The Sidetracking Meta-Principle

Luís Moniz Pereira, José Júlio Alferes, Carlos Damásio

{lmp,jja,cd}@fct.unl.pt

AI Centre, Uninova and DCS, U. Nova de Lisboa
2825 Monte da Caparica, Portugal

Abstract

The sidetracking principle is nothing but an instance of the well-known principle of procrastination, advising postponement of the problematic until the inevitable has been dealt with, in the hope that the problematic will either be no longer an issue or becomes less problematic. The aim of this paper is to show how the sidetracking principle as a method and technique is applicable to meta-interpreters for a variety of logic programming semantics. To this effect, we present sidetracking for logic programs in the form of an abstract and modular meta-interpreter schema which can be specialized for particular logic programming semantics. Besides, as a programming principle, it can benefit from, and be beneficial to, other declarative concepts and methods.

We illustrate the approach and techniques by applying them to a variety of such semantics. In particular to one of the first implementations of Well Founded and Extended Stable Models Semantics, developed by us. This is, on its own, an important contribution of this paper regarding declarative logic programming implementation techniques.

Introduction and motivation

"In that which I now propose, we will discard the interior points [...], and concentrate our attention upon its outskirts. Not the least unusual error in investigations such as these is the limiting of enquiry to the immediate, with total disregard of the collateral or circumstancial events." [Poe, Edgar Allan 1908]

The sidetracking principle is nothing but an instance of the well-known principle of procrastination, advising postponement of the problematic until the inevitable has been dealt with, in the hope that the problematic will either be no longer an issue or becomes less problematic.

A sidetracking strategy for logic programs was first detailed in [Pereira and Porto 79a] and implemented in Prolog in [Pereira and Porto 79b]. In [Kowalski 79] (p.93), the equivalent "principle of eager consideration of subgoals" was formulated.

Sidetracking is an appropriately evocative designation in the logic programming context. Instead of pushing ahead at a non-deterministic choice point in the present goal/subgoal track of a derivation, one sidetracks to a lateral goal in the resolvant, namely a deterministic one, i.e. having at most one clause matching it.

The aim of this paper is to show how the sidetracking principle as a method and technique is applicable to meta-interpreters for a variety of logic programming semantics. Besides, as a programming principle, it can benefit from, and be beneficial to, other declarative concepts and methods.

To this effect, we present sidetracking for logic programs in the form of an abstract and modular meta-interpreter schema which can be specialized for particular logic programming semantics. We illustrate the approach and techniques by applying them to a variety of such semantics. In particular to one of the first implementations of Well Founded and Extended Stable Models Semantics. This is, on its own, an important contribution of this paper regarding declarative logic programming implementation techniques. The resulting meta-interpreters were successfully tested.

Sidetracking has been used in the Andorra parallel logic programming language [Janson and Haridi 91], embedded at the C implementation level, under the name of "Andorra Principle".

The structure of this paper is as follows. In section 1 we describe sidetracking for pure Prolog as it was defined in [Pereira and Porto 79a], and produce a meta-interpreter. Next we proceed to abstract and apply this sidetracking meta-interpreter to other logic programming semantics. In particular, in section 3, we present an implementation of Well Founded and Extended Stable Models Semantics. Some final considerations close the paper.

1. Sidetracking - the principle applied to logic programming and a simple implementation

In this section we describe the sidetracking strategy, first detailed in [Pereira and Porto 79a] and implemented in Prolog in [Pereira and Porto 79b], and present an interpreter (for pure Prolog) based on the one in [Pereira 84].

Sidetracking is defined as a strategy for the execution of pure Prolog programs which is essentially dynamic, since it relies upon runtime evaluation of the deterministic or non-deterministic character of the goal invocation awaiting activation.

A goal call is said to be non-deterministic, at some stage of the derivation, if at that stage it matches more than one clause head, and deterministic otherwise, i.e. it unifies with at most one head.

An execution step is the replacement of a single goal in the conjunction of goals (or resolvant) that represents a derivation stage, by the conjunction of goals which constitutes the body of a clause activated by matching the goal. This activation and matching may cause instantiation of variables of the goal also present in other pending goals in the resolvant, possibly influencing the matching possibilities of those goals, and consequently their character with respect to determinism.

The main idea behind sidetracking is to postpone, at any stage, the replacement of non-deterministic goals until all pending deterministic ones have been replaced. This is an instance of the principle of procrastination, advising postponement of the problematic until the inevitable has been dealt with, in the hope that the problematic will either be no longer an issue or becomes less problematic. Alternatively, the sidetracking meta-principle can be stated as "tackle the inevitable first; then consider options". This translates into "inevitability-first" search strategies for logic programming.

The main advantages of this (pointed out in [Pereira and Porto 79a]) are:

- First, doing all replacements of deterministic goals first, before activating non-deterministic ones, avoids the failure and reactivation of such deterministic goals when backtracking now takes place only among the non-deterministic goals that follow.
- Second, if a failing deterministic replacement path exists then we are sure to spot that failure before any (useless) choice for non-deterministic goals is ever made.

1.1. The sidetracking procedure

Starting with the goal statement, and throughout the execution, we assume we have always a conjunction of goals (the resolvant) one of whose goals is considered the current goal (at the start it will be the leftmost goal in the conjunction).

An execution step is performed as follows:

(1) If the current goal call is non-deterministic then the next goal in the resolvant becomes the current goal; otherwise, it is replaced in the resolvant by the body of its matching clause if there is one, and then the next goal in the resolvant becomes the current goal. If there is no matching clause for a deterministic goal activation, we are in presence of a failure, and backtracking must ensue, to a goal activation and replacement that was non-deterministic (whose occurrence is described next). Occurrences of the special goal **true** are eliminated from the conjunction as they are found.

(2) When the end of the resolvant is reached two possibilities arise. Either at least one deterministic goal was found, and in this case the whole resolvant is searched again for more deterministic goals; or no deterministic goal was found, and in this case the current goal is then replaced by the body of one of its matching clauses, and execution restarts at (1).

(3) A solution to the problem is found if the resolvant becomes empty. o

1.2. The chocolate meta-interpreter

In order to present an interpreter for pure Prolog programs using sidetracking, we begin by rewriting the vanilla meta-interpreter using the replacement method. We call this interpreter the chocolate meta-interpreter.

The main predicate of this interpreter is `C#RC`, meaning that the resolvant conjunction `C` can be replaced by the resolvant conjunction `RC`. The code for it is[1]:

```
true#true :- !.
(A,B)#NC :- !, A#NA, B#NB, conjunction(NA,NB,NC).
G#NG :- clause(G,BG), BG#NG.
```

where `conjunction(NA,NB,NC)` is a predicate that produces the conjunction of `NA` and `NB` as the new resolvant, eliminating any excessive occurrences of `true`.

Remark: the above cuts can be eliminated, but with a loss of efficiency.

```
conjunction(true,A,A) :- !.
conjunction(A,true,A) :- !.
conjunction(A,B,(A,B)).
```

Remark: the above cuts can be eliminated with recourse to system predicate `\=/2`.

Finally, a top goal is solved if it can be replaced by true, i.e.: o

1.3. The sidetracking meta-interpreter

It is now easy to write the chocolate interpreter with sidetracking strategy. The inner loop of searching for deterministic goals is similar to the chocolate meta-interpreter, except that it only replaces the deterministic goals, and leaves the non-deterministic ones as they were.

```
true#true :- !.
(A,B)#NC :- !, A#NA, B#NB, conjunction(NA,NB,NC).
G#NG :- deterministic(G), !, clause(G,BG), BG#NG.
G#G.
```

The cut in the third clause ensures that no useless backtracking is made into deterministic goals[2].

[1]Note the one-to-one correspondence between the clauses of this interpreter and of the vanilla one.

[2] If `(A#NA, B#NB)` in the second clause is replaced by `(side(A,NA), side(B,NB)),` the search for deterministic goals is depth first, instead of breadth first, as above. This is more efficient because normally the influence of variable bindings on determinism is greater from left to right than vice-versa; also, less rewriting in general is performed on goal conjunctions, since the resolvant is divided into smaller subresolvants.

In order to ensure that if deterministic goals are found then the whole resolvant is searched again for more deterministic goals, we introduce the predicate `side(C,ND)`, where `C` is a conjunction of goals, and `ND` is the conjunction of non-deterministic goals resulting from all replacements of deterministic goals in `C`.

```
% cut included for efficiency
side(true,true) :- !.
side(C,ND) :- C#NC, (C == NC, ND = NC; C \== NC, side(NC,ND).
```

Note that the conjunction `C` is replaced by itself iff no deterministic goal replacement takes place during the search for them in `C`[3].

Now a top level is needed for dealing with non-deterministic replacements.

```
demo(true).
demo(G) :- G \= true, side(G,ND), replace_one_nd(ND,NG), demo(NG).

replace_one_nd(true,true) :- !.
replace_one_nd((A,B),(RA,B)) :- !, replace_one_nd(A,RA).
replace_one_nd(A,RA) :- clause(A,RA).
```

The only predicate missing to complete the interpreter is `deterministic(Goal)`, which determines whether a goal call is deterministic. One possible away of implementing this in Prolog is[4]:

```
% if there is at least one matching clause make a note of it
deterministic(G) :- one(clause(G,B)), assert(one_clause(B)), fail.

% if there is another one the call is non-deterministic
deterministic(G) :- retract(one_clause(B)), clause(G,BB), B \= BB, !, fail.

% otherwise; one_clause/1 will have been abolished in any case
deterministic(G).

% one solution is enough
one(G) :- G, !.
```

2. Sidetracking applied to other semantics for logic programs

In this section we generalize the vanilla meta-interpreter, and make the corresponding generalizations to the sidetracking interpreter, in order to compute with the sidetracking strategy semantics other than that of pure Prolog, i.e. we will apply sidetracking to Prolog meta-interpreters, for other logic program semantics. This generalization is made in such a way that the code for sidetracking is completely independent from the semantics it implements. The particular semantics is specified by Prolog predicates called from the sidetracker. This has the advantages of modularity of code, and that the code responsible for the sidetracking strategy does not need rewriting when the semantics is changed.

We end up by describing a schema for applying the sidetracking strategy to interpreters, general enough for implementing many well-known semantics. At the end of this section we exemplify how the schema is specified to compute some of them. In the next section we obtain, based on this schema, a sidetracking interpreters for the well founded semantic [Van

[3]Remarks: (1) For efficiency `C \== NC` can be omitted if a cut is inserted at the end of the first disjunct. (2) Another option (cf. [Pereira and Porto 79b]) is to couple with each resolvant pass a new control variable, which is bound whenever a deterministic goal call is found; the == test is then replaced by a `var` test.

[4]Remarks: (1) Since the interpreter only rewrites goals, it does not affect the semantics of programs; in fact, its clauses can be regarded as logical theorems. (2) For efficiency, memoizing techniques could be used for `deterministic/1`.

Gelder et al. 1990] and the stationary (or extended stable models) semantics [Przymusinska and Przymusinski 1990; Przymusinski 91] by providing their specific semantic predicates which specialize the sidetracking schema.

2.1. Generalizing the sidetracking interpreter with abstraction and pruning

The first generalization we make is in what regards the replacement of a single goal. In fact, there is no need to require that the replacement of a single goal be made by the body of a definite clause for it. Consider, for example, that clauses are of the form `clause(`A_1 / … / A_n`)`, where the A_is are classical literals. In a classical theorem prover, a goal `G` for which there is a clause statement of the form `clause(`A_1 / … / ¬G / … / A_n`)`[5] can be replaced by A_1 / … / A_n. Another example is well founded semantics, where the rule for replacing negative literals is more elaborate (c.f. section 3).

Since nothing in the interpreters, both the vanilla and the sidetracker, depends on the form of clauses, the generalization of clause form is achieved simply by substituting every goal of the form `clause(A,B)` by `rule(A,B)`, where `rule(A,B)` is a predicate that depends on the semantics to be implemented. Note the definition of `deterministic(G)` now must refer to `rule` instead of `clause`.

Another useful generalization is the notion of context of a goal. For example in SL-Resolution any goal can cancel with a clause for it or with complement ancestor. Here, its ancestors can be viewed as context for the goal.

In order to keep this notion more general, the sidetracking schema should not impose anything about the data structure of the context. Thus two predicates must be defined when implementing any particular semantics context: `initial_Cx(I)`, which determines the structure of context and its initial element; and `add_to_Cx(G,Cx,NewCx)`, which determines how a goal `G` is added to a context `Cx`, giving the new context `NewCx` as a result .

Changing the vanilla meta-interpreter in order to define a particular notion of context is quite straightforward. We just add to predicate `demo` an extra argument that accumulates the context, i.e.:

```
demo(true,_) :- !.
demo((A,B),Cx) :- !, demo(A,Cx), demo(B,Cx).
demo(G,Cx) :- rule(G,RG), add_to_Cx(G,Cx,NewCx), demo(RG,NewCx).
```

and a top level that introduces the initial context:

```
demo(G) :- initial_Cx(In), demo(G,In).
```

For the sidetracking schema we must redefine the structure of the resolvant to be, instead of a conjunction of goals, a conjunction of pairs goal/context: `gc(Goal,Cx)`. Modification of the schema in order to add the ancestor to the context every time a goal is replaced is rendered by:

```
demo(G) :- initial_Cx(In), demo(gc(G,In)).
demo(gc(true,_)) :- !.
demo(G) :- side(G,ND), replace_one_nd(ND,NG), demo(NG).

replace_one_nd(gc(true,Cx),gc(true,Cx)).
replace_one_nd((A,B),(RA,B)) :- replace_one_nd(A,RA).
replace_one_nd(gc(A,Cx),RA) :-
      rule(A,NA), add_to_Cx(A,Cx,NewCx), insert_Cx(NA,NewCx,RA).

side(gc(true,Cx),gc(true,Cx)) :- !.
side(C,ND) :- C#NC, (C == NC, ND = NC; C \== NC, side(NC,ND).
```

[5]Where ¬G is the logically complementary literal of G.

```
gc(true,Cx)#gc(true,Cx).
(A,B)#NC :- A #NA, B#NB, conjunction(NA,NB,NC).
gc(G,Cx)#NG :- deterministic(G), rule(G,BG), add_to_Cx(G,Cx,NewCX),
               insert_Cx(BG,NewCx,RG),!, RG#NG.
R#R.
```

with the appropriate changes to `conjunction`, and where `insert_Cx(C,Cx,CCx)` is a predicate that builds `CCx` as the conjunction of all `gc(Goal,Cx)`, for every `Goal` in conjunction `C`.

Of course, contextual information like this does not change anything in the interpreters, because, up to now, contexts are never used. In order to make use of this contextual information we introduce a context sensitive pruning device. For example, for the SL-Resolution rule of cancelling a goal with its complement ancestor, pruning is used to solve the goal. Also, pruning combined with contextual information about ancestors is used for semantics with loop detection, but this time to fail a goal subsumed by an ancestor.

To accomplish this generalization, we introduce in the schema calls to the predicate `pruning(G,Cx,Control)`, defined according to the semantics being implemented, where `Control` can be either `true` or `fail` depending on the pruning result we want. This is done simply by introducing, just before the clause for replacing a single goal, a clause for pruning it, i.e.:

```
gc(G,Cx)#gc(true,Cx) :- pruning(G,Cx,Control), !, Control.
```

before the clause `gc(G,Cx)#NG :- deterministic(G), … .`

Some interpreters also make use of global information, i.e. information that is common to the whole resolvant. An example is the WFM interpreter described in section 3. This global information can also be used for extracting information about the whole derivation, such as the AND derivation tree, the number of total goal replacements, etc.

In order to provide this generalization in the sidetracking schema, we introduce two arguments to predicates, one being the global information before activating the goal, and the other being the global information after replacing the goal.

We also pass this information to the pruning device. This is needed for example in the WFM interpreter.

To keep the schema independent of the data structure coding this information, like we did for contexts, we introduce two predicates that must be defined when implementing any particular semantics: `initial_Int(I)`, which determines the structure of global information and its initial element; and `add_to_Int(G,Int,NewInt)`, which determines how replacing a goal `G` changes the global information `Int` giving as a result the new global information `NewInt`.

The complete schema is listed below.

2.2. The sidetracking schema with goal contexts and global information

```
demo(G) :- initial_Cx(Cx), initial_Int(Ii), demo(gc(G,In),Ii,Io).
demo(gc(true,_),Ii,Io) :- !, add_to_Int(true,Ii,Io).
demo(G,Ii,Io) :-
      side(G,ND,Ii,I), replace_one_nd(ND,NG,I,II), demo(NG,II,Io).
replace_one_nd(gc(true,Cx),gc(true,Cx),Ii,Io) :- !, add_to_Int(true,Ii,Io).
replace_one_nd((A,B),(RA,B),Ii,Io) :- replace_one_nd(A,RA,Ii,Io).
replace_one_nd(gc(A,Cx),RA,Ii,Io) :-
      rule(A,NA), add_to_Cx(A,Cx,NewCx), insert_Cx(NA,NewCx,RA),
      add_to_Int(G,Ii,Io).

side(gc(true,Cx),gc(true,Cx),Ii,Io) :- !, add_to_Int(true,Ii,Io).
side(C,ND,Ii,Io) :-
      r(C,Ii)#r(NC,I), (C == NC, ND = NC; C \== NC, side(NC,ND,I,Io).
r(gc(true,Cx),Ii)#r(gc(true,Cx),Io) :- !, add_to_Int(true,Ii,Io).
r((A,B),Ii)#r(NC,Io) :- !,
      r(A,Ii)#r(NA,I), r(B,I)#r(NB,Io), conjunction(NA,NB,NC).
r(gc(G,Cx),Ii)#r(gc(true,Cx),Io) :-
      pruning(G,Ii,Cx,Control), !, Control, add_to_Int(true,Ii,Io).
r(gc(G,Cx),Ii)#r(NG,Io) :-
      deterministic(G), rule(G,BG), add_to_Cx(G,Cx,NewCX),
      insert_Cx(BG,NewCx,RG), add_to_Int(G,Ii,I), !, r(RG,I)#r(NG,Io).
GCI#GCI. o
```

The rest of this section will be devoted to exhibiting some example of completion of this schema in order to comply with some particular semantics. Note that when specification predicates are added to the sidetracking schema we obtain an interpreter for the specified semantics that uses sidetracking as a strategy.

Example 2.3: Pure prolog with loop detection

For this interpreter clauses are represented as usual, `clause(Head,Body)`, and the rule of replacement is also the usual one. So:

```
rule(G,B) :- clause(G,B).
```

The contextual information needed is the list of ancestors.

```
initial_Cx([]).
add_to_Cx(G,Cx,[G|Cx]).
```

The rule for pruning is that the derivation should fail if an ancestor subsumes the goal:

```
pruning(G,_,Cx,fail) :- subsumes(Cx,G)
```

where `subsumes(Cx,G)` enacts some subsumption algorithm.

There is no need for global information for it is irrelevant. Thus:

`initial_Int(_).` and `add_to_Int(_,_,_).` o

Example 2.4: Pure Prolog with loop detection that counts the number of replacements

Clauses for `rule/2`,`initial_Cx/1`,`add_to_Cx/3` and `pruning/4` are the same as in the last example. Now global information is needed.

```
initial_Int(0).
add_to_Int(true,I,I).
add_to_Int(G,Ii,Io) :- G \= true, Io is Ii + 1.o
```

Example 2.5: Classical Logic in propositional SL-Resolution

Here clauses are represented as `clause(L)`, where `L` is a list of literals, and stands for the disjunction of its elements. A literal is an atom A or its negation ¬A. As the schema is built with reference to clauses with a head and a body, predicate `rule` must function as if all contrapositives variants of a clause were present. Thus:

```
rule(G,B) :- clause(L), variant_clause(G,L,B).
```

where `variant_clause` produces a definite clause variant with `G` as the head.

```
variant_clause(G,[G|B],CB) :- compl_list(B,CB).
variant_clause(G,[A|B],(CA,NB)) :- compl(A,CA), variant_clause(G,B,NB).

compl_list([],[]).
compl_list([G|T],(CG,CT)) :- compl(G,CG), compl_list(T,CT).

compl(¬A,A) :- !.
compl(A,¬A).
```

Context is the ancestors list. Thus its definition is the same as above.

Pruning is necessary for loop detection:

```
pruning(G,_,Cx,fail) :- subsumes(Cx,G).
```

and for ancestor resolution:
`pruning(G,_,Cx,true) :- compl(G,CG), member(CG,Cx).`[6]

There is no need for global information. Thus predicates concerning it are as in example 2.3. o

3. WFS and XSM Interpreter

Using the meta-interpreter previously presented, we describe in this section one of the first interpreters implementing an inovative decision procedure for the Well Founded and the Extended Stable Model (or Stationary, [Przymusinski 91]) Semantics. This is, on its own, an important contribution of this paper regarding declarative logic programming implementation techniques. Well Founded Semantics (WFS) [Van Gelder et al. 90] and the Extended Stable Model Semantics (XSMS), introduced by [Przymusinska and Przymusinski 90], are 3-valued semantics defined for the class of all normal logic programs. We'll use the derivation procedure rules set forth in [Pereira et al. 91a] to write this new interpreter, and to do so we must first review and introduce some definitions. These procedures compute whether some literal belongs to the Well Founded Model (WFM) or to some Extended Stable Model (XSM) of a general logic program.

Definition 3.1 *(general logic program)*

A general logic program [Lloyd 87] Π is a finite set of clauses of the form H ♦ L1,...,Ln where n _ 0, H is an atom and each Li is a literal. A unit clause, H ♦, stands for H ♦ **true**, with **true** an atom satisfied in all models of Π. A negative literal is syntactical represented by not A, where A is an atom.o

Definition 3.2 *(ground program)*

The set of all ground instances of the clauses of a logic program Π, with respect to its Herbrand Universe, is denoted by ground(Π).o

Definition 3.3 *(ground literals of a program)*

The set of ground literals in program Π, Lit(Π), is the set of literals in ground(Π).o

We need Lit(Π) and ground(Π) because both of the ensuing top-down derivation procedures are only defined, without loss of generality, for propositional rules.

Definition 3.4 *(interpretations)*

A positive (resp. negative) signed (partial) interpretation I of Π is a set of positive (resp. negative) signed literals from ground(Π).

Definition 3.5 *(context)*

A context C is an ordered set of positive or negative interpretations $S_1S_2...S_n$. The length of context C, i.e., the number of interpretations in C, is denoted by |C|. The interpretation S_i of context C, with $0 \leq i \leq |C|$, is denoted by ${}^{i}C$. We say that L □ ΔC iff L □ ${}^{|C|}C$. o

Definition 3.6 *(adding literal to context)*
Consider the context C, $S_1S_2...S_n$, and a literal L. The addition of a literal to a context, C+L, is defined by: $C+L = S_1S_2...S_n\{L\}$, iff G and S_n have different signs;
$C+L = S_1S_2...(S_n \approx \{L\})$, otherwise. o

[6] For non-propositional SL-resolution this isn't complete since there would be the need to backtrack into `member/2`. The search for an ancestor would be tackled in `replace_one_nd`/4.

Definition 3.7 *(C-formula)*

A contextual formula (C-formula) is a pair C|F, where C is a context and F is a C-expression, i.e. resolvant, inductively defined as follows:

- An atom is a C-expression.
- If E is a C-expression then not(E)[7] is also a C-expression.
- If E1 and E2 are C-expressions then (E1,E2) is a C-expression.
- nothing else is a C-expression.

An empty C-formula is the C-formula C|**true**. o

Now, we can define the already mentioned top-down derivation procedures. We begin with the simpler one, the WFM-derivation, the derivation procedure for computing whether a literal belongs to the WFM of a program.

Definition 3.8 *(C-derivation)*

Given a logic program Π, let R_j = <C_j|F_j;I_j> where C_j|F_j is a C-formula and I_j a set of literals in Lit(Π). A C-derivation is a sequence from <C_i|F_i;I_i> to <C_n|F_n;I_n> such that for any <C_k|F_k;I_k>, $i \leq k \leq n$, some derivation rules apply. o

Definition 3.9 *(WFM-derivation)*

There is a WFM-derivation for G in Π iff there is a C-derivation using the rules below from <ϕ|G;ϕ> to <C|**true**;I> for some C and I; the literals in I also belong to the WFM[8]. When applying the following rules we assume $C_{k+1} = C_k$ and $I_{k+1} = I_k$, unless stated otherwise.

Rules for negative C-expressions:

D1.1 If F_k=not G, and there is no rule G ♦ B in ground(Π) then
R_{k+1} = <C_k+F_k|**true**;I_k ≈ {F_k}>.

D1.2 If F_k=not G, not G □ ΔC_k, and G □ I_k, then F_{k+1} = **true**.

D1.3 If F_k=not (not G) then R_{k+1} = <C_k|G;I_k>.

D2.1 If F_k=not G, not G □ ΔC_k, G □ I_k, and there are r rules in ground(Π)
G_1 ♦ B_{11}, ..., B_{m1}
...
G_r ♦ B_{1r}, ..., $B_{m'r}$
with G_i as head, $1 \leq i \leq r$, then
R_{k+1} = <C_k+F_k|~G_1,...,~G_r;I_k ≈ {F_k}>
where each ~G_i is shorthand for not(B_{1i}, ..., B_{mi}).

D2.2 If F_k = not(G_1,...,G_m), then R_{k+1} = <C_k|not G_i;I_k> for some $1 \leq i \leq m$.

Rules for non negative C-expressions:

D3.1 If F_k = G, where G is an atom, G □ ΔC_k, and not G □ I_k, then for some rule G ♦ B_1,...,B_m □ ground(Π), R_{k+1} = <C_k+F_k#(B_1,...,B_m);I_k ≈ {F_k}>.

D4.1 F_k = (g,G) then R_{k+1} = <C_k|G;I_{gk} >
if there is a derivation from <C_k|g;I_k > to <C|G;I_{gk} >. o

In [Pereira et al., 1991a] it is proved that G □ WFM(Π) iff there is a WFM-derivation for G in Π, that is, showing the soundness and completeness of this derivation procedure.

[7] We will discard the brackets where it is not ambiguous to do so.

[8] In fact any such I is a support set for C, as defined in [Pereira et al. 91b].

The other derivation procedure, for XSMS, reported in the same paper, succeeds iff a literal G is in some XSM of Π.

Definition 3.10 *(XSM-derivation)*

There is a XSM-derivation for G in Π iff there is a sequence from <ϕ|G;ϕ> to <C|true;I> for some C and I. The rules to generate this sequence are obtained from the WFM-derivation rules by adding a new rule, D3.2, and changing rule D1.2 to D'1.2.

D'1.2 If F_k=not G, G □ I_k and for some i, F_k □ ${}^iC_k$, $1 \le i \le |C_k|$ then F_{k+1} = **true**.
D3.2 If F_k = G, where G is an atom, not G □ I_k and for some i, G □ ${}^iC_k$, $1 \le i < |C_k|$ then F_{k+1} = **true**. o

We are now in a position to define our meta-interpreters for WFM and XSM with embedded sidetracking. We need only to specify the six predicates `initial_Cx`, `add_to_Cx`, `initial_Int`, `add_to_Int`, `pruning` and `rule`.

As the reader may have noticed, there is a direct correspondence between the context C_j, and interpretation I_j, appearing in the definition of R_j for a WFM-derivation and a XSM-derivation, and the Cx and I arguments of the sidetracking meta-interpreter. The data structure we use to represent a context is a list of lists. An interpretation I_j is simply a list. Their initial values are defined by the clauses:

```
initial_Cx( [[]] ).
initial_Int([]).
```

The predicate `add_to_Cx` performs, as expected, the addition of a goal to a context. It is a direct translation of definition 3.6 into Prolog:

```
add_to_Cx(G,[[]],[[G]]) :- !.
add_to_Cx(G,[[L|RI]|Others], [[G,L|RI]|Others]) :- same_sign(G, [L|RI]), !.
add_to_Cx(G,Cx, [[G]|Cx] ).

% same_sign decides whether a literal has the same sign of an
% interpretation of a context
same_sign(not _,[not _|_]) :- !.
same_sign(G,[C|_]) :- not functor(G, not, 1), not functor(C, not, 1).
```

The addition of a goal G to an interpretation I is just the fronting of G to I:

```
add_to_Int( G, I, [G|I] ).
```

The sidetracking schema meta-interpreter defines `rule` as a goal rewriting predicate. This rewriting operation is performed, once some preconditions are verified, in rules D1.1, D1.3, D2.2, D3.1 and D4.1 of both derivation procedures. The implementation of the required rewriting is:

```
% D1.1, D1.3, D2.1 and D2.2
rule(not G,B) :- !, clause_neg(G,B).
rule(G,B) :- clause(G,B).                % D3.1

% D1.1, D1.3, D2.1 and D2.2
clause_neg(G,Body) :- findall(B, clause(G,B) , L), one_from_each(L,Body).

% i.e. constructs all possible "bodies" with a literal from each
% clause
one_from_each([B],CSB) :- !, member_conj(SB,B), compl(SB,CSB).
one_from_each([B|T],(CSB,ST)) :-
    !, member_conj(SB,B), compl(SB,CSB), one_from_each(T,ST).
one_from_each([],true).                  % D1.1

% Selects, in turn, each literal from a clause
member_conj(A,(A,_)).
member_conj(A,(_,B)) :- !, member_conj(A,B).
member_conj(A,A).
```

```
% Rule D1.3
compl(not G,G) :- !.
compl(G,not G).
```

Notes: The process of expanding a negative C-expression (predicate `clause_neg`) is guided by rules D2.1 and D2.2. The rewriting of a non-negative C-expression by the second clause of rule above is obvious and corresponds to the application of rule D3.1. The expansion of a conjunction of C-expressions, rule D4.1, is already contemplated in the sidetracker.

After the definition of the rewriting rules we must now ensure that their preconditions are satisfied, this and the implementation of the remaining rules is achieved with the `pruning` predicate.

```
% Consistency check in rules D1.2 (or D'1.2), D2.1, D3.1 and D3.2
pruning(G,I,_,fail) :- compl(G,CG), member(CG,I).

% Negative goal with fact fails, just for efficiency
pruning(not G,_,_,fail) :- clause(G,true).

% Rule D1.2; usage of the double not saves space...
pruning(not G, _, [LCx|_], true) :-
      not not ( same_sign(not G,LCx), member(not G,LCx) ).

% Loop detection ensuring not G □ΔCk in rule D2.1, and G □ ΔCk in
% D3.1 for the WFM-interpreter.
pruning(G,_,Cx,fail) :-
    flag(wfm), in_other_Cx(G,Cx).   % flag tests whether wfm case

% Loop detection on positive goals: test G □ ΔCk, rule D3.2 for the
% XSM-interpreter.
pruning(G, _, [Last|_], fail) :-
      flag(xsm), same_sign(G,Last), member(G,Last).   % xsm case

% rules D'1.2 and and D3.2 for XSM-interpreter.
pruning(G,_,Cx,true) :-
    flag(xsm), compl(G,CG), in_other_Cx(G,Cx).        % xsm case

% test of membership of goal G in a context
in_other_Cx(G,[[G|_]|_]).
in_other_Cx(G,[[H|T]|Cx]):- not same_sign(G,[H|T]), !, in_other_Cx(G,Cx).
in_other_Cx(G,[[_|S]|Cx]):- !, in_other_Cx(G,[S|Cx]).
in_other_Cx(G,[[]|Cx]) :- in_other_Cx(G,Cx).
```

4. Final considerations

Next we raise some points of interest regarding the sidetracking meta-principle.

• One possible generalization of the sidetracking principle is in what regards the definition of determinism. Above, we said that a deterministic goal is one that activates at most one clause. Activation of a clause, however, should not be construed simply as the successful matching of the goal with the clause head. Additional conditions may be imposed. The motivation for doing so is that some tests on arguments of a goal cannot be performed by unification alone, but may be carried out within the body of the clause, up to some desired point (as in the notion of guard). Consequently, a goal may match several clause heads and yet be deterministic in the sense that on closer inspection, up to their guards, only one clause may execute the goal. This generalization is also mentioned in the context of the Andorra language [Janson and Haridi 91].

• The sidetracking strategy avoids useless backtracking, thus avoiding in many cases the need for intelligent backtracking. However, in the general case, intelligent backtracking can still improve sidetracking execution. An example of this can be found in [Pereira and Porto 79a].

• Sidetracking can be viewed as a type of priority control strategy, where priority is given to deterministic goals. Of course, priority can be generalized for conditions other than determinism.

• For languages with "negation as failure", it is appropriate to consider deterministic negative goal activations, `not G`, whenever there are no clauses for `G`. This can be achieved via predicate `rule/2`, as illustrated for the WFS-interpreter. Furthermore, if delaying of non-ground negative goals is desired, care must be taken to do so by postponing them as long as possible (except in the foregoing case), during the non-deterministic goal choice step, and by never considering them as prioritary.

An implementation for priority could be:
```
priority(not G) :- not rule(G,_).
priority(not G) :- at_least_one_var(G), !, fail.
priority(G) :- deterministic(G).
```

• A sidetracking meta-interpreter can be called from inside a program only on those goals for which sidetracking is desired.

Acknowledgements

We thank ESPRIT BRA COMPULOG (no. 3012), Instituto Nacional de Investigação Científica, Junta Nacional de Investigação Científica e Tecnológica, and Gabinete de Filosofia do Conhecimento for their support.

References

[Janson and Haridi 91] Janson S., S. Haridi (1991). Programming Paradigms of the Andorra Kernel Language. Proceedings of the International Logic Programming Symposium'91, San Diego USA, Saraswat and Ueda (eds.), MIT Press.

[Kowalski 71] Kowalski, R. (1971). Linear Resolution with Selection Function, Artificial Intelligence, vol. 2, pp. 227-260.

[Kowalski 79] Kowalski, R. (1979). Logic for Problem Solving, North Holland.

[Lloyd 87] Lloyd, J. (1987). Foundations of Logic Programming, 2nd edition, Springer-Verlag.

[Pereira and Porto 79a] Pereira L.M. and A. Porto. (1979). Intelligent backtracking and sidetracking in Horn clause programs - the theory. Technical report 2/79 CIUNL, Departamento de Informática, U. Nova de Lisboa.

[Pereira and Porto 79b] Pereira L.M. and A. Porto. (1979). Intelligent backtracking and sidetracking in Horn clause programs - the implementation. Technical report 3/79 CIUNL, Departamento de Informática, U. Nova de Lisboa.

[Pereira 84] Pereira, L.M. (1984). Logic Control with Logic. In "Implementations of Prolog", J.Campbell (ed.), Ellis Horwood.

[Pereira et al. 1991a] Pereira, L. M., J. N. Aparício and J. J. Alferes. (1991). Derivation Procedures for Extended Stable Models. Proceedings of the International Joint Conference on A.I., Morgan Kaufmann.

[Pereira et al. 1991b] Pereira, L. M., J. J. Alferes and J. N. Aparício. (1991). Contradiction Removal within Well Founded Semantics. Proceedings of the 1st Int. Workshop on Logic Programming and Nonmonotonic Reasoning, Nerode, Marek and Subrahmanian (eds.), MIT Press.

[Poe, Edgar Allan 1908] Poe, E. A. (1908). The mystery of Marie Roget - a sequel to "The murders in the Rue Morgue". In "Tales of mystery and imagination", p.435. Everyman's Library, Dent, London 1971.

[Przymusinska and Przymusinski 1990] Przymusinska, H. and T. Przymusinski. (1990). Semantic Issues in Deductive Databases and Logic Programs. In "Formal Techniques in Artificial Intelligence", pp. 321-367, R.Banerji (ed.), North Holland.

[Przymusinski 91] Przymusinski T. (1991). The Stationary semantics for logic programs. Workshop on Deductive Databases, ILPS'91, San Diego.

[Van Gelder et al. 1990] Van Gelder, A., K. A. Ross and J. S. Schlipf. (1990). The Well-Founded Semantics for General Logic Programs. J.ACM.