NewArity,NCls,(NH:-Formula),DL),
   add_mode_option(H, NH, true, true, yes).

% Replace all heads in a procedure by new heads with one extra argument,
% keeping the same body.
replace_heads([], [], _, _, _).
replace_heads([Cl|Cls], [(NH:-B)|NCls], NewName, NewArity, ExtraArg) :-
   split(Cl, H, B),
   dup_head(H, NewName, NewArity, ExtraArg, NH),
   replace_heads(Cls, NCls, NewName, NewArity, ExtraArg).

% Duplicate a head & add one argument to the end:
dup_head(Head, NewName, NewArity, LastArg, NewHead) :-
        functor(Head, _, Arity),
        functor(NewHead, NewName, NewArity),
        match_all(1, Arity, Head, NewHead),
        arg(NewArity, NewHead, LastArg).

% *** transform_cut_cls(Cls, CCls, DCL, DF, AF)
% Transform all occurrences of ’!’ to deep or shallow cuts.
% DCL is a label bound to the arguments of all deep cuts.
% DF (Deep Flag) = output true iff there is a deep cut in Disj.
% AF (Any Flag)  = output true iff there is ANY cut in Disj.
% Neither of these flags takes if-then-else’s into account, because
% they have no effect on what happens at the top level.
% This predicate is identical to the one below, except that it handles
% a list of clauses instead of a disjunction that may contain if-then-else.
transform_cut_cls([], [], _, false, false).
transform_cut_cls([A|Cls], [(H:-CB)|CCls], DCL, DF, AF) :-
   split(A, H, B),
   standard_conj(B, SB),
   transform_cut_conj(SB, CB, true, DCL, DF1, AF1, true, _),
   transform_cut_cls(Cls, CCls, DCL, DF2, AF2),
   or(DF1, DF2, DF),
   or(AF1, AF2, AF).

% *** transform_cut_disj(Disj, CDisj, DCLbl, DF, AF, TF)
transform_cut_disj(Disj, (’$cut_load’(IfLbl),CDisj), DCLbl, DF, AF, TF) :-
   contains_if(Disj), !,
   transform_cut_disj(Disj, CDisj, DCLbl, IfLbl, DF, AF, TF).
transform_cut_disj(Disj, CDisj, DCLbl, DF, AF, TF) :-
   transform_cut_disj(Disj, CDisj, DCLbl, _,     DF, AF, TF).

transform_cut_disj(fail, fail, _, _, false, false, _).
transform_cut_disj((A->B;Disj), (CA;CDisj), DCLbl, IfLbl, DF, AF, TF) :-
   transform_cut_conj(A, CA,   CA2, DCLbl, DF1, AF1, TF, MidTF),
   insert_cut(MidTF, IfLbl, CA2, CA3),
   transform_cut_conj(B, CA3, true, DCLbl, DF2, AF2, MidTF, _),
   transform_cut_disj(Disj, CDisj, DCLbl, IfLbl, DF3, AF3, TF),
   or(DF1, DF2, DF3, DF),
   or(AF1, AF2, AF3, AF).
transform_cut_disj((A;Disj), (CA;CDisj), DCLbl, IfLbl, DF, AF, TF) :-
   transform_cut_conj(A, CA, true, DCLbl, DF1, AF1, TF, _),
   transform_cut_disj(Disj, CDisj, DCLbl, IfLbl, DF2, AF2, TF),
   or(DF1, DF2, DF),
   or(AF1, AF2, AF).

% *** transform_cut_conj(Conj, CConj, CLink, DCL, DF, AF, InTF, OutTF)
% DF, AF are outputs, true if there exists a deep cut or any cut, false
% otherwise.  TF is an input, a running flag that remains true
% as long as only tests are encountered.  A disjunction makes TF false.
transform_cut_conj(true, L, L, _, false, false, TF, TF).
transform_cut_conj((G,Conj), (G,CConj), L, DCL, DF, AF, InTF, OutTF) :-
   test(G), !,
   transform_cut_conj(Conj, CConj, L, DCL, DF, AF, InTF, OutTF).
transform_cut_conj((G,Conj), (CG,CConj), L, DCL, DF, AF, InTF, OutTF) :-
   disj_p(G), !,
   transform_cut_disj(G, CG, DCL, DF1, AF1, false),
   transform_cut_conj(Conj, CConj, L, DCL, DF2, AF2, false, OutTF),
   or(DF1, DF2, DF),
   or(AF1, AF2, AF).
transform_cut_conj((!,Conj), CConj, L, DCL,true,true,InTF,OutTF) :-
   insert_cut(InTF, DCL, CConj, C2),
   transform_cut_conj(Conj, C2, L, DCL, _, _, InTF, OutTF).
transform_cut_conj((G,Conj), (G,CConj), L, DCL, DF, AF, _, OutTF) :-
   \+disj_p(G), \+test(G), \+G=!,
   transform_cut_conj(Conj, CConj, L, DCL, DF, AF, false, OutTF).

insert_cut( true, Lbl, ('$cut_shallow'(Lbl),Link), Link).
insert_cut(false, Lbl,    ('$cut_deep'(Lbl),Link), Link).

contains_if(((A->B);_)) :- !.
contains_if((_;D)) :- contains_if(D).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

unify.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Code generation for unification.
% by Peter Van Roy

% Unification with a term is unraveled into a tree of explicit code.
% For clarity and ease of development this code is written using the
% Extended DCG notation.

% This code can be sped up somewhat by getting the type of an argument X
% first and then doing a simple case selection on it, instead of redoing
% the ctest selections everywhere.

% This module reuses code for write mode unification of the last argument.
% This reduces nested code size from O(N*N) to O(N) when there is deep
% recursion in the last argument of nested compound terms. This is especially
% useful for long immediate lists.

% Modes are taken into account to reduce the number of moves, trails, derefs,
% and multiway branches.  A mode is given as a logical formula F.

% Unitialized variables are taken into account.  They are given as modes
% uninit(reg,V,[]) and uninit(mem,V,[]) in F.

% Argument ISF contains the variables that have already
% been initialized upon entry of this code (except for the uninit variables).
% The others are initialized by this code when first encountered.  For example,
% in a head unification ISF contains the set of head arguments.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Extended DCG notation declarations:

% *** Description of accumulators:
% sf = set of variables that are currently initialized (i.e. NOT including um).
% vl = list of variables used in order of occurrence.  For register allocation.
% form = logical mode formula that currently holds.
% code = list of instructions generated.
% offset = offset into block of writemode code being generated.
% gnd = the set of all variables implied to be ground by the formula.
% *** Description of passed arguments:
% fail = the failure address.
% type = the type of the compound term being unified in read-mode unification.
% um = the set of uninitialized memory variables.
% top = the variable X during the writemode unification of X=Term.
% aun = allowed uninit, vars allowed to be created uninit (see clause_code.pl).

% *** Accumulators:
acc_info(    sf, V,  InSF,  OutSF, includev(V,InSF,OutSF)).
acc_info(   gnd, X, InGnd, OutGnd, includev(X,InGnd,OutGnd)).
acc_info(   drf, D, InDrf, OutDrf, includev(D,InDrf,OutDrf)).
acc_info(    vl, V,   Out,     In, Out=[V|In]).
acc_info(  form, F,   InF,   OutF, update_formula(F,InF,OutF)).
acc_info(  code, I,   Out,     In, Out=[I|In]).
acc_info(offset, S, InOff, OutOff, (OutOff is InOff+S)).

acc_info(newvar, V,   Out,     In, Out=[V|In]).
acc_info(  conj, C,   Out,     In, Out=(C,In)).

% *** Passed arguments:
pass_info(fail).
pass_info(type).
pass_info(aun, []).
pass_info(um, []).
pass_info(top, none).

% *** Predicates:
pred_info(       unify, 1,   [           aun,sf,vl,form,fail,code]).
pred_info(     unify_2, 2,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(     unify_3, 2,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(unify_3_conj, 1,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(unify_nonvar, 2,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(unify_nonvar, 4,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(   unify_var, 2,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(    unify_rm, 5,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(    unify_rm, 8,   [drf,gnd,um,aun,sf,vl,     fail     ]).
pred_info(    unify_wm, 4,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(    unify_wm, 8,   [drf,gnd,um,       vl,     fail     ]).
pred_info(  unify_args, 5,   [drf,gnd,um,aun,sf,vl,form,fail,code,type]).
pred_info(  unify_args, 6,   [drf,gnd,um,aun,sf,vl,form,fail,code,type]).
pred_info(   unify_arg, 7,   [drf,gnd,um,aun,sf,vl,form,fail,code,type]).
pred_info(  init_unify, 4,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info( xinit_unify, 4,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info(uninit_unify, 4,   [drf,gnd,um,aun,sf,vl,form,     code]).
pred_info( unify_var_i, 2,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info( unify_var_i, 4,   [drf,gnd,um,aun,sf,vl,form,fail,code]).
pred_info( unify_var_u, 2,   [drf,gnd,um,    sf,vl,form,     code]).

% Writemode routines:
pred_info(    create_arg, 2, [           sf,vl,form,     code]).
pred_info(    new_var_nf, 1, [           sf,vl,          code]).
pred_info(    new_var_nf, 2, [           sf,vl,          code]).
pred_info(new_umemvar_nf, 1, [           sf,vl,          code]).
pred_info(       new_var, 1, [           sf,vl,form,     code]).
pred_info(       new_var, 2, [           sf,vl,form,     code]).
pred_info(      new_vars, 1, [           sf,vl,form,     code]).
pred_info(  new_var_list, 2, [           sf,vl,form,     code]).
pred_info(     writemode, 1, [           sf,vl,          code]).
pred_info(     writemode, 5, [    um,aun,sf,vl,form,     code]).
pred_info(   b_writemode, 2, [    um,aun,sf,vl,form,     code,top,offset]).
pred_info(   b_writeargs, 4, [    um,aun,sf,vl,form,     code,top,offset]).
pred_info(    fill_slots, 2, [    um,aun,sf,vl,form,     code,top]).
pred_info(initialize_var, 2, [    um,aun,sf,vl,form,     code,top]).
pred_info(  push_if_init, 2, [    um,aun,sf,vl,form,     code,top]).
pred_info( is_uninit_mem, 1, [    um,    sf,   form          ]).

% Small utilities:
pred_info(          move_arg, 5, [drf,gnd,sf,vl,form,code,type]).
pred_info(          move_arg, 3, [drf,gnd,sf,vl,form,code,type]).
pred_info(    gnd_drf_update, 2, [drf,gnd,      form]).
pred_info(        drf_update, 2, [drf,          form]).
pred_info(      pragma_align, 2, [code]).
pred_info(        pragma_tag, 2, [code]).
pred_info(   cond_pragma_tag, 2, [code]).
pred_info(   pragma_tag_form, 1, [form,code]).
pred_info(         align_pad, 1, [code]).
pred_info(       uninit_dest, 3, [um]).
pred_info(         cond_pref, 1, [vl]).
pred_info(       uninit_bind, 2, [um,vl,form,code]).
pred_info(       uninit_move, 2, [um,   form,code]).
pred_info(      term_formula, 2, [form]).
pred_info(fence_if_nosurvive, 0, [vl]).
pred_info(     unify_depth_2, 3, [conj,newvar]).
pred_info(     unify_depth_2, 5, [conj,newvar]).

% These routines are defined externally:
pred_info( remove_uninit, 1, [form]).
pred_info( remove_uninit, 2, [form]).
pred_info(          ct_u, 1, [form,gnd]).
pred_info(         ctest, 1, [form]).
pred_info(         rtest, 2, [   form,fail,code]).
pred_info(      rtest_in, 2, [   form,fail,code]).
pred_info(   rtest_deref, 2, [vl,form,     code]).
pred_info(rtest_in_deref, 2, [vl,form,     code]).
pred_info(          rt_d, 2, [vl,form,     code,drf]).
pred_info(       rt_in_d, 2, [vl,form,     code,drf]).

% Versions of ctest, rtest_deref, rtest_in_deref that use local information:
% This one is better able to see if an argument is nonvar:
ct_u(nonvar(X)) -->> Gnd/gnd, {inv(X, Gnd)}, !.
ct_u(Test)      -->> ctest(Test).

% These are better able to see if an argument is deref:
rt_d(X, X) -->> Drf/drf, {inv(X, Drf)}, !.
rt_d(X, T) -->> rtest_deref(X, T).

rt_in_d(X, X) -->> Drf/drf, {inv(X, Drf)}, !.
rt_in_d(X, T) -->> rtest_in_deref(X, T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Debugging hooks ***

u(T) :- u(X=T, [X]).

u(X=T, ISF) :- u(X=T, ISF, true).

u(X=T, ISF, F) :- u(X=T, ISF, _, _, F).
```

```
u(X=T, ISF, OSF, VList, F) :- u(X=T, ISF, OSF, fail, VList, F).

ul(List, ISF, OSF, F) :- ul(List, ISF, OSF, F).
ul(List, ISF, OSF, VList, F) :- ul(List, ISF, OSF, fail, VList, F).

% Print the unification of a variable with a term assuming
% all variables in ISF are initialized.
u(X=T, ISF, OSF, Fail, VList, F) :-
   ul((X=T,true), ISF, OSF, Fail, VList, F).

ul(List, ISF, OSF, Fail, VList, F) :-
   unify_list(List, ISF, OSF, Fail, VList, [], F, Code, []),
   write_code(Code).

% Generate code for a list of unifications:
unify_list(true, ISF, ISF, Fail, Vl, Vl, IF) --> !.
unify_list((X=T,List), ISF, OSF, Fail, Vs, Vt, IF) -->
   unify(X=T, ISF, MSF, Vs, Vt, IF, MF, Fail),
   unify_list(List, MSF, OSF, Fail, Vt, Vl, MF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Main Entry Points ***
% Given:
%  ISF set of variables that occurred before execution (In So Far).
%      This also contains the uninit variables.  However, the internal
%      SF contains only initialized variables, NO uninit variables.
%  IF  the mode formula true before execution (In Formula).
%  Faillabel to jump to on failure.
% Calculates:
%  UM  set of uninit mem variables in IF.
% Returns code for the unification X=Y, along with:
%  OSF set of variables after execution that have occurred before
%      execution (Out So Far),  including uninits in the unification.
%  VList   list of all variables’ usages, for register allocation.
%  OF  the updated IF, with uninit goals removed (Out Formula).
%      A variable is only uninit once, after that it becomes init.
unify(X=Y, ISF, OSF, IF) --> unify(X=Y, ISF, OSF, _, IF).
unify(X=Y, ISF, OSF, VList, IF) -->
   unify(X=Y, VList, ISF, OSF, IF, _).
unify(X=Y, VList, ISF, OSF, IF, OF) -->
   unify(X=Y, ISF, OSF, VList, [], IF, OF).
unify(X=Y, ISF, OSF, Vs, Vl, IF, OF) -->
   {varset(X=Y, Aun)},
   unify(X=Y, Aun, ISF, OSF, Vs, Vl, IF, OF, fail).

% Call from clause_code.pl:
% Aun = set of variables allowed to be created as uninit.
unify_goal(Unify, VList, Aun, ISF, OSF, IF, OF) -->
   unify(Unify, Aun, ISF, OSF, VList, [], IF, OF, fail).

% *** Interface between the unification compiler & rest of compiler ***
% This top level call interfaces between the form of SF needed in unification
% compilation (which contains only inits) and in the rest of the compiler
% (which contains all variables with a value, init and uninit).
% Also calculates Gnd & UM, and updates the mode formula.
unify(X=Y) -->>
   % Internal InSF contains only initialized variables:
   insert(ExtInSF, IntInSF):sf,
   IF/form,
   {uninit_set(IF, UnAll)},
   {diffv(ExtInSF, UnAll, IntInSF)},
       % Call to the internal unification compiler:
       {grounds_in_form(IF, Gnd)},
       {rderef_set(IF, Drf)},
       {intersectv(Drf, Gnd, StayDrf)},
       {uninit_set_type(mem, IF, UM)},
       unify_2(X, Y):[drf(StayDrf,_),gnd(Gnd,_),um(UM)],
   % External OutSF contains both init and uninit variables:
   insert(IntOutSF, ExtOutSF):sf,
   {varset(X=Y, Vars), unionv(IntOutSF, Vars, ExtOutSF)},
   % Update the mode formula:
   {intersectv(UnAll, IntOutSF, OutInit)},
   remove_uninit(OutInit),
   update_formula(X=Y, ExtInSF):form.
unify((U1,U2)) -->> unify(U1), unify(U2).

% *** Internal Routines ***

% This starts with two stub routines, followed by
% unify_var, uninit_unify and init_unify, which do the main work.
% Also includes unify_rm and unify_wm, which do the read and
% write mode compilation, and writemode, which creates the
% block of moves for write mode creation of compound terms.

% Take care of the order of terms X and Y and of unifying two nonvars:
unify_2(X, Y) -->> {   var(X),    var(Y)}, !, unify_var(X, Y).
unify_2(X, Y) -->> {   var(X), nonvar(Y)}, !, unify_3(X, Y).
unify_2(X, Y) -->> {nonvar(X),    var(Y)}, !, unify_3(Y, X).
unify_2(X, Y) -->> {nonvar(X), nonvar(Y)}, !, unify_nonvar(X, Y).

% Unifying two nonvariables together:
unify_nonvar(X, Y) -->> {functor(X, N, A), functor(Y, N, A)}, !,
   unify_nonvar(X, Y, 1, A).
unify_nonvar(X, Y) -->> [fail]:code, [fail]:form.

unify_nonvar(X, Y, I, A) -->> {I>A}, !.
unify_nonvar(X, Y, I, A) -->> {I=<A}, !,
   {arg(I, X, Xi)},
   {arg(I, Y, Yi)},
   unify_2(Xi, Yi),
   {I1 is I+1},
   unify_nonvar(X, Y, I1, A).

% Take care of distinction that X is initialized vs. uninitialized
% and do the depth limiting transformation for the unification
% of an initialized variable that is not known to be ground.
```

```
unify_3(X, Y) -->> SF/sf, {\+inv(X, SF)}, !,
   uninit_unify(X, Y, nonlast, _),
   [X]:sf.
unify_3(X, Y) -->> Gnd/gnd, {inv(X, Gnd)}, !,
   init_unify(X, Y, nonlast, _).
unify_3(X, Y) -->>
   {unify_depth(X, Y, NewY, _Vars, Conj)},
   init_unify(X, NewY, nonlast, _),
   unify_3_conj(Conj).

% The added unifications take advantage of a full unify_2
% (which is effectively unify_var or uninit_unify):
unify_3_conj(true) -->> [].
unify_3_conj((X=Y,Conj)) -->> unify_2(X, Y), unify_3_conj(Conj).

% *** Unifying two variables together:
unify_var(X, Y) -->> SoFar/sf, {inv(X, SoFar),   inv(Y, SoFar)}, !,
   unify_var_i(X, Y).
unify_var(X, Y) -->> SoFar/sf, {inv(X, SoFar), \+inv(Y, SoFar)}, !,
   uninit_dest(Y, YDest, Flag),
   cond_pref(Flag),
   [X,Y]:vl,
   map_formula(X, Y):form,
   cond_pragma_tag(Flag, Y),
   [move(X,YDest)]:code,
   [Y]:sf.
unify_var(X, Y) -->> SoFar/sf, {\+inv(X, SoFar),   inv(Y, SoFar)}, !,
   uninit_dest(X, XDest, Flag),
   cond_pref(Flag),
   [Y,X]:vl,
   map_formula(Y, X):form,
   cond_pragma_tag(Flag, X),
   [move(Y,XDest)]:code,
   [X]:sf.
unify_var(X, Y) -->> SoFar/sf, {\+inv(X, SoFar), \+inv(Y, SoFar)}, !,
   unify_var_u(X, Y).

% Conditional preference: if Flag is yes, then create a ’pref’ flag
% in the varlist.  This is a suggestion to regalloc for improving
% register allocation.  It does not affect correctness.
cond_pref(yes) -->> !, [pref]:vl.
cond_pref( no) -->> !, [].

% Conditional tag pragma: insert a variable tag pragma if it’s a store.
cond_pragma_tag(yes, _) -->> [].
cond_pragma_tag(no,  X) -->> pragma_tag(X, var).

% Location of an uninit X’s value depends on whether it is in
% memory or not:
uninit_dest(X,  X, yes) -->> UM/um, \+inv(X, UM), !.
uninit_dest(X, [X], no) -->> UM/um,   inv(X, UM), !.

% Create a new formula containing terms for both X and Y by mapping
% the existing formula.  Relies on the fact that Y remains dereferenced
% if X is dereferenced, even if Y is uninit(mem).  This condition is
% enforced by clause_code.
map_formula(X, Y, InF, OutF) :-
   map_terms([X], [Y], InF, InFY),
   union_formula(InF, InFY, OutF).

% Both variables are initialized:
% There are many special cases handled here which result in better code.
% This routine is careful to make sure that the oldest variable is always
% bound to the youngest.
unify_var_i(X, Y) -->> Form/form, {atomic_value(Form, X, A)}, !,
   unify_2(Y, A).
unify_var_i(X, Y) -->> Form/form, {atomic_value(Form, Y, A)}, !,
   unify_2(X, A).
unify_var_i(X, Y) -->> ctest(atomic(X)), ctest(atomic(Y)), !,
   rtest_in_deref(X, T),
   rtest_in_deref(Y, U),
   [T,U]:vl,
   Fail/fail,
   [equal(T,U,Fail)]:code.
unify_var_i(X, Y) -->>
   Form/form,
   {unify_flag(Form, X, Xf)},
   {unify_flag(Form, Y, Yf)},
   unify_var_i(Xf, Yf, X, Y).

unify_var_i(var, nonvar, X, Y) -->> !,
   rtest_in_deref(X, T),
   rtest_in(trail(T), T),
   [Y,T]:vl,
   pragma_tag_form(T),
   [move(Y,[T])]:code.
unify_var_i(nonvar, var, X, Y) -->> !,
   rtest_in_deref(Y, T),
   rtest_in(trail(T), T),
   [X,T]:vl,
   pragma_tag_form(T),
   [move(X,[T])]:code.
unify_var_i(Xf, Yf, X, Y) -->>
   rtest_in_deref(X, T),
   rtest_in_deref(Y, U),
   [pref,T,U]:vl,
   fence_if_nosurvive,
   Fail/fail,
        [unify(T,U,Xf,Yf,Fail)]:code, % *** Call to general unify ***
   remove_vars:form.    % Because binding has many effects.

% Insert ’fence’ if general unification kills all registers:
% This is needed for register allocation.  See regalloc.
fence_if_nosurvive -->> {  survive(’$unify’(_,_))}, !.
fence_if_nosurvive -->> {\+survive(’$unify’(_,_))}, !, [fence]:vl.

% Flag whether X is known to be variable, nonvariable or not:
% (This allows some optimization in the translator.)
```

```
unify_flag(Form, X, nonvar) :- implies(Form, nonvar(X)), !.
unify_flag(Form, X,    var) :- implies(Form, var(X)), !.
unify_flag(Form, X,    '?').

% Both variables are uninitialized:
unify_var_u(X, Y) -->> UM/um, {inv(X, UM), inv(Y, UM)}, !,
   [X,Y]:sf,
   [deref(X),deref(Y),var(X),var(Y)]:form,
   [X,pref,X,Y]:vl,
   pragma_tag(X, var),
   [move(X,[X])]:code,
   pragma_tag(Y, var),
   [move(X,[Y])]:code.
unify_var_u(X, Y) -->> UM/um, {inv(X, UM), \+inv(Y, UM)}, !,
   [X,Y]:sf,
   [deref(X),deref(Y),var(X),var(Y)]:form,
   [X,pref,X,Y]:vl,
   pragma_tag(X, var),
   [move(X,[X])]:code,
   [move(X,Y)]:code.
unify_var_u(X, Y) -->> UM/um, {\+inv(X, UM), inv(Y, UM)}, !,
   [X,Y]:sf,
   [deref(X),deref(Y),var(X),var(Y)]:form,
   [pref,Y,X,X]:vl,
   [move(Y,X)]:code,
   pragma_tag(X, var),
   [move(X,[X])]:code.
unify_var_u(X, Y) -->> UM/um, {\+inv(X, UM), \+inv(Y, UM)}, !,
   new_var_nf(X, Y).

% *** Uninit_unify assumes X has not yet been initialized:
uninit_unify(X, Term, Last, [Lbl|LLbls]) -->> {compound(Term)}, !,
   uninit_bind(X, Term),
   writemode(Last, X, Term, Lbl, LLbls).
uninit_unify(X, Term, Last, LLbls) -->> {atomic(Term)}, !,
   uninit_bind(X, Term).
uninit_unify(X, Var, Last, LLbls) -->> {var(Var)}, !,
   unify_var(X, Var).

% Bind X to a nonvariable and update the mode formula:
uninit_bind(X, Term) -->>
   [X]:vl,
   {make_word(Term, Word)},
   uninit_move(Word, X),
   term_formula(X, Term).

% Move to an uninit variable:
% Recognizes that the destination of an uninit_mem variable is in memory.
uninit_move(Word, X) -->> UM/um, {\+inv(X, UM)}, !,
   [deref(X)]:form,
   [move(Word,X)]:code.
uninit_move(Word, X) -->> UM/um,   {inv(X, UM)}, !,
   pragma_tag(X, var),
   [move(Word,[X])]:code.
```

```
% Update the formula for X assuming it is unified with Term:
term_formula(X, Term) -->> {nonvar(Term)}, !,
   {functor(Term, Na, Ar)},
   ['$name_arity'(X,Na,Ar)]:form.
term_formula(X, Term) -->> {var(Term)}, !.

% *** Init_unify assumes X has already been initialized:

% Hook to trim mode formula for speed:
init_unify(X, T, L, LL) -->>
   insert(InF, InInitF):form,
   SoFar/sf,
   {split_formula(_, SoFar, X=T, InF, InInitF, RestF)},
   xinit_unify(X, T, L, LL),
   insert(OutInitF, OutF):form,
   {combine_formula(OutInitF, RestF, OutF)}.

xinit_unify(X, Term, Last, [Lbl|LLbls]) -->> {atomic(Term)}, ct_u(nonvar(X)),!,
   rtest_deref(X, T),
   unify_rm(T, Term, Last, LLbls, no).
xinit_unify(X, Term, Last, [Lbl|LLbls]) -->>{compound(Term)}, ct_u(nonvar(X)),!,
   rtest_deref(X, T),
   {term_test(Term, T, Test)},
   rtest(Test, T),
   unify_rm(T, Term, Last, LLbls, no).
xinit_unify(X, Term, Last, [Lbl|LLbls]) -->> {nonvar(Term)}, ctest(var(X)), !,
   rtest_deref(X, T),
   unify_wm(T, Term, Last, LLbls).
xinit_unify(X, Term, Last, [Lbl|LLbls])-->>{compile_option(uni),atomic(Term)},!,
   rtest_deref(X, T),
   rtest_in(trail_if_var(T), T),
   {term_tag(Term, Tag)},
   {make_word(Term, _, Word)},
   Fail/fail,
   [T]:vl,
   [unify_atomic(T,Word,Fail)]:code.
% This clause merges the output formulas from unify_rm and unify_wm:
xinit_unify(X, Term, Last, [Lbl|LLbls]) -->> {nonvar(Term)}, !,
   rtest_deref(X, T),
   {term_tag(Term, Tag)},
   {term_test(Term, T, Test)},
   Fail/fail,
   [T]:vl,
   [switch(unify,Tag,T,Write,Read,Fail)]:code,
   insert(Form, OutF):form,
   {update_formula(var(T), Form, WForm)},
   SoFar/sf,
   unify_wm(T, Term, Last, LLbls, SoFar, WForm, OutFW, Write),
   {update_formula(Test, Form, RForm)},
   unify_rm(T, Term, Last, LLbls, yes, RForm, OutFR, Read),
   {intersect_formula(OutFW, OutFR, OutF)}.
xinit_unify(X, Term, Last, [Lbl|LLbls]) -->> {var(Term)}, !,
   unify_var(X, Term).
```

```
% wm hook: Ignore output of sf.  Return vl, code list and output of form.
% It is assumed that using aun=[] guarantees that wm and rm initialize
% the same variables.
unify_wm(X, T, Last, LLbls, SF, IF, OF, Code) -->>
   unify_wm(X,T, Last, LLbls):[aun([]),form(IF,OF),sf(SF,_),code(Code,[])].

% rm hook: Return sf, vl, code list, and output of form.
unify_rm(X, T, Last, LLbls, Opt, IF, OF, Code) -->>
   unify_rm(X, T, Last, LLbls, Opt):[form(IF,OF),code(Code,[])].

% *** Unify_rm assumes X is a dereferenced nonvariable at run-time and Term is
%     a nonvariable at compile-time.
%     Matching between X and Term’s tags is assumed to be done elsewhere.
unify_rm(X, Term, Last, LLbls, Opt) -->> {structure(Term)}, !,
   [X]:vl,
   {functor(Term, N, A), Term=..[N|Args]},
   rtest_in(’$name_arity’(X,N,A), X),
   unify_args(Args, 1, X, LLbls, Opt):type(structure).
unify_rm(X, Term, Last, LLbls, Opt) -->> {cons(Term)}, !,
   [X]:vl,
   {Term=[Hd|Tl], Args=[Hd,Tl]},
   unify_args(Args, 0, X, LLbls, Opt):type(cons).
unify_rm(X, Term, Last, LLbls, Opt) -->> {atom(Term)}, !,
   [X]:vl,
   rtest_in(’$name_arity’(X,Term,0), X).
unify_rm(X, Term, Last, LLbls, Opt) -->> {number(Term)}, !,
   [X]:vl,
   rtest_in(’$name_arity’(X,Term,0), X).

% *** Unify_wm assumes X is initialized, unbound, and dereferenced at run-time
%     and Term is a nonvariable at compile-time.
unify_wm(X, Term, Last, [Lbl|LLbls]) -->>
   {compound(Term)}, !,
   [X]:vl,
   rtest_in(trail(X), X),
   {term_tag(Term, Tag)},
   pragma_tag(X, var),
   [move(Tagˆr(h),[X])]:code,
   writemode(Last, X, Term, Lbl, LLbls).
unify_wm(X, Term, Last, LLbls) -->>
   {atom(Term)}, !,
   [X]:vl,
   {term_tag(Term, Tatm)},
   rtest_in(trail(X), X),
   pragma_tag(X, var),
   [move(TatmˆTerm,[X])]:code.
unify_wm(X, Term, Last, LLbls) -->>
   {number(Term)}, !,
   [X]:vl,
   rtest_in(trail(X), X),
   pragma_tag(X, var),
   [move(Term,[X])]:code.
```

```
% Unification of the arguments of a structure.
% Two optimizations are done:
% 1. Use single move instruction when an argument is a first-occurrence
%    variable. (Why doesn't preferred allocation make this unnecessary?)
% 2. Group pairs of moves together to facilitate creating double word loads.
%    InV gives arguments whose moves have already been created.
unify_args(Args, I, X, LLbls, Opt) -->> unify_args([], Args, I, X, LLbls, Opt).

unify_args( [],  [], I, X, LLbls, Opt) -->> !.
unify_args(InV, [A], I, X, LLbls, no) -->> !,
   unify_arg(InV, _, A, I, X, nonlast, LLbls).
unify_args(InV, [A], I, X, LLbls, yes) -->> !,
   unify_arg(InV, _, A, I, X, last, LLbls).
unify_args( [], [A,B|Args], I, X, LLbls, Opt) -->>
   {align(2), 0 is I mod 2}, !,
   move_arg(X, I, Y1),
   {I1 is I+1},
   move_arg(X, I1, Y2),
   unify_args([Y1,Y2], [A,B|Args], I, X, LLbls, Opt).
unify_args(InV, [A,B|Args], I, X, LLbls, Opt) -->> !,
   unify_arg(InV, OutV, A, I, X, nonlast, _),
   {I1 is I+1},
   unify_args(OutV, [B|Args], I1, X, LLbls, Opt).

% Unification of one structure argument:
% Optimize the case when A is a first-occurrence variable.
% (Note that sf may become unsorted because of this, requiring a sort.)
unify_arg(InV, OutV, A, I, X, Last, LLbls) -->> {var(A)},
   SoFar/sf, {\+inv(A,SoFar)},
   UM/um, {\+inv(A,UM)}, !,
   move_arg(InV, OutV, X, I, A),
   sort:sf.
unify_arg(InV, OutV, A, I, X, Last, LLbls) -->>
   move_arg(InV, OutV, X, I, Y),
   init_unify(Y, A, Last, LLbls).

% Get argument only if it hasn't been done earlier:
move_arg([Y|Vs], Vs, X, I, Y) -->> [].
move_arg([],     [], X, I, Y) -->> move_arg(X, I, Y).

% Get argument I of a structure:
move_arg(X, I, Y) -->>
   Type/type,
   pragma_tag(X, Type),
   pragma_align(X, I),
   [move([X+I],Y)]:code,
   [X,Y]:vl,
   [Y]:sf,
   gnd_drf_update(Y, X).

pragma_tag(X, Type) -->>
   {tag(Type, Tag)},
   [pragma(tag(X,Tag))]:code.
```

```
% Insert tag pragma from mode formula if the tag is known:
pragma_tag_form(X) -->> F/form, {get_tag(F, X, Tag)}, !,
   [pragma(tag(X,Tag))]:code.
pragma_tag_form(_) -->> [].

pragma_align(X, I) -->>
   {align(K), 0 is I mod K}, !,
   [pragma(align(X,K))]:code.
pragma_align(X, I) -->> [].

% Update gnd and drf sets and the mode formula:
gnd_drf_update(Y, X) -->> Gnd/gnd, {inv(X, Gnd)}, !, [Y]:gnd,
   combine_formula(ground(Y)):form,
   drf_update(Y, X).
gnd_drf_update(_, _) -->> [].

% Only add Y to rderef set if X is ground:
drf_update(Y, X) -->> Drf/drf, {inv(X, Drf)}, !, [Y]:drf,
   combine_formula(rderef(Y)):form.
drf_update(_, _) -->> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generate a minimal sequence of moves to the global heap to
% create the structure Term (this is write mode unification),
% given that all initialized variables are in ISF.

% *** Create the term and return an argument for it:
create_arg(Var, Var) -->> {var(Var)}, SoFar/sf, {inv(Var, SoFar)}, !.
create_arg(Var, Var2) -->> {var(Var)}, SoFar/sf, {\+inv(Var, SoFar)}, !,
   new_var_nf(Var, Var2).
create_arg(Term, Word) -->> {atomic(Term)}, !, {make_word(Term, Word)}.
create_arg(Term, X) -->> {compound(Term)}, !,
   term_formula(X, Term),
   combine_formula(deref(X)):form,
   [X]:vl,
   [X]:sf, % Added Jan. 11
   {make_word(Term, Word)},
   [move(Word,X)]:code,
   writemode(Term).

% Create an unbound variable:
% (Does not update formula)
new_var_nf(X) -->>
   [X]:sf,
   [X,X]:vl,
   {tag(var, Tvar)},
   [move(Tvar^r(h),X)]:code,
   [pragma(push(variable))]:code,
   [push(X,r(h),1)]:code,
   {align_calc(1, _, Pad)},
   align_pad(Pad).

% Create two identical unbound variables:
```

```
% (Does not update formula)
new_var_nf(X, Y) -->>
   {X\==Y}, !,
   [X,Y]:sf,
   [X,pref,X,Y,Y]:vl,
   {tag(var, Tvar)},
   [move(Tvar^r(h),X)]:code,
   [move(Tvar^r(h),Y)]:code,
   [pragma(push(variable))]:code,
   [push(Y,r(h),1)]:code,
   {align_calc(1, _, Pad)},
   align_pad(Pad).
new_var_nf(X, Y) -->>
   {X==Y}, !,
   new_var_nf(X).

% Create an unbound variable:
% (Also updates formula)
new_var(X) -->>
   combine_formula((deref(X),var(X))):form,
   new_var_nf(X).

% Create two identical unbound variables:
% (Also updates formula)
new_var(X, Y) -->>
   {X\==Y}, !,
   combine_formula((deref(X),var(X),deref(Y),var(Y))):form,
   new_var_nf(X, Y).
new_var(X, Y) -->>
   {X==Y}, !,
   new_var(X).

% Create an uninit mem variable:
% (Does not update formula)
new_umemvar_nf(X) -->>
   [X]:sf,
   [X,X]:vl,
   {tag(var, Tvar)},
   [move(Tvar^r(h),X)]:code,
   [adda(r(h),1,r(h))]:code,
   {align_calc(1, _, Pad)},
   align_pad(Pad).

% Create unbound variables for those variables in VarBag that do not
% yet exist.  Both initialized and uninitialized variables are created
% depending on what is in the formula.  It is assumed that SF contains
% all variables that have a value, including both inits and uninits.
new_vars(VarBag) -->>
   {sort(VarBag, Vars)},
   SF/sf, {diffv(Vars, SF, NewVars)},
   F/form,
   {uninit_set_type(mem, F, UMemVars)},
   {uninit_set_type(reg, F, URegVars)},
   {diffv(NewVars, URegVars, IVars)},
   new_var_list(IVars, UMemVars).

new_var_list([], _) -->> [].
new_var_list([V|Vars], UMemVars) -->> {inv(V, UMemVars)}, !,
   new_umemvar_nf(V),
   new_var_list(Vars, UMemVars).
new_var_list([V|Vars], UMemVars) -->> {\+inv(V, UMemVars)}, !,
   new_var(V),
   new_var_list(Vars, UMemVars).

% *** Create a term on the heap:
writemode(Term) -->> writemode(nonlast, none, Term, _, _):[form(true,_)].

% *** Most general routine, used as part of unify above:
% Last flag:
%    Last = last: writemode is a jump to Lbl.
%    Last = nonlast: writemode is a full block; it creates LLbls, a list of
%                    labels for nested last arguments to jump to.
% Init flag:
%    Init = init: initialize all new variables in the code.
%    Init = uninit: the new variables are uninit(mem).
writemode(last, X, Term, Lbl, _) -->> !,
   [jump(Lbl)]:code.
writemode(nonlast, X, Term, _, LLbls) -->> !,
   [pragma(push(term(Size)))]:code,
   b_writemode(Term, LLbls):[offset(0,Size),top(X)].

% Create the Block of moves:
b_writemode(Str, LLbls) -->> {structure(Str)}, !,
   tag(atom, Tatm),
   {functor(Str, F, A)},
   [pragma(push(structure(A)))]:code,
   [push(Tatm^(F/A),r(h),1)]:code,
   {Str=..[F|Args]},
   {S is A+1},
   {align_calc(S, Size, Pad)},
   Offset/offset,
   {T is Offset+1},
   [Size]:offset,
   fill_slots(Args, Offsets),
   align_pad(Pad),
   b_writeargs(Args, T, Offsets, LLbls).
b_writemode(Cons, LLbls) -->> {cons(Cons)}, !,
   {Cons=[Hd|Tl], Args=[Hd,Tl]},
   {align_calc(2, Size, Pad)},
   Offset/offset,
   {T is Offset},
   [Size]:offset,
   [pragma(push(cons))]:code,
   fill_slots(Args, Offsets),
   align_pad(Pad),
   b_writeargs(Args, T, Offsets, LLbls).
b_writemode(Atom, []) -->> {atomic(Atom)}, !.
b_writemode(Var,  []) -->> {var(Var)}, !.
```

```
b_writeargs([], _, [], []) -->> !.
b_writeargs([A], S, [IO], [Lbl|LLbls]) -->> !,
   Offset/offset,
   {IO is Offset-S},
   [label(Lbl)]:code,
   b_writemode(A, LLbls).
b_writeargs([A,B|Args], S, [IO|Offsets], LLbls) -->> !,
   Offset/offset,
   {IO is Offset-S},
   {S1 is S+1},
   b_writemode(A, _),
   b_writeargs([B|Args], S1, Offsets, LLbls).

fill_slots([], []) -->> !.
fill_slots([A|Args], [Off|Offsets]) -->> !,
   {make_word(A, Off, Word)},
   initialize_var(A, Word),
   fill_slots(Args, Offsets).

make_word(C, Off, (Tcom^(r(h)+Off))) :- compound(C), term_tag(C, Tcom), !.
make_word(A, Off, (Tatm^A))          :- atom(A), term_tag(A, Tatm), !.
make_word(N, Off, N)                 :- number(N), !.
make_word(V, Off, (V))               :- var(V), !.

make_word(C, (Tcom^r(h))) :- compound(C), term_tag(C, Tcom), !.
make_word(A, (Tatm^A))    :- atom(A), term_tag(A, Tatm), !.
make_word(N, N)           :- number(N), !.
make_word(V, (V))         :- var(V), !.

% Initialize first-time variables in write mode:
initialize_var(V, Word) -->> is_uninit_mem(V), !,
   remove_uninit(mem, [V]),
   combine_formula(var(V)):form,
   [V,V]:vl,
   [V]:sf,
   pragma_tag(V, var),
   [move(V,[V])]:code,
   [push(Word,r(h),1)]:code.
initialize_var(V, Word) -->> {var(V)}, SoFar/sf, {\+inv(V, SoFar)}, !,
   {tag(var, Tvar)},
   [V,V]:vl,
   [move(Tvar^r(h),V)]:code,
   push_if_init(V,Word).
initialize_var(V, Word) -->> {var(V)}, SoFar/sf, {  inv(V, SoFar)}, !,
   [V]:vl,
   [push(Word,r(h),1)]:code.
initialize_var(V, Word) -->> {nonvar(V)}, !,
   [push(Word,r(h),1)]:code.

is_uninit_mem(V) -->> {var(V)}, UM/um, SF/sf, {inv(V,UM), \+inv(V,SF)}, !.
is_uninit_mem(V) -->> {var(V)}, ctest(uninit(mem,V)), !.

% Create an uninit(mem) OR an init. var on the heap:
```

```
push_if_init(V, Word) -->> Us/aun, inv(V, Us), !,
   [adda(r(h),1,r(h))]:code,
   X/top,
   combine_formula(uninit(mem,V,[X])):form.
push_if_init(V, Word) -->> Us/aun, \+inv(V, Us), !,
   [push(Word,r(h),1)]:code,
   [V]:sf,
   combine_formula((var(V),deref(V))):form.

%*** Predicates used for compound term alignment:

% Calculate alignment value:
% Size is least integer >= S and multiple of alignment constant K:
% Pad is the difference between Size and S.
align_calc(S, Size, Pad) :-
   align(K),
   Size is (S+K-1)//K*K,
   Pad is Size-S.

% Generate pad instruction:
align_pad(0) -->> !.
align_pad(P) -->> {P=\=0}, !, [pad(P)]:code.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Depth limiting in unification: (assumes X is always a variable)
% Ensures that unifications are only nested up to the global depth limit.
% This is done because compilation time and code size for deeply nested
% unifications goes as the square of the size of the term.
% A subterm Y at the depth limit is replaced by a variable Z and
% the goals V=Y, V=Z after the original unification goal.
% Then V=Y is done in writemode and V=Z is done with general unification.
% Also returns the set of the variables Z.
unify_depth(X, Y, NewY, VSet, Flat) :-
   unify_depth_2(0, Y, NewY, Conj, true, VBag, []),
   sort(VBag, VSet),
   flat_conj(Conj, Flat).

unify_depth_2(_, Y, Z) -->> {simple(Y)},             !, {Z=Y}.
unify_depth_2(D, Y, Z) -->> {depth_limit(L), D>=L}, !, [V=Y]:conj, [V=Z]:conj,
   [Z]:newvar,
   {comment(['Found structure ',Y,' at depth ',D])}.
unify_depth_2(D, Y, Z) -->>
   {functor(Y, Na, Ar)},
   {functor(Z, Na, Ar)},
   {D1 is D+1},
   unify_depth_2(1, Ar, D1, Y, Z).

unify_depth_2(I, Ar, D, Y, Z) -->> {I>Ar}, !.
unify_depth_2(I, Ar, D, Y, Z) -->> {I=<Ar}, !,
   {arg(I, Y, Yi)},
   {arg(I, Z, Zi)},
   unify_depth_2(D, Yi, Zi),
   {I1 is I+1},
   unify_depth_2(I1, Ar, D, Y, Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

utility.pl

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California

% Miscellaneous utility predicates

:- dynamic(compile_cputime/3).
:- dynamic(gensym_integer/1).

% Numeric parameters:
max_int(32767).
min_int(-32768).

% Split a clause into its head and body:
split(Clause, Head, Body) :-
        (Clause=(Head:-Body)
         -> true
          ; Clause=Head, Body=true
        ).

directive((:-_)).

stree(stree(_,_,_,_,_,_)).

% Find the first argument number containing the given term:
% Fail otherwise.
% Arguments have same order as arg/3, but different instantiation pattern.
find_arg(N, Head, Arg) :-
        functor(Head, _, Arity),
        find_arg(1, Arity, N, Head, Arg).

find_arg(I, Arity, N, Head, A) :- I=<Arity, arg(I, Head, Arg), Arg==A,  !, N=I.
find_arg(I, Arity, N, Head, A) :- I=<Arity, arg(I, Head, Arg), Arg\==A, !,
        I1 is I+1,
        find_arg(I1, Arity, N, Head, A).

outfilename(InFile, Suffix, OutFile) :-
   name(InFile, InList),
   (append(InPre, ".",_InSuf, InList) ->
       append(InPre, ".", Suffix, OutList);
       append(InList, ".", Suffix, OutList)),
   name(OutFile, OutList).

newfilename(InFile, Suffix, OutFile) :-
   name(InFile, InList),
   append(InList, Suffix, OutList),
   name(OutFile, OutList).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Transitive graph closure utilities:
% Utilities based on Warshall’s algorithm, from O’Keefe, page 89.
% *** Find the set of all ground variables implied by a formula.
% This starts with explicit ground modes and follows unifications.
grounds_in_form(InF, AllGnd) :-
   ground_set(InF, BaseGnd),
   vars_in_unify(BaseGnd, InF, AllGnd).

% *** Find the set of all variables related to a Term.
% This follows unifications, i.e. if X is in Term, and X=Term2 occurs,
% then the vars in Term2 are also related.
vars_in_unify(Term, InF, Vars) :-
        make_graph((X=Term,InF), Graph, []),
        warshall(Graph, Closure),
        member_w(X-Vars, Closure).

% Warshall’s algorithm from O’Keefe, page 89:
warshall(Graph, Closure) :-
        warshall(Graph, Graph, Closure).

warshall([], Closure, Closure) :- !.
warshall([V-_|G], E, Closure) :-
        member_w(V-Y, E),
        warshall(E, V, Y, NewE),
        warshall(G, NewE, Closure).

warshall([], _, _, []) :- !.
warshall([X-Neibs|G], V, Y, [X-NewNeibs|NewG]) :-
        inv(V, Neibs), !,
        unionv(Neibs, Y, NewNeibs),
        warshall(G, V, Y, NewG).
warshall([X-Neibs|G], V, Y, [X-Neibs|NewG]) :-
        warshall(G, V, Y, NewG).

member_w(V-Y, [W-Y|_]) :- V==W, !.
member_w(V-Y, [_|E]) :- member_w(V-Y, E).

% Convert a conjunction of X=Term goals and others into a graph:
make_graph((A,B)) --> !, make_graph(A), make_graph(B).
make_graph(Goal) --> graph_node(Goal).

graph_node(X=Term) --> !, {varset(Term, Vars)}, [X-Vars].
graph_node(X==Term) --> !, {varset(Term, Vars)}, [X-Vars].
graph_node(Goal) --> [].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Set operations:
% Implementation inspired by R. O’Keefe, Practical Prolog.
% Sets must be represented as sorted lists without duplicates.
% Predicates with ’v’ suffix work with sets containing uninstantiated vars.

% *** Intersection
intersectv([], _, []).
intersectv([A|S1], S2, S) :- intersectv_2(S2, A, S1, S).
```

```
intersectv_2([], _, _, []).
intersectv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        intersectv_3(Order, A, S1, B, S2, S).

intersectv_3(<, _, S1, B, S2,     S) :- intersectv_2(S1, B, S2, S).
intersectv_3(=, A, S1, _, S2, [A|S]) :- intersectv(S1, S2, S).
intersectv_3(>, A, S1, _, S2,     S) :- intersectv_2(S2, A, S1, S).

intersectv_list([], []).
intersectv_list([InS|Sets], OutS) :- intersectv_list(Sets, InS, OutS).

intersectv_list([]) --> [].
intersectv_list([S|Sets]) --> intersectv(S), intersectv_list(Sets).

% ** Disjoint sets
disjointv([], _).
disjointv([A|S1], S2) :- disjointv_2(S2, A, S1).

disjointv_2([], _, _).
disjointv_2([B|S2], A, S1) :-
   compare(Order, A, B),
   disjointv_3(Order, A, S1, B, S2).

disjointv_3(<, _, S1, B, S2) :- disjointv_2(S1, B, S2).
disjointv_3(>, A, S1, _, S2) :- disjointv_2(S2, A, S1).

% *** Difference
diffv([], _, []).
diffv([A|S1], S2, S) :- diffv_2(S2, A, S1, S).

diffv_2([], A, S1, [A|S1]).
diffv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        diffv_3(Order, A, S1, B, S2, S).

diffv_3(<, A, S1, B, S2, [A|S]) :- diffv(S1, [B|S2], S).
diffv_3(=, A, S1, _, S2,     S) :- diffv(S1, S2, S).
diffv_3(>, A, S1, _, S2,     S) :- diffv_2(S2, A, S1, S).

% *** Union
unionv([], S2, S2).
unionv([A|S1], S2, S) :- unionv_2(S2, A, S1, S).

unionv_2([], A, S1, [A|S1]).
unionv_2([B|S2], A, S1, S) :-
        compare(Order, A, B),
        unionv_3(Order, A, S1, B, S2, S).

unionv_3(<, A, S1, B, S2, [A|S]) :- unionv_2(S1, B, S2, S).
unionv_3(=, A, S1, _, S2, [A|S]) :- unionv(S1, S2, S).
unionv_3(>, A, S1, B, S2, [B|S]) :- unionv_2(S2, A, S1, S).
```

```
% *** Element inclusion
includev(A, S1, S) :- includev_2(S1, A, S).

includev_2([], A, [A]).
includev_2([B|S1], A, S) :-
        compare(Order, A, B),
        includev_3(Order, A, B, S1, S).

includev_3(<, A, B, S1, [A,B|S1]).
includev_3(=, _, B, S1,   [B|S1]).
includev_3(>, A, B, S1,    [B|S]) :- includev_2(S1, A, S).

% *** Element exclusion
% Modified not to use extra heap space
excludev(A, S1, S) :- excludev_2(S1, A, S, S1).

excludev_2([], _, [], _).
excludev_2([B|S1], A, S, BS1) :-
   compare(Order, A, B),
   excludev_3(Order, A, B, S1, S, BS1).

excludev_3(<, _, _,  _,    BS1, BS1).
excludev_3(=, _, _, S1,     S1,   _).
excludev_3(>, A, B, S1,  [B|S],   _) :- excludev_2(S1, A, S, S1).

% *** Subset
subsetv([], _).
subsetv([A|S1], [B|S2]) :-
        compare(Order, A, B),
        subsetv_2(Order, A, S1, S2).

subsetv_2(=, _, S1, S2) :- subsetv(S1, S2).
subsetv_2(>, A, S1, S2) :- subsetv([A|S1], S2).

% For unordered lists S1:
small_subsetv([], _).
small_subsetv([A|S1], S2) :- inv(A, S2), small_subsetv(S1, S2).

% *** Membership
inv(A, [B|S]) :-
        compare(Order, A, B),
        inv_2(Order, A, S).

inv_2(=, _, _).
inv_2(>, A, S) :- inv(A, S).

% *** Remove adjacent identical elements:
uniqvar([], []).
uniqvar([A|List], Uniq) :- uniqvar(A, List, Uniq, []).

uniqvar(A, []) --> !, [A].
uniqvar(A, [B|List]) -->
   one_uniqvar(A, B),
   uniqvar(B, List).
```

```
one_uniqvar(A, B) --> {A==B}, !.
one_uniqvar(A, B) --> {A\==B}, [A], !.

% *** Filter all nonvariables from a list, keeping only the variables:
filter_vars(L, Vs) :- filter_vars(L, Vs, []).

filter_vars([]) --> [].
filter_vars([V|L]) --> {var(V)}, !, [V], filter_vars(L).
filter_vars([V|L]) --> {nonvar(V)}, !,   filter_vars(L).

% Filter all vars & carry along corresponding elements of second list:
filter_vars([], [], [], []).
filter_vars([X|L1], [Y|L2], [X|V1], [Y|V2]) :-
   var(X), !,
   filter_vars(L1, L2, V1, V2).
filter_vars([X|L1], [_|L2], V1, V2) :-
   nonvar(X), !,
   filter_vars(L1, L2, V1, V2).

% *** Gather all variables used in a term: (in a set or a bag)
varset(Term, VarSet) :- varbag(Term, VB), sort(VB, VarSet).
varbag(Term, VarBag) :- varbag(Term, VarBag, []).

varbag(Var) --> {var(Var)}, !, [Var].
varbag(Str) --> {nonvar(Str), !, functor(Str,_,Arity)}, varbag(Str, 1, Arity).

varbag(_Str, N, Arity) --> {N>Arity}, !.
varbag(Str, N, Arity) --> {N=<Arity}, !,
   {arg(N, Str, Arg)}, varbag(Arg),
   {N1 is N+1},
   varbag(Str, N1, Arity).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Logical set operations:

union([A|S1], [B|S2], [A|S])     :- A@<B, !, union(S1,     [B|S2], S).
union([A|S1], [B|S2], [A|S])     :- A=B,  !, union(S1,     S2,     S).
union([A|S1], [B|S2], [B|S])     :- B@<A, !, union([A|S1], S2,     S).
union([], S, S) :- !.
union(S, [], S).

intersect([A|S1], [B|S2], S)     :- A@<B, !, intersect(S1,     [B|S2], S).
intersect([A|S1], [B|S2], [A|S]) :- A=B,  !, intersect(S1,     S2,     S).
intersect([A|S1], [B|S2], S)     :- B@<A, !, intersect([A|S1], S2,     S).
intersect([], _, []) :- !.
intersect(_, [], []).

diff([A|S1], [B|S2], [A|S])      :- A@<B, !, diff(S1,     [B|S2], S).
diff([A|S1], [B|S2], S)          :- A=B,  !, diff(S1,     S2,     S).
diff([A|S1], [B|S2], S)          :- B@<A, !, diff([A|S1], S2,     S).
diff([], _, []) :- !.
diff(S, [], S).
```

```
in(A, [A|_]) :- !.
in(A, [B|S]) :- A@>B, in(A, S).

include(A, [B|S], [A,B|S]) :- A@<B, !.
include(A, [B|S], [B|S])   :- A==B, !.
include(A, [B|S], [B|S1])  :- A@>B, !, include(A, S, S1).
include(A, [], [A]).

subset([A|S1], [B|S2]) :- A=B,  !, subset(S1,     S2).
subset([A|S1], [B|S2]) :- A@>B, !, subset([A|S1], S2).
subset([], _).

not_disjoint([A|S1], [A|S2]) :- !.
not_disjoint([A|S1], [B|S2]) :- A@>B, !, not_disjoint([A|S1], S2).
not_disjoint([A|S1], [B|S2]) :- A@<B, !, not_disjoint(S1, [B|S2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Unique and duplicate element utilities:

% Return the set of entries that have duplicates:
filter_dups(Bag, Dups) :-
   list_to_key(Bag, KeyBag),
   keysort(KeyBag, KeySet),
   filter_dups(KeySet, Dups, []).

filter_dups([]) --> !.
filter_dups([V1-_,V2-_,V3-_|KeySet]) --> {V1==V2,V2==V3}, !,
        filter_dups([V2-_,V3-_|KeySet]).
filter_dups([V1-_,V2-_|KeySet]) --> {V1==V2}, !,
        [V1], filter_dups(KeySet).
filter_dups([V1-_|KeySet]) --> !,
        filter_dups(KeySet).

% Return the set of entries that are unique:
% This code is used in peephole to get the unique labels.
filter_uniq(Bag, Uniq) :-
        list_to_key(Bag, KeyBag),
        keysort(KeyBag, KeySet),
   filter_uniq(none, none, none, KeySet, Uniq, []).

filter_uniq(none, none, none, []) --> !.
filter_uniq(   A,    A,    B, KS) --> !,
   {get_next(KS, C, KS2)},
   filter_uniq(A, B, C, KS2).
filter_uniq(   A,    B,    B, KS) --> !,
   {get_next(KS, C, KS2)},
   {get_next(KS2, D, KS3)},
   filter_uniq(B, C, D, KS3).
filter_uniq(   A,    B,    C, KS) --> {B=e(X)}, [X],
   {get_next(KS, D, KS2)},
   filter_uniq(B, C, D, KS2).
```

```
get_next([], none, []).
get_next([A-_|KeySet], e(A), KeySet).

% Convert a list into a keylist, ready for keysort:
list_to_key([], []).
list_to_key([V|Bag], [V-V|KeyBag]) :- list_to_key(Bag, KeyBag).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Table utilities:

% This code implements a mutable array, represented as a binary tree.

% Access a value in logarithmic time and constant space:
% This predicate can be used to create the array incrementally.
get(node(N,W,L,R), I, V) :- get(N, W, L, R, I, V).

get(N, V, _, _, I, V) :- I=N, !.
get(N, _, L, R, I, V) :-
   compare(Order, I, N),
   get(Order, I, V, L, R).

get(<, I, V, L, _) :- get(L, I, V).
get(>, I, V, _, R) :- get(R, I, V).

% Access a value:
% This predicate cannot be used to create the array incrementally,
% but it is faster than get/3.
fget(node(N,W,L,R), I, V) :-
   compare(Order, I, N),
   fget_2(Order, I, V, W, L, R).

fget_2(<, I, V, _, L, _) :- fget(L, I, V).
fget_2(=, _, V, W, _, _) :- V=W.
fget_2(>, I, V, _, _, R) :- fget(R, I, V).

% Update an array in logarithmic time and space:
set(node(N,_,LT,RT),  I, V, node(N,V,LT,RT)) :- I=N, !.
set(node(N,VL,LT,RT), I, V, node(N,VL,LNew,RT)) :- I@<N, !, set(LT, I, V, LNew).
set(node(N,VL,LT,RT), I, V, node(N,VL,LT,RNew)) :- I@>N, !, set(RT, I, V, RNew).
set(leaf,             I, V, node(I,V,leaf,leaf)).

% Update an array in logarithmic time and space:
% This is faster than set/4.
fset(leaf,          I, V, node(I,V,leaf,leaf)).
fset(node(N,W,L,R), I, V, node(N,NW,NL,NR)) :-
   compare(Order, I, N),
   fset_2(Order, I, V, W, L, R, NW, NL, NR).

fset_2(<, I, V, W, L, R, W, NL, R) :- fset(L, I, V, NL).
fset_2(=, I, V, _, L, R, V, L, R).
fset_2(>, I, V, W, L, R, W, L, NR) :- fset(R, I, V, NR).

% Prevent any further insertions in the array:
```

```
seal(leaf) :- !.
seal(node(_,_,L,R)) :- seal(L), seal(R).

% Create a sealed array from a list with entries Node-Value
% or from a list of Nodes (in which case the values are the
% same as the nodes):
create_array(List, Tree) :-
   (key_list(List)
   -> KeyList=List
   ;  list_to_key(List, KeyList)
   ),
   random_permute(KeyList, Random),
        list_to_tree(Random, Tree).

list_to_tree([], T) :- seal(T).
list_to_tree([Node-Val|L], T) :- get(T, Node, Val), list_to_tree(L, T).

key_list([]).
key_list([_-_|_]).

% Succeeds for a non-empty array:
non_empty_array(node(_,_,_,_)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generation of unique atoms:

% This predicate is dynamic:
gensym_integer(1).

init_gensym :-
   abolish(gensym_integer, 1),
   asserta(gensym_integer(1)).

% Create a unique atom:
gensym(Atom) :- gensym("$internal_sym_", Atom).

% Create a unique atom with given prefix:
gensym(Pre, Atom) :-
   gennum(N),
   name(N, NL),
   append(Pre, NL, AL),
   name(A, AL), !,
   Atom=A.

% Create a unique atom with given prefix and with 'Name/Arity' embedded:
gensym(Pre, Name/Arity, Atom) :-
   atomic(Name),
   atomic(Arity), !,
   name(Name, L1),
   name(Arity, L2),
   append(Pre, L1, "/", L2, "_", Pre2),
   gensym(Pre2, Atom).
gensym(Pre, AnAtom, Atom) :-
```

```
   atomic(AnAtom), !,
   name(AnAtom, L),
   append(Pre, L, Pre2),
   gensym(Pre2, Atom).
gensym(Pre, Other, Atom) :-
   warning(['Erroneous second argument to gensym/3: ',Other]),
   gensym(Pre, Atom).

% Create a term NewHead which is Head with new name and extra arguments:
new_head(Prefix, Head, ExtraArgs, NewHead) :-
        functor(Head,Name,Arity),
        Head=..[Name|Args],
        append(Args, ExtraArgs, NewArgs),
        gensym(Prefix, Name/Arity, NewName),
        NewHead=..[NewName|NewArgs].

% Return the next integer:
gennum(N) :-
        gensym_integer(N),
        abolish(gensym_integer, 1),
        N1 is N+1,
        asserta(gensym_integer(N1)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Instantiation of variables in a term to letters:
% This creates the atoms A, B, ..., Z, A1, B1, ..., Z1, A2, B2, etc
% for the different variables, making the term much easier to read
% for people.

% Instantiate all variables in a term and create a new term:
% The old term is unchanged.
inst_vars(Term, Inst) :- copy(Term, Inst), inst_vars(Inst).

% Instantiate all variables in a term:
inst_vars(Term) :-
   varset(Term, Vars),
   inst_vars_list(0, Vars).

% Writeq of an element or a list of elements as separate Prolog terms:
inst_writeqs(X) :-   cons(X), !, inst_writeq_list(X).
inst_writeqs(X) :- \+cons(X), !, inst_writeq(X), wn('.').

inst_writeq_list([]).
inst_writeq_list([X|L]) :- inst_writeq(X), wn('.'), inst_writeq_list(L).

% Writeq of a term, Quintus version:
q_inst_writeq(Term) :-
   compile_option(system(quintus)), !,
   copy(Term, Copy),
   numbervars(Copy, 0, _),
   writeq(Copy).
q_inst_writeq(Term) :-
   inst_writeq(Term).
```

```
% Writeq of a term:
% This version does not handle operator priorities, but the output is
% otherwise nicer:
inst_writeq(Term) :-
   % Copy ensures alphabetic order of variable names:
   copy(Term, Copy),
   varset(Copy, Vars),
   inst_vars_names_list(0, Vars, Names),
   inst_writeq(Copy, Vars, Names).

inst_writeq(Term, Vars, Names) :- var(Term), !,
   memberv2(Term, Vars, Name, Names),
   write(Name).
inst_writeq(Term, Vars, Names) :- atomic(Term), !, writeq(Term).
inst_writeq(Term, Vars, Names) :- cons(Term), !,
   write('['),
   inst_writeq_listterm(Term, Vars, Names).
inst_writeq(Term, Vars, Names) :- binary_op(Term, A, Op, B), paren_op(Op), !,
   list_op(Term, Op, List, []),
   length(List, N),
   (N>1 -> write('('); true),
   inst_writeq_oplist(List, Op, Vars, Names),
   (N>1 -> write(')'); true).
inst_writeq(Term, Vars, Names) :- binary_op(Term, A, Op, B), !,
   inst_writeq(A, Vars, Names),
   write(Op),
   inst_writeq(B, Vars, Names).
inst_writeq(Term, Vars, Names) :- structure(Term), !,
   functor(Term, Na, Ar),
   writeq(Na), write('('),
   inst_writeq(1, Ar, Term, Vars, Names),
   write(')').

inst_writeq_listterm(X, Vars, Names) :- \+list(X), !,
   write('|'),
   inst_writeq(X, Vars, Names),
   write(']').
inst_writeq_listterm([], Vars, Names) :- !,
   write(']').
inst_writeq_listterm([Head|Tail], Vars, Names) :- cons(Tail), !,
   inst_writeq(Head, Vars, Names),
   write(','),
   inst_writeq_listterm(Tail, Vars, Names).
inst_writeq_listterm([Head|Tail], Vars, Names) :- \+cons(Tail), !,
   inst_writeq(Head, Vars, Names),
   inst_writeq_listterm(Tail, Vars, Names).

inst_writeq_oplist([], _, _, _).
inst_writeq_oplist([Head|Tail], Op, Vars, Names) :- cons(Tail), !,
   inst_writeq(Head, Vars, Names),
   write(Op),
   inst_writeq_oplist(Tail, Op, Vars, Names).
inst_writeq_oplist([Head|Tail], Op, Vars, Names) :- nil(Tail), !,
   inst_writeq(Head, Vars, Names).

list_op(Term, Op) --> {nonvar(Term)},
   {binary_op(Term, A, Op, B)}, !,
   list_op(A, Op),
   list_op(B, Op).
list_op(Term, _) --> [Term].

paren_op(’,’).
paren_op(’;’).
paren_op((:-)).

binary_op((A,B),   A,   ’,’, B).
binary_op((A;B),   A,   ’;’, B).
binary_op((AˆB),   A,   ’ˆ’, B).
binary_op((A=B),   A,   ’=’, B).
binary_op((A\=B),  A,  ’\=’, B).
binary_op((A:-B),  A,  (:-), B).
binary_op((A->B),  A,  (->), B).
binary_op((A=..B), A, ’=..’, B).
binary_op((A@<B),  A,  ’@<’, B).
binary_op((A@=<B), A, ’@=<’, B).
binary_op((A@>B),  A,  ’@>’, B).
binary_op((A@>=B), A, ’@>=’, B).
binary_op((A<B),   A,   ’<’, B).
binary_op((A=<B),  A,  ’=<’, B).
binary_op((A>B),   A,   ’>’, B).
binary_op((A>=B),  A,  ’>=’, B).
binary_op((A=:=B), A, ’=:=’, B).
binary_op((A=\=B), A, ’=\=’, B).
binary_op((A\==B), A, ’\==’, B).
binary_op((A==B),  A,  ’==’, B).
binary_op((A+B),   A,   ’+’, B).
binary_op((A-B),   A,   ’-’, B).
binary_op((A/B),   A,   ’/’, B).
binary_op((A*B),   A,   ’*’, B).
binary_op((A//B),  A,  ’//’, B).
binary_op((A/\B),  A,  ’/\’, B).
binary_op((A\/B),  A,  ’\/’, B).
binary_op((A mod B), A, ’ mod ’, B).
binary_op((A is B),  A,  ’ is ’, B).

inst_writeq(I, Ar, _, _, _) :- I>Ar, !.
inst_writeq(I, Ar, Term, Vars, Names) :- I=:=Ar, !,
   arg(I, Term, X),
   inst_writeq(X, Vars, Names).
inst_writeq(I, Ar, Term, Vars, Names) :- I<Ar, !,
   arg(I, Term, X),
   inst_writeq(X, Vars, Names),
   write(’,’),
   I1 is I+1,
   inst_writeq(I1, Ar, Term, Vars, Names).

inst_vars_list(_, []) :- !.
inst_vars_list(I, [V|Vars]) :- !,
   var_name(I, V),
   I1 is I+1,
   inst_vars_list(I1, Vars).

inst_vars_names_list(_, [], []) :- !.
inst_vars_names_list(I, [V|Vars], [W|Names]) :- !,
   var_name(I, W),
   I1 is I+1,
   inst_vars_names_list(I1, Vars, Names).

% Get a unique variable name from an integer:
var_name(Int, Atom) :- 0=<Int, Int<26, !,
   Ascii is Int+"A",
   name(Atom,[Ascii]).
var_name(Int, Atom) :- Int>=26, !,
   Letter is (Int mod 26) + "A",
   Number is (Int // 26),
   name(Number, List),
   name(Atom, [Letter|List]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Copy a term keeping the structure but using fresh variables:
% Timing notes:
% 1. Copy1 is by far the fastest; for a moderately sized structure
%    takes ˜67ms, whereas copy2 takes 2000 ms. (Quintus V1.6 comp. on Sun-3/50).
% 2. Copy2 and copy3 are almost identical in speed.  Copy2 wins if there
%    are many shared variables, otherwise copy3 wins.
% 3. Timing comparisons:
%    (Time in ms on rigel, a slow Sun 3/50, with Quintus 2.0 and C-Prolog 1.5)
%  Routine      Simple          Complex
%       Quintus C-Prolog Quintus C-Prolog
%  copyB     4.2      20.         58   200
%  copy1     8.5       3.3        65   180
%  copy2     5.1      74.        985 15000
%  copy3     4.2      59.        750 11200
%
%    Simple structure: a(A,A,a,a(A))
%    Complex structure: I, where:
%         A=a(X,Y),B=b(A,A),C=c(B,B),D=d(C,C),E=e(D,D),
%         F=f(E,E),G=g(F,F),H=h(G,G),I=i(H,H)
% This causes large virtual mem. usage, because it is used
% inside of code which asserts/retracts compile_option/1: (?)
% copy(T1, T2) :- compile_option(system(quintus)), !, copy_term(T1, T2).
copy(T1, T2) :- copyB(T1, T2).

% Implementation B: using bagof.
copyB(Term1, Term2) :-
   bagof(Term1, true, [Term2]).

% Implementation 1: using assert/abolish.
copy1(Term1, Term2) :-
   assert(global_copy(Term1)),
   global_copy(X),
   abolish(global_copy, 1), !,
   X=Term2.

% Implementation 2: using a symbol table.
copy2(Term1, Term2) :-
   varset(Term1, Set), make_sym(Set, Sym),
   copy2(Term1, Term2, Sym), !.

% Implementation 3: using varbag symbol table.
copy3(Term1, Term2) :-
   varbag(Term1, Bag), make_sym(Bag, Sym),
        copy2(Term1, Term2, Sym), !.

copy2(V1, V2, Sym) :- var(V1), !, retrieve_sym(V1, Sym, V2).
copy2(X1, X2, Sym) :- nonvar(X1), !,
   functor(X1,Name,Arity),
   functor(X2,Name,Arity),
   copy2(X1, X2, Sym, 1, Arity).

copy2(_X1,_X2,_Sym, N, Arity) :- N>Arity, !.
copy2(X1, X2, Sym, N, Arity) :- N=<Arity, !,
   arg(N, X1, Arg1),
   arg(N, X2, Arg2),
   copy2(Arg1, Arg2, Sym),
   N1 is N+1,
   copy2(X1, X2, Sym, N1, Arity).

retrieve_sym(V, [p(W,X)|_Sym], X) :- V==W, !.
retrieve_sym(V, [_|Sym], X) :- retrieve_sym(V, Sym, X).

make_sym([], []).
make_sym([V|L], [p(V,_)|S]) :- make_sym(L, S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Term mapping operations:
% If the size of the Args lists becomes large then they should be stored
% in trees or in hash tables.

map_vars(L1, L2, T1, T2) :-
   filter_vars(L1, L2, V1, V2),
   map_terms(V1, V2, T1, T2).

% Perform the mapping AArgs->XArgs on the term AT to get XT.
% I.e. replace all occurrences in AT of terms in AArgs by
% the corresponding terms of XArgs to get XT.
map_terms(AArgs, XArgs, AT, XT) :- memberv2(AT, AArgs, XT, XArgs), !.
map_terms(AArgs, XArgs, AT, XT) :- var(AT), !, XT=AT.
map_terms(AArgs, XArgs, AT, XT) :- nonvar(AT), !,
   functor(AT, Name, Arity),
   functor(XT, Name, Arity),
   map_terms_seq(1, Arity, AArgs, XArgs, AT, XT).
```

```
% For sequencing down a structure:
map_terms_seq(I, Arity, _, _, _, _) :- I>Arity, !.
map_terms_seq(I, Arity, AArgs, XArgs, AT, XT) :- I=<Arity, !,
   arg(I, AT, ArgA),
   arg(I, XT, ArgX),
   map_terms(AArgs, XArgs, ArgA, ArgX),
   I1 is I+1,
   map_terms_seq(I1, Arity, AArgs, XArgs, AT, XT).

% Same as above and collect the list of AArgs members that were found in AT:
map_terms(AArgs, XArgs, AT, XT) --> {memberv2(AT, AArgs, XT, XArgs)}, !, [AT].
map_terms(AArgs, XArgs, AT, XT) --> {var(AT)}, !, {XT=AT}.
map_terms(AArgs, XArgs, AT, XT) --> {nonvar(AT)}, !,
   {functor(AT, Name, Arity)},
   {functor(XT, Name, Arity)},
   map_terms_seq(1, Arity, AArgs, XArgs, AT, XT).

map_terms_seq(I, Arity, _, _, _, _) --> {I>Arity}, !.
map_terms_seq(I, Arity, AArgs, XArgs, AT, XT) --> {I=<Arity}, !,
   {arg(I, AT, ArgA)},
   {arg(I, XT, ArgX)},
   map_terms(AArgs, XArgs, ArgA, ArgX),
   {I1 is I+1},
   map_terms_seq(I1, Arity, AArgs, XArgs, AT, XT).

% Map a list of instructions and get a difference list back:
% Also add tag pragmas before the instructions.
% This is used in clause_code to implement dereference chain length reduction.
map_instr_list([], _, _) --> [].
map_instr_list([I|AT], AArgs, XArgs) -->
   {map_terms(AArgs, XArgs, I, J, MapBag, [])},
   pragma_tag_list(MapBag),
   [J],
   map_instr_list(AT, AArgs, XArgs).

pragma_tag_list([]) --> [].
pragma_tag_list([X|Bag]) --> pragma_tag(X, var), pragma_tag_list(Bag).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List operations:

append(A, B, C) :- difflist(A, C, B).

append([], B, C, D) :- append(B, C, D).
append([X|A], B, C, [X|D]) :- append(A, B, C, D).

append([], B, C, D, E) :- append(B, C, D, E).
append([X|A], B, C, D, [X|E]) :- append(A, B, C, D, E).

append([], B, C, D, E, F) :- append(B, C, D, E, F).
append([X|A], B, C, D, E, [X|F]) :- append(A, B, C, D, E, F).

c_append(true, C, C).
c_append((X,C1), C2, (X,C3)) :- c_append(C1, C2, C3).

d_append(fail, D, D).
d_append((X;D1), D2, (X;D3)) :- d_append(D1, D2, D3).

% Converting lists to difference lists:
% (Also usable for DCG’s.)
difflist([]) --> [].
difflist([X|L]) --> [X], difflist(L).

% Converting conjunctions to difference lists:
conjlist((A,C)) --> !, conjlist(A), conjlist(C).
conjlist(A) --> [A].

% Insert a list:
insert(L) --> difflist(L).
insertlist(L) --> difflist(L).

% Insert a difference list:
insert(Head, Tail, Head, Tail).

% Inline addition & subtraction:
add(A, B, C) :- C is B+A.
sub(A, B, C) :- C is B-A.

% Delete an element from a list:
delete([], _, []).
delete([X|L], X, D)  :- !, delete(L, X, D).
delete([Y|L], X, [Y|D]) :- delete(L, X, D).

reverse(A, B) :- reverse(A, [], B).

reverse([X|A], L, B) :- reverse(A, [X|L], B).
reverse([], B, B).

last([X], X).
last([_|L], X) :- last(L, X).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

memberv(X, [Y|L]) :-
   (X==Y
    -> true
     ; memberv(X, L)
   ).

memberv2(A, [B|_], X, [X|_]) :- A==B, !.
memberv2(A, [_|L], X, [_|Y]) :- memberv2(A, L, X, Y).

% List of integers from L to H (low to high):
range_list(I, J,    []) :- I>J, !.
range_list(I, J,   [J]) :- I=J, !.
range_list(I, J, [I|L]) :- I<J, !, I1 is I+1, range_list(I1, J, L).
```

```
% List of integers from H to L (high to low):
downrange_list(I, J,    []) :- I<J, !.
downrange_list(I, J,   [J]) :- I=J, !.
downrange_list(I, J, [I|L]) :- I>J, !, I1 is I-1, downrange_list(I1, J, L).

% Pseudo-random permutation of a list in O(n log n) time:
random_permute(List, Perm) :-
   ran_keys(List, 23141, RanKeys),
   keysort(RanKeys, KeyList),
   key_to_list(KeyList, Perm).

ran_keys([], _, []).
ran_keys([I|List], K, [K-I|Ran]) :-
   random_step(K, Kp),
   ran_keys(List, Kp, Ran).

random_step(K, Kp) :- Kp is (K*1237+2116) mod 44449.
% These numbers are too big for C-Prolog:
% random_step(K, Kp) :- Kp is (K*123456+66390) mod 314159.

key_to_list([], []).
key_to_list([_-X|KeyList], [X|List]) :- key_to_list(KeyList, List).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Conjunction and disjunction operations:

member_conj(X, (C,_)) :- member_conj(X, C), !.
member_conj(X, (_,C)) :- member_conj(X, C), !.
member_conj(X, X).

memberv_conj(X, (C,_)) :- memberv_conj(X, C), !.
memberv_conj(X, (_,C)) :- memberv_conj(X, C), !.
memberv_conj(X, Y) :- \+Y=(_,_), X==Y, !.

reverse_conj(A, B) :- reverse_conj(A, true, B).

reverse_conj((X,A), L, B) :- reverse_conj(A, (X,L), B).
reverse_conj(true, B, B).

append_conj(true, L, L).
append_conj((X,L1), L2, (X,L3)) :- append_conj(L1, L2, L3).

append_conj(true, B, C, D) :- append_conj(B, C, D).
append_conj((X,A), B, C, (X,D)) :- append_conj(A, B, C, D).

append_conj(true, B, C, D, E) :- append_conj(B, C, D, E).
append_conj((X,A), B, C, D, (X,E)) :- append_conj(A, B, C, D, E).

last_conj((L,C), L) :- all_true(C), !.
last_conj((_,C), L) :- last_conj(C, L).

% Intersecting two conjunctions:
```

```
intersectv_conj(A, B, C) :- intersectv_conj(A, B, C, true).

intersectv_conj((C1,C2), Conj) --> !,
   intersectv_conj(C1, Conj),
   intersectv_conj(C2, Conj).
intersectv_conj(A, Conj) --> {\+A=(_,_), memberv_conj(A, Conj)}, !, co(A).
intersectv_conj(A, Conj) --> {\+A=(_,_)}, !.

% Combining two conjunctions without redundancy:
% Don’t ever use this to update a mode formula unless there are
% no predicates like var(X) which may become false.  Such an update must
% recognize these predicates.
unionv_conj(Conj1, _, fail) :- memberv_conj(fail, Conj1), !.
unionv_conj(_, Conj2, fail) :- memberv_conj(fail, Conj2), !.
unionv_conj(Conj1, Conj2, Comb) :- unionv_conj(Conj1, Conj2, Comb, Conj2).

unionv_conj((A,B), Conj) --> !, unionv_conj(A, Conj), unionv_conj(B, Conj).
unionv_conj(true, Conj) --> !.
unionv_conj(Goal, Conj) --> {succeeds(Goal)}, !.
unionv_conj(Goal, Conj) --> {\+succeeds(Goal), \+Goal=(_,_), \+Goal=true}, !,
        conj_if_nomember(Goal, Conj).

conj_if_nomember(Goal, Conj) --> {memberv_conj(Goal, Conj)}, !, [].
conj_if_nomember(Goal, Conj) --> {\+memberv_conj(Goal, Conj)}, !, co(Goal).

% Flatten a conjunction and terminate it with ’true’:
flat_conj(Conj, FConj) :- flat_conj(Conj, FConj, true).

flat_conj(true) --> !.
flat_conj((A,B)) --> !, flat_conj(A), flat_conj(B).
flat_conj(A) --> co(A).

% Squeeze a conjunction as small as possible:
squeeze_conj((A,B),S) :- \+all_true(B),!,squeeze_conj(B, X), flat_conj(A, S, X).
squeeze_conj((A,B),S) :- \+all_true(A),!,squeeze_conj(A, X), flat_conj(B, S, X).
squeeze_conj(A, true) :- all_true(A), !.
squeeze_conj(A, A).

all_true(true).
all_true((A,B)) :- all_true(A), all_true(B).

% Replace the beginning of Conj with RepConj:
replace_start_conj(true, Conj, Conj).
replace_start_conj((G,RepConj), (_,Conj), (G,NewConj)) :-
   replace_start_conj(RepConj, Conj, NewConj).

% Split a disjunction into first choice and other choices:
split_disj((First;Rest), First, Rest) :- !.
split_disj(       First, First, fail).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Some new type-checking predicates:
```

```
% Succeed if argument is ground:
% Could use map(ground, L).
ground(A) :- nonvar(A), functor(A, _, N), ground(N, A).

ground(0, _) :- !.
ground(N, A) :- N>0, !,
   arg(N, A, X),
   ground(X),
   N1 is N-1,
   ground(N1, A).

nil(X)  :- atom(X), X=[].
cons(X) :- nonvar(X), X=[_|_].
list(X) :- (nil(X); cons(X)).
structure(X) :- nonvar(X), \+atomic(X), \+X=[_|_].

compound(X)  :- nonvar(X), \+atomic(X).

simple(X) :- var(X), !.
simple(X) :- atomic(X), !.

negative(X) :- integer(X), X<0.
nonnegative(X) :- integer(X), X>=0.
positive(X) :- integer(X), X>0.
nonpositive(X) :- integer(X), X=<0.

full_list([]).
full_list([_|L]) :- full_list(L).

make_list(A, [A]) :- \+list(A), !.
make_list(A,   A) :-   list(A), !.

conj_p(X) :- nonvar(X), (X=(_,_); X=true), !.

disj_p(X) :- nonvar(X), (X=(_;_); X=fail), !.

strong_disj_p(X) :- nonvar(X), X=(_;_).

conj_p((A,B), A, B).
disj_p((A;B), A, B).

co(G, (G,R), R).
di(G, (G;R), R).

unify_p(_=_).

call_p(G) :- \+unify_p(G).

% Split a unification goal into its parts:
split_unify(X=Y, X, Y).
split_unify(Y=X, X, Y).

split_unify(X, Y, X, Y).
split_unify(X, Y, Y, X).
```

```
split_unify_v(X=Y, X, Y) :- var(X).
split_unify_v(X=Y, Y, X) :- var(Y).

split_unify_v(X, Y, X, Y) :- var(X).
split_unify_v(X, Y, Y, X) :- var(Y).

split_unify_v_nv(X=Y, X, Y) :- var(X), nonvar(Y).
split_unify_v_nv(X=Y, Y, X) :- var(Y), nonvar(X).

split_unify_v_nv(X, Y, X, Y) :- var(X), nonvar(Y).
split_unify_v_nv(X, Y, Y, X) :- var(Y), nonvar(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Numeric utilities:

range(L,L,H).
range(L,I,H) :- L<H, L1 is L+1, range(L1,I,H).

downrange(L,H,H).
downrange(L,I,H) :- L<H, H1 is H-1, downrange(L,I,H1).

% This version needs O(N log N) time and O(log N) stack space for C-Prolog
% Old range needs O(N*N) time and O(N) local stack space on C-Prolog.
crange(L,L,L).
crange(L,I,H) :-
   L<H,
   M is (L+H)//2,
   crange(L,I,M).
crange(L,I,H) :-
   L<H,
   M is (L+H)//2+1,
   crange(M,I,H).

gen_integer(I) :- range(1, I, 10000).

% The following definitions of maximum and minimum can be used as a test of
% the compiler.  If the actual call uses integers then the predicates should
% be compiled with integer comparisons instead of general term comparisons.

% Maximum of two terms:
max(A, B, A) :- A@>=B, !.
max(A, B, B) :- B@>A.

maximum(A, B, C) :- max(A, B, C).

% Minimum of two terms:
min(A, B, A) :- A@=<B, !.
min(A, B, B) :- B@<A.

minimum(A, B, C) :- min(A, B, C).

min_integer(A, B, A) :- A<B, !.
min_integer(A, B, B) :- A>=B, !.

% Maximum of a list of numbers:
maxlist([M1|L], M) :- maxlist(L, M1, M), !.

maxlist([], M, M).
maxlist([M1|L], M2, M) :- max(M1, M2, M3), maxlist(L, M3, M).

% Minimum of a list of numbers:
minlist([M1|L], M) :- minlist(L, M1, M), !.

minlist([], M, M).
minlist([M1|L], M2, M) :- min(M1, M2, M3), minlist(L, M3, M).

% List length utilities:

shorter_than([], N) :- N>0.
shorter_than([_|L], N) :- N>0, N1 is N-1, shorter_than(L, N1).

longer_than([_|_], N) :- N=<0.
longer_than([_|L], N) :- N>0, N1 is N-1, longer_than(L, N1).

% Number of non-fail choices in a disjunction:
length_disj(Disj, N) :- length_disj(Disj, 0, N).

length_disj(fail) --> !.
length_disj((A;B)) --> !, length_disj(A), length_disj(B).
length_disj(G) --> {\+disj_p(G)}, !, add(1).

% Number of non-true and non-cut goals in a conjunction:
length_conj(Conj, N) :- length_conj(Conj, 0, N).

length_conj(true) --> !.
length_conj((A,B)) --> !, length_conj(A), length_conj(B).
length_conj(Cut) --> {cut_p(Cut)}, !.
length_conj(G) --> {\+conj_p(G)}, !, add(1).

% Number of user-defined goals & tests in a conjunction:
% Count all tests up to the first non-test, then count all the rest,
% excluding the cuts.  Cuts are much less important for code size.
length_test_user(Conj, T, U) :-
   length_test(Conj, Rest, 0, T),
   length_conj(Rest, U).

length_test( true, true) --> !.
length_test((T,B), Body) --> {test(T)}, !, add(1), length_test(B, Body).
length_test(    T, true) --> {test(T)}, !, add(1).
length_test( Body, Body) --> [].

% Logical utilities:

or( true,  true,  true).
or( true, false,  true).
or(false,  true,  true).
or(false, false, false).

or(A, B, C, D) :- or(A, B, X), or(X, C, D).

and( true,  true,  true).
and( true, false, false).
and(false,  true, false).
and(false, false, false).

and(A, B, C, D) :- and(A, B, X), and(X, C, D).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Timing utilities for Quintus and C-Prolog:

% 1. Simple utilities:
t(A) :- A is cputime.
t(A,B) :- B is cputime-A.

ct(A) :- A is cputime.
ct(A,B) :- B is cputime-A.

qt :- statistics(runtime,_).
qt(Time) :- statistics(runtime,[_,Time]).
qt(_,Time) :- statistics(runtime,[_,Time]).

% 2. More sophisticated method that
%    compensates for the time taken by an empty loop.
% Method of use:
% | ?- time(1000), Query, fail.
% Time = 221.65 ms

% *** Quintus timing:
qtime(N) :- qtime(N, _).

qtime(N, Time) :-
   qtime_empty(N, Empty),
   qtime_loop(N, Empty, Time).

qtime_empty(N, _) :-
   statistics(runtime, _), range(1,_,N), fail.
qtime_empty(_, Empty) :-
   statistics(runtime, [_,Empty]).

qtime_loop(N, _, _) :-
   statistics(runtime, _), range(1,_,N).
qtime_loop(N, Empty, Time) :-
   statistics(runtime, [_,Delta]),
   Time is (Delta-Empty)/N,
   read(_),
   w('Time per query = '),w(Time),w(' ms'),
   (Time<10.0
    -> w(' (0.0 - 0.05 ms too high)')
     ; true
   ), nl.

% *** C-Prolog timing:
ctime(N) :-
   T1 is cputime,
   ctime_empty(N),
   T2 is cputime,
   ctime_loop(N, T1, T2).

ctime_empty(N) :- crange(1,_,N), fail.
ctime_empty(_).

ctime_loop(N, _, _) :- crange(1,_,N).
ctime_loop(N, T1, T2) :-
   T3 is cputime,
   Delta is T3-T2,
   Empty is T2-T1,
   Time is (Delta-Empty)*1000/N,
   w('Time per query = '),w(Time),w(' ms'),
   (Time<100.0
    -> w(' (0.0 - 0.5 ms too high)')
     ; true
   ), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Writing compiled code in human-readable or Prolog-readable form:

write_code([]) :- !.
write_code(L) :- \+compile_option(write), !.
write_code(L) :-
   compile_option(write),
   \+compile_option(flat), !,
   write_code(L, 4).
write_code(L) :-
   compile_option(write),
   compile_option(flat), !,
   write_output(L).

write_code([], _).
write_code([I|L], N) :- write_code(I, L, N).

% Write in human-readable form:
write_code(switch(unify,Type,V,Wbr,Rbr,Fail), L, N) :- !,
        N1 is N+4,
        tab(N), w('switch('),w(V),wn(') {'),
        tab(N), tag(var, Tvar), w(Tvar), wn(':'),
        write_code(Wbr, N1),
        tab(N), w(Type),wn(':'),
        write_code(Rbr, N1),
        tab(N), w('else: '),wn(Fail),
        tab(N), wn('}'),
        write_code(L, N).
write_code(label(Lbl), L, N) :-
```

```
   N1 is N-4,
   tab(N1), wq(Lbl), wn(’:’),
        write_code(L, N).
write_code(procedure(Name), L, N) :-
   N1 is N-4,
   tab(N1), nl, wqn(procedure(Name)), nl,
        write_code(L, N).
write_code(Instr, L, N) :-
   \+(Instr=label(Lbl)),
   \+(Instr=procedure(_)),
        \+(Instr=switch(unify,Type,V,Wbr,Rbr,Fail)),
        tab(N), wqn(Instr),
        write_code(L, N).

% Write in Prolog-readable form:
write_output([]).
write_output([I|Code]) :-
   nl_if_procedure(I),
   tab_if_nonlbl(I), wq(I), wn(’.’),
   write_output(Code).

tab_if_nonlbl(label(_)) :- !.
tab_if_nonlbl(procedure(_)) :- !.
tab_if_nonlbl(I) :-
   \+I=label(_),
   \+I=procedure(_), !,
   put(9).

nl_if_procedure(procedure(_)) :- !, nl.
nl_if_procedure(_).

% Write the source code if ’source’ option is active:
write_source(S, M) :- \+compile_option(source), !.
write_source(S, M) :- compile_option(source),
   inst_vars(M, Modes),
   xwrite_modes(Modes),
   inst_vars(S, Source),
   xwrite_source(Source).

xwrite_modes([]) :- !.
xwrite_modes(Modes) :- cons(Modes), !,
   nl,
   write(’% Modes:’), nl,
   xwrite_clauses(Modes), nl,
   write(’% Source:’).
xwrite_modes(Mode) :- \+cons(Mode), !,
   xwrite_modes([Mode]).

xwrite_source([]) :- !.
xwrite_source(Source) :- cons(Source),
   nl,
   xwrite_clauses(Source).

xwrite_clauses([]).
xwrite_clauses([Cl|Cls]) :-
   write('%    '),write(Cl),write('.'), nl,
   xwrite_clauses(Cls).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Writing utilities:

w(X)   :- write(X).
wq(X)  :- writeq(X).
wn(X)  :- write(X), nl.
wqn(X) :- writeq(X), nl.

% Printing messages (there are four levels of information):
% comment: for information only
% aesthetic: remark on internal compiler structure
% warning: possible problem, but compiled correctly.
% error: possible problem, not necessarily compiled correctly.
comment(L)   :- compile_option(comment), !,
   make_msg(L, Msg), cm, msg_list(Msg, cm).
comment(L)   :- \+compile_option(comment), !.
aesthetic(L) :- make_msg(L, Msg), ae, msg_list(Msg, ae).
warning(L)   :- make_msg(L, Msg),  w, msg_list(Msg,  w).
error(L)     :- make_msg(L, Msg),  e, msg_list(Msg,  e).

% Print messages, using variables numbered according to the head :
comment(Head, L)   :- compile_option(comment), !,
   make_msg(Head, L, Msg), cm, msg_list(Msg, cm).
comment(Head, L)   :- \+compile_option(comment), !.
aesthetic(Head, L) :- make_msg(Head, L, Msg), ae, msg_list(Msg, ae).
warning(Head, L)   :- make_msg(Head, L, Msg),  w, msg_list(Msg,  w).
error(Head, L)     :- make_msg(Head, L, Msg),  e, msg_list(Msg,  e).

make_msg(Head, L, Msg) :- inst_vars(s(Head,L), s(_,C)), make_list(C, Msg).
make_msg(L, Msg) :- inst_vars(L, C), make_list(C, Msg).

msg_list([], _) :- !, nl.
msg_list([X|Msg], K) :- msg_one(X, K), msg_list(Msg, K).

msg_one(tab, _) :- !, put(9).
msg_one(nl, cm) :- !, nl, cms.
msg_one(nl, ae) :- !, nl, aes.
msg_one(nl,  w) :- !, nl,  ws.
msg_one(nl,  e) :- !, nl,  es.
msg_one(X,   _) :- !, w(X).

cm  :- w('% *** ').
cms :- w('% *** ').
ae  :- w('% *** Aesthetic: ').
aes :- w('% ***            ').
w   :- w('% *** Warning: ').
ws  :- w('% ***          ').
e   :- w('% *** Error: ').
es  :- w('% ***        ').
```

```
f   :- wn(' ***').

% Debug messages:
wd(Msg) --> {wd(Msg)}.

wd(Msg) :- compile_option(debug), !, w(Msg).
wd(Msg) :- \+compile_option(debug).

write_debug(Msg) --> {write_debug(Msg)}.

write_debug(Msg) :- wd(Msg), nl_debug.

nl_debug :-
   (compile_option(debug)
   -> nl
   ;  true
   ).

write_list([]).
write_list([X|L]) :- w(X), nl, write_list(L).

% Check of correctness:
not_used(Here) :- w(Here), wn('- not used!').
not_used(Here) --> {not_used(Here)}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Statistics option:

init_stats :-
   abolish(compile_cputime,3),
   get_cputime(U), !,
   assert(compile_cputime(start,none,U)).
init_stats.

done_stats :-
   get_cputime(S), !,
   retract(compile_cputime(OldPlace,OldN,OldU)),
   nl,w('% Cputime between '),w(OldPlace),write_num(OldN),
   w(' and finish is '),
   Time is S-OldU,
   wn(Time),
   ttyflush_quintus.
done_stats.

stats(A, B) --> {stats(A, B)}.

stats(Place, N) :-
   get_cputime(S),
   compile_option(stats(L)),
   (member(Place, L); L=Place),
   !,
   retract(compile_cputime(OldPlace,OldN,OldU)),
   w('% Cputime between '),w(OldPlace),write_num(OldN),
   w(’ and ’),w(Place),write_num(N),
   w(’ is ’),
   Time is S-OldU,
   wn(Time),
   ttyflush_quintus,
   get_cputime(U),
   assert(compile_cputime(Place,N,U)), !.
stats(_, _).

ttyflush_quintus :- the_system(quintus), !, ttyflush.
ttyflush_quintus.

write_num(none) :- !.
write_num(N) :- w(’-’), w(N).

get_cputime(Time) :- the_system(cprolog), !,
   Time is cputime.
get_cputime(Time) :- the_system(quintus), !,
   statistics(runtime,[T,_]), % First arg is absolute, second is relative.
   Time is T/1000.0.
get_cputime(Time) :- \+the_system(_),
   wn(’% I do not know how to get the cputime in this system.’),
   Time is 0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```