Chapter 5

Using Tabling in XSB: A Tutorial Introduction

XSB has two ways of evaluating predicates. The default is to use Prolog-style evaluation, but by using various declarations a programmer can use also tabled resolution which allows for a different, more declarative programming style than Prolog. In this section we discuss the various aspects of tabling and how it is implemented in XSB. Our aim in this section is to provide a user with enough information to be able to program productively in XSB. It is best to read this tutorial with a copy of XSB handy, since much of the information is presented through a series of exercises.

For the theoretically inclined, XSB uses SLG resolution which can compute queries to non-floundering normal programs under the well-founded semantics [44], and is guaranteed to terminate when these programs have the *bounded term-depth property.* This tutorial covers only enough of the theory of tabling to explain how to program in XSB. For those interested, the web site contain papers covering in detail various aspects of tabling (often through the links for individuals involved in XSB). An overview of SLG resolution, and practical evaluation strategies for it are provided in [8, 41, 36, 19]. The engine of XSB, the SLG-WAM, is described in [35, 33, 18, 38, 7, 14] as it is implemented in Version 2.0 and its performance analyzed. Examples of large-scale applications that use tabling are overviewed in [42, 9, 12].

5.1 XSB as a Prolog System

Before describing how to program using tabling it is perhaps worthwhile reviewing some of the goals of XSB

1. To execute tabled predicates at the speed of compiled Prolog.
2. To ensure that the speed of compiled Prolog is not slowed significantly by adding the option of tabling.
3. To execute that the functionality of Prolog is not compromised by support for tabling.

4. To provide Prolog functionality in tabled predicates whenever it is semantically sensible to do so.

5. To provide standard predicates to manipulate tables taken as objects in themselves.

Goals 1 and 2 are addressed by XSBs engine, which in Version 2.0 is based on a memory-copying version of a virtual machine called the SLG-WAM. The overhead for SLD resolution using this machine is negligible. Thus when XSB is used simply as a Prolog system (i.e., no tabling is used), it is reasonably competitive with other Prolog implementations based on a WAM emulator written in C or assembly. For example, XSB Version 1.6 is about two to three times slower than Quintus 3.1.1 or emulated SICStus Prolog 3.1.

Goals 3, 4 and 5 have been nearly met, but there are a few instances in which interaction of tabling with a Prolog construct has been accomplished, or is perhaps impossibe. Accordingly we discuss these instances throughout this chapter. XSB is still under development however, so that future versions may support more transparent mixing of Prolog and tabled code (e.g. allowing tabled predicates in the scope of $'\backslash +'$ `/1`) or adding Prolog functionality to tabled predicates (e.g. allowing non-ground negation in `tnot/1`).

5.2 Tabling in Definite Programs

Definite programs, also called Horn Clause Programs, are those programs without negation — In XSB, this means without the `\+/1`, `fail_if/1`, `not/1` or `tnot/1` operators. Consider the Prolog program

```
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

together with the query `?- path(1,Y)`. This program has a simple, declarative meaning: there is a path from `X` to `Y` if there is a path from `X` to some node `Z` and there is a path from `Z` to `Y`, or if there is a direct path from `X` to `Y`. Prolog, however enters into an infinite loop when computing an answer to this query. The inability of Prolog to answer such queries, which arise frequently, comprises one of its major limitations as an implementation of logic.

A number of approaches have been developed to address this problem by reusing partial answers to the query `path(1,Y)` [17, 43, 3, 45, 46]. The ideas behind these algorithms can be described in the following manner. First, the implementation keeps track of all calls to tabled predicates, (or *tabled subgoals* such as `path(1,Y)` in the above example. Whenever a new tabled subgoal S is called, a check is first made to see whether S is in the table. If so, S is resolved against answers in the table; if not S is entered into the table and the subgoal is resolved against program clauses, as in Prolog. Answers are handled in the same way. When an answer to a tabled subgoal S is derived a check is made against the table for S to see if the answer is there. If the answer isn't in the table for S, the answer is added and scheduled to be returned to all instances where S has been called; if the answer is already in the table, the evaluation simply fails and backtracks to generate more answers.

Predicates can be declared tabled in a variety of ways. A common form is the compiler directive

:-`table` $p_1/n_1, \ldots, p_k/n_k$.

where p_i is a predicate symbol and n_i is an integer representing the arity of p_i. This directive can be added to the file containing the predicate to be tabled and then to compile the file.

Exercise 5.2.1 *Unless otherwise noted, the file table_examples.P in the directory* `$XSB_DIR/examples` *contains all code for the running examples in this section. Consult the file into XSB and type the query*

```
?- path(1,Y).
```

and continue hitting semi-colons until you have exhausted all answers. Type the query again. Can you guess why the order of answers is different? Now type

```
?- abolish_all_tables.
```

and retry the path query.

Exercise 5.2.2 *If you are curious, try rewriting the path query as it would be written in Prolog. Will it now terminate for the provided* `edge/2` *relation? (Remember, in XSB you can always hit* `<ctrl>-C` *if you go into an infinite loop).*

The return of answers in tabling aids in filtering out redundant computations – indeed it is this property which makes tabling terminate for many classes of programs. The *same generation* program furnishes a case of the usefulness of tabling for optimizing a Prolog program.

Exercise 5.2.3 *If you are* still *curious, load in the file* `cyl.P` *in the* `$XSB_DIR/examples` *directory using the command.*

```
?- load_dync(cyl.P).
```

and then type the query

```
?- same_generation(X,X),fail.
```

Now rewrite the `same_generation/2` *program so that it does not use tabling and retry the same query what happens? (Be patient — or use* `<ctrl>-C`*).*

The examples stress two differences between tabling and SLD resolution beyond termination properties. First, that each solution to tabled subgoal is returned only once — a property that is helpful not only for `path/2` but also for `same_generation/2` which terminates in Prolog. Second, because answers are sometimes obtained using program clauses and sometimes using answers, answers may be returned in an unaccustomed order.

In the language of tabling, the first instance of a tabled subgoal S is called a *generator* subgoal, and is expanded using program clauses as in SLD resolution (Prolog). Subsequent instances of S are referred to as *consuming* subgoals and are expanded using answers in the table for S instead of program clauses. Because *consuming* subgoals resolve against unique answers rather than repeatedly against program clauses, tabling will terminate whenever (1) a finite number of subgoals are encountered in query evaluation, (2) each of these subgoals have a finite number of answers. Indeed, it can be proven that for any program with the *bounded term depth property* (roughly, where all terms generated in a program have a maximum depth), SLG computation will terminate. These programs include the important class of *Datalog* programs.

Variant and Subsumptive Tabling The above description gives the general idea of how tabling affects definite programs but is imprecise on certain points. In XSB, a subgoal subgoals S_2 can use a table from S_1 if S_1 is a *variant* of S_2, that is, if S_1 and S_2 are the same up to variable renaming. Other tabling strategies may allow S_2 to use the table of S_1 if S_2 not more general than S_1, or S_2 is *subsumed by* S_1:

Example 5.2.1 *The terms* `p(f(Y),X,1)` *and* `p(f(Z),U,1)` *are variants, but* `p(f(Y),X,1)` *and* `p(f(Z),Z,1)` *are not. In fact, the former subsumes the latter.*

Just as a subsumption or variance relation can be used to decide when one subgoal can use the table of another, the two relations can be used to determine when an answer should be returned. In XSB's engine, a derived answer A will be considered new and returned to a subgoal S only if A is not a variant of some other previously derived answer for S. In Version 2.0 of XSB, subgoal subsumption is not supported: although work on an engine that includes subgoal subsumption is nearing completion. Answer subsumption, however, can be flexibly programmed as discussed in Section 5.4 [1].

Cuts and Tabling Tabling integrates well with most Prolog functionality, even for non-pure Prolog predicates. Meta-logical predicates like `var/1`, and predicates with side-effects like `read/1` and `write/1` can be used freely in tabled predicates as long as it is remembered that only the first call to a goal will execute program clauses: the rest will look up answers from a table.

The use of cuts with tabling is more problematic, as can be seen from the following exercise.

Exercise 5.2.4 *Consider the program*

```
:- table cut_p/1,cut_q/1,cut_r/0,cut_s/0.

cut_p(X):- cut_q(X),cut_r.
cut_r:- cut_s.
cut_s:- cut_q(_).
cut_q(1).        cut_q(2).

once(Term):- call(Term),!.
```

[1] We also note that the library `subsumes` contains routines for checking variance and subsumption.

What solutions are derived for the goal `?- p(X)`*? Suppose that* `cut_p/1` *were rewritten as*

```
p1(X):- q1(X),once(r1).
```

How should this cut over a table affect the answers generated for `cut_p/1`*? What happens if you rewrite* `p/1` *in this way and compile it in XSB?*

The solution Version 2.0 of XSB takes to the problem posed in Exercise 5.2.4 is to check whether a tabled predicate statically lies in the scope of a cut at compile time. If so, the compilation is aborted. However, cuts are allowed within tabled predicates, subject (as always) to the restriction that the scope of a cut cannot include a call to a tabled predicate.

Example 5.2.2 *An example of using cuts in a tabled predicate is a tabled meta-interpreter.*

```
:- table demo/1.

demo(true).
demo((A,B)):-!,demo(A),demo(B).
demo(C):-call(C).
```

More elaborate tabled meta-interpreters can be extremely useful, for instance to implement various extensions of definite or normal programs.

In Version 2.0 of XSB a "cut" over tables occurs only when the user makes a call to a tabled predicate from the interpreter level, but does not generate all solutions. In such a case, the user will see the warning `"Removing incomplete tables..."` appear. Any complete tables will not be removed. They can be abolished by using one of XSB's predicates for abolishing tables.

Potential Pitfalls in Tabling While the judicious use of tabling can make some programs faster, its indiscriminate use can make other programs slower. Naively tabling `append/3` is one case

```
append([],L,L).
append([H|T],L,[H|T1]) :- append(T,L,T1).
```

can, in the worst case, copy N sublists of the first and third arguments into the table, transforming a linear algorithm into a quadratic one.

Exercise 5.2.5 *If you need convincing that tabling can sometimes slow a query down, type the query:*

```
?- genlist(1000,L),prolog_append(L,[a],Out).
```

and then type the query

```
?- genlist(1000,L),table_append(L,[a],Out).
```

`append/3` *is a particularly bad predicate to table. Type the query*

```
?- table_append(L,[a],Out).
```

(i.e. with no `genlist/2` *and backtrack through a few answers. Will* `table_append/3` *ever succeed for this predicate? Why not?*

Suppose DCG predicates (Section 8) are defined to be tabled. How is this similar to tabling append?

Another issue to be aware of when using tabling in XSB is tracing. XSB's tracer is a standard 4-port tracer, that interacts with the engine at each call, exit, redo, and failure of a predicate (see Chapter 7). When tabled predicates are traced, these events may occur in unexpected ways, as the following example shows.

Exercise 5.2.6 *Consider a tabled evaluation when the query* `?- a(0,X)` *is given to the following program*

```
:- table mut_ret_a/2, mut_ret_b/2.
mut_ret_a(X,Y):- mut_ret_d(X,Y).
mut_ret_a(X,Y):- mut_ret_b(X,Z),mut_ret_c(Z,Y).

mut_ret_b(X,Y):- mut_ret_c(X,Y).
mut_ret_b(X,Y):- mut_ret_a(X,Z),mut_ret_d(Z,Y).

mut_ret_c(2,2).      mut_ret_c(3,3).

mut_ret_d(0,1).      mut_ret_d(1,2).     mut_ret_d(2,3).
```

`mut_ret_a(0,1)` *can be derived immediately from the first clause of* `mut_ret_a/2`*. All other answers to the query depend on answers to the subgoal* `mut_ret_b(0,X)` *which arises in the evaluation of the second clause of* `mut_ret_a/2`*. Each answer to* `mut_ret_b(0,X)` *in turn depends on an answer to* `mut_ret_a(0,X)`*, so that the evaluation switches back and forth between deriving answers for* `mut_ret_a(0,X)` *and* `mut_ret_b(0,X)`*.*

Try tracing this evaluation, using creep and skip. Do you find the behavior intuitive or not?

Table Directives and Declarations Often it is tedious to decide which predicates must be tabled. To address this, XSB can automatically table predicates in files. The declaration `auto_table` chooses predicates to table to assist in termination, while `suppl_table` chooses predicates to table to optimize data-oriented queries. Both are explained in Section 3.8.4.

Exercise 5.2.7 *The reader may have noted that the command* `table` *was referred to as a directive, while* `auto_table` *and* `suppl_table` *were both referred to as declarations. The difference is that* *the user can execute a directive at the command line but not a compiler declaration. For instance, restart XSB and at the XSB prompt, type the directive*

```
?- table(dyn_path/2).
```

and

```
?- load_dyn(dyn_examples).
```

Try the queries to `path/2` *of the previous examples. Note that it is important to dynamically load* `dyn_examples.P` *— otherwise the code in the file will be compiled without knowledge of the tabling declaration.*

5.3 Stratified Normal Programs

Normal programs extend definite programs to include default negation, which posits a fact as false if all attempts to prove it fail. As shown in Example 1.0.1, which presented one of Russell's paradoxes as a logic program, the addition of default negation allows logic programs to express contradictions. As a result, some assertions, such as `shaves(barber,barber)` may be undefined, although other facts, such as `shaves(barber,mayor)` may be true. Formally, the meaning of normal programs may be given using the *well-founded semantics* and it is this semantics that XSB adopts for negation.

The Intuition behind Stratified Programs Before considering the full well-founded semantics, we discuss how XSB can be used to evaluate programs with *stratified negation.* Intuitively, a program uses stratified negation whenever there is no recursion through negation. Indeed, most programmers, most of the time, use stratified negation.

Exercise 5.3.1 *The program*

```
win(X):- move(X,Y),tnot(move(Y)).
```

is stratified when the `move/2` *relation is a binary tree. This can be seen by loading the file* `$XSB_DIR/examples/tree` *along with* `table_examples.P` *and typing the query*

```
?- win(1).
```

`win(1)` *calls* `win(2)` *through negation,* `win(2)` *calls* `win(4)` *through negation, and so on, but no subgoal ever calls itself recursively through negation.*

The previous example of `win/1` over a binary tree is a simple instance of a stratified program, but it does not even require tabling. A more complex example is presented below.

Exercise 5.3.2 *Consider the query* `?- lrd_s` *to the following program*

```
lrd_p:- lrd_q,tnot(lrd_r),tnot(lrd_s).
lrd_q:- lrd_r,tnot(lrd_p).
lrd_r:- lrd_p,tnot(lrd_q).
lrd_s:- tnot(lrd_p),tnot(lrd_q),tnot(lrd_r).
```

Should `lrd_s` *be true or false? Try it in XSB. Using the intuitive definition of "stratified" as not using recursion through negation, is this program stratified? Would the program still be stratified if the order of the literals in the body of clauses for* `lrd_p`, `lrd_q`, *or* `lrd_r` *were changed?*

The rules for `p`, `q` and `r` are involved in a positive loop, and no answers are ever produced. Each of these atoms can be failed, thereby proving `s`. Exercise 5.3.2 thus illustrates an instance of how tabling differs from Prolog in executing stratified programs since Prolog would not fail finitely for this program.

Completely Evaluated Subgoals Knowing when a subgoal is completely evaluated can be useful when programming with tabling. Simply put, a subgoal S is *completely evaluated* if an evaluation can produce no more answers for S. The computational strategy of XSB makes great use of complete evaluation so that understanding this concept and its implications can be of great help to a programmer.

Consider a simple approach to incorporating negation into tabling. Each time a negative goal is called, a separate table is opened for the negative call. This evaluation of the call is carried on to termination. If the evaluation terminates, its answers if any, are used to determine the success of failure of the calling goal. This general mechanism underlies early formulations for tabling stratified programs [21, 40]. Of course this method may not be efficient. Every time a new negative goal is called, a new table must be started, and run to termination. We would like to use information already derived from the computation to answer a new query, if at all possible — just as with definite programs.

XSB addresses this problem by keeping track of the *state* of each subgoal in the table. A call can have a state of *complete*, *incomplete* or *not_yet_called*. Calls that do have table entries may be either *complete* or *incomplete*. A subgoal in a table is marked *complete* only after it is determined to be completely evaluated; otherwise the subgoal is *incomplete*. If a tabled subgoal is not present in the table, it is termed *not_yet_called*. XSB contains predicates that allow a user to examine the state of a given table (Section 6.12).

Using these concepts, we can overview how tabled negation is evaluated for stratified programs. If a literal `tnot(S)` is called, where `S` is a tabled subgoal, the evaluation checks the state of `S`. If `S` is *complete* the engine simply determines whether the table contains an answer for `S`. Otherwise the engine *suspends* the computation path leading to `tnot(S)` until `S` is completed (and calls `S` if necessary). Whenever a suspended subgoal `tnot(S)` is completed with no answers, the engine resumes the evaluation at the point where it had been suspended. We note that because of this behavior, tracing programs that heavily use negation may produce behavior unexpected by the user.

`tnot/1` vs. `'\+'/1` Subject to some semantic restrictions, an XSB programmer can intermix the use of tabled negation (`tnot/1`) with Prolog's negation (`'\+'/1`, or equivalently `fail_if/1` or `not/1`). These restrictions are discussed in detail below — for now we focus on differences in behavior or these two predicates in stratified programs. Recall that `'\+'(S)` calls S and if S has a solution, Prolog , executes a cut over the subtree created by `'\+'(S)`, and fails. `tnot/1` on the other hand, does not execute a cut, so that all subgoals in the computation path begun by the negative call will be completely evaluated. The major reason for not executing the cut is to insure that XSB evaluates ground queries to Datalog programs with negation with polynomial data complexity. As seen in Section 5.2, this property cannot be preserved if negation "cuts" over tables.

There are other small differences between `tnot/1` and `'\+'/1`illustrated in the following exercise.

Exercise 5.3.3 *In general, making a call to non-ground negative subgoal in Prolog may be unsound (cf. [29]), but the following program illustrates a case in which non-ground negation is sound.*

```
ngr_p:- \+ ngr_p(_).
ngr_p(a).
```

Its tabled analog is

```
:- table ngr_tp/1.
ngr_tp:- tnot(ngr_tp(_)).
ngr_tp(a).
```

Version 2.0 of XSB will flounder on the call to `ngr_tp`*, but not on the call to* `ngr_p/0`*.*

The description of `tnot/1` *in Section 6.4 describes other small differences between* `'\+'/1`*and* `tnot/1` *as implemented in XSB.*

Before leaving the subject of stratification, we note that the concepts of stratification also underly XSB's evaluation of tabled findall: `tfindall/3`. Here, the idea is that a program is stratified if it contains no loop through tabled findall (See the description of predicate `tfindall/3` on page 83).

5.3.1 Non-stratified Programs

As discussed above, in stratified programs, facts are either true or false, while in non-stratified programs facts may also be undefined. XSB represents undefined facts as *conditional answers.*

Conditional Answers

Exercise 5.3.4 *Consider the behavior of the* `win/1` *predicate from Exercise 5.3.1.*

```
win(X):- move(X,Y),tnot(move(Y)).
```

when the when the `move/2` *relation is a cycle. Load the file* `$XSB_DIR/examplescycle1k.P` *into XSB and again type the query* `?- win(1)`*. Does the query succeed? Try* `tnot(win(1))`*.*

Now import `get_residual/2` *via the command*

```
?- import get_residual/2 from tables.
```

Can you guess what is happening with this non-stratified program?

The predicate `get_residual/2` (Section 6.12) unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the subgoal is taken to be conditional on the truth of the elements in the delay list. Thus `win(1)` is conditional on `tnot(win(2))`, `win(2)` in `tnot(win(3))` and so on until `win(1023)` which is conditional on `win(1)`.

From the perspective of the well-founded semantics, `win(1)` is undefined. Informally, true answers in the well-founded semantics are those that have a (tabled) derivation. False answers are those for which all possible derivations fail — either finitely as in Prolog or by failing positive loops. `win(1)` fits in neither of these cases – there is no proof of `win(1)`, yet it does not fail in the sense given above and is thus undefined.

However this explanation does not account for why undefined answers should be represented as conditional answers, or why a query with a conditional answer *and* its negation should both succeed. These features arise from the proof strategy of XSB, which we now examine in more detail.

Exercise 5.3.5 *Consider the program*

```
:- table simpl_p/1,simpl_r/0,simpl_s/0.
simpl_p(X):- tnot(simpl_s).

simpl_s:- tnot(simpl_r).
simpl_s:- simpl_p(X).

simpl_r:- tnot(simpl_s),simpl_r.
```

Is `simpl_p(X)` *true for any* `X`*? Try the query* `?- simpl_p(X)` *– be sure to backtrack through all possible answers. Now try the query again. What could possibly account for this behavior?*

At this point, it is worthwhile to examine closely the evaluation of the program in Exercise 5.3.5. The query `simpl_p(X)` calls `simpl_s` and `simpl_r` and executes the portion of the program shown below in bold:

simpl_p(X):- tnot(simpl_s).

simpl_s:- tnot(simpl_r).
simpl_s:- simpl_p(X).

simpl_r:- tnot(simpl_s),*simpl_r.*

Based on evaluating only the bold literals, the three atoms are all undefined since they are neither proved true, nor fail. However if the evaluation could only look at the literal in italics, *simpl_r*, it would discover that *simpl_r* is involved in a positive loop and, since there is only one clause for *simpl_r*, the evaluation could conclude that the atom was false. This is exactly what XSB does, `delays` the evaluation of `tnot(simpl_s)` in the clause for `simpl_r` and looks ahead to the next literal in the body of that clause. This action of looking ahead of a negative literal is called *delaying*. A delayed literal is moved into the *delay list* of a current path of computation. Whenever an answer is derived, the delay list of the current path of computation is copied into the table. If the delay list is empty, the answer is unconditional; otherwise it is conditional. Of course, for definite programs any answers will be unconditional — we therefore omited delay lists when discussing such programs.

In the above program, delaying occurs for the negative literals in clause for `simpl_p(X)`, `simpl_s`, and `simpl_r`. In the first two cases, conditional answers can be derived, while in the third, `simpl_r` will fail as mentioned above. Delayed literals eventually become evaluated through *simplification*. Consider an answer of the form

```
simpl_p(X):- tnot(simpl_s)|
```

where the `|` is used to represent the end of the delay list. If, after the answer is copied into the table, `simpl_s` turns out to be false, (after being initially delayed), the answer can become unconditional. If `simpl_s` turns out to be true, the answer should be removed, it is false.

In fact, it is this last case that occurs in Exercise 5.3.5. The answer

```
simpl_p(X):- tnot(simpl_s)|
```

is derived, and returned to the user (XSB does not currently print out the delay list). The answr is then removed through simplification so that when the query is re-executed, the answer does not appear.

We will examine in detail how to alter the XSB interface so that evaluation of the well-founded semantics need not be confusing. It is worthwhile to note that the behavior just described is uncommon.

Version 2.0 of XSB handles dynamically stratified programs through delaying negative literals when it becomes necessary to look to their right in a clause, and then simplifying away the delayed literals when and if their truth value becomes known. However, to ensure efficiency, literals are never delayed unless the engine determines them to not to be stratified under the LRD-stratified evaluation method.

When Conditional Answers are Needed A good Prolog programmer uses the order of literals in the body of a clause to make her program more efficient. However, as seen in the previous section, delaying can break the order that literals are evaluated within the body of a clause. It then becomes natural to ask if any guarantees can be made that XSB is not delaying literals unnecessarily.

Such a guarantee can in fact be made, using the concept of *dynamic stratification* [31]. Without going into the formalism of dynamic stratification, we note that a program is dynamically stratified if and only if it has a two-valued model. It is also known that computation of queries to dynamically stratified programs is not possible under any fixed strategy for selecting literals within the body of a clause. In other words, some mechanism for breaking the fixed-order literal selection strategy must be used, such as delaying.

However, by redefining dynamic stratification to use an arbitrary fixed-order literal selection strategy (such as the left-to-right strategy of Prolog), a new kind of stratification is characterized, called *Left-to-Right Dynamic Stratification*, or *LRD-stratification*. LRD-stratified is not as powerful as dynamic stratification, but is more powerful than other fixed-order stratification methods, and it can be shown that for ground programs, XSB delays only when programs are not LRD-stratified. In the language of [36] XSB is *delay minimal*.

Programming in the Well-founded Semantics XSB delays literals for non-LRD-stratified programs and later simplifies them away. But how can the programmer determine when all simplification has been done? One method is to use local evaluation, discussed below in Section 5.4.1. A second method is to make a top-level call for a predicate, `p` as follows:

```
?- p,fail ; p.
```

when the second `p` in this query is called, all simplification on `p` will have been performed. However, this query will succeed if `p` is true *or* undefined.

Exercise 5.3.6 *Write a predicate* `wfs_call(?Tpred,?Val)` *such that if* `Tpred` *is a ground call to a tabled predicate,* `wfs_call(?Tpred,?Val)` *calls* `Tpred` *and unifies* `Val` *with the truth value of* `Tpred` *under the well-founded semantics.*

How would you modify `wfs_call(?Tpred,?Val)` *so that it properly handled cases in which* `Tpred` *is non-ground.*

Trouble in Paradise: Answer Completion The engine for XSB performs both Prolog style and answer resolution, along with delay and simplification. What it does not do is to perform an operation called *answer completion* which is needed in certain (pathological?) programs.

Exercise 5.3.7 *Consider the following program:*

```
:- table p/1,r/0,s/0.
ac_p(X):- ac_p(X).
ac_p(X):- tnot(ac_s).

ac_s:- tnot(ac_r).
ac_s:- ac_p(X).

ac_r:- tnot(ac_s),ac_r.
```

Using either the predicate from Exercise 5.3.6 or some other method, determine the truth value of `ac_p(X)`*. What should the value be? (hint: what is the value of* `ac_s/1`*?).*

For certain programs, XSB will delay a literal (such as `ac_p(X)` that it will not be able to later simplify away. In such a case, an operation, called *answer completion* is needed to remove the clause

```
p(X):- p(X)|
```

Without answer completion, XSB may consider some answers to be undefined rather than false. It is thus is sound, but not complete for terminating programs to the well-founded semantics. Answer completion is not available for Version 2.0 of XSB, as it is expensive and the need for answer completion arises rarely in practice. However answer completion will be included at some level in future versions of XSB.

5.3.2 On Beyond Zebra: Implementing Other Semantics for Non-stratified Programs

The Well-founded semantics is not the only semantics for non-stratified programs. XSB can be used to (help) implement other semantics that lie in one of two classes. 1) Semantics that extend the well-founded semantics to include new program constructs; or 2) semantics that contain the well-founded partial model as a submodel.

An example of a semantics of class 1) is (WFSX) [2], which adds explicit (or provable) negation to the default negation used by the Well-founded semantics. The addition of explicit negation in WFSX, can be useful for modeling problems in domains such as diagnosis and hierarchical reasoning, or domains that require updates [26], as logic programs. WFSX is embeddable into the well-founded semantics; and this embedding gives rise to an XSB meta-interpreter, or, more efficiently, to the preprocessor described in Section *Extended Logic Programs* in Volume 2. See [42] for an overview of the process of implementing extensions of the well-founded semantics.

An example of a semantics of class 2) is the stable model semantics. Every stable model of a program contains the well-founded partial model as a submodel. As a result, the XSB can be used to evaluate stable model semantics through the *residual program*, to which we now turn.

The Residual Program Given a program P and query Q, the residual program for Q and P consists of all (conditional and unconditional) answers created in the complete evaluation of Q.

Exercise 5.3.8 *Consider the following program.*

```
:- table ppgte_p/0,ppgte_q/0,ppgte_r/0,ppgte_s/0,
         ppgte_t/0,ppgte_u/0,ppgte_v/0.
ppgte_p:- ppgte_q.          ppgte_p:- ppgte_r.

ppgte_q:- ppgte_s.          ppgte_r:- ppgte_u.
ppgte_q:- ppgte_t.          ppgte_r:- ppgte_v.

ppgte_s:- ppgte_w.          ppgte_u:- undefined.
```

```
ppgte_t:- ppgte_x.          ppgte_v:- undefined.

ppgte_w:- ppgte(1).         ppgte_x:- ppgte(0).
ppgte_w:- undefined.        ppgte_x:- undefined.

ppgte(0).

:- table undefined/0.
undefined:- tnot(undefined).
```

Write a routine that uses `get_residual/2` *to print out the residual program for the query* `?- ppgte_p,fail`*. Try altering the tabling declarations, in particular by making* `ppgte_q/0`*,* `ppgte_r/0`*,* `ppgte_s/0` *and* `ppgte_t/0` *non-tabled. What effect does altering the tabling declarations have on the residual program?*

When XSB returns a conditional answer to a literal L, it does not propagate the delay list of the conditional answer, but rather delays L itself, even if L does not occur in a negative loop. This has the advantage of ensuring that delayed literals are not propagated exponentially through conditional answers.

Stable Models Stable models are one of the most popular semantics for non-stratified programs. The intuition behind the stable model semantics for a ground program P can be seen as follows. Each negative literal $notL$ in P is treated as a special kind of atom called an *assumption.* To compute the stable model, a guess is made about whether each assumption is true or false, creating an assumption set, A. Once an assumption set is given, negative literals do not need to be evaluated as in the well-founded semantics; rather an evaluation treats a negative literal as an atom that succeeds or fails depending on whether it is true or false in A.

Example 5.3.1 *Consider the simple, non-stratified program*

```
writes_manual(terry)-¬writes_manual(kostis),has_time(terry).
writes_manual(kostis)-¬writes_manual(terry),has_time(kostis).
has_time(terry).
has_time(kostis).
```

there are two stable models of this program: in one `writes_manual(terry)` *is true, and in another* `writes_manual(kostis)` *is true. In the Well-Founded model, neither of these literals is true. The residual program for the above program is*

```
writes_manual(terry)-¬writes_manual(kostis).
writes_manual(kostis)-¬writes_manual(terry).
has_time(terry).
has_time(kostis).
```

Computing stable models is an intractable problem, meaning that any algorithm to evaluate stable models may have to fall back on generating possible assumption sets, in pathological cases. For a ground program, if it is ensured that residual clauses are produced for *all* atoms, using the residual program may bring a performance gain since the search space of algorithms to compute stable models will be correspondingly reduced. In fact, by using XSB in conjunction with a Stable Model generator, Smodels [30], an efficient system has been devised for model checking of concurrent systems that is 10-20 times faster than competing systems [28].

5.4 Tabled Aggregation

The following shortest path predicate is a modification of the `path/2` predicate of Section 5.2:

```
:- table path/3.
path(X,Y,C) :- path(X,Z,C1), edge(Z,Y,C2), C is C1 + C2.
path(X,Y,C) :- edge(X,Y,C).
```

Exercise 5.4.1 `path/3` *has a simple declarative meaning: it computes the path between two vertices of a graph along with the cost of the path. Since* `path/3` *is tabled would you expect it to terminate? Try the query* `?- path(1,5,X)` *over the graph provided in the file* `table_examples.P`.

If we could use tabling to compute the path with least cost, or the shortest path, the program would not only omit extraneous information, but it would also terminate. Recall that for simple horn programs, variant-based tabling ensures termination by only returning a given answer A once, and failing on subsequent derivations of A. If this strategy could be extended so that the engine only returned a new answer if it was minimal, termination could be ensured. The XSB predicate, `filterReduce(?Pred,+Binary_operator,+Identity,Value)`, does just this.

Exercise 5.4.2 *The use of* `filterReduce/4` *can be seen most easily through an example such as the following, (which uses a closely related predicate* `filterReduce1/4`*).*

```
shorter_path(X,Y,C) :- filterReduce1(sp(X,Y),min,infinity,C).

sp(X,Y,C) :- shorter_path(X,Z,C1),
             edge(Z,Y,C2),C is C1 + C2.
sp(X,Y,C) :- edge(X,Y,C).

min(X,Y,Y):- \+ number(X),!.
min(X,Y,X):- \+ number(Y),!.
min(One,Two,Min):- One > Two -> Min = Two ; Min = One.
```

Note that the library predicate `filterReduce1/4` *is tabled, so that neither* `sp/3` *nor* `shorter_path/3` *need be tabled. Now try the query* `shorter_path(1,5,C)`.

`filterReduce1((?Pred,+Binary_operator,+Identity,Value)`, forms a new predicate out of `Pred` and `Value` to get a new predicate to call. `Binary_Operator` must define a binary function in which the first two arguments determine the third. `Id` must be the identity of `Binary_operator`. `Value` becomes the result of applying `Op` to all the elements in the table that are variants of `Pred`. In our case, when a new answer `sp(X,Y,C)` is derived within `filterReduce1/4`, the later predicate returns only when `C` is a shorter path for `X` and `Y` than any so far derived.

While `shorter_path/4` terminates, it returns non-optimal solutions, and these solutions can in principle be costly — [18] cites a case in which the shorter path program, which should be less than cubic in the number of vertices in a graph, has exponential complexity because of the non-optimal solutions that are returned. Fortunately, this has an easy solution.

Exercise 5.4.3 *The actual* `shortest_path` *program has the following definition.*

```
filterReduce(Call,Op,Id,Res) :- filterReduce1(Call,Op,Id,Res), fail.
filterReduce(Call,Op,Id,Res) :- filterReduce1(Call,Op,Id,Res).

shortest_path(X,Y,C) :- filterReduce(sp(X,Y),min,infinity,C).

sp(X,Y,C) :- shortest_path(X,Z,C1),
             edge(Z,Y,C2),C is C1 + C2.
sp(X,Y,C) :- edge(X,Y,C).

min(X,Y,Y):- \+ number(X),!.
min(X,Y,X):- \+ number(Y),!.
min(One,Two,Min):- One > Two -> Min = Two ; Min = One.
```

Once again try the query `shortest_path(1,5,C)`*.*

By simply failing out of `filterReduce1/4` and then rereading the maximal value from the table, an efficient `shortest_path` algorithm is derived, whose complexity is roughly cubic in the number or vertices of the graph. This solution is not general for all predicates, but does work for deriving the shortest path. A more general solution is provided in Section 5.4.1.

`filterReduce/4` is an extremely useful predicate. It can write database aggregation functions, such as min, max, count, sum, and average. However, it can also be used to implement paraconsistent and quantitative reasoning through Generalized Annotated Programs [23], as detailed in the section on GAPs in Volume 2 of this manual.

Several predicates perform tabled aggregation besides `filterReduce/4`. One of these is the predicate `filterPO1(?Pred,?Preference_structure,+Partial_order)`. Analoguosly to `filterReduce1/4` if `Pred` is an n-ary predicate, `filterPO/4` forms a (n+1)-ary predicate `Pred1` whose last argument is `Preference_structure` and whose functor and all other arguments are determined by `Pred`. `filterPO(?Pred,?Preference_structure,+Partial_order)`, then calls `Pred1` and for each return of `Pred1` fails if there is some answer already in the table for `filterPO1/4` such that the first n arguments of `Pred` in the tabled answer unify with the first n arguments of `Pred` in the return and whose preference structure (last argument) is preferred to that of the return. A case study in the use of `filterPO/4` to construct preference logic grammars can be found in [10].

5.4.1 Local Evaluation

For the shortest path example, simply failing until a minimal answer was derived and then returning that solution was an effective technique for computing the shortest path. However, this approach will not always work. As we have seen in Exercise 5.2.6, programs can consist of sets of mutually recursive predicates and in principle these sets can be arbitrarily large. If these computations are to use tabled aggregation, the approach taken by `filterReduce/4` will not suffice. To see this, we make the notion of mutual recursion more precise. A tabled computation can be viewed as a directed graph, in which there is a link from one non-completed tabled predicate $P1$ to a non-completed tabled predicate $P2$ if $P2$ (or $tnot(P2)$) is called by $P1$. Of course, this graph constantly changes through an evaluation as resolution proceeds, subgoals are completed, and so on. Any directed graph can be uniquely partitioned into a set of maximal *strongly connected components* or SCCs, and these sets correspond to sets of mutually recursive predicates. The SCCs then, are reminiscent of the LRD-stratifiedstratification discussed in Section 5.3.1, except that both positive and negative links are counted as dependencies. From this view, to optimally compute tabled aggregation, non-optimal answers from a given subgoal S must be returned within the SCC of S, but not outside the SCC. This action is performed by *Local Scheduling*.

It is illustrative to compare local scheduling to *Batched Scheduling* the default scheduling of XSB. Batched scheduling returns answers as they are derived, and resembles Prolog's tuple at a time scheduling. Local scheduling was shown to be quite efficient in terms of time and space in [18], and is the fastest scheduling strategy that we know of for computing a sequence of answers. The same paper also introduced Local Scheduling, which computes all answers for each SCC and return only the best answer (or answers) out of the SCC, when the SCC is completely evaluated — exactly the thing for tabled aggregation.

XSB can be configured to use local scheduling via the configuration option `--enable-local-scheduling` and remaking XSB. This will not affect the default version of XSB, which will also remain available.