Minimal Model Tabling in Mercury

Zoltan Somogyi[1] and Konstantinos Sagonas[2]

[1] Department of Computer Science and Software Engineering, University of Melbourne, Australia
[2] Department of Information Technology, Uppsala University, Sweden
zs@cs.mu.OZ.AU kostis@it.uu.se

Abstract. Prolog systems such as XSB have proven that tabling can be quite helpful in a variety of tasks, especially if it is efficiently implemented and fully integrated in the language. Implementing tabling in Mercury poses special challenges for several reasons. First, Mercury is both semantically and culturally quite different from Prolog. While decreeing that tabled predicates must not include cuts (or Prolog-style negations) is acceptable in a Prolog system, it is not acceptable in Mercury, since if-then-elses and existential quantification have sound semantics and are used very frequently both by programmers and by the compiler. The Mercury implementation thus has no option but to handle interactions of tabling with Mercury's language features safely. Second, the Mercury implementation is vastly different from the WAM, and many of the differences (e.g. storing values directly in stack slots without indirection, the absence of a trail) have significant impact on the implementation of tabling. In this paper, we describe how we adapted the copying approach to tabling to implement minimal model tabling in Mercury.

1 Introduction

By now, it is widely recognized that tabling adds power to a logic programming system. By avoiding repeated subcomputations, it often significantly improves the performance of applications, and by terminating more often it allows for a more natural and declarative style of programming. It is therefore not a fluke that tabling has so far been used for a variety of tasks ranging from program analysis via abstract interpretation [4, 5] to model checking [11], learning via statistical abduction [15] and various other forms of non-monotonic reasoning. All these applications would have been significantly more difficult to program without tabling.

When deciding which tabling mechanism to adopt, an implementor is faced with various choices. The first one concerns the resolution strategy. Tabling resolution strategies which are *linear*, such as SLDT [19] and DRA [8], are relatively easy to implement, but they are also relatively ad hoc and often perform recomputation. On the other hand, *proper* tabled resolution strategies, such as OLDT [17] and SLG [3], avoid recomputation, but their implementation is significantly more challenging because they require the introduction of a suspension/resumption mechanism into the basic execution engine.

In the framework of the WAM [18], there are two main techniques to implement suspension/resumption. The one employed both in XSB [14] and in Yap [12], that of the SLG-WAM [13], implements suspension via *stack freezing* and resumption using an extended trail mechanism called *forward trail*. The SLG-WAM mechanism relies heavily on features specific to the WAM, and imposes a small but non-negligible overhead on *all* programs, not just the ones which use tabling. The other main mechanism, CAT (the Copying Approach to Tabling [6]), completely avoids this overhead; it leaves the WAM stacks unchanged and implements suspension/resumption by incrementally saving and restoring the WAM areas that proper tabling execution needs to preserve in order to avoid recomputation.

For Mercury, we chose to base tabling on SLG resolution, partly because when we started this work (eight years ago) linear tabling resolution strategies did not exist, but more importantly because we wanted to make the Mercury tabling mechanism premium quality, no matter how much implementation effort this required. We decided to restrict the implementation to the subset of SLG that

handles stratified programs (i.e., we compute the *minimal model* of these programs). As implementation platform we chose CAT. The main reason for this choice is because the alternatives conflict with basic assumptions of the Mercury implementation. For example, Mercury has no trail to freeze, let alone a forward one; also freezing the stack *à la* SLG-WAM breaks Mercury's invariant that a call to a predicate which can succeed at most once leaves the stack unchanged. Simply put, CAT is the tabling mechanism requiring the fewest, most isolated changes to the Mercury implementation. This has the additional benefit that it allows us to set up the system to minimize the impact of tabling on the performance of programs and program components that do not use tabling. This is in line with Mercury's general philosophy of "no distributed fat", which requires that, if possible, programs should not pay (in performance) for features they do not use.

Contributions The contributions of this paper are as follows:

- We fully describe the implementation of minimal model tabling in Mercury, a language which is strongly typed and moded, and describe the additional optimizations that can be performed when tabling is introduced in such an environment.
- As a by-product, we adapt the CAT mechanism to a different implementation technology, one which is closer to the execution model of conventional languages than the WAM.
- We mention how we ensure the safety of tabling's interactions with Mercury's if-then-else and existential quantification, constructs that would require the use of cut in Prolog.

Overview To make the paper self-contained, the next two sections briefly review the Mercury language and its implementation, respectively. Section 4 shows the various forms of tabling in Mercury, followed by the paper's main section (§ 5) which fully describes the implementation of minimal model tabling and the technical issues that needed to be addressed. A brief performance comparison with other Prolog systems with tabling appears in § 6, and the paper ends with some concluding remarks.

2 The Mercury language: Capsule description

Mercury is a pure logic programming language intended for the creation of large, fast, reliable programs. While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different due to Mercury's purity, its type, mode, determinism and module systems, and its support for evaluable functions.

Mercury has a strong Hindley-Milner type system very similar to Haskell's. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference).

The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the argument passed by the caller must be a ground term. If output, the argument passed by the caller must be a distinct free variable, which the predicate or function will instantiate to a ground term. It is possible for a predicate or function to have more than one mode; the usual example is `append`, which has two principal modes: `append(in,in,out)` and `append(out,out,in)`. We call each mode of a predicate or function a *procedure*. The Mercury compiler generates different code for different procedures, even if they represent different modes of the same predicate or function. The mode checking pass of the compiler is responsible for reordering conjuncts as necessary to ensure that for each variable the goal that generates the value of the variable comes before all goals that use this value. This means that for each variable in each procedure, the compiler knows exactly which atomic goal (call or unification) in that procedure makes that variable ground.

Each procedure has a determinism, which puts limits on the number of its possible solutions. Procedures with determinism *det* succeed exactly once; *semidet* procedures succeed at most once; *multi* procedures succeed at least once; while *nondet* procedures may succeed any number of times.

A complete description of Mercury can be found in the language reference manual [9].

3 The Mercury implementation

The front end of the Mercury compiler performs type checking, mode checking and determinism analysis. Programs without any errors are then subject to program analyses and transformations (such as the one being presented in Sect. 5) before being passed on to a backend for code generation.

The Mercury compiler has several backends. The original backend [16] generates very low level C code; its output is essentially assembly code in C syntax for portability. Newer backends [10] generate higher level code that can be translated to Java or the intermediate language of Microsoft's .NET platform. So far, tabling is implemented only for the original, low level backend, because it is the only one that allows us to explicitly manipulate stacks (see Sect. 5.3). The following is a brief introduction to the parts of this backend relevant to this paper.

The abstract machine targeted by the low level backend has three main data areas: a heap and two stacks. In the absence of a native Mercury garbage collector, the heap is managed by the Boehm-Demers-Weiser conservative garbage collector for C [2]. Since this collector was not designed for logic programming systems, it has no notion of backtracking, and does not support any mechanism to deallocate all the memory blocks allocated since a specific point in time. Unlike Prolog, Mercury thus cannot recover memory by backtracking and must recover all allocated blocks via garbage collection.

The two stacks of the Mercury abstract machine are called the *det stack* and the *nondet stack*. In most programs, most procedures can succeed at most once. This means that one cannot backtrack into a call to such a procedure after the procedure has succeeded, and therefore there is no need to keep around the arguments and local variables of the call after the initial success (or failure, for semidet procedures). Mercury therefore puts the stack frames of such procedures on the det stack, which is managed exactly like the stacks of conventional imperative languages, i.e. in strict LIFO fashion.

Procedures that can succeed more than once have their stack frames allocated on the nondet stack, from which stack frames are removed only when procedures fail. Since the stack frames of such calls stick around when the call succeeds, the nondet stack is not a true LIFO stack; it is more like a spaghetti stack. Given a clause `p(...) :- q(...), r(...), s(...)`, where `p`, `q` and `r` are all nondet or multi, the stack will contain the frames of `p`, `q` and `r` in order just after the call to `r`. After `r` succeeds and control returns to `p`, the frames of the calls to `q` and `r` are still on the stack. The Mercury abstract machine thus has two registers to point to the nondet stack: `maxfr` always points to the top frame, while `curfr` points to the frame of the currently executing call. (If the currently executing call uses the det stack, then `curfr` points to the frame of its most recent ancestor that uses the nondet stack.)

There are two kinds of frames on the nondet stack: *ordinary* and *temporary*. An ordinary frame is allocated for a procedure that can succeed more than once, i.e. whose determinism is nondet or multi. Such a frame is equivalent to the combination of a choice point and an environment in a Prolog implementation based on the WAM [18]. Ordinary nondet stack frames have five fixed slots and a variable number of other slots. The other slots hold the values of the variables of the procedure, including its arguments; these are accessed via offsets from `curfr`. The fixed slots are the following:

`prevfr` The previous frame slot points to the stack frame immediately below this one. (Both Mercury stacks grow toward higher addresses.) The difference between the address of a nondet stack frame and the address in its `prevfr` slot is the size of that frame.

`redoip` The redo instruction pointer slot contains the address of the instruction to which control should be transferred when backtracking into (or within) this call.

`redofr` The redo frame pointer slot contains the address that should be assigned to `curfr` when backtracking jumps to the address in the `redoip` slot.

`succip` The success instruction pointer slot contains the address of the instruction to which control should be transferred when the call that owns this stack frame succeeds.

`succfr` The success frame pointer slot contains the address of the stack frame that should be assigned to `curfr` when the call owning this stack frame succeeds; this will be the stack frame of its caller.

The `redoip` and `redofr` slots together constitute the failure continuation, while the `succip` and `succfr` slots together constitute the success continuation.

In the example above, both `q`'s and `r`'s stack frames have the address of `p`'s stack frame in their `succfr` slots, while their `succip` slots point to the instructions in `p` after their respective calls.

Suppose the definition of `q` is as in the program below.

```
:- pred q(int::in, int::out) is multi.
q(A, A).
q(A, A+1).
q(A, A+2).
```

The compiler will convert this multi-clause definition into a disjunction. Throughout the execution of `q`, the `redofr` slot of `q`'s stack frame will point to `q`'s stack frame. When control enters `q`, the `redoip` slot will contain the address of the first instruction in the implementation of the second disjunct. When control enters the second disjunct, the `redoip` slot will be updated to contain the address of the first instruction of the third disjunct. When control enters the third disjunct, the `redoip` slot will be updated to contain the address of the failure handler in the Mercury runtime system. This code removes the top frame from the nondet stack, sets `curfr` from the value in the `redofr` slot of the frame that is now on top, and jumps to the instruction pointed to by its `redoip` slot.

This system can handle predicates whose bodies have any number of disjunctions as long as those disjunctions are properly nested, like in Fig. 1(a). However, it cannot handle non-nested disjunctions, because one `redoip` slot cannot store the state of more than one non-nested disjunction. Non-nested disjunctions are implemented using temporary nondet stack frames, which have only `prevfr`, `redoip` and `redofr` slots. When control enters the first disjunct of the second disjunction in `s` (Fig. 1(b)), the nondet stack will contain these frames, in order:

1. The ordinary stack frame of this call to `s`, whose `redoip` records the next alternative in the first disjunction.
2. The ordinary stack frame of this call to `r`.
3. Any other nondet stack frames left by the call to `r`.
4. The temporary stack frame whose `redoip` records the next alternative in `s`'s second disjunction.

```
:- pred p(int::in, int::out) is multi.

p(A, A).
p(A, O) :-
    q(A, B),
    ( O = B ; O = B + 1 ).
```

(a) Disjunctions are properly nested

```
:- pred s(int::in, int::out) is multi.

s(A, O) :-
    ( B = A ; B = A + 1 ),
    r(B, C),
    ( O = C ; O = C * 2 ).
```

(b) Disjunctions are not properly nested

Fig. 1. Mercury programs with nested disjunctions

A given call to a procedure that lives on the nondet stack will create exactly one ordinary frame and zero or more temporary frames on the nondet stack. The `redoip` slots of these frames will all point either to the failure handler in the Mercury runtime system, or to an instruction in the code of the procedure itself. The `redofr` slots of these frames will all point to the ordinary frame of the call, which contains all the call's variables, for use by the code `redoip` points to.

The stack slot assigned to a variable will contain garbage before the variable is generated. Afterward, it will contain the value of the variable, which may be an atomic value such as an integer or a pointer to the heap. Since the compiler knows the state of instantiation of every visible variable at every program point, the code it generates will never look at stack slots containing garbage. This means that backtracking does not have to reset variables to unbound, which in turn means that the Mercury implementation does not need a trail.

4 Tabling in Mercury

In all existing tabling systems, some predicates are designated as *tabled* by means of a declaration and use tabled resolution for their evaluation; all other predicates are *non-tabled* and are evaluated using SLD. Mercury also follows this scheme, but it supports three different forms of tabled evaluation: memoization (automatic caching), loop checking, and minimal model evaluation.

For predicates that can succeed at most once (i.e. whose determinism is det or semidet), the effect of tabling is similar to memoization in functional languages: when used judiciously, it can increase performance. The idea is to remember the first invocation of each call (henceforth referred to as a *generator*) and its computed result in tables (in a *call* and an *answer table* respectively), so that subsequent identical calls (referred to as the *consumers*) can use the remembered answer without repeating the computation. Mercury supports this form of tabling though the `'pragma memo'` declaration; Fig. 2(a) shows one such example.

Another use of tabling is for *loop detection*; see e.g. Fig. 2(b). If a predicate with a `'pragma loop_check'` declaration makes a recursive call to itself with identical input arguments, execution will throw an exception with a message about the infinite loop. Implementing both loop checking and memoization is relatively straightforward and is done with simplified variants of the transformation we present in Sect. 5. The challenge they pose is the design of efficient data structures for the tables. Due to space limitations, we will not further describe these forms of tabling in this paper.

The really interesting uses of tabling in (stratified) logic programs occur when one is interested in computing the answers of tabled predicate calls according to the *minimal model* semantics. In Mercury, programmers can request this form of tabled evaluation for a predicate using the `'pragma minimal_model'` declaration. An example (for some appropriate definition of `edge/2`) is the familiar `path` predicate of Fig. 3.

Predicates with `minimal_model` pragmas are required to satisfy two requirements not normally imposed on all Mercury predicates. The first requirement is that the set of values computed by the

```
:- func mfib(int) = int.
:- pragma memo(mfib/1).

mfib(N) = F :-
    ( N < 2 -> F = 1
    ; F = mfib(N-1) + mfib(N-2)
    ).
```

(a) Tabling purely for efficiency

```
:- pred return(int::in) is semidet.
:- pragma loop_check(return/1).

return(X) :- no_return(X).
return(42).

no_return(X) :- return(X).
```

(b) Tabling purely for avoiding loops

Fig. 2. Two types of tabling fully implemented in Mercury but not further considered in this paper

```
:- pred path(int::in, int::out) is nondet.
:- pragma minimal_model(path/2).

path(A, B) :- edge(A, B).
path(A, B) :- edge(A, C), path(C, B).
```

Fig. 3. Transitive closure predicate in Mercury using minimal model tabling

predicate for its output arguments is completely determined by the values of the input arguments. This means that the predicate must not do I/O; it must also be *pure*, i.e., free of side-effects such as updating the value of a global variable through the foreign function interface. The second restriction is that each argument of a minimal model predicate must be either fully input (ground at call and at return) or fully output (free at call, ground at return). In other words, partially instantiated arguments and arguments of unknown instantiation (a mode system extension used for constraint programming [1]) are not allowed. How this restriction affects the implementation of minimal model tabling in Mercury is discussed in the following section.

When a call to a minimal model predicate is made, a check must be made to see whether it exists in the call table or not. In the terminology of SLG resolution [3], this takes place using the NEW SUBGOAL operation. If the subgoal s is new, it is entered in the table and this call, as the subgoal's generator, will use PROGRAM CLAUSE RESOLUTION to derive answers. The generator will use the NEW ANSWER operation to record each answer it computes in a global data structure called the *answer table* of s. If, on the other hand, (a variant of) s already exists in the table, this call is a consumer and will resolve against answers from the subgoal's answer table. Answers are fed to the consumer one at a time through the ANSWER RETURN operation.

Because in general it cannot be known *a priori* how many answers a minimal model tabled call will get in its table, and because there can be mutual dependencies between (sets of) generators and consumers, correctly computing the minimal model requires:

1. a mechanism to retain (or reconstruct) and reactivate the execution environments of consumers until there are no more answers for them to consume, and
2. a mechanism for returning answers to consumers and determining when the evaluation of a (generator) subgoal is *complete*, i.e. when it has produced all its answers.

As mentioned, we chose the CAT suspension/resumption mechanism as the basis for Mercury's minimal model tabling implementation. However, we had to adapt it to Mercury and extend it in order to handle existential quantification and negated contexts in the manner described in the next section. For completion, we chose the *incremental completion* approach followed by XSB. A subgoal can be determined complete if all program clause resolution has finished and all instances of this subgoal have resolved against all derived answers. However, as there might exist dependencies between subgoals, these have to be taken into account by maintaining and examining (a conservative approximation of) the subgoal dependency graph, finding a set of subgoals that depend only on each other, completing them together, and then repeating the process until there are no incomplete subgoals. We refer to these sets of subgoals as *scheduling components*. The generator of one of the subgoals in the component (typically the oldest one) is called the component's *leader*.

5 The implementation of minimal model tabling in Mercury

First, we describe the implementation of minimal model evaluation for the subset of Mercury without if-then-else, negation or quantification; we will consider those constructs later.

5.1 The minimal model tabling transformation and tabling data structures

Mercury allows programmers to use impure constructs to implement a pure interface, simply by making a promise to this effect. The implementation of minimal model tabling exploits this capability. Given a pure predicate, like e.g., `path/2`, a compiler pass transforms its body by surrounding it with impure and semipure code; see Fig. 4 (impure code may write global variables; semipure code may only read them). Since the transformed code implements the same semantics as the original but is more complete (e.g., it terminates even for queries for which the original body could loop forever), the compiler promises that transformed code actually behaves as a pure goal.[3]

```
path(A, B) :-
  promise_pure (
    pickup_call_table_root_for_path_2(CallTableRoot),
    impure lookup_insert_int(CallTableRoot, A, CallNode1),
    impure subgoal_setup(CallNode1, Subgoal, Status),
    ( % switch on 'Status'
      Status = new,
      (
        impure mark_as_active(Subgoal),
        % original body of path/2 in the two lines below
        edge(A, C),
        ( C = B ; path(C, B) ),
        semipure get_answer_table(Subgoal, AnswerTableRoot),
        impure lookup_insert_int(AnswerTableRoot, B, AnswerNode1),
        impure answer_is_not_duplicate(AnswerNode1),
        impure new_answer_block(Subgoal, 1, AnswerBlock),
        impure save_answer(AnswerBlock, 0, B)
      ;
        impure completion(Subgoal),
        fail
      )
    ;
      Status = complete,
      semipure return_all_answers(Subgoal, AnswerBlock),
      semipure restore_answer(AnswerBlock, 0, B)
    ;
      Status = active,
      impure suspend(Subgoal, AnswerBlock),
      semipure restore_answer(AnswerBlock, 0, B)
    )
  ).
```

Fig. 4. The minimal model tabling program transformation illustrated on the predicate of Fig. 3

As mentioned, arguments of minimal model tabling must be either fully input or fully output. This considerably simplifies the implementation of call tables. SLG resolution considers two calls to represent the same subgoal if they are *variants* (i.e., are identical up to variable renaming). In Mercury, this is the case if and only if the two calls have the same ground terms in their input argument positions, because the output arguments of a call are always distinct variables. Conceptually, the call table of a predicate with n input arguments is a tree with $n + 1$ levels. Level 0 contains only the root node. Each node on level 1 corresponds to a value of the first input argument that the predicate has been called with; in general, each node on level k corresponds to a combination of the values of the first k input arguments that the predicate has been called with. Thus each node on level n uniquely identifies a subgoal. We store all the information we have about each subgoal in a *subgoal structure*. We reach the subgoal structure of a given subgoal through a pointer in the subgoal's level n node in the call table. The subgoal structure has several fields (cf. Fig. 5), which we will discuss as we go along:

[3] Although tabling modifies the state by updating globally accessible data structures (the tables), this is not an observable side-effect as Mercury provides no table inspection built-ins like XSB's `get_calls` and `get_returns`.

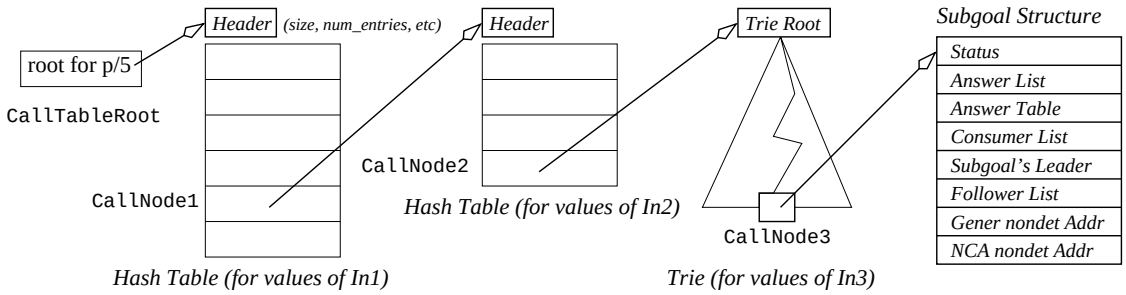


Fig. 5. Data structures created for the calls of predicate `p/5`

- the subgoal's status (*new*, *active* or *complete*)
- the chronological list of the subgoal's answers computed so far
- the root of the subgoal's answer table
- the list of the consumers of this subgoal
- the leader of the clique of subgoals this subgoal belongs to
- if this subgoal is the leader, the list of its followers
- the address of the generator's frame on the nondet stack
- the address of the nondet stack frame that is the youngest ancestor of both this generator and all its consumers; we call this the nearest common ancestor (NCA)

The transformed body of a minimal model predicate starts by looking up the call table to see whether this subgoal has been seen before or not. Given a predicate declared as:

```
:- pred p(int::in, string::in, int::out, t1::in, t2::out) is nondet.
:- pragma minimal_model(p/5).

p(In1, In2, Out1, In3, Out2) :-
    ...
```

the minimal model tabling transformation inserts the following code at the start of its procedure body:

```
pickup_call_table_root_for_p_5(CallTableRoot),
impure lookup_insert_int(CallTableRoot, In1, CallNode1),
impure lookup_insert_string(CallNode1, In2, CallNode2),
impure lookup_insert_user(CallNode2, In3, CallNode3),
impure subgoal_setup(CallNode3, Subgoal, Status)
```

`CallTableRoot`, `CallNode1`, `CallNode2` and `CallNode3` are all pointers to nodes in the call tree at levels 0, 1, 2 and 3 respectively; see Fig. 5. `CallTableRoot` points to the global variable generated by the Mercury compiler to serve as the root of the call table for this procedure. This variable is initialized to NULL, indicating no child nodes yet. The first call to `p/5` will cause `lookup_insert_int` to create a hash table in which every entry is NULL, and make the global variable point to it. `lookup_insert_int` will then hash `In1`, create a new slot in the indicated bucket (we currently use separate chaining to avoid fixed limits on the sizes of tables) and return the address of the new slot as `CallNode1`. At later calls, the hash table will exist, and by then we may have seen the then current value of `In1` as well; `lookup_insert_int` will perform a lookup if we have and an insertion if we have not. Either way, it will return the address of the slot selected by `In1`.

The process then gets repeated with the other input arguments. The predicates being called are different because Mercury uses different representations for different types. Unlike the WAM, the Mercury runtime system cannot look at the first argument of a hypothetical generic `lookup_insert` predicate and say "that's an integer" or "that's a string": a given bit pattern can be an integer in one call and a string in another. This affects how the hash table works: we hash integers directly but we hash the characters of a string, not its address.

User-defined types Values of user-defined types consist of a function symbol applied to zero or more arguments. In a strongly typed language such as Mercury, the type of a variable directly determines the set of function symbols that variable can be bound to. The data structure we use to represent a function symbol from user-defined types is therefore a trie. If the function symbol is a constant, we are done. If it has arguments, then `lookup_insert_user` processes them one by one the same way we process the arguments of predicates, using the slot selected by the function symbol to play the role of the root. In this way, the path in the call table from the root to a leaf node representing a given subgoal has exactly one trie or hash table on it for each function symbol in the input arguments of the subgoal; their order is given by a preorder traversal of those function symbols.

Polymorphic types This scheme works for monomorphic predicates because at each node of the tree, the type of the value at that node is fixed, and the type determines the mechanism we use to table values of that type (integer hash table, string hash table or float hash table for builtin types, a trie for user-defined types). For polymorphic predicates (whose signatures include type variables) the caller passes extra arguments identifying the actual types bound to those type variables [7]. We table these arguments first; they have a fixed type whose structure is similar to that of terms of user-defined types. Once we have followed the path from the root to the level of the last of these arguments, we have arrived at what is effectively the root of the table for a given monomorphic instance of the predicate's signature, and we proceed as described above.

5.2 The tabling primitives

The `subgoal_setup` primitive ensures the presence of the subgoal's subgoal structure. If this is a new subgoal, then `CallNode3` will point to a table node containing NULL. In that case, `subgoal_setup` will allocate a new subgoal structure, initializing its fields to reflect the current situation, it will update the table node pointed to by `CallNode3` to point to this new structure, and will return this same pointer as `SubgoalVar`. If this is not the first call to this procedure with these input arguments, then `CallNode3` will point to a table node that contains a pointer to the previously allocated subgoal structure, so `subgoal_setup` will just return this pointer.

`subgoal_setup` returns not just `SubgoalVar`, but also the subgoal's status. When first created, the status of the subgoal is set to *new*. It becomes *active* when a generator has started work on it and becomes *complete* once it is determined that the generator has produced all its answers.

What the transformed procedure body does next depends on the subgoal's initial status. If the status is *active* or *complete*, the call becomes one of the subgoal's consumers. If it is *new*, the call becomes the subgoal's generator and executes the original body of the predicate after changing the subgoal's status to *active*. When an answer is generated, we check whether we have derived this answer before. We do this by using `get_answer_table` to retrieve the root of the answer table from the subgoal structure, and inserting the output arguments into this table one by one, just as we inserted the input arguments one by one into the call table. The node on the last level of the answer table thus uniquely identifies this answer.

`answer_is_not_duplicate` looks up this node. If the tip of the answer table selected by the output argument values is NULL, then this is the first time we have computed this answer for this subgoal, and `answer_is_not_duplicate` succeeds. Otherwise it fails. (To make later calls fail, successful calls to `answer_is_not_duplicate` replace the tip with a non-NULL value.) We thus get to the call to `new_answer_block` only if the answer we just computed has not been seen before.

`new_answer_block` adds a new element to the end of the subgoal's chronological list of answers, the new element being a fresh new block with room for the given number of output arguments. The

call to `new_answer_block` is then followed by a call to `save_answer` for each output argument to fill in the slots of the answer block.

When the last call to `save_answer` returns, the transformed code of the tabled predicate succeeds. When backtracking returns control to the tabled predicate, it will drive the original predicate body to generate more and more answers. In programs with a finite minimal model, the answer generation will eventually stop, and execution will enter the second disjunct, which invokes the `completion` primitive. This will make the answers generated so far for this subgoal available to any consumers that are waiting for such answers. This may generate more answers for this subgoal if the original predicate body makes a call, directly or indirectly, to this same subgoal. The `completion` primitive will drive this process to a fixed point using the algorithm we describe in Sect. 5.5 and then mark the subgoal as *complete*. Having already returned all the answers of this subgoal from the first disjunct, execution fails out of the body of the transformed predicate.

If the initial status of the subgoal is *complete*, we call `return_all_answers`, which succeeds once for each answer in the subgoal's chronological list of answers. For each answer, the calls to `restore_answer` pick up the individual output arguments put there by `save_answer`.

If the initial status of the subgoal is *active*, then this call is a consumer but the generator is not known to have all its answers. We therefore call the `suspend` primitive. `suspend` has the same interface as `return_all_answers`, but its implementation is completely different. Whereas `return_all_answers` simply iterates through a pre-existing list of answer blocks, the implementation of `suspend` is quite complicated. We invoke the `suspend` primitive when we cannot continue computing along the current branch of the SLD tree. The main task of the suspension operation is therefore to record the state of the current branch of the SLD tree to allow its exploration later, and then simulate failure of that branch, allowing the usual process of backtracking to switch execution to the next branch. Sometime later, the `completion` primitive will restore the state of this branch of the SLD tree, feed the answers of the subgoal to it, and let the branch compute more answers if it can.

5.3 Suspension of consumers

The `suspend` primitive starts by creating a *consumer structure* and adding it to the current subgoal's list of consumers. The consumer structure has three fields: a pointer to this subgoal's subgoal structure (available in `suspend`'s `Subgoal` argument), an indication of which answers of the subgoal this consumer has consumed so far, and the saved state of the SLD branch of the consumer.

Making a copy of all the data areas of the Mercury abstract machine (det stack, nondet stack, heap and registers) would clearly be sufficient to record the state of the SLD branch, but equally clearly it would also be overkill. To minimize overhead, we want to record only the parts of the state that contain needed information which can change between the suspension of this SLD branch and any of its subsequent resumptions. For consumer suspensions, the preserved saved state is as follows.

Registers Of all the abstract machine registers used for parameter passing, the only one that contains live data is that containing `Subgoal`. The special purpose abstract machine registers (`maxfr`, `curfr`, the det stack pointer `sp`, and the return address register `succip`) do need to be part of the saved state.

Heap With Mercury's current conservative collector, heap space is recovered only by garbage collection and never by backtracking. This means that a term on the heap will naturally hang around as long as a pointer to it exists, regardless of whether that pointer is in a current stack or in a saved copy. Moreover, in the absence of instantiations of heap terms and destructive updates, this data will stay unchanged. This in turn means that, unlike a WAM-based implementation of CAT, Mercury's

implementation of minimal model tabling *does not need to save or restore any part of the heap*. (We argue for the correctness of doing so below.) In practice this is a big win, since it is typically the heap which is the largest area. The tradeoff is that we may need to save more data from the stacks, because the mapping from variables to values (the current substitution) is stored entirely in stack slots.

Stacks The way Mercury uses stack slots is a lot closer to the runtime systems of imperative languages than to the WAM. First of all, there are no links between variables because the mode system does not allow two free variables to be unified. So, binding a variable to a value affects only the stack slot holding the variable. Another difference with Prolog concerns the timing of parameter passing. If a predicate `p` makes the call `q(A)`, and the definition of `q` has a clause with head `q(B)`, then in Prolog, `A` would be unified with `B` at the time of the call, and any unification inside `q` that binds `B` would immediately update `A` in `p`'s stack frame. In Mercury, by contrast, there is no information flow between caller and callee except at call and (successful) return. At call, the caller puts the input arguments into abstract machine registers and the callee picks them up; at return, the callee puts the output arguments into those registers and the caller picks them up. Each invocation puts the values it picks up into a slot of its own stack frame when it next executes a call. The important point is that the only code that modifies a stack frame is the code of the predicate that created that stack frame.

CAT saves the frames on the stacks between the stack frame of the generator (excluded) and the consumer (included), and uses the WAM trail to save and restore addresses and values of variables which have been bound since the creation of a consumer's generator. (Only trail entries referring to the unsaved parts of the heap and local stack need to be preserved by copying. Saving those pointing to the saved parts is obviously redundant.) Mercury has no variables on its heap, but without a mechanism like the trail to guide the selective copying of stack slots which might change values, it must make sure that suspension saves information in *all* stack frames that could be modified between the suspension of a consumer and its resumption by its generator. The deepest frame on the nondet stack that this criterion requires us to save is the frame of the *nearest common ancestor* (NCA) of the consumer and the generator. We find the NCA by initializing two pointers to point to the consumer and generator stack frames, and repeatedly replacing whichever pointer is higher with the `succfr` link of the frame it points to, stopping when the two pointers are equal.

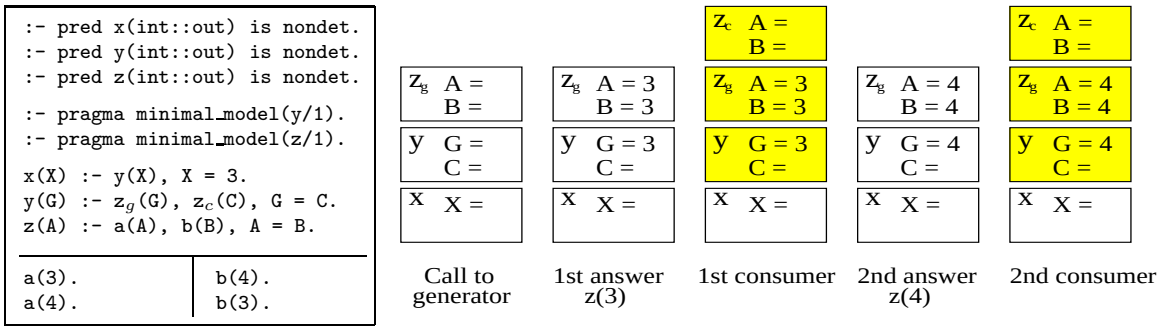
Note that we *must* save the stack frame of the NCA because the variable bindings in it may have changed between the suspension and the resumption. To see this, consider the code in Fig. 6, in which the first call to `z` in the body of `y` is a generator while the second `z` call is a consumer (hence their subscripts), and `y` is their nearest common ancestor.

- When $\texttt{z}_c$ first suspends, the value of `G` in `y`'s stack frame is 3.
- After $\texttt{z}_c$'s first suspension, the generator $\texttt{z}_g$ returns 4 to `y`, which stores it as the new value of `G`.
- When $\texttt{z}_g$ finishes generating its last answer, it resumes the consumer $\texttt{z}_c$. When $\texttt{z}_c$ returns, its return value `C`, is tested for equality with the stored value of `G`.
- The stored value of `G` in `y` will be 4, unless the resumption of $\texttt{z}_c$ restores `y`'s stack frame to the state it had when the consumer was suspended.

Note that it is also possible for the nearest common ancestor of the generator and consumer to be a procedure that lives on the det stack. Consider for example

```
p(A, B) :- ( if some [X, Y] ( q(A, X), q(A, Y), X = 2 * Y ) then B = yes else B = no ).
```

where `p` is det while `q` is a minimal model nondet procedure. In such cases, the algorithm we gave above for finding the NCA will actually find the stack frame that was on top of the nondet stack when the det stack frame of the NCA (in this case `p`) was created. Since `p` may not know whether `q` is tabled,



(a) Tabled program in Mercury

(b) Nondet stack while executing `x(X)`; shaded frames are those copied

Fig. 6. Information on the stack (growing upwards) while executing the program on the left.

we cannot make p perform any special action to mark its stack frame; not without slowing down all procedures that live on the det stack. We prefer instead to slow down all procedures that live on the nondet stack. Since in our experience there are many fewer such procedures, this is the lesser of the two evils. In minimal model grades we extend each nondet stack frame with an extra slot holding the value of the det stack pointer at the time when the nondet stack frame was created. By saving everything on the det stack above the stack pointer value recorded in the computed, nondet NCA's stack frame, we guarantee that we will save the stack frame of the real, det NCA.

Correctness of saved information from the stacks The variables in the nondet stack frames below the NCA and in the det stack frame the NCA points to fall into two classes. The variables that were bound when the NCA call was made cannot have their bindings changed unless the goal that made the binding fails. This can happen only if everything to their right has also failed. Since the call to NCA and thus the generator are to the right of this code, the generator must have failed. This can happen in only two circumstances. The first is that the generator has computed all the answers to its subgoal and has fed them all back to their consumers. In that case, none of those consumers will ever be restored; in fact, their saved states will have been deleted. The second circumstance is that the generator is simulating failure because it has yielded to another generator which is now its leader. In such situations, which we will discuss in the next section, we do in fact save beyond this NCA (we save to the nearest common ancestor of the consumer and the leader generator.)

On the other hand, the variables that were not yet bound when the NCA call was made may be bound after the NCA succeeds, but if backtracking ever returns control to the NCA and through it to the consumer, those bindings will have been implicitly unbound by then.

Therefore, whether or not a variable in a nondet stack frame below the NCA is bound when the NCA is called, we do *not* need to save its value when suspending a consumer. Since the contents of the fixed slots in those frames do not need to be saved either, the segment of the nondet stack that we need to save ends at the nearest common ancestor.

Correctness of not saving information from the heap The only two ways that a heap term pointed to from a saved copy of a stack frame could be changed between the saving of that stack frame and its restoration are variable instantiation and destructive update. We rule out both of these through the use of *grades*. A grade is a set of settings for the compilation options that modify the Mercury abstract machine in some fashion; for example, the minimal model tabling grade adds some new stacks (as we will see later). The Mercury implementation ensures that all parts of the program are compiled with the same grade. In minimal model grades, compilation options and program constructs that could result in non-ground terms on the heap (required for plugging in constraint solvers [1]) are disallowed, as are optimizations that destructively update ground heap terms.

5.4 Maintenance of subgoal dependencies and their influence on suspensions

We have described suspension as if consumers will be scheduled only by their nearest generator. This is indeed the common case, but as explained in Sect. 4 there are also situations in which subgoals are mutually dependent and cannot be completed on an individual basis. To handle such cases, Mercury maintains a stack-based approximation of dependencies between subgoals, in the form of scheduling components. For each scheduling component (a group subgoals that may depend on each other), its *leader* is the youngest generator G_L for which all consumers younger than G_L are consumers of generators that are not older than G_L.

Of all scheduling components, the one of most interest is that on the top of the stack. This is because it is the one whose consumers will be scheduled first. We call its leader the *current leader*.

The maintenance of scheduling components is reasonably efficient. Information about the leader of each subgoal and the leader's *followers* is maintained in the subgoal structure (cf. Fig. 5). Besides creation of a new generator (in which case the generator becomes the new current leader with no followers), this information possibly changes whenever execution creates a consumer suspension. If the consumer's generator, G, is the current leader or is younger than the current leader, no change of leaders takes place. If G is older than the current leader, a *coup* happens, G becomes the current leader, and its scheduling component gets updated to contain as its followers the subgoals of all generators younger than G. In either case, the saved state for the consumer suspension will be till the NCA of the consumer and the current leader. This generalizes the scheme described in the previous section.

However, because a coup can happen even after the state of a consumer has been saved, we also need a mechanism to extend the saved consumer states. The mechanism we have implemented is simple and consists of extending the saved state of all consumers upon change of leaders. When a coup happens, the saved state of all followers (consumers and generators) of the old leader is extended to the stack frame of the NCA of each follower and the new leader. (The NCA point up to which followers have copied their state is maintained in the consumer and subgoal structures.) Unlike CAT which tries to share the trail, heap, and local stack segments it copies [6], in Mercury we have not (yet) implemented a mechanism to share the copied stack segments. Note, however, that the space problem is not as severe in Mercury as it is in CAT, because in Mercury there is no trail and no information from the heap is ever copied, which means that heap segments for consumers are naturally "shared".

On failing back to a generator which is a leader, scheduling of answers to all its followers will take place, as described below. When the scheduling component gets completed, execution will continue with the immediately older scheduling component whose leader will then become the current leader.

5.5 Resumption of consumers and completion

The main body of the `completion` primitive consists of three nested loops: over all subgoals in the current scheduling component $\mathcal{S}$, over all consumers of these subgoals, and over all answers to be returned to those consumers. The code in the body of the nested loop arranges for the next unconsumed answer to be returned to a consumer of a subgoal in $\mathcal{S}$. It does this by restoring the stack segments saved by the `suspend` primitive, putting the address of the relevant answer block into the abstract machine register assigned to the return value of `suspend`, restoring the other saved abstract machine registers, and branching to the return address stored in `suspend`'s stack frame. Each consumer resumption thus simulates a return from the call to `suspend`.

Since restoring the stack segments from saved states of consumers clobbers the state of the generator that does the restoring (the leader of $\mathcal{S}$), the `completion` primitive first saves the leader's own state, which consists of saving the nondet stack down to the oldest NCA of the leader generator and any of the consumers it schedules, and saving the det stack to the point indicated by this nondet frame.

Resumption of a consumer essentially restores the saved branch of the SLD search tree, but restoring its saved stack segments *intact* is not a good idea. The reason is that leaving the `redoip` slots of the restored nondet stack frames unchanged resumes not just the saved branch of the SLD search tree, but also the departure points of all the branches going off to its right. Those branches have been explored immediately after the suspension of the consumer, because suspension involves simulating the failure of the consumer, thus initiating backtracking. When we resume the consumer to consume an answer, we do not want to explore the exact same alternatives again, since this could lead to an arbitrary slowdown. We therefore replace all the `redoip`s in saved nondet stack segments to make them point to the failure handler in the runtime system. This effectively cuts off the right branches, making them fail immediately. Given the choice between doing this pruning once when the consumer is suspended or once for each time the consumer is resumed, we obviously choose the former.

This pruning means that when we restore the saved state of a consumer, only the success continuations are left intact, and thus the only saved stack frames the restored SLD branch can access are those of the consumer's ancestors. Any stack frames that are not the consumer's ancestors have effectively been saved and restored in vain. The advantage of this approach is that the code to save and restore stack segments is quite fast (has a low constant factor). In addition, the direct correspondence between the original and saved stack copies makes the code of the pruning operation much easier to write compared to an approach that would try to remove those unnecessary frames. For the programs we have looked at so far, our approach looks to be the right tradeoff.

When a resumed consumer has consumed all the currently available answers, it fails out of the restored segment of the nondet stack. We arrange to get control when this happens by setting the `redoip` of the very oldest frame of the restored segment to point to the code of the `completion` primitive. When `completion` is re-entered in this way, it needs to know that the three-level nested loop has already started and how far it has gone. We therefore store the state of the nested loop in a global record. When this state indicates that we have returned all answers to all consumers of subgoals in $\mathcal{S}$, we have reached a fixed point. At this time, we mark all subgoals in $\mathcal{S}$ as *complete* and we reclaim the saved states of all their consumers and generators.

5.6 Existential quantification

Mercury supports existential quantification. This construct is usually used to check whether a component of a data structure possesses a specific property as in the code fragment below:

```
( if some [Element] ( member(Element, List), test(Element) ) then ... else ... )
```

It is typically the case that the code inside the quantification's scope may have more than one solution, but the code outside the quantification only wants to check whether a solution *exists* without caring about the number of solutions or their bindings. One can thus convert a goal that can succeed more than once into one that can succeed at most once by existentially quantifying all its output variables.

Mercury implements quantifications of that form using what we call a *commit* operation, which some Prologs call a *once* operation. The operation saves `maxfr` when it enters the goal and restores it afterward, throwing away all the stack frames that have been pushed onto the nondet stack in the meantime. The interaction with minimal model tabling arises from the fact that the discarded stack frames can include the stack frame of a generator. If this happens, the commit removes all possibility of the generator being backtracked into ever again, which in turn may prevent the generation of answers and completion of the corresponding subgoal. Without special care, all later calls of that subgoal will become consumers who will wait forever for the generator to schedule the return of their answers.

To handle such situations, we introduce of a new stack which we call the *cut stack*. This stack always has one entry for each currently active existentially quantified goal; new entries are pushed onto it when such a goal is entered and popped when that goal either succeeds or fails. Each entry contains a pointer to a list of generators. Whenever a generator is created, it is added to the list in the entry on the current top of the cut stack. When the goal inside the commit succeeds, the code that pops the cut stack entry checks its list of generators. For all generators whose status is not *complete*, we erase all trace of their existence and reset the call table node that points to the generator's subgoal structure back to a null pointer. This allows later calls to that subgoal to become new generators.

If the goal inside the commit fails, the failure may have been due to the simulated failure of a consumer inside that goal. When the state of the consumer is restored, it may well succeed, which means that any decision the program may have taken based on the initial failure of the goal may be incorrect. When the goal inside the commit fails, we therefore check whether any of the generators listed in the cut stack entry about to be popped off have a status other than *complete*. Any such generator must have consumers whose failure may not be final, so we throw an exception in preference to computing incorrect results. Note that this can happen only when the leader of the incomplete generator's scheduling component is outside the existential quantification.

5.7 Possibly negated contexts and aggregates

The interaction of tabling with cuts and Prolog-style negation is notoriously tricky. Many implementation papers on tabling ignore the issue altogether, considering only the definite subset of Prolog. An implementation of tabling for Mercury cannot duck the issue. Typical Mercury programs rely extensively on if-then-elses, and if-then-elses involve negation: "`if` C `then` T `else` E" is semantically equivalent to $(C \wedge T) \vee (\neg\exists C \wedge E)$. Of course, operationally the condition is executed only once. The condition C is a possibly negated context: it is negated only if it has no solutions. Mercury implements if-then-else using a *soft cut*: if the condition succeeds, it cuts away the possibility of backtracking to the else part only.

If C fails, execution should continue at the else part of the if-then-else. This poses a problem for our implementation of tabling, because the failure of the condition does not necessarily imply that C has no solution: it may also be due to the suspension of a consumer called (directly or indirectly) somewhere inside C as in the code below.

```
p(...) :- t_g(...), ( if ( ..., t_c(...), ... ) then ... else ... ), ...
```

If $\texttt{t}_c$ suspends and is later resumed to consume an answer, the condition may evaluate to true. However, by then the damage will have been done, because we will have executed the code in the `else` part.

We have not yet implemented a mechanism that will let us compute the correct answer in such cases, because any such mechanism would need the ability to transfer the "generator-ship" of the relevant subgoal from the generator of `t` to its consumer. However, we *have* implemented a mechanism that guarantees that incorrect answers will not be computed. This mechanism is the *possibly-negated-context stack*, or *pneg stack* for short. We push an entry onto this stack when entering a possibly negated context such as the condition of an if-then-else. The entry contains a pointer to a list of consumers, which is initially empty. When creating a consumer, we link the consumer into the list of the top entry on the pneg stack. When we enter the else part of the if-then-else, we search this list looking for consumers that are suspended. Since suspension simulates failure without necessarily implying the absence of further solutions, we throw an exception if the search finds such a consumer. If not, we simply pop the entry of the pneg stack. We also perform the pop on entry to the then

part of the if-then-else. Since in that case there is no risk of committing to the wrong branch of the if-then-else, we do so without looking at the popped entry.

There are two other Mercury constructs that can compute wrong answers if the failure of a goal does not imply the absence of solutions for it. The first is negation. We handle negation as a special case of if-then-else: $\neg G$ is equivalent to "`if` G `then fail else true`". The other is the generic all-solutions primitive `builtin_aggregate`, which serves as the basic building block for all the user-visible all-solutions predicates, such as `solutions/2`. `builtin_aggregate` itself is implemented as an impure Mercury predicate based on the idea of the failure-driven loop shown in Fig. 7(a).

```
builtin_aggregate(P, ..., Answers) :-
  impure ⟨setup code⟩
  (
    P(Answer),
    impure ⟨record Answer⟩
    fail
  ;
    impure ⟨pickup all Answers⟩
  ).
```

(a) In non minimal model grades

```
builtin_aggregate(P, ..., Answers) :-
  impure ⟨setup code⟩
  impure ⟨push pneg stack entry⟩                    %(1)
  ( P(Answer),
    impure ⟨record Answer⟩
    fail
  ; impure ⟨check for suspensions, pop pneg stack⟩  %(2)
    impure ⟨pickup all Answers⟩
  ).
```

(b) In minimal model grades

Fig. 7. Implementation of aggregates without and with support for minimal model tabling

To guard against `builtin_aggregate` mistaking the failure of `P` due to a suspension somewhere inside it as implying the absence of solutions to `P`, we treat its loop as the condition of an if-then-else. We surround it with the code we normally insert at the start of the condition *(1)* and the start of the else part *(2)* in Fig. 7(b). This gives all-solutions predicates the same protection as if-then-elses.

Entries on both the cut stack and the pneg stack contain a field that points to the stack frame of the procedure invocation that created them, which is of course also responsible for removing them. When saving stack segments or extending saved stack segments, we save an entry on the cut stack or the pneg stack if the nondet stack frame they refer to is in the saved segment of the nondet stack.

6 Performance evaluation

We ran several benchmarks to measure the performance of Mercury with tabling support, but space limitations allow presenting only some of them here; some more information appears in the appendix.

Overhead of the minimal model grade We compiled the Mercury compiler in two grades that differ only in that one supports minimal model tabling. Enabling support for minimal tabling without using it (the compiler has no minimal model predicates) increases the size of the compiler executable by about 5%. On the standard benchmark task for the Mercury compiler, compiling six of its own largest modules, moving to a minimal model grade slows the compiler down by about 25%. (For comparison, enabling debugging leads to a 455% increase in code size and a 135% increase in execution time.) Our analysis shows that virtually all of this cost in both space and time is incurred by the extra code we have to insert around possibly negated contexts; the extra code around commits and the larger size of nondet stack frames have no measurable overheads. For details, see the appendix.

While a 25% hit on execution time may seem a lot, one must remember that a given amount of overhead added to a fast system will result in a larger percentage slowdown than the same absolute overhead added to a slower one. The performance results in [16] indicate that Mercury is faster than other Prolog systems by integer factors, for real programs as well as benchmarks. Even with this 25% overhead, Mercury is still much faster than any current Prolog system.

Table 1. Times (in secs) to execute various versions of transitive closure and same generation

				chain			cycle		
bench	size	query	iter	XSB	XXX	Merc	XSB	XXX	Merc
tc_lr	2000	+-	200	0.24	0.20	0.22	0.23	0.19	0.27
tc_lr	4000	+-	200	0.50	0.40	0.58	0.50	0.40	0.55
tc_lr	8000	+-	200	1.02	0.83	1.25	1.01	0.85	1.19
tc_lr	16000	+-	200	2.04	1.73	2.11	2.08	1.66	2.30
tc_lr	32000	+-	200	4.18	3.59	4.97	4.20	3.57	5.07
tc_lr	64000	+-	200	8.60	7.30	9.15	8.78	7.44	9.34
tc_lr	2000	--	1	2.18	2.04	2.93	5.00	4.83	5.63
tc_rr	2000	--	1	1.71	1.46	9.33	5.31	5.23	25.22

bench	size	query	iter	XSB	XXX	Merc
sg	$24 \times 24 \times 2$	+-	10	14.34	8.60	5.64
sg	$24 \times 24 \times 2$	--	10	41.51	24.01	15.96

Comparison against other implementations of tabling We compare the minimal model grade of Mercury rotd-09-05-2005 (based on CAT) against XSB 2.7.1 (based on the SLG-WAM) and the XXX system (derived from XSB but based on CHAT).[4] Mercury's scheduling strategy is *batched*-like, while the other two systems use *local scheduling* [13] by default. All benchmarks were run on a 2.4 GHz P4-based laptop with 512 Mb of memory running Linux. Times in Table 1 were obtained by running the benchmark at least eight times, discarding the lowest and highest values, and averaging the rest.

The first set of benchmarks consists of left- and right-recursive versions of transitive closure. In each case, the edge relation is a chain or a cycle. In a chain of size n, there are $n - 1$ edges of the form $k \rightarrow k + 1$ for $0 \leq k < n$; in a cycle of size n, there is also an edge $n \rightarrow 0$. We use two query forms: the query with the first argument input and the second output (+-) and the open query with both arguments output (--). The number of solutions is linear in the size of the data for the +- query and quadratic for --. Each entry in Table 1 shows how long it takes for a given system to run the specified query on the specified data iter times. Each system uses a failure driven loop or its equivalent to generate all answers of the query. The tables are reset between iterations.

The six rows for the +- query on left recursive transitive closure show the runtimes of all three systems to be linear in the size of the data, which is as expected. Also, on left recursion, regardless of query, all three systems are pretty much on the same performance ballpark. On right recursion, Mercury is slower than the other two systems due to saving and restoring stack segments of consumers, and having to do so more times due to its different scheduling strategy.

In some sense, it is unfortunate that not all systems implement the same scheduling strategy. However, local evaluation (i.e., postponing the return of answers to the generator until the subgoal is complete) is not compatible with the pruning that Mercury's execution model requires in existential and possibly negated contexts, constructs not properly handled in Prolog systems with tabling.

On the same generation (sg) benchmark where consumer suspensions are not created (variant subgoals are only encountered when the subgoals are completed), Mercury is clearly the fastest system.

It is very difficult to draw general conclusions from these synthetic tabled benchmarks (also notice they are all Datalog programs), but the following observations can safely be made:

1. Mercury has opted to pay a cost for suspension/resumption to avoid penalizing non-tabled execution;
2. Mercury can be a lot faster that other tabled Prolog systems in programs where only few consumer suspensions are encountered;
3. How faster Mercury is on *real* tabled programs depends to a great extent on the number of consumer suspensions and on employing a scheduling strategy that can avoid some of them.

[4] We also wanted to include Yap in the comparison, but in running the benchmarks with Yap 4.4.4, we experienced some problems we do not yet understand. We expect that we will include Yap numbers in the final version of the paper.

We intend to explore the latter issue as future work.

7 Concluding remarks

Adapting the implementation of minimal model tabling to Mercury has been a challenge because the Mercury abstract machine is very different from the WAM. We have based our implementation on CAT because it is the only recomputation-free approach to tabling that does not make assumptions that are invalid in Mercury. However, even CAT required significant modifications to work properly with Mercury's stack organization, its mechanisms for managing variable bindings, and its type-specific data representations. We have described all these in this paper as well as describing two new mechanisms, the cut and the pneg stack, which allow for safe interaction of tabling with language constructs such as if-then-else and existential quantification. Finally, note that Mercury also provides new opportunities for optimization of tabled programs. For example, the strong mode system greatly simplifies variant checking and the type system allows for a type-tailored design of tabling data structures.

In keeping with Mercury's orientation towards industrial-scale systems, our design objective was maximum performance on large programs containing some tabled predicates, not maximum performance on the tabled predicates themselves. The distinction matters, because it requires us to make choices that minimize the impact of tabling on non-tabled predicates even when these choices slow down tabled execution. We have been broadly successful in achieving this objective. Since supporting minimal model tabling is optional, programs that do not use it are not affected at all. Even in programs that do use tabling, non-tabled predicates only pay the cost of one new mechanism: the one ensuring the safety of interactions between minimal model tabling and negation. Even with this cost, Mercury is much faster than any Prolog implementation that supports tabling.

The results on microbenchmarks focusing on the performance of minimal model tabled predicates themselves show Mercury to be quite competitive with existing tabling systems. It is faster on some benchmarks, slower on some others, and quite similar on the rest, even though Mercury currently lacks some obvious optimizations, such as sharing stack segment extensions among consumers. How the system behaves on real tabled applications, written in Mercury rather than Prolog, remains to be seen. But one should not underestimate neither the difficulty nor the importance of adding proper tabling to a high-performance LP system and the power that this brings to the system.[5]

Acknowledgements We thank Bart Demoen for discussions on the copying approach to tabling for Mercury at the start of this work, Oliver Hutchinson for his work on the infrastructure of tabling in Mercury, and Michael Day for the original version of the program in Fig. 6. The research of the first author has been partially supported by the Australian Research Council and by Microsoft. The research of the second author has been partially supported by the Swedish Research Council.

References

1. R. Becket, M. Garcia de la Banda, P. Stuckey, K. Marriott, M. Wallace, and Z. Somogyi. Adding constraint solving to Mercury. Submitted to ICLP 2005, May 2005.
2. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, Jan. 1996.

[5] The system we have described is available in the next Mercury release (0.12) as well as in recent releases-of-the day from the Mercury web site.

4. M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. *Springer International Journal of Software Tools for Technology Transfer*, 2(1):29–45, Nov. 1998.
5. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126. ACM Press, May 1996.
6. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. *J. of Functional and Logic Program.*, Nov. 1999.
7. T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run time type information in Mercury. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, pages 224–243, Sept. 1999.
8. H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer, Nov./Dec. 2001.
9. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
10. F. Henderson and Z. Somogyi. Compiling Mercury to high-level C code. In N. Horspool, editor, *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France, Apr. 2002. Springer-Verlag.
11. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification*, volume 1254 of *LNCS*, pages 143–154. Springer, July 1997.
12. R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Program.*, 5(1 & 2):161–205, Jan. 2005.
13. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Prog. Lang. Syst.*, 20(3):586–634, May 1998.
14. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, May 1994.
15. T. Sato and Y. Kameya. Statistical abduction with tabulation. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *LNCS*, pages 567–587. Springer, 2002.
16. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. of Logic Program.*, 26(1–3):17–64, Oct./Dec. 1996.
17. H. Tamaki and T. Sato. OLD resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98. Springer-Verlag, July 1986.
18. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, Oct. 1983.
19. N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Implementation of a linear tabling mechanism. *J. of Functional and Logic Program.*, 2001(10), 2001.

A More information on the benchmarks

When the paper is published, the information in this appendix will be available from the Mercury web site, from a link right next to the link to this paper.

This set of benchmarks attempts to analyze the source of the difference between the performance of the `asm_fast.gc` and `asm_fast.gc.mm` grades. Each block in Table 2 identifies the parameters with which the given version of the Mercury compiler was built. The final number gives the time (in seconds) it took that version of the compiler to compile six files, which happen to be six of the largest modules of the compiler itself, totalling 34,426 lines of code. This time was obtained by getting each tested version of the compiler to compile those six files twelve times, discarding the highest and lowest times, and averaging the other ten times.

The unmodified `asm_fast.gc` configuration is listed as version 1, while the unmodified `asm_fast.gc.mm` configuration is version 7. The other five versions are in between: they have some of the extra overheads incurred by the `asm_fast.gc.mm` version, but not all of them. We therefore expect version 1 to be the fastest and version 7 to be the slowest. That this is not so is due to cache effects, which make it possible for a version of a program that executes a strict subset of the instructions executed by another version of the program to nevertheless take more time to run.

Table 2. The Mercury compiler compiled and run in different configurations: performance results

#	How obtained	Size of executable	Time

1	`EXTRA_MCFLAGS =`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc`
		8,928,116	49.41

2	`EXTRA_MCFLAGS =`
	`EXTRA_CFLAGS = -DMR_USE_MINIMAL_MODEL_STACK_COPY_EXTRA_SLOT`
	`GRADE = asm_fast.gc`
		8,928,724	48.53

3	`EXTRA_MCFLAGS = --no-allow-hijacks`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc`
		8,924,020	48.62

4	`EXTRA_MCFLAGS = --disable-mm-pneg`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc.mm`
		8,940,648	48.69

5	`EXTRA_MCFLAGS = --disable-mm-cut`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc.mm`
		9,338,248	62.56

6	`EXTRA_MCFLAGS = --disable-mm-pneg --disable-mm-cut`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc.mm`
		8,936,552	48.83

7	`EXTRA_MCFLAGS =`
	`EXTRA_CFLAGS =`
	`GRADE = asm_fast.gc.mm`
		9,342,344	61.95

Version 2 differs from version 1 in making each frame on the nondet stack contain an extra slot, which is filled in from the current value of the det stack pointer. This does not add any measurable overhead (in fact it gets a speedup).

Version 3 differs from version 1 in not allowing the compiler to employ the optimization of using the `redoip` slot of a nondet stack frame to record the current state of several nested disjunctions; in version 3, every disjunction, even nested ones, will have a temporary nondet stack frame created for it. Minimal model tabling needs this to happen because it needs to see the state of every disjunction in the `redoip` slot of some frame on the nondet stack to allow it clobber that state when pruning right branches of the SLD tree. In the presence of the optimization, this is not possible. Disabling the optimization does not add any measurable overhead (in fact it gets a speedup).

Version 4 is in the `asm_fast.gc.mm` grade, but differs from version 7 in not generating code for pushing entries on the nondet stack when entering possibly negated contexts and popping them off and inspecting them when leaving such contexts. This gets a huge savings compared to version 7, and does not add any measurable overhead compared to version 1 (in fact it is slightly faster than version 1). This proves that pretty much all of the overhead of minimal model tabling for this program (the Mercury compiler) is in the handling of pneg contexts.

Version 5 is also in the `asm_fast.gc.mm` grade, but differs from version 7 in not generating code for pushing entries on the cut stack when entering commit contexts and popping them off and inspecting them when leaving such contexts. This does not get any saving to version 7 (in fact it is slightly slower).

Version 6 is also in the `asm_fast.gc.mm` grade, but differs from version 7 in missing both the code dealing with pneg contexts and the code dealing with commit contexts. The result is essentially identical to version 4, which shows that commit contexts have no measurable impact on performance.

The story is similar with respect to executable size. The `asm_fast.gc.mm` version of the compiler (version 7) is 414 Kb larger than the `asm_fast.gc` version (version 1). Of this increase, only about 12 Kb remains in version 4, which shows that all the rest is due to the code surrounding possibly negated contexts.