Tabling in Mercury: Design and Implementation

Zoltan Somogyi[1] and Konstantinos Sagonas[2]

[1] NICTA Victoria Laboratory,
Department of Computer Science and Software Engineering, University of Melbourne, Australia
[2] Department of Information Technology, Uppsala University, Sweden
zs@csse.unimelb.edu.au kostis@it.uu.se

Abstract. For any LP system, tabling can be quite handy 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 is acceptable in a Prolog system, it is not acceptable in Mercury, since if-then-elses and existential quantification have sound semantics for stratified programs 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. 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 tabling in Mercury.

1 Introduction

By now, it is widely recognized that tabling adds power to logic programming. 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. As a result, many Prolog systems (e.g., XSB, YAP, and B-Prolog) nowadays offer some form of tabling. Mercury is a language with an efficient implementation and comes with a module and a strong type system that ease the development of industrial-scale applications. Like Prolog systems with tabling, Mercury aims to encourage a more declarative style of programming than "plain" Prolog. This paper discusses implementation aspects of adding tabling to Mercury.

When deciding which tabling mechanism to adopt, an implementor is faced with several choices. Linear tabling strategies [11, 3] are relatively easy to implement (at least for Prolog), but they are also relatively ad hoc and often perform recomputation. Tabled resolution strategies such as OLDT [9] and SLG [1] are guaranteed to avoid recomputation, but their implementation is challenging because they require the introduction of a suspension/resumption mechanism into the basic execution engine.

In the framework of the WAM [10], currently there are two main techniques to implement suspension/resumption. The one employed both in XSB and in YAP [5], that of the SLG-WAM [6], implements suspension via *stack freezing* and resumption using an extended trail mechanism called the *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 [2], 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. We decided to restrict the implementation to the subset of SLG that handles stratified programs. We chose CAT as implementation platform, because the alternatives conflict with basic assumptions of the Mercury implementation. For example, Mercury has no trail to freeze, let alone a forward one, and freezing the stack *à la* SLG-WAM breaks Mercury's invariant that calls to deterministic predicates leave the stack unchanged. CAT is simply 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 program components that do not use tabling; given an appropriate static analysis, the overhead can be completely eliminated.

This paper documents the implementation of tabling in Mercury (we actually aim to compute a specific minimal model of stratified programs: the perfect model). We describe how we adapted the CAT (Copying Approach to Tabling) mechanism to a different implementation technology, one which is closer to the execution model of conventional languages than the WAM, and present the additional optimizations that can be performed when tabling is introduced in such an environment. Finally, 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.

The next section reviews Mercury and its implementation. Section 3 introduces tabling in Mercury, followed by the paper's main section (Section 4) which describes the implementation of tabling in detail. A brief performance comparison with other Prolog systems with tabling implementations based on SLG resolution appears in Section 5.

2 A Brief Introduction to Mercury

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. 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 the Mercury language can be found at http://www.cs.mu.oz.au/research/mercury/information/doc-latest/mercury_ref.

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 Section 4) before being passed on to a backend for code generation.

The Mercury compiler has several backends. So far, tabling is implemented only for the original backend which generates low level C code [7], because it is the only one that allows us to explicitly manipulate stacks (see Section 4.3). The abstract machine targeted by this low level backend has three main data areas: a heap and two stacks. The heap is managed by the Boehm-Demers-Weiser conservative garbage collector for C. Since this collector was not designed for logic programming systems, it does not support any mechanism to deallocate all the memory blocks allocated since a specific point in time. Thus Mercury, unlike Prolog, does not recover memory by backtracking and recovers all memory blocks via garbage collection.

The Mercury abstract machine has two stacks: 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 thus 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 in strict LIFO fashion.

Procedures that can succeed more than once have their stack frames allocated on the nondet stack. These 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. 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 [10]. 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 five fixed slots are:

`prevfr` The previous frame slot points to the stack frame immediately below this one. (Both stacks grow higher.)

`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 of 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.

The compiler converts multi-clause predicate definitions into disjunctions. When executing in the code of a disjunct, the `redoip` slot points to the first instruction of the next disjunct or, if this is the last disjunct, to the address of the failure handler whose 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 address in its `redoip` slot. Disjunctions other than the outermost one are implemented using temporary nondet stack frames, which have only `prevfr`, `redoip` and `redofr` slots [8].

The stack slot assigned to a variable contains garbage before the variable is instantiated; afterward, it contains the value of the variable. 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.

3 Tabling in Mercury

In tabling systems, some predicates are declared *tabled* 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 (caching), loop checking, and minimal model evaluation. We concentrate on the last form, which is the most interesting and subsumes the other two.

The idea of tabling is to remember the first invocation of each call (henceforth referred to as a *generator*) and its computed results in tables (in a *call table* and an *answer table* respectively), so that subsequent identical calls (referred to as the *consumers*) can use the remembered answers without repeating the computation. Mercury programmers who are interested in computing the answers of tabled predicate calls according to the *perfect model* semantics can use the `'minimal_model'` pragma. An example is the usual `path` predicate on the right.

```
:- 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).
```

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 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 observable side-effects such as updating the value of a global variable through the foreign function interface. The second 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 are not allowed. How this restriction affects the implementation of tabling in Mercury is discussed in the following section.

When a call to a minimal model predicate is made, the program must check whether the call exists in the call table or not. In SLG terminology [1], 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 ANSWER RETURN operations.

Because in general it is not known *a priori* how many answers a tabled call will get in its table, and because there can be mutual dependencies between generators and consumers, the implementation requires: (a) a mechanism to retain (or reconstruct) and reactivate the execution environments of consumers until there are no more answers for them to consume, and (b) 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 tabling implementation. However, we had to adapt it to Mercury and extend it in order to handle existential quantification and negated contexts. For completion, we chose the *incremental completion* approach described in [6]. A subgoal can be determined to be 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 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 some subgoal (typically the oldest) in the component is called the component's *leader*.

4 The Implementation of Tabling in Mercury

4.1 The tabling transformation and its supporting data structures

Mercury allows programmers to use impure constructs to implement a pure interface, simply by making a promise to this effect. The tabling implementation exploits this capability. Given a pure predicate such as `path/2`, a compiler pass transforms its body by surrounding it with impure and semipure code as shown in Fig. 3 (impure code may write global variables; semipure code may only read them). Note that the compiler promises that the transformed code behaves as a pure goal, since the side-effects inside are not observable from the outside.

As mentioned, the arguments of tabled procedures 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., 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.

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 in the code shown in Fig. 1, the minimal model tabling transformation inserts the code shown on the same figure at the start of its procedure body.

```
:- 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) :-
    ...
```

```
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)
```

Fig. 1. Type-directed program transformation for arguments of tabled calls

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 the following eight fields (cf. Fig. 2), which we will discuss as we go along: 1) the subgoal's status (*new, active* or *complete*); 2) the chronological list of the subgoal's answers computed so far; 3) the root of the subgoal's answer table; 4) the list of the consumers of this subgoal; 5) the leader of the scheduling component this subgoal belongs to; 6) if this subgoal is the leader, the list of its followers; 7) the address of the generator's frame on the nondet stack; and 8) the address of the youngest nondet stack frame that is an ancestor of both this generator and all its consumers; we call this the nearest common ancestor (NCA).

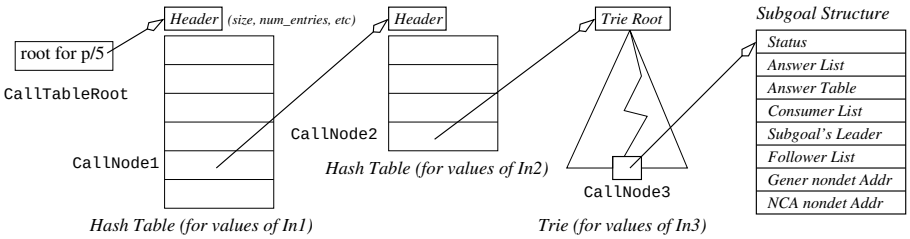

Fig. 2. Data structures created for the calls of predicate `p/5`

In the code of Fig. 3, `CallTableRoot`, `CallNode1`, `CallNode2` and `CallNode3` are all pointers to nodes in the call tree at levels 0, 1, 2 and 3 respectively; see Fig. 2. `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. The `lookup_insert_int` call will then hash `In1`, create a new slot in the indicated bucket (or in one of its overflow cells) 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. For example, integers are hashed directly but we hash the characters of a string, not its address.)

```
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, AnsTabRoot),
        impure lookup_insert_int(AnsTabRoot, B, AnsNode1),
        impure answer_is_not_duplicate(AnsNode1),
        impure new_answer_block(Subgoal, 1, AnsBlock),
        impure save_answer(AnsBlock, 0, B)
      ;
        impure completion(Subgoal),
        fail
      )
    ;
      Status = complete,
      semipure return_all_answers(Subgoal, AnsBlock),
      semipure restore_answer(AnsBlock, 0, B)
    ;
      Status = active,
      impure suspend(Subgoal, AnsBlock),
      semipure restore_answer(AnsBlock, 0, B)
    )
  ).
```

Fig. 3. Example of the tabling transformation on path/2

User-defined types Values of these 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*, a data structure which has extensively been used in tabled systems [4]. 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 node 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, string 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. We first table these arguments, which are terms of a builtin type. 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.

4.2 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 (a) allocate a new subgoal structure, initializing its fields to reflect the current situation, (b) update the table node pointed to by `CallNode3` to point to this new structure, and (c) return this same pointer as `Subgoal`. 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 `Subgoal`, 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 this answer is new. 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, as we inserted the input arguments 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 the call succeeds. Otherwise it fails. (To make later calls fail, `answer_is_not_duplicate` sets the tip to non-NULL on success.) We thus get to call `new_answer_block` only if the answer we just computed is new.

`new_answer_block` adds a new item to the end of the subgoal's chronological list of answers, the new item being a fresh new memory 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 perfect 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 (see Sect. 4.5) and then mark the subgoal

as *complete*. Having already returned all answers of this subgoal from the first disjunct, execution fails out of the body of the transformed predicate.

If the subgoal is initially *complete*, we call `return_all_answers`, which succeeds once for each answer in the subgoal's chronological list of answers. For each answer, calls to `restore_answer` pick up the 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 much more 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.

4.3 Suspension of consumers

The `suspend` primitive starts by creating a *consumer structure* and adding it to the current subgoal's list of consumers. This structure has three fields: a pointer to this subgoal's subgoal structure (available in `suspend`'s `Subgoal` argument), an indication of which answers this consumer has consumed so far, and the saved state 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 The special purpose abstract machine registers (`maxfr`, `curfr`, the det stack pointer `sp`, and the return address register `succip`) all need to be part of the saved state, but of all the general purpose machine registers used for parameter passing, the only one that contains live data and thus needs to be saved is the one containing `Subgoal`.

Heap With Mercury's 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 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*. This is a big win, since the heap is typically the largest area. The tradeoff is that we 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. Binding a variable to a value thus affects only the stack slot holding the variable. Another difference 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 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 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 `fr` is the code of the procedure that created `fr`.

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. 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.

Two technical issues deserve to be mentioned. 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. Also, it is possible for the nearest common ancestor of the generator and consumer to be a procedure that lives on the det stack. The expanded version of this paper [8] gives examples of these situations, motivates the implementation alternatives we chose to adopt, and argues for the correctness of saving (only) this information for consumers.

4.4 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 Section 3 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. 2). 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.

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 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. Unlike CAT which tries to share the trail, heap, and local stack segments it copies [2], in Mercury we have not (yet) implemented sharing of the copied stack segments. It is our intention to implement and evaluate such a mechanism. However, note 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.

4.5 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 loops 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 part of the det stack allocated since the creation of this nondet frame. To provide the information required for the second part of this operation, we extend every ordinary nondet stack frame with a sixth slot that contains the address of the top of the det stack at the time of the nondet stack frame's creation.

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 a 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 `redoips` 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.

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 reentered 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 memory occupied by the saved states of all their consumers and generators.

4.6 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:

```
:- pred list_contains_odd_number(list(int)::in) is semidet.
list_contains_odd_number(List) :- some [N] (member(N, List), odd(N)).
```

Typically the code inside the quantification may have more than one solution, but the code outside only wants to check whether a solution *exists* without caring about the number of solutions or their bindings. One can thus convert a multi or nondet goal into a det or semidet goal 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 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 currently on 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.

4.7 Possibly negated contexts

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. 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 (the condition may succeed more than once).

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 $\mathtt{t}_c$ 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, or something equivalent. 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 and abort execution. 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 could 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 of Mercury's all-solutions predicates. The implementation of `builtin_aggregate` uses a failure driven loop. To ward against `builtin_aggregate(Closure, ...)` mistaking the failure of `call(Closure)` due to a suspension somewhere inside `Closure` as implying the absence of solutions to `Closure`, we treat the loop body as the condition of an if-then-else, i.e. we surround it with the code we normally insert at the start of the condition and the start of the else part (see [8] for the details).

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.

5 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.

Overhead of the grade with full tabling support We compiled the Mercury compiler in two grades that differ in that one supports minimal model tabling, the form of tabling discussed in this paper, by including the cut and pneg stacks and the extra slot on nondet stack frames, and while the other, lacking these extras, supports only the other forms of tabling (memoization and loop checking). Enabling support for minimal model 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 with full tabling support 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.) First of all, it should be mentioned that paying this 25% cost in time happens only if the user selects a grade with minimal model tabling support: programs that do not use minimal model tabling at all can use the default `asm_fast.gc` grade and thus not pay any cost whatsoever. Moreover, this 25% is probably an upper limit. (See also the results in Table 3 which overall show less than 19% overhead.) 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 (see the data in [8]). If we had an analysis that could determine that tabled predicates are not involved (directly or indirectly) in a possibly negated context, this overhead could be totally avoided for that context. We are now working on such an analysis.

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

			chain				cycle			
benchmark	size	iter	XSB	XXX	YAP	Mercury	XSB	XXX	YAP	Mercury
tc_lr +-	4K	200	0.62	0.51	0.28	0.58	0.63	0.52	0.28	0.59
tc_lr +-	8K	200	1.24	1.05	0.62	1.27	1.27	1.07	0.62	1.30
tc_lr +-	16K	200	2.57	2.15	1.51	2.47	2.62	2.12	1.48	2.61
tc_lr +-	32K	200	5.25	4.41	3.78	5.23	5.20	4.44	3.78	5.07
tc_lr --	2K	1	2.58	2.46	1.25	3.20	6.22	6.30	2.88	6.24
tc_rr --	2K	1	2.21	2.04	2.94	10.27	6.35	5.85	6.00	27.48

Comparison against other implementations of tabling We compared the minimal model grade of Mercury (using rotd-06-10-2005, based on CAT) against XSB (2.7.1, based on the SLG-WAM), the XXX system (derived from XSB but based on CHAT) and YAP (version in CVS at 28 July 2005, based on SLG-WAM). XSB and XXX use *local scheduling* [6] in the default configuration while YAP uses *batched scheduling*. Mercury's scheduling strategy is similar but not identical to batched scheduling. All benchmarks were run on an IBM ThinkPad R40 laptop with a 2.0 GHz Pentium4 CPU and 512 Mb of memory running Linux. All times were obtained by running each benchmark 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 `--`. The second set consists of versions of the same generation predicate with full indexing (i) or Prolog-style first-argument indexing only (p), with the same two kinds of queries. Each table entry shows how long it takes for a given system to run the specified query on the specified data *iter* times (iter=50 for the `sg` benchmarks). The tables are reset between iterations. In Tables 1 and 2, benchmarks use a failure driven loop or its equivalent to perform the iterations, while in Table 3 they use a tail-recursive driver predicate.

Table 2. Times (in secs) to execute various versions of same generation

benchmark	XSB	XXX	YAP	Mercury
sg i +-	1.21	1.32	0.34	1.05
sg i --	3.53	3.89	1.07	2.43
sg p +-	83.56	58.17	34.58	32.14
sg p --	237.58	161.08	77.63	92.64

The rows for the `+-` query on left recursive transitive closure show all runtimes to be linear in the size of the data, as expected. Also, on left recursion, regardless of query, YAP is fastest, and XSB, XXX and Mercury are pretty similar. On right recursion, Mercury is slower than the other systems due to saving and restoring stack segments of consumers, and having to do so more times due to its different scheduling strategy (YAP doesn't do save/restore). 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 quantifications, a construct not properly handled in Prolog systems with tabling. On the same generation (`sg`) benchmark, in which consumer suspensions are not created (variant subgoals are only encountered when the subgoals are completed), Mercury is clearly much faster than XSB and XXX, although it is still beaten by YAP in three cases out of four. Two reasons why Mercury's usual speed advantage doesn't materialize here are that (1) these benchmarks spend much of their time executing tabling's primitive operations, which are in handwritten C code in all four systems, and (2) the Prolog systems can recover the memory allocated by an iteration by resetting the heap pointer, whereas in Mercury this can be done only by garbage collection. (Although the benchmark programs are Datalog, the all-solutions predicate used by the benchmark harness allocates heap cells.)

Table 3. Times (in secs) to execute some standard untabled Prolog benchmarks

benchmark	cqueen	crypt	deriv	nrev	primes	qsort	queen	query	tak	total
iterations	60K	30K	500K	300K	150K	300K	2K	100K	1K	
Mercury plain	1.92	5.44	5.61	7.99	6.43	6.37	4.77	0.70	0.52	39.8
Mercury tabled	3.26	7.17	4.96	7.08	8.80	7.41	5.83	0.89	1.80	47.2
YAP	9.16	9.14	4.08	4.53	20.89	15.35	12.40	6.44	12.50	94.5
XXX	15.27	10.86	8.08	6.94	31.66	21.72	22.09	17.46	17.30	151.4
XSB	23.64	17.23	11.58	16.71	thrashes	32.83	34.56	29.65	24.05	$>$ 190.3

Table 3 shows the performance of the same four systems on nine standard Prolog benchmarks that do not use tabling, taken from [7]. Mercury is clearly the fastest system by far, even when minimal model tabling is enabled but not used. It is beaten only on nrev and deriv, which spend *all* their time in predicates that are tail recursive in Prolog but not in Mercury.

It is very difficult to draw detailed conclusions from these small benchmarks, but we can safely say that we succeeded in our objective of concentrating the costs of tabling on the predicates that use tabling, reducing the performance of untabled predicates by at most 25%. We can confidently expect Mercury to be much faster than Prolog systems on programs in which relatively few consumer suspensions are encountered. The speed of Mercury relative to tabled Prolog systems on *real* tabled programs will depend on what fraction of time they spend in tabled predicates.

Our most promising avenues for further improvement of tabling in Mercury are clearly (1) improving the speed of saving and restoring suspensions and (2) implementing a scheduling strategy that reduces the number of suspensions and resumptions.

6 Concluding Remarks

Adapting the implementation of 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. These constructs are either not available or not properly handled in other tabled LP systems.

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 support for 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.

The results on microbenchmarks focusing on the performance of the basic tabled primitives themselves show tabling in Mercury to be quite competitive with that of other high-performance 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 tabling 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. Performing such a comparison across different languages is not a trivial task because many applications of tabling often rely on features (e.g., inspection of tables during runtime or dynamic modifications of the Prolog database) which are not available in Mercury. But one should not underestimate either the difficulty or the importance of adding proper tabling in a safe way to a truly declarative, high-performance LP system and the power that this brings to it.

References

1. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, Jan. 1996.
2. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. *J. of Functional and Logic Programming*, Nov. 1999.
3. H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of ICLP'01*, pages 181–196, Nov/Dec. 2001.
4. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *J. of Logic Programming*, 38(1):31–54, Jan. 1999.
5. R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1 & 2):161–205, Jan. 2005.
6. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. on Prog. Lang. Syst.*, 20(3):586–634, May 1998.
7. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. of Logic Progr.*, 26(1–3):17–64, 1996.
8. Z. Somogyi and K. Sagonas. Minimal model tabling in Mercury. Available at `http://www.cs.mu.oz.au/research/mercury/information/papers.html`, 2005.
9. H. Tamaki and T. Sato. OLD resolution with Tabulation. In *ICLP '86*, pages 84–98. 1986.
10. D. H. D. Warren. An abstract Prolog instruction set. Tech. Rep. 309, SRI International, 1983.
11. N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan and J.-H. You. Implementation of a linear tabling mechanism. *J. of Functional and Logic Programming*, 2001(10), 2001.