Evaluation in DATR is co-NP-hard

2 NP-Hardness: Boolean satisfiability

Here we show that DATR is NP-hard by providing a theory which recognizes boolean formulae which are satisfiable. In the path notation we adopt, wffs are represented in Reverse Polish notation ('RPN' or 'postfix'), and variables are represented by strings over the alphabet 0, . . ., 9, enclosed in square brackets. As an example, the formula (1) could be represented by the path (2):

(1) $(v_1 \vee v_2) \wedge (v_3 \wedge \neg v_2)$

(2) `<[ 1 ] [ 2 ] | [ 3 ] [ 2 ] ~ & &>`

A formula such as (2), read left to right, is a program for a stack evaluator, but the stack machine we have in mind processes only constants, and not variables. We replace the variable references by applying an *assignment function*. An assignment function is a mapping which assigns a value of 0 or 1 to every variable in the formula, denoting its truth value. An assignment function can be represented as a binary sequence of the same length as the number of variable occurrences in the wff. An assignment is 'valid' (*i.e.*, a valid substitution) if it assigns the same value to each occurrence of a given variable. Here are examples of two value assignments for formula (2), the second of which is invalid because v_2 is not uniquely mapped to a truth value:

```
wff                 <[ 1 ] [ 2 ] | [ 3 ] [ 2 ] ~ & &>
valid assignment    <  0     1       0     1       >
invalid assignment  <  0     0       0     1       >
```

We say that an assignment function satisfies a formula *iff.*:

(a) it is valid; and

(b) the result of executing the formula instantiated by the assignment function on a stack machine is 1.

The set of assignment functions for a particular formula with n variable occurrences are the binary strings of length n; there are 2^n such assignment functions. The ones we are interested in, the solutions, are those which satisfy a formula by the above definition.

The algorithm we present for testing satisfiability of a wff can be broadly broken down into two components: generating an enumeration of possible assignment functions; and testing each for satisfaction against the two conditions given above. By expressing this computation as the result of a DATR query, we show that the time complexity of DATR query evaluation is at least NP-hard. We will define a node `Satisfy` at which the value of a path representing a wff is an assignment function which satisfies the wff, and the empty list if no such assignment exists. Thus the type of theorems we wish to derive are illustrated by:

```
Satisfy: <[ 1 2 3 4 ] ~ ;> = 0
         <[ 3 ] [ 3 ] & ;> = 1 1
         <[ 3 ] [ 3 ] ~ & ;> = ()
         <[ 3 ] ~ [ 3 ] ~ [ 2 ] & | ;> = 0 0 0.
```

The first wff is satisfied by the assignment $\{v_{1234} = 0\}$; the second by $\{v_3 = 1\}$, the third is not satisfiable; and the last by $\{v_3 = 0,\ v_2 = 0\}$. The symbol ; is an argument terminator.

A recognizer for the language of satisfiable wffs is node `Satisfiable`, defined in terms of `Satisfy` as follows:

```
Satisfiable:
    <> == <if Satisfy ;>
    <if ;> == false
    <if> == true.
```

Theorems of `Satisfiable` are of the following form:

```
Satisfiable: <[ 1 ] [ 1 ] | ;> = true
             <[ 1 ] [ 1 ] ~ & ;> = false.
```

As we use the primitives defined in **`args.dtr`** and **`arglogic.dtr`** (developed in Moser (1992a)), we first define a preamble of symbols which will be recognized by the path-as-argument manipulation primitives. (A concise listing of the node definitions is given in Appendix B.3.) The DATR variable **`$varletter`** denotes the alphabet for propositional variable names. The complete repertoire of terminals (**`$terminal`**) are the boolean values 0, and 1, the variable name alphabet, logical operators **`&`** and `|`, and the negation symbol `~`, as well as the square brackets used to delimit propositional variables and the letter `a`, the (singleton) alphabet over which we will represent distance lists (see below):

```
#vars $operator: & | ~.
#vars $varletter:  0 1 2 3 4 5 6 7 8 9.
#vars $terminal: a 0 1 2 3 4 5 6 7 8 9 & | ~ [ ].
```

We also require declarations of the symbols we use to represent the argument terminator and
menrih1rationst

The definition of `Generate` is as follows:

```
Length:<> == Arg1.
Formula:<> == Arg2.
Assn_fn:<> == Arg3.
Left_branch_result:<> == Arg1.

Generate:
    <;> == Test:< Formula ; Assn_fn ; !>
    <a> == <if Generate:<Length:<> ; Formula ; Assn_fn 0 ; !> ;>
    <if ;> == Generate:<Length:<> ; Formula:<> ; Assn_fn:<> 1 ; !>
    <if> == Left_branch_result:<>.
```

The key to understanding `Generate` is to see what these statements entail when the query path is of the form

<a^n ; *formula* ; *assn_fn* ;>

The first statement of node `Generate` defines its value at terminal nodes, where the distance list is nil ($n = 0$). In this case,

Generate:<; *formula* ; *assn_fn* ;> = Test:<*formula* ; *assn_fn* ;>

and `Test` returns the assignment function if it is a solution, or the empty list `()` if it is not.

The second statement prefix (`<a>`) is matched when $n > 0$, which are non-terminal nodes in the search tree. Here

Generate:<a a^{n-1} ; *formula* ; *assn_fn* ;>
= Generate:<if Generate:<a^{n-1} ; *formula* ; *assn_fn* 0 ;> ; a^{n-1} ; *formula* ; *assn_fn* ; >

Note that the argument list is carried forward as a default extension, and the matched prefix `<a>` is chopped, thus effectively decrementing the distance by one. It is chopped explicitly in the embedded inheritance (`Length:<>`), and implicitely when appended as a default extension.

The `Generate`

```
Distance_list:
   <$operator> ==  <>
   <[ > == <scan>
   <scan $varletter> == <scan>
   <scan ]> == (a <>)
   <;> == ().
```

`Distance_list` simply scans along an argument outputting an `a` for each `]` encountered. The definition of `Satisfy` is:

```
Satisfy:<> == Generate:<Distance_list:<Arg1 ; !> ; Arg1 ; ; ! >.
```

`Satisfy` takes a single argument and calls `Generate` on a 3-argument list, the distance list, the formula, and the label on the root of the search tree – which is nil. If `Generate` evaluates to either an empty list or a satisfying label on a terminal node, we will have theorems such as

```
Satisfy: <[ 3 ] ~ [ 3 ] ~ [ 2 ] & | ;> = 0 0 0.
```

to indicate that the formula $\neg v_3 \vee (\neg v_3 \wedge \neg v_2)$ is satisfied by the assignment $\{v_3 = 0, v_2 = 0\}$.

Assuming that the intermediate nodes are appropriately defined (the reader is welcome to work through the definitions in Appendix B), we now have a test for boolean satisfiability, and conclude that DATR query evaluation is at least NP-hard – that is, the time complexity of query evaluation is as high as that for any problem in NP.

3 Query evaluation is co-NP-hard

To prove that evaluation is co-NP-hard we need only show that the complement of an NP-hard language can be recognized by a DATR theory. The node `Not_satisfiable` recognizes the complements of the set of satisfiable boolean wffs:

```
Not_Satisfiable: <> == Not:<Satisfiable>.
```

Theorems of this node are the negation of theorems of `Satisfiable`:

```
Not_Satisfiable: <v1 v1 | ;> = false
                 <v1 v1 ~ & ;> = true.
```

We have thus derived two lower bounds, NP-hard and co-NP-hard. The question of whether one is 'worse' than the other (*i.e.*, whether NP $\subseteq$ co-NP) is an open question in complexity theory. While we have established a lower bound, we cannot derive an upper bound on the complexity of DATR query evaluation from further study of this particular problem. If this particular theory could be evaluated in nondeterministic polynomial time, we could only conclude that this problem is in NP, not that an arbitrary DATR theory is necessarily in NP. The fact that an NP-complete problem can be solved in DATR does not guarantee that problems with a higher order complexity could not also be solved in DATR.

We conjecture that further work will show that DATR query evaluation is in fact undecidable, and than DATR theories which recognize any recursively enumerable language can be defined.

References

[Aho *et al.* 1974] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[Evans and Gazdar 1989a] Roger Evans and Gerald Gazdar. Inference in DATR. In *ACL Proceedings, 4th European Conference*, pages 1–9, Manchester, 1989.

[Evans and Gazdar 1989b] Roger Evans and Gerald Gazdar. The semantics of DATR. In Anthony G. Cohn, editor, *Proceedings of the Seventh Conference of the Society for Artificial Intelligence and the Simulation of Behaviour*, pages 79–87, Brighton, UK, 1989.

[Evans and Gazdar 1990]

A he stack evaluator

We begin the node definitions by defining a 'bit logic', following Gazdar (1990) and Moser (1992b):

```
AND: <0 0>  == 0        OR: <0 0>  == 0        NOT: <0>  == 1
     <0 1>  == 0            <0 1>  == 1             <1>  == 0.
     <1 0>  == 0            <1 0>  == 1
     <1 1>  == 1.           <1 1>  == 1.
```

Next we define the stack evaluator. Node `Eval` takes two arguments, a fully instantiated formula, and a stack, initially empty.[1] `Eval:<`*formula* `;` *stack* `;>` evaluates the formula (or 'runs the program') and returns the resulting top of the stack upon termination.

In the definition of `Eval`

also followed by a terminating `;`. The argument list terminator is appended to separate the two-argument list from the default extension. The logical operators also have the default extension added to their query paths, but their behaviour depends only upon the first two atoms' values.

An example theorem is `Eval:<1 1 & 1 0 & & ; ;> = 0`. That is, the following program leaves a `0` on top of the stack:

```
Push 1
Push 1
And
Push 1
Push 0
And
And
```

Some other theorems of `Eval` are:

```
Eval: <1 ; ;> = 1
      <0 ; ;> = 0
      <0 ~ ; ;> = 1
      <1 0 | ; ;> = 1
      <1 1 & 1 0 & & ; ;> = 0
      <1 1 & 0 1 ~ & | ; ;> = 0.
```

B

```
% Eval:<formula ; stack ;> = 0 or 1 (false or true)
%  where
%     formula: is a boolean expression consisting of a sequence of symbols
%              over $terminal. The formula is in Reverse Polish Notation
%              (RPN). To evaluate the expression
%                 (1 & 1) | (0 | ~1)
%              the RPN formula would be
%                  1 1 & 0 1 ~ | |
%     stack:   is a stack with the bottom at right and top at left.
%              It must be initially empty.
%
% Sample theorem:
%      Eval: <1 1 & 0 1 ~ | | ; ;> = 1.
%
% Argument names:
Formula1:<> == Arg1.
Stack1:<> == Arg2.

Eval: <;> == Top:<>                                   % return top of stack.
      <1> == Eval:< Formula1:<> ; 1 Stack1:<> ; ! >   % Push 1 on stack
      <0> == Eval:< Formula1:<> ; 0 Stack1:<> ; !>    % Push 0 on stack
      <&> == Eval:< Formula1:<>                       % Push AND of top 2 items
                    ;
                    AND:< Top:<Stack1:<>>
                          Second:<Stack1:<>> >
                    Pop:< Pop:< Stack1:<> ; > ;  > ; !
                  >
      <|> == Eval:< Formula1:<>                       % Push OR of top 2 items
                    ;
                    OR:< Top:<Stack1:<> ; >
                         Second:<Stack1:<> ; > >
                    Pop:<Pop:<Stack1:<> ;> ; > ; !
                  >
      <~> == Eval:< Formula1:<>                       % Push NOT of top item
                    ;
                    NOT:< Top:<Stack1:<> ;> >
                    Pop:<Stack1:<> ;> ;  !
                  >.

% --- Some theorems ------------
%
% Eval: <1 ; ;> = 1
%       <0 ; ;> = 0
%       <0 ~ ; ;> = 1
%       <1 0 | ; ;> = 1
%       <1 1 & 1 0 & & ; ;> = 0
%       <1 1 & 0 1 ~ & | ; ;> = 1.
```

B.2 satisfy-u.dtr - Find solutions to propositional calculus formulae

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %
% File:            satisfy-u.dtr                                     %
% Purpose:         Boolean satisfiability with an unlimited supply   %
%                  of propositional variables.                       %
% Authors:         Lionel Moser, January 1992.                       %
% Documentation:   HELP *datr                                        %
% Related Files:   lib datr; args.dtr; arglogic.dtr; stackeval.dtr   %
% Version:         1.00                                              %
%     Copyright (c) University of Sussex 1992.  All rights reserved. %
%                                                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This theory recognises membership in boolean formulas in the set of
% satisfiable formulas. Variables are represented as sequences over alphabet
% 0, ..., 9, enclosed in square brackets, and the formula must be in
% postfix notation. e.g.,
% formula (v13 & v12) would be represented by <[ 1 3 ] [ 1 2 ] &>.
% The variables are simply distinct sequences, not arithmetic values.
% This provides the unlimited supply of variables.
%
% Assignment functions do not assign values to operators, only
% variables. Thus (0 0 0) would be an assignment function for
% formula ([ 3 ] ~ [ 2 ] ~ [ 3 ] &). An assignment function is `valid'
% if it assigns each variable uniquely.
%
%  The formulas and assignment functions look like this:
%
%        wff     <[ 1 2 ] [ 3 2 ] & [ 3 4 ] [ 3 2 ] ~ & |>
%        assn_fn <      0       1         0       1>  valid
%        assn_fn <      0       0         0       1>  invalid: [ 3 2 ]
%                                                     is not uniquely mapped
%
% An assignment function satisfies a formula if it is valid and the formula
% under that substitution evaluates to 1, which we check be evaluating it
% on the stack evaluator (stackeval.dtr).

% terminals are the alphabet we use:
%     a          - used for distance list
%     0,1        - boolean values
%     0-9        - alphabet for variable names
%     [ ]        - variable name delimiters
%     & | ~      - logical operators AND, OR, and NOT.

#vars $operator: & | ~.                        % logical operators.
#vars $varletter:  0 1 2 3 4 5 6 7 8 9.        % Td(-)T10.56020Td(alp7Td(1)T10.317d(&)T7(a)T57
```

```
#vars $!: !.
#vars $;: ;.

#load '../ARGS.v5/args.dtr'.        % tools for argument manipulation.
#load '../ARGS.v5/arglogic.dtr'.    % simple logic
#load 'stackeval.dtr'.              % boolean logic stack evaluator

%%%%%% Primitives to deal with propositional variables of the form
%%%%%% [ x1 x2 x3 ... xn ]

% Variable extractor
% Varbl:<[ x1 ... xn ]> == [ x1 ... xn ].
Varbl:<> == '**** ERROR: (Varbl) Invalid symbol.'
      <$varletter> == ($varletter <>)
      <[> == ([ <>)
      <]> == ].

% Extract variable or operator.
% Varbl_or_op returns the first item in the list, whether it's a
% variable or operator.
Varbl_or_op: <> == '**** ERROR: (Varbl_or_op) Invalid symbol.'
   <[> == Varbl
   <$operator>== $operator.

% Like Rest, removes one item, which is either a variable or operator.
% Pop_varbl_or_op:<[ 1 2 3 ] & | ;> = (& |).
Pop_varbl_or_op: <> == '**** ERROR: (Pop_varbl_or_op) unexpected symbol'
   <[> == Pop_varbl
   <$operator> == Arg1:<>.

% Like Rest, where first 'item' is a variable.
% Pop_varbl:<[ 1 2 3 ] & | ;> = (& |).
Pop_varbl: <> == '**** ERROR: (Pop_varbl) unexpected symbol'
   <[> == <>
   <$varletter> == <>
   <]> == Arg1:<>.

% Typeof returns the type of its operand, either var or op.
Typeof: <> == '**** ERROR: (Typeof) Invalid symbol'
   <$operator> == op
   <[> == var.

% Distance_list takes a formula and returns a sequence of a's of
% the same length; a variable has length 1.
Distance_list: <> == '**** ERROR: (Distance_list) Invalid symbol'
   <$operator> ==  <>
   <[ > == <scan>
   <scan $varletter> == <scan>
   <scan ]> == (a <>)
   <;> == ().
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Satisfy:<Formula ;> = () or an assn_fn.
%
Satisfy:<> == Generate:<Distance_list:<Arg1 ; !> ; Arg1 ; ; ! >.

% Some theorems:
%   Satisfy: <[ 1 2 3 4 ] ~ ;> = (0)
%            <[ 1 2 3 4 ] [ 1 2 3 4 ] | ;> = (1 1)
%            <[ 1 2 3 4 5 ] [ 1 2 3 4 ] | ;> = (0 1)
%            <[ 3 ] [ 3 ] | ;> = (1 1)
%            <[ 3 ] [ 3 ] & ;> = (1 1)
%            <[ 3 ] [ 3 ] ~ & ;> = ()
%            <[ 3 ] ~ [ 3 ] ~ [ 2 ] & | ;> = (0 0 0)
%            <[ 3 ] [ 3 ] [ 3 ] & & ;> = (1 1 1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate: generates and tests possible solutions.
%
% Generate:<Length ; Formula ; Assn_fn ;> = some Assn_fn or () if none exists.
%    where
%        Length  -  <Sn Sn-1 ... S0> (Si is any terminal symbol);
%        Formula -  a wff over variables and operators;
%        Assn_fn -  a partial assignment function.
%    satisfying
%        length(Length) + length(Assn_fn) = length(Formula).

% Length is a distance list, used to measure the length (n) of the assignment
% function needed. Any sequence of terminal symbols of length n will do;
% Satisfy passes a copy of the formula itself. Generate recurses to the bottom
% left of the powerset tree, and at leaf nodes it passes the assignment
% function of length n and the formula to Test, which returns the empty list
% "()" if it is not a solution, or returns the assignment function itself if
% it is. If Test returns a non-nil list, then the assignment function
% satisfied the formula, and this is returned.  Otherwise, the right branch is
% descended.  Negative path extension is used to determine what result was
% returned from Test.

% Left branches have label zero (ie, 0 is appended to the assignment function)
% and right branches have label one (ie, 1 is appended to the assignment
% function).
%
% The initial call must provide a nil list as Assn_fn, which will be appended
% to as Generate recurses down to terminal nodes of the search tree.
%
% A simple version of this algorithm is illustrated in file ‘powertest.dtr’.

% Argument names: we name the arguments as this makes it easier to read the
% main node definitions. Argument names must be unique in the theory. The
% integer at the end is arbitrary (but unique).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test:<Formula ; Assn_fn ;> == Assn_fn or ()
%
% Test returns nil if the assignment function is not a solution, and returns
% the assignment function if it is a solution. Test first calls Assn_fn_valid to
% determine whether the assignment function is a valid substitution (ie,
% whether it assigns the same value to each occurrence of each variable). If
% not, it returns a nil list. Otherwise, it instantiates the formula using the
% assignment function (ie, substitutes each variable with the value it is
% assigned by the assignment function), evaluates the instantiated formula on
% the stack evaluator (see 'stackeval.dtr'). If this result is 0 it returns ();
% if it is 1 it returns Assn_fn.

% Test:<Formula ; Assn_fn ;> == Assn_fn or ()

% argument names
FormulaT:<>==Arg1.
Assn_fnT:<>==Arg2.

Test:
   <> == <If:<Assn_fn_valid:<FormulaT ; Assn_fnT ; !>>>
   <then> == <result Eval:< Instantiate:<FormulaT:<> ;
                                         Assn_fnT:<> ; ; !>
                                 ; ; !
                               >
                          ;
                          Assn_fnT:<> ; !>
      <result 0> == ()
      <result 1> == Assn_fnT:<>
   <else> == ().

% Sample theorems:
% (1) Initial call
%     Test:<[ 3 ] [ 4 ] [ 3 ] ~ & | ; 0 1 0 ;> = (0 1 0).
% (2) Assn_fn is valid and satisfies the formula:
%     Test:<then [ 3 ] [ 4 ] [ 3 ] ~ & | ; 0 1 0 ;> = (0 1 0)
%          <result 1 ; 0 1 0 ;> = (0 1 0).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Instantiate:< Formula ; Assn_fn ; Inst_formula ; > == Instantiated_formula
%   where
%      Formula        - is a sequence of variables and operators
%      Assn_fn        - is a sequence of bits (same length as Formula)
%      Inst_formula   - is the part of the Formula which has already
%                       been instantiated.
%
%  Note: Inst_formula must be initially nil.
%
% Instantiate takes a formula (a sequence of variables and operators) and an
% assignment function (a binary string of the same length). It returns the
% instantiation of the formula under the assignment function. Operators, which
% have assignments, are not mapped, since when we evaluate the instantiated
% formula we won't care about the value of any operator. Inst_formula (the
% instantiated formula so far) must be initially nil.
%
% e.g. Instantiate:<[ 3 ] [ 4 ] [ 3 ] ~ & | ; 0 1 0 ; ;>= (0 1 0 ~ & |).

% Instantiate:< Formula ; Assn_fn ; Inst_formula ; > == Instantiated_formula

% Argument names
FormulaI:<> == Arg1.
Assn_fnI:<> == Arg2.
Inst_formulaI:<> == Arg3.

Instantiate: <> == '**** ERROR: (Instantiate) Invalid argument'
   <; ;> == Inst_formulaI    % Formula & Assn_fn exhausted.
   <;> == '**** ERROR: (Instantiate) Assignment function too short'  % NPE

                  % Remove var from Formula and its mapping from Assn_fn,
                  % append the mapping of the var to the assn_fn, and recurse.
   <[>         == Instantiate:< Pop_varbl:<FormulaI ;> ;;>  [illegible]

                                                                    % Instantiate:<
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Assn_fn_valid:<Formula ; Assn_fn ;> =
%    true    -  if Assn_fn is valid;
%    false   -  if Assn_fn is invalid.
%
% An assignment function is valid if it assigns the same value to each
% occurrence of each variable in the formula, operators excepted. Operators are
% ignored. The technique is rather inefficient: Assn_fn is valid if it assigns
% the first symbol correctly and is valid on the rest of the formula. If the
% first symbol is a variable, we test its mapping in the rest of the list,
% remove the first symbol from both the assignment function and the formula,
% and recurse. So every occurrence of a variable is checked against the rest of
% the list.

% Assn_fn_valid:<Formula ; Assn_fn ;>

% argument names
FormA:<> == Arg1.
Assn_fnA:<> == Arg2.

Assn_fn_valid: <> == '**** ERROR: (Assn_fn_valid) Invalid symbol'
   <; ;> == true  % Formula and Assn_fn exhausted
   <;> == true
   <[> == <If:< Var_map_ok:< First:<Assn_fnA ;> ;   % First symbol in
                             FormA ;                % Formula is var;
                             Varbl ;                % check its assn
                             Assn_fnA ; !           % for consistency
                           >    % var is prefixed to default extn so
              > [ >             % that entire formula is preserved.

   <$operator> == Assn_fn_valid:< FormA:<> ;       % ditto for op;
                                  Assn_fnA:<> ; !> % no check.
   <then> == % mapping of first symbol is okay. Check rest.
             Assn_fn_valid:< Pop_varbl_or_op:<FormA:<> ;> ;
                             Rest:<Assn_fnA:<> ;> ; !>
   <else> == false.

% Sample theorems:
% (1) Initial call
%     Assn_fn_valid:<[ 3 ] [ 4 ] [ 3 ] ~ & | ; 0 1 0 ;> = true.
% (2) During recursion
%     Assn_fn_valid:<then [ 4 ] [ 3 ] ~ & | ; 1 0 ;> =  true.
% (3) During recursion
%     Assn_fn_valid:<~ & | ; ;> = true.
% (4) Recursion termination
%     Assn_fn_valid:<; ;> = true.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Var_map_ok: <Val ; Form ; Var ; Assn_fn ;> = true/false
% where
%    Val:     is either 0 or 1
%    Form:    is the formula being tested.
%    Var:     is the name of the variable
%    Assn_fn: is as assignment function, a la Montague, i.e., assn_fn
%             is a model for the formula.
%
% Var_map_ok tests whether all occurrences of variable Var in formula Form
% are assigned the value Val by the assignment function Assn_fn.
% For example,
%   Var_map_ok: <0 ; [ 2 ]  [ 1 ]  [ 2 ]  ; [ 2 ]  ; 0 0 0 ;> = true
% as assignment function (0 0 0) applied to the formula assigns
% every occurrence of variable [ 2 ] the value 0. However,
%   Var_map_ok: <0 ; [ 2 ] [ 1 ] [ 2 ] ; [ 2 ] ; 0 1 1 ;> = false
% because not every occurrence of variable [ 2 ] is assigned value 0
% (the second % occurrence is assigned value 1).
%
```

```
% Var_map_ok: <Val ; Form ; Var ; Assn_fn ;>

% Argument names
ValV:<>  == Arg1.
FormV:<> == Arg2.
VarV:<>  == Arg3.
AssnV:<> == Arg3:<Pop_arg>.  % args.dtr only defines up to 3.

Var_map_ok:
   <0 ; ;> == true   % Formula = nil
   <1 ; ;> == true   % ditto

   % if assn_fn finished so are we.
   <> == <done First:<AssnV> ; Arglist !>
   <done 0 ;> == <more>
   <done 1 ;> == <more>
   <done ;> == true

   % Is first symbol of Formula the Var we are checking?
   <more> == <1 If:< Equal:< Varbl_or_op:<FormV:<> ; > ; VarV:<> ;> > >

   <1 then> == % Yes. Is it assigned the right value?
                 <2 If:< Equal:<ValV:<> ; First:<AssnV:<>> ; > > >
       <2 then> == <reduce var>
       <2 else> == false

   <1 else> == <reduce Typeof:<FormV:<> ;> >
       <reduce op> == Var_map_ok:< ValV:<> ;
                                   Rest:<FormV:<> ; ! > ;
                                   VarV:<> ;
                                   AssnV:<>  ;
                               ! >
       <reduce var> == Var_map_ok:< ValV:<> ;
                                    Pop_varbl:<FormV:<> ; ! > ;
                                    VarV:<> ;
                                    Rest:<AssnV:<> ;> ;
                                    ! >.
% Sample theorems:
%
% Var_map_ok: <Val ; Form ; Var ; Assn_fn ;>
% (1) Initial call
%     Var_map_ok:<0 ; [ 3 ] ~ & | ; [ 3 ] ; 0 ;> = true.
% (2) During recursion
%     Var_map_ok:<1 then 0 ; [ 3 ] ~ & | ; [ 3 ] ; 0 ;> = true.
% (3) during recursion
%      Var_map_ok:<more 0 ; [ 3 ] ~ & | ; [ 3 ] ; 0 ;> = true.
% (5) during recursion
%     Var_map_ok:<done 0 ; 0 ; [ 3 ] ~ & | ; [ 3 ] ; 0 ;> = true.
% (6) during recursion
%     Var_map_ok:<reduce op 0 ; & | ; [ 3 ] ; ;> = true.
```

```
% (4) Recursion termination
%     Var_map_ok:<0 ; ; [ 3 ] ; ;> = true.
```

B.3 Primitives used by `satisfy-u.dtr` and `stackeval.dtr`

To simp6-ckeval.dtr

```
% Pv_to_; returns a path-to-value conversion, stopping at the end of
% the first argument.
Pv_to_;: <;> == ()
         <$terminal> == ($terminal <>).

% Reverse returns the ;-delimited argument reversed, minus the delimiter.
Reverse: <;> == ()
         <$terminal> == (<> $terminal).

% Remove_last retruns first ;-delimited arg minus last symbol.
Remove_last: <> == Reverse:<Rest:<Reverse ;> ;>.

% Equal:<arg1 ; arg2 ;> == true/false
Equal:
   <; ;> == true
   <;> == false
   <> == < If:< Tequal:<First First:<Arg2 ;>>> >
   <then> == Equal:<Rest:<Arg1:<> ;> ; Rest:<Arg2:<> ;> ; !>
   <else> == false.

% Tequal:<atom atom>  == true/false    (terminals equal)
Tequal:
   <$terminal $terminal> == true
   <$terminal> == false.

% If:<condition> == then/else
If: <true> == then
    <false> == else.

Not: <true> == false
     <false> == true.
```