CICLOPS 2005

Proceedings of the Fifth Colloquium on Implementation of Constraint and LOgic Programming Systems

Christian Schulte Fernando Silva Ricardo Rocha
(Eds.)

Technical Report DCC-2005-08

Departamento de Ciência de Computadores, Faculdade de Ciências
&
Laboratório de Inteligência Artificial e Ciência de Computadores
Universidade do Porto

Rua do Campo Alegre, 823 – 4150 Porto, Portugal
Tel: (+351) 226078830 – Fax: (+351) 226003654
`http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html`

CICLOPS 2005

Proceedings of the Fifth Colloquium on Implementation of Constraint and LOgic Programming Systems

Sitges, Barcelona, Spain

October 5, 2005

Christian Schulte Fernando Silva Ricardo Rocha
(Eds.)

Preface

This volume contains the papers presented at the fifth *Colloquium on Implementation of Constraint and LOgic Programming Systems* (CICLOPS 2005), held in Sitges (Barcelona), Spain, in October 2005, as a satellite workshop of ICLP 2005.

CICLOPS is a workshop that aims at discussing and exchanging experiences on the design, implementation, and optimization of logic and constraint (logic) programming systems, or systems intimately related to logic as a means to express computations. The workshop continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with in Budapest (1993), Ithaca (1994) and Portland (1995), the Compulog Net workshops on Parallelism and Implementation Technologies held in Madrid (1993 and 1994), Utrecht (1995) and Bonn (1996), the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages held in Port Jefferson (1997), Manchester (1998), Las Cruces (1999), and London (2000), and recently the Colloquium on Implementation of Constraint and LOgic Programming Systems in Paphos (2001), Copenhagen (2002), Mumbai (2003), and Saint-Malo (2004), and the CoLogNet Workshops on Implementation Technology for Computational Logic Systems held in Madrid (2002), and Pisa (2003).

This year, we received 7 paper submissions, of which 6 were accepted for presentation and inclusion in this volume. The workshop coordinators wish to thank the program committee for the effort and willingness put in the evaluation of the papers, and also the organization of ICLP 2005 for their valuable help. Thanks should go also to the authors of the submitted papers for their contribution to make CICLOPS alive.

We hope that the workshop will bring together researchers interested on implementation technology for different aspects of logic and constraint-based languages and systems, in order to contribute to make this event a meeting point for a fruitful exchange of ideas and feedback on recent developments.

August 2005,

Christian Schulte
Fernando Silva
Ricardo Rocha

Workshop Coordinators

Christian Schulte	Royal Institute of Technology, Sweden
Fernando Silva	University of Porto, Portugal
Ricardo Rocha	University of Porto, Portugal

Program Committee Members

Bart Demoen	K. U. Leuven, Belgium
Christian Schulte	KTH Royal Institute of Technology, Sweden
David S. Warren	State University of New York, USA
Enrico Pontelli	New Mexico State Univeristy, USA
Fernando Silva	University of Porto, Portugal
Inês Dutra	Federal University of Rio de Janeiro, Brazil
Manuel Carro	Technical University of Madrid, Spain
Mats Carlsson	Swedish Institute of Computer Science, Sweden
Ricardo Rocha	University of Porto, Portugal
Vitaly Lagoon	University of Melbourne, Australia
Warwick Harvey	Imperial College, IC-Parc, UK

Referees

Bart Demoen, Christian Schulte, David S. Warren, Enrico Pontelli, Fernando Silva, Inês Dutra, Manuel Carro, Mats Carlsson, Ricardo Lopes, Ricardo Rocha, Vitaly Lagoon and Warwick Harvey.

Web Page

`http://www.dcc.fc.up.pt/ciclops05`

Table Of Contents

The Implementation of Minimal Model Tabling in Mercury (extended abstract)*

Zoltan Somogyi
Dept of Comp. Science and Soft. Eng.
University of Melbourne, Australia
zs@cs.mu.OZ.AU

Konstantinos Sagonas
Dept of Information Technology
Uppsala University, Sweden
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 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. As a result, many LP systems (e.g., XSB, YAP, B-Prolog, and TALS) nowadays offer some form of tabling.

When deciding which tabling mechanism to adopt, an implementor is faced with several choices. Linear tabling strategies such as SLDT [11] and DRA [4] 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], 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 (the Copying Approach to Tabling [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 and compute their *minimal model*. 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 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 for features they do not use.

This extended abstract documents the implementation of minimal model tabling in Mercury (we actually aim to compute a specific minimal model: the perfect model). We describe how we adapted the CAT mechanism to a different

*See [8] for an expanded version with background material, correctness arguments, and more information on the implementation.

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 briefly reviews the Mercury language and its implementation. Section 3 introduces the various forms of tabling in Mercury, followed by the paper's main section (Section 4) which describes the implementation of minimal model tabling in detail. A brief performance comparison with other Prolog systems with tabling appears in Section 5, and the paper ends with some concluding remarks.

2 The Mercury Language and Implementation: 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. 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 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 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. 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 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 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.

The compiler converts all multi-clause predicate definitions into a disjunction. 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 instruction pointed to by its `redoip` slot. Disjunctions other than the outermost one are implemented using temporary nondet stack frames, which have only `prevfr`, `redoip` and `redofr` slots. For more details, see [8].

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.

3 Tabling in Mercury

In 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 (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. In (stratified) Mercury programs, programmers who are interested in computing the answers of tabled predicate calls according to the *minimal model* semantics, they can use the '`pragma minimal_model`' declaration. 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 minimal model 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

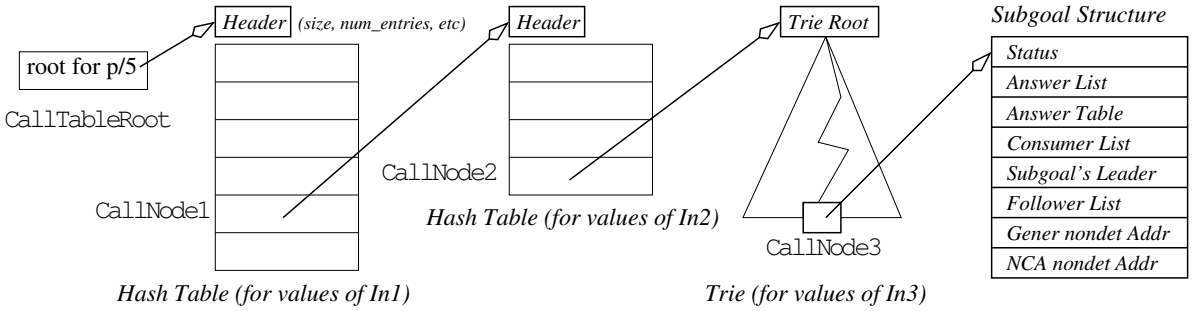

Figure 1: Data structures created for the calls of predicate `p/5`

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 minimal model 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 minimal model 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 one of the subgoals in the component (typically the oldest one) is called the component's *leader*.

4 The Implementation of Minimal Model 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 implementation of minimal model tabling 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 figure 2 (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 aren't observable from the outside.

As mentioned, the arguments of minimal model 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 on the left below, the minimal model tabling transformation inserts the code shown on the right 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)
```

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. 1), 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 clique of subgoals 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).

```
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)
    )
  ).
```

Figure 2: Example of the minimal model tabling transformation

In the code on the right, `CallTableRoot`, `CallNode1`, `CallNode2` and `CallNode3` are all pointers to nodes in the call tree at levels 0, 1, 2 and 3 respectively; see Fig. 1. `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 (or one of its overflow cells in general) 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. We hash integers directly but we hash the characters of a string, not its address.)

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. 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 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 [3]. 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, just 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, 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 is new.

`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 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 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 (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 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 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. (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.

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. 1). 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 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.

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 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.

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 fragment:

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

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 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 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. 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 $\mathtt{t}_c$ 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. 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`. The implementation of `builtin_aggregate` in the Mercury standard library uses a failure-driven loop. To guard against

			chain				cycle			
bench	size	iter	XSB	XXX	YAP	Mer	XSB	XXX	YAP	Mer
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

bench	XSB	XXX	YAP	Mer
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

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

bench iter	cqueen 60K	crypt 30K	deriv 1.5M	nrev 800K	primes 150K	qsort 300K	queen 2K	query 100K	tak 1K
Mer asm_fast.gc	2.01	5.49	17.16	20.49	6.33	6.26	5.59	0.75	0.54
Mer asm_fast.gc.mm	3.17	7.10	14.96	18.06	8.84	6.80	6.77	1.01	1.80
YAP	9.15	8.95	11.71	11.62	20.42	15.28	12.31	6.24	12.52
XXX	14.54	10.94	24.72	18.11	31.29	21.91	21.52	17.16	17.85
XSB	23.63	17.68	32.44	44.15	thrashes	32.51	34.26	29.42	23.95

Table 2: Times (in secs) to execute some standard untabled Prolog benchmarks

`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 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 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 minimal model tabling compilation grade: programs that do not use 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. 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 [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.

Comparison against other implementations of tabling We compared the minimal model grade of Mercury (rotd-28-07-2005, 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], 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 entry in Table 1 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 Table 1, each system uses a failure driven loop or its equivalent to perform the iterations, while in Table 2 they use a tail-recursive driver predicate.

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 C 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 done only by gc.

Table 2 shows the performance of the same four systems on nine standard Prolog benchmarks that do not use tabling, taken from [7]. (The tenth benchmark from that set, poly, leads to memory exhaustion on XSB and XXX and a core dump on YAP.) Mercury is clearly the fastest system by far, even when minimal model tabling is enabled (but not used). It is beaten only on nrev, which is tail recursive in Prolog but not in Mercury, and on deriv, on which (due to a currently broken optimization) Mercury creates many unnecessary stack frames. While Mercury executed all nine benchmarks in a total time of 64s (in grade asm_fast.gc, which cannot do minimal model tabling) to 69s (in asm_fast.gc.mm, which can), YAP required 108s, XXX 178s and XSB 238s (not counting primes).

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, leaving the performance of untabled predicates mostly unaffected. 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 are clearly (1) improving the speed of saving and restoring suspensions and (2) implementing a scheduling strategy that reduces the number of suspensions required.

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. 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.

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 either the difficulty or the importance of adding proper tabling to a high-performance LP system and the power that this brings to the system.

The system we have described in this paper is available in recent releases-of-the day from the Mercury web site.

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] T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run time type information in Mercury. In *Proceedings of PPDP'99*, pages 224–243, Sept. 1999.

[4] 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.

[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 Transactions on Programming Languages and Systesm*, 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 Programming*, 26(1–3):17–64, Oct/Dec. 1996.

[8] Z. Somogyi and K. Sagonas. Minimal model tabling in Mercury. Extended version of this paper, available at: `http://www.cs.mu.oz.au/research/mercury/information/papers.html`, Aug 2005.

[9] H. Tamaki and T. Sato. OLD resolution with Tabulation. In *Proceedings of ICLP '86*, pages 84–98. July 1986.

[10] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA, Oct. 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.

Pruning Extensional Predicates in Deductive Databases

Tiago Soares Ricardo Rocha Michel Ferreira
DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
{tiagosoares,ricroc,michel}@ncc.up.pt

Abstract

By coupling Logic Programming with Relational Databases, we can combine the higher expressive power of logic with the efficiency and safety of databases in dealing with large amounts of data. An important drawback of such coupled systems is the fact that, usually, the communication architecture between both systems is not *tight* enough to support a completely transparent use of extensionally (or relationally) defined predicates in the logic program. Such an example is the *cut operation*. This operation is used quite often in Prolog programs, both for efficiency and semantic preservation. However, its use to prune choice-points related to database predicates is discouraged in existing coupled systems. Typically, SQL queries result sets are kept outside WAM data structures and the cut implementation leaves the database result set pointer (cursor) open and the result set data structure allocated in memory, instead of closing the cursor and deallocating memory. In this work we focus on the transparent use of the cut operation over database predicates, describing the implementation details in the context of the coupling between the YapTab system and the MySQL RDBMS. Our approach can be generalised to handle not only database predicates but also any predicate that requires generic actions upon cuts.

1 Introduction

The similarities between logic based languages, such as Prolog, and relational databases have long been noted. There is a natural correspondence between relational algebra expressions and Prolog predicates, and between relational tuples and Prolog facts [7]. The main motivation behind a deductive database system, based on the marriage of a logic programming language and a relational database management system (RDBMS), is the combination of the inference capabilities of the logic language with the ultra-efficient data handling of the RDBMS. The implementation of such deductive database systems follows the following four general methods [1, 14, 6]: **(i)** coupling of an existing logic system implementation to an existing RDBMS; **(ii)** extending an existing logic system with some facilities of a RDBMS; **(iii)** extending an existing RDBMS with some features of a logic language; and **(iv)** tightly integrating logic programming techniques with those of RDBMS's.

An example of a tightly integrated implementation is the Aditi system [15], and examples of coupled implementations are the Coral [8] and XSB [10] systems. Regarding coupled systems, the literature usually distinguishes between *tightly* and *loosely* coupled systems. This classification is far from being clear, though. Sometimes the tight or loose adjectives are used regarding the degree of integration from the programming perspective, while sometimes the terms are used regarding the transparency of the use of logic predicates defined by database relations.

The use of explicit SQL queries in logic predicates would characterize a loosely coupled system, while the specification of database queries transparently in logic syntax would characterize a tightly coupled system. The usual meaning of the *tight* and *loose* terms in the computer industry is, however, different. Systems are tightly coupled if they cannot function separately and loosely coupled if they can. Under this definition, the XSB system coupled with a RDBMS results in a loosely coupled system.

The coupling approach, by keeping the deductive engine separated from the RDBMS, offers a number of advantages: deductive capabilities can be used with arbitrary RDBMS's and the deductive system can profit from future and independent developments of the RDBMS; furthermore, there are also no extra overheads for programs that do not access data stored in a relational database. The coupling approach also presents, however, some important drawbacks as compared with integrated systems. The most relevant is the communication overhead to get external data from the database server to be unified with logic goals. This overhead is extremely significant and makes impracticable the use of *relation-level access* (where SQL queries are generated for each logic goal) with all but toy applications. *View-level access* (where relational join operations of logic goals are executed by the RDBMS) very importantly reduces this communication overhead, particularly if it allows the efficient use of the RDBMS indices [4].

Another important drawback of loosely or tightly coupled systems is the fact that the communication architecture between both systems is normally based on an interface layer which lacks important features necessary to improve both the efficiency and the transparent integration of the logic system and the database system. For instance, the application programming interfaces of RDBMS's do not provide adequate mechanisms to send *sets* of tuples back and forth from the logic system to the database server, and this has been shown to be crucial to attain good performance [5].

In addition to efficiency concerns, the existing communication architectures between coupled systems are also not *tight* enough to support a completely transparent use of relationally defined predicates in the logic program. Consider, for example, the *cut operation*. This operation is extremely used in Prolog programs, both for efficiency and semantic preservation. However, its use after a database defined predicate can have undesired effects, with the current interface architectures of coupled systems. The undesired effects can be so significant that systems such as XSB clearly state on the programmers' manual that cut operations should be used very carefully with relationally defined predicates [11]:

> *"The XSB-ODBC interface is limited to using 100 open cursors. When XSB systems use database accesses in a complicated manner, management of open cursors can be a problem due to the tuple-at-a-time access of databases from Prolog, and due to leakage of cursors through cuts and throws. Often, it is more efficient to call the database through set-at-a-time predicates such as findall/3, and then to backtrack through the returned information."*

In this work we focus on the transparent use of the cut operation over database predicates, describing the implementation details in the context of the coupling between the YapTab system [9] and the MySQL RDBMS [16]. YapTab is a tabling system that extends Yap's engine [12] to support tabled evaluation for definite programs. The remainder of the paper is organized as follows. First, we briefly describe the cut semantics of Prolog and the problem arising from its use to prune database predicates. Next, we present how Yap interfaces MySQL through their C API's. Then, we describe the needed extension to the interface architecture in order to deal with pruning for database predicates. At the end, we outline some conclusions.

2 Pruning Database Predicates

Ideally, it should be possible to use predicate facts defined extensionally in database relations exactly as predicate facts defined in a Prolog program. In particular, for the cut operation, it should be possible to prune predicates independently of how they are defined. In this section we show why this is a problem for current coupled systems.

2.1 Cut Semantics

Cut is a system built-in predicate that is represented by the symbol '**!**'. Its execution results in pruning all the branches to the right of the cut scope branch. The *cut scope branch* starts at the current node and finishes at the node corresponding to the predicate containing the cut.

Figure 1 gives a general overview of cut semantics by illustrating the left to right execution of an example with cuts. The query goal `a(X)` leads the computation to the first alternative of predicate `a/1`, where `!(a)` means a cut with scope node `a`. Predicate `b(X)` is then called and suppose that it succeeds in its first alternative binding `X` with `b1`. Next, `!(a)` gets executed and all the right branches until node `a` are pruned. As a consequence, the nodes for `a` and `b` can be removed.

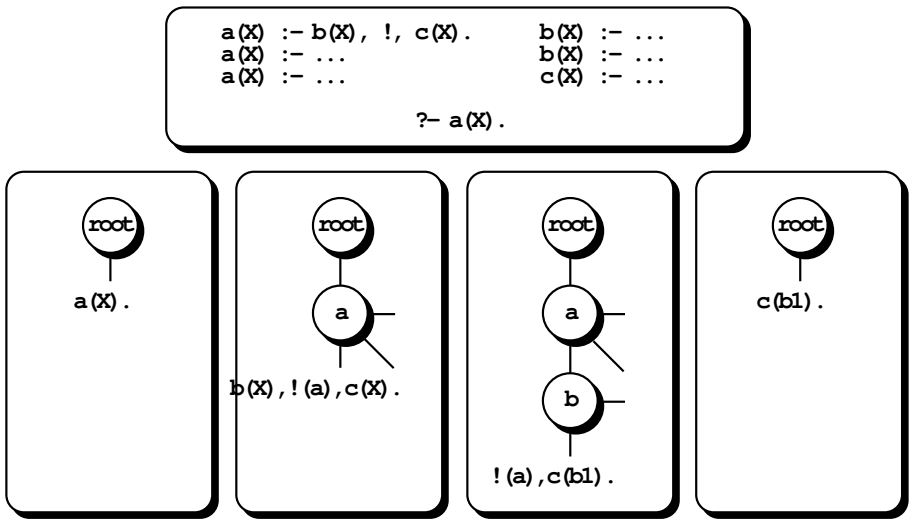

Figure 1: Cut semantics

2.2 Pruning a Query Result Set

In the coupling interface between a logic system and a database system, logic goals are usually translated into SQL queries, which are then sent to the database system. To improve performance, a number of logic goals can be combined on a single SQL query, replacing a relation-level access for what is known as view-level access. The database system then receives the query, processes it, and sends the resulting tuples back to the logic system.

MySQL offers two alternatives for sending these resulting tuples to the client program that generated the query: **(i)** store the set of tuples on a data structure on the database server side and send each tuple *tuple-at-a-time* to the client program; or **(ii)** store the set of tuples on a data structure on the client side sending the all set of tuples at once. Logic systems also have two alternatives to deal with these sets of tuples: **(i)** use them *tuple-at-a-time*, exactly as is done for

normal facts; or **(ii)** use them *set-at-a-time*, typically in a Prolog list structure [3]. Building this list for thousands or millions of tuples can lead to memory allocation problems and is very time consuming. Furthermore, the transparent use of facts defined in database relations is lost if they imply a set-at-a-time use.

For tuple-at-a-time, the obvious method of accessing the tuples in the result set of a query is to use the backtracking mechanism, which iteratively increments the database result set pointer (cursor) and fetches the current tuple. Using this tuple-at-a-time access, the deallocation of the data structure holding the result set, whether on the server or on the client side, is performed when the last tuple on the result set has been reached.

The problem is when, during the tuple-at-a-time navigation, a cut operation occurs before reaching the last tuple. If this happens, the result set cannot be deallocated. This can cause a lack of cursors and, more important, a lack of memory due to a number of very large non-deallocated data structures. Consider the example described in Fig. 1 and assume now that predicate `b/1` is a database predicate. The computation of `b(X)` will query the database for the correspondent relation and bind `X` with the first tuple that matches the query. Next, we execute `!(a)` and, as mentioned before, it will disable the action of backtracking for node `b`. The result set with the facts for `b` will remain in memory, although it will never be used.

To solve this problem, we propose an extension to the interface architecture that allows to associate cut procedures with database predicates, in such a way that the system transparently executes them when a cut operation occurs. With this functionality, we can thus use these procedures to deallocate the result set for the pruned database predicates and therefore avoid the problems discussed above. In what follows we briefly describe the Yap interface and then we detail our approach to extend the interface in order to deal with pruning for database predicates.

3 The C Language Interface to Yap Prolog

Like other Prolog Systems, Yap provides an interface for writing predicates in other programming languages, such as C, as external modules. An important feature of this interface is how we can define predicates. Yap distinguishes two kinds of predicates: *deterministic predicates*, which either fail or succeed but are not backtrackable, and *backtrackable predicates*, which can succeed more than once.

Deterministic predicates are implemented as C functions with no arguments which should return zero if the predicate fails and a non-zero value otherwise. They are declared with a call to `YAP_UserCPredicate()`, where the first argument is the name of the predicate, the second the name of the C function implementing the predicate, and the third is the arity of the predicate.

For backtrackable predicates we need two C functions: one to be executed when the predicate is first called, and other to be executed on backtracking to provide (possibly) other solutions. They are similarly declared, but using instead `YAP_UserBackCPredicate()`. When returning the last solution, we should use `YAP_cut_fail()` to denote failure, and `YAP_cut_succeed()` to denote success. The reason for using `YAP_cut_fail()` and `YAP_cut_succeed()` instead of just returning a zero or non-zero value, is that otherwise, when backtracking, our function would be indefinitely called. For a more exhaustive description on how to interface C with Yap please refer to [13].

3.1 Writing Backtrackable Predicates in C

To explain how the C interface works for backtrackable predicates we will use a small example from the interface between Yap and MySQL. We present the `db_row(+ResultSet,?ListOfArgs)`

predicate, which fetches tuples from a query result set, tuple-at-a-time through backtracking, and unifies the list in `ListOfArgs` with the current tuple. To do so, first a `yap_mysql.c` module should be created. Next the module should be compiled to a shared object and then loaded under Yap by executing `load_foreign_files([yap_mysql],[],init_predicates)`. After that, the `db_row/2` predicate becomes available. The code for this module is shown next in Fig. 2.

```
#include "Yap/YapInterface.h"                    // header file for the Yap interface to C

void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row, c_db_row, 2, 0);
}

int c_db_row(void) {                                    // db_row: ResultSet -> ListOfArgs
  int i, arity;
  YAP_Term arg_result_set, arg_list_args, head;
  MYSQL_ROW row;
  MYSQL_RES *result_set;

  arg_result_set = YAP_ARG1;
  arg_list_args = YAP_ARG2;
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  arity = mysql_num_fields(result_set);
  if ((row = mysql_fetch_row(result_set)) != NULL) {                    // get next tuple
    for (i = 0; i < arity; i++) {
      head = YAP_HeadOfTerm(arg_list_args);
      arg_list_args = YAP_TailOfTerm(arg_list_args);
      if (!YAP_Unify(head, YAP_MkAtomTerm(YAP_LookupAtom(row[i]))))
        return FALSE;
    }
    return TRUE;
  } else {                                                              // no more tuples
    mysql_free_result(result_set);
    YAP_cut_fail();
    return FALSE;
  }
}
```

Figure 2: The C code for the `db_row/2` predicate

Figure 2 shows some of the key aspects about the Yap interface. The include statement makes available the macros for interfacing with Yap. The `init_predicates()` procedure tells Yap, by calling `YAP_UserBackCPredicate()`, the predicate defined in the module. The function `c_db_row()` is the implementation in C of the desired predicate. We can define a function for the first time the predicate is called and another for calls via backtracking. In this example the same function is used for both calls. Note that this function has no arguments even though the predicate being defined has two. In fact the arguments of a Prolog predicate written in C are accessed through the macros `YAP_ARG1`, ..., `YAP_ARG16` or with `YAP_A(N)` where `N` is the argument number.

The `c_db_row()` function starts by converting the first argument (`YAP_ARG1`) to the correspondent pointer to the query result set (`MYSQL_RES *`). The conversion is done by the `YAP_IntOfTerm()` macro. It then fetches a tuple from this result set, through `mysql_fetch_row()`, and checks if the last tuple as been already reached. If not, it calls `YAP_Unify()` to attempt the unification of values in each attribute of the tuple (`row[i]`) with the respective elements in `arg_list_args`. If unification fails it returns `FALSE`, otherwise returns `TRUE`. On the other hand, if the last tuple has been already reached, it deallocates the result set, as mentioned before, calls `YAP_cut_fail()` and returns `FALSE`.

For simplicity of presentation, we omitted type checking procedures over MySQL attributes that must be done to convert each attribute to the appropriate term in Yap. For some predicates it is also useful to preserve some data structures across different backtracking calls. This can be done by calling `YAP_PRESERVE_DATA()` to associate such space and by calling `YAP_PRESERVED_DATA()` to get access to it later. With these two macros we can easily share information between backtracking. This example does not need this preservation, as the cursor is maintained in the result set structure.

3.2 The Yap Implementation of Backtrackable Predicates

In Yap, a backtrackable predicate is compiled using two WAM-like instructions, `try_userc` and `retry_userc`, as follows:

```
try_userc c_first arity extra_space
retry_userc c_back arity extra_space
```

Both instruction have three arguments: the `c_first` and `c_back` arguments are pointers to the C functions associated with the backtrackable predicate, `arity` is the arity of the predicate, and `extra_space` is the memory space used by the `YAP_PRESERVE_DATA()` and `YAP_PRESERVED_DATA()` macros.

When Yap executes a `try_userc` instruction it uses the choice point stack to reserve as much memory as given by the `extra_space` argument, next it allocates and initializes a new choice point (see Fig. 3), and then it executes the C function pointed by the `c_first` argument. Later, if the computation backtracks to such choice point, the `retry_userc` instruction gets loaded from the `CP_AP` choice point field and the C function pointed by the `c_back` argument is then executed.

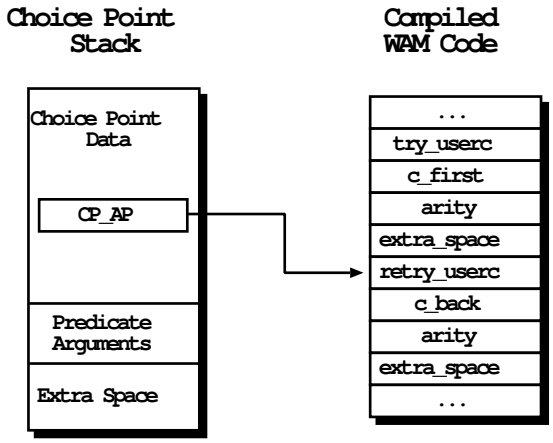

Figure 3: Yap support for backtrackable predicates

In order to repeatedly execute the same `c_back` function when backtracking to this choice point, the `retry_userc` instruction maintains the `CP_AP` field pointing to itself. This is the reason why we should use `YAP_cut_succeed()` or `YAP_cut_fail()` when returning the last solution for the predicate, as otherwise the choice point will remain in the choice point stack and the `c_back` function will be indefinitely called.

The execution of the `YAP_PRESERVE_DATA()` and `YAP_PRESERVED_DATA()` macros in the C functions corresponds to calculate the starting position of the reserved space associated with the `extra_space` argument. For both functions, this is the address of the current choice point pointer plus its size and the arity of the predicate.

4 Extending the Yap C Interface to Deal with Pruning

In this section, we discuss how we can solve the problem of pruning database predicates. We consider two different approaches. In the first approach it is the user that is responsible to specify an action to be executed just before a cut operation over a database predicate. The second approach is our original proposal where the system transparently executes a cut procedure when a cut operation occurs. As we will see, this approach can be used, not only, to handle database predicates but also to handle any predicate that requires a generic action over a cut.

4.1 Handling Cuts Explicitly

In this approach it is the user that has to explicitly call a predicate to be executed when a cut operation is performed over special predicates. The idea is as follows: before executing a cut operation that potentially prunes over databases predicates, the user must execute a predicate that releases beforehand the result sets for the predicates to be pruned.

Consider again the example from Fig. 1 and assume that `b(X)` is a database predicate. We might think of a simple solution: every time a database predicate is first called, we store the pointer for its result set in a stack frame. Then, we could implement a C predicate, `db_free_result/0` for example, that deallocates the result set in the top of the stack. Having this, we could adapt the code for the first clause of predicate `a/1` to:

```
a(X) :- b(X), db_free_result, !, c(X).
```

To use this approach, the user must be careful to always include a call to `db_free_result/0` before a cut operator. A possible alternative, would be to define a new predicate, `db_cut/0` for example, and use it to replace the `db_free_result/0` and the cut.

```
a(X) :- b(X), db_cut, c(X).

db_cut :- db_free_result, !.
```

But, unfortunately, this would not work because the cut operator in the `db_cut/0` definition will not prune the choice point for `b(X)` in the first clause of `a/1`. Another problem with the `db_free_result/0` approach occurs if we call more than one database predicates before a cut. Consider the following definition for the predicate `a/1`, where `b1/1` and `b2/1` are database predicates.

```
a(X) :- b1(X), b2(Y), db_free_result, !, c(X).
```

The `db_free_result/0` will only deallocate the result set for `b2/1`, leaving the result set for `b1/1` pending. A possible solution for this problem is to *mark* beforehand the range of predicates to protect. We can thus implement a C predicate, `db_cut_mark/0` for example, that *marks* were to cut to, and change the `db_cut/0` predicate to call recursively the `db_free_result/0` predicate for all the result sets within the mark left by the last `db_cut_mark/0` predicate.

```
a(X) :- db_cut_mark, b1(X), b2(Y), db_cut, !, c(X).
```

This final solution solves the problem of cursor leaks when pruning database predicates. However, in this approach the transparency in the use of relationally defined predicates is lost. The user has to explicitly include extra annotations in the code to control the range of predicates to cut.

4.2 Handling Cuts Transparently

We next present our approach to handle cuts transparently. As we shall see, this requires minor changes to the Yap interface and engine. First, we have extended the procedure used to declare backtrackable predicates, `YAP_UserBackCPredicate()`, to include an extra C function. Remember that for backtrackable predicates we used two C functions: one to be executed when the predicate is first called, and another to be executed upon backtracking. The extra function is where the user should declare the function to be executed in case of a cut, which for database predicates will involve the deallocation of the result set. Declaring and implementing this extra function is the only thing we need to do to take advantage of this approach. Thus, from the user's point of view, dealing with standard predicates or relationally defined predicates is then equivalent.

With this extra C function, the compiled code for a backtrackable predicate now includes a new WAM-like instruction, `cut_userc`, which is used to store the pointer to the extra C function, the `c_cut` argument.

```
try_userc c_first  arity extra_space
retry_userc c_back arity extra_space
cut_userc c_cut arity extra_space
```

When Yap executes a `try_userc` instruction it now also allocates space for a cut frame data structure (see Fig. 4). This data structure includes two fields: `CF_previous` is a pointer to the previous cut frame on stack and `CF_inst` is a pointer to the `cut_userc` instruction in the compiled code for the predicate. A top cut frame variable, `TOP_CF`, always points to the youngest cut frame on stack. Frames form a linked list through the `CF_previous` field.

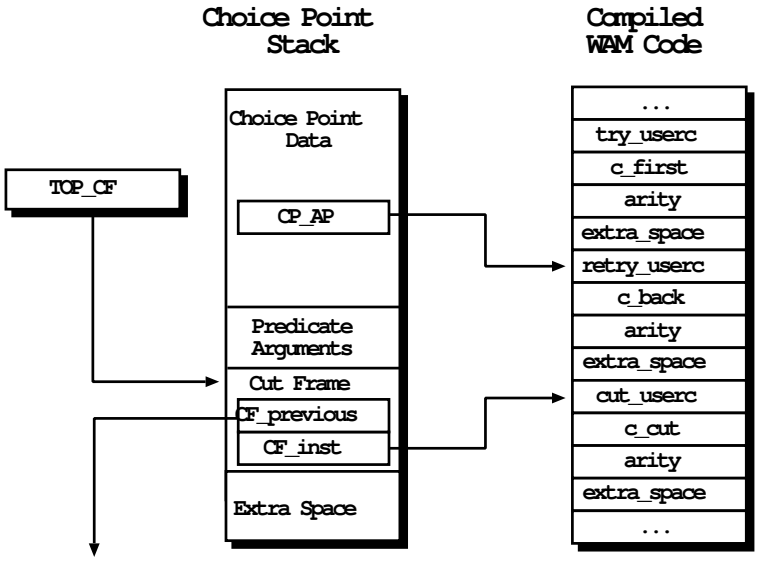

Figure 4: Yap support for handling cuts transparently with backtrackable predicates

By putting the cut frame structure below the associated choice point, we can easily detect the predicates being pruned when a cut operation occurs. To do so, we extended the implementation of the cut operation to start by executing a new `userc_cut_check()` procedure (see Fig. 5). Remember that a cut operation receives as argument the choice point to cut to. Thus, starting from the `TOP_CF` variable and going through the cut frames, we can check if a cut frame will be pruned. If so, we load the `cut_userc` instruction stored in the correspondent `CF_inst` field in order to execute the cut

function pointed by the c_cut argument. In fact, in our implementation, the cut_userc instruction is never executed. As an optimization, the CF_inst field points directly to the cut function.

```
void userc_cut_check(choiceptr cp_to_cut_to) {
  while (TOP_CF < cp_to_cut_to) {
    execute_c_function(TOP_CF->CF_inst);
    TOP_CF = TOP_CF->CF_previous;
  }
  return;
}
```

Figure 5: The pseudo-code for the userc_cut_check() procedure

The process described above is done before executing the original code for the cut instruction, that is, before updating the global registry B (pointer to the current choice point on stack). This is important to prevent the following situation. If the cut function executes a YapCallProlog() macro to call the Prolog engine from C, this might has the side-effect of allocating new choice points on stack. Thus, if we had updated the B register beforehand, we will potentially overwrite the cut frames stored in the pruned choice points and avoid the possibility of executing the correspondent cut functions.

As a final remark, note that the cut function can also call the YAP_PRESERVED_DATA() macro to access the data store in the extra space. We thus need to access the extra space from the cut frames. This is why we store the cut frames above the extra space. The starting address of the extra space is thus obtained by adding to the pointer of the current cut frame its size.

To show how the extended interface can be used to handle cuts transparently, we next present in Fig. 6 the code for generalising the db_row/2 predicate to perform the cursor closing upon a cut.

First, we need to define the function to be executed when a cut operation occurs. An important observation is that this function will be called from the cut instruction, and thus it will not be able to access the Prolog arguments, YAP_ARG1 and YAP_ARG2, as described for the c_db_row() function. However, we need to access the pointer to the correspondent result set in order to deallocate it. To solve this, we can use the YAP_PRESERVE_DATA() macro to preserve the pointer to the result set. As this only needs to be done when the predicate is first called, we defined a different function for this case. The YAP_UserBackCPredicate() macro was thus changed to include a cut function, c_db_row_cut(), and to use a different function when the predicate is first called, c_db_row_first(). The c_db_row() function is the same as before (see Fig. 2). The last argument of the YAP_UserBackCPredicate() macro defines the size of the extra space for the YAP_PRESERVE_DATA() and YAP_PRESERVED_DATA() macros.

The c_db_row_first() function is an extension of the c_db_row() function. The only difference is that it uses the YAP_PRESERVE_DATA() macro to store the pointer to the given result set in the extra space for the current choice point. On the other hand, the c_db_row_cut() function uses the YAP_PRESERVED_DATA() macro to be able to deallocate the result set when a cut operation occurs. With these two small extensions, the db_row/2 predicate is now protect against cuts and can be safely pruned by further cut operations.

5 Concluding Remarks

We discussed the problem of pruning database predicates in logic programming with relational database coupled systems. The existing communication architectures between coupled systems

```
void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row_first, c_db_row,
                         c_db_row_cut, 2, sizeof(MYSQL_RES *));
}

int c_db_row_first(void) {
  MYSQL_RES **extra_space;
  ...                                          // the same as for the c_db_row function
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  YAP_PRESERVE_DATA(extra_space, MYSQL_RES *);              // initialize the extra space
  *extra_space = result_set;                     // store the pointer to the result set
  ...                                          // the same as for the c_db_row function
}

int c_db_row(void) {
  ...                                                                  // the same as before
}

void c_db_row_cut(void) {
  MYSQL_RES **extra_space, *result_set;

  YAP_PRESERVED_DATA(extra_space, MYSQL_RES *);
  result_set = *extra_space;                          // get the pointer to the result set
  mysql_free_result(result_set);
  return;
}
```

Figure 6: The C code for protecting the `db_row/2` predicate against cuts

either do not handle cuts or rely on the user to explicitly protect the database predicates from potential cut operations. This is a relevant problem as the existing coupling architectures between a logic system and a RDBMS do not deallocate result sets upon a cut operation over and extensional predicates. We tested a simple program were a cut was executed over a large database extensional predicate, using XSB and Ciao Prolog [2] coupled to MySQL and we rapidly reached memory overflow because of non-deallocated result sets.

In this work, we proposed a new approach where pruning operators can be used independently of the predicates being pruned, whether they are defined extensionally in database relations or as Prolog facts. In our proposal the Prolog engine transparently executes a user-defined function when a cut operation occurs. To do so, we have extended the Yap interface to include an extra C function when declaring backtrackable predicates. This extra function is where the user defines the actions to be executed when a cut operation prunes the predicate. The Prolog engine then transparently executes the function when a cut operation prunes over the predicate. Thus, from the user's point of view, it only needs to declare that function once when defining the predicate. This approach is applicable to any predicate that requires protection against pruning.

Acknowledgements

This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

[1] M. Brodie and M. Jarke. On Integrating Logic Programming and Databases. In *Expert Database Workshop*, pages 191–207. Benjamin Cummings, 1984.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lpez, and G. Puebla. *Ciao Prolog System Manual*. Available from `http://clip.dia.fi.upm.es/Software/Ciao`.

[3] C. Draxler. Accessing Relational and NF^2 Databases Through Database Set Predicates. In *UK Annual Conference on Logic Programming*, Workshops in Computing, pages 156–173. Springer Verlag, 1992.

[4] M. Ferreira, R. Rocha, and S. Silva. Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 71–82, 2004.

[5] J. Freire. Practical Problems in Coupling Deductive Engines with Relational Databases. In *International Workshop on Knowledge Representation meets Databases*, 1998.

[6] F. Maier, D. Nute, W. Potter, J. Wang, M. J. Twery, H. M. Rauscher, P. Knopp, S. Thomasma, M. Dass, and H. Uchiyama. PROLOG/RDBMS Integration in the NED Intelligent Information System. In *Confederated International Conferences DOA, CoopIS and ODBASE*, volume 2519 of *LNCS*, page 528. Springer-Verlag, 2002.

[7] K. Parsaye. Logic Programming and Relational Databases. *IEEE Database Engineering Bulletin*, 6(4):20–29, 1983.

[8] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: A Deductive Database Programming Language. In *International Conference on Very Large Data Bases*. Morgan Kaufmann, 1992.

[9] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[10] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[11] K. Sagonas, D. S. Warren, T. Swift, P. Rao, S. Dawson, J. Freire, E. Johnson, B. Cui, M. Kifer, B. Demoen, and L. F. Castro. *XSB Programmers' Manual*. Available from `http://xsb.sourceforge.net`.

[12] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.

[13] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[14] P. Singleton and P. Brereton. Storage and Retrieval of First-order Terms Using a Relational Database. In *British National Conference on Databases*, volume 696 of *LNCS*, pages 199–219. Springer-Verlag, 1993.

[15] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The Aditi Deductive Database System. Technical Report 93/10, School of Information Technology and Electrical Engineering, Univ. of Melbourne, 1993.

[16] M. Widenius and D. Axmark. *MySQL Reference Manual: Documentation from the Source.* O'Reilly Community Press, 2002.

Functional Notation and Lazy Evaluation in Ciao

Amadeo Casas[1] Daniel Cabeza[2] Manuel Hermenegildo[1,2]

amadeo@cs.unm.edu, herme@unm.edu,
{dcabeza, herme}@fi.upm.es

[1]Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, Albuquerque, NM, USA.
[2]School of Computer Science, T. U. Madrid (UPM), Madrid, Spain

Abstract

Certain aspects of functional programming provide syntactic convenience, such as having a designated implicit output argument, which allows function call nesting and sometimes results in more compact code. Functional programming also sometimes allows a more direct encoding of lazy evaluation, with its ability to deal with infinite data structures. We present a syntactic functional extension of Prolog covering function application, predefined evaluable functors, functional definitions, quoting, and lazy evaluation. The extension is also composable with higher-order features. We also highlight the Ciao features which help implementation and present some data on the overhead of using lazy evaluation with respect to eager evaluation.

Keywords: Declarative Languages, Logic Programming, Functional Programming, Logic-Functional Programming, Lazy Evaluation.

1 Introduction

Logic Programming offers a number of features, such as nondeterminism and partially instantiated data structures, that give it expressive power beyond that of functional programming. However, certain aspects of functional programming provide in turn syntactic convenience. This includes for example having a syntactically designated output argument, which allows the usual form of function call nesting and sometimes results in more compact code. Also, lazy evaluation, which brings the ability to deal with infinite (non-recursive) data structures [22, 1], while subsumed by logic programming features such as delay declarations, enjoys a more direct encoding in functional programming. Bringing this syntactic convenience to logic programming can result in a more compact program representation in certain cases and is therefore a desirable objective.

With this objective in mind, in this paper we present a design for an extensive functional syntactic layer for logic programs and its implementation in the Ciao system [4].[1] While the idea of adding functional features to logic programming systems is clearly not new [2, 3, 21], and there are currently a good number of systems which integrate functions into some form of logic programming, such as Oz [16] and Mercury [25] (developed simultaneously and independently from Ciao), or perform a full integration of functional and logic programming, such as Curry [15, 20], we feel that our proposal and its implementation has peculiarities which make it interesting in itself, for its application to standard Prolog systems or otherwise.

[1]The design described herein actually includes some minor changes with respect to the current version which will be available in the upcoming release of Ciao.

Our target system, Ciao, offers a complete Prolog system, supporting ISO-Prolog, with a novel modular design [9] which allows both restricting and extending the language. This design has allowed us to add functional features to it completely at the source (Prolog) level, and without modifying at all the compiler or the low-level machinery. The implementation is made by means of a number of Ciao *packages* [9], i.e., through the same extension mechanism by which other syntactic and semantic extensions are supported in Ciao, including constraints, objects, feature terms/records, persistence, several control rules, etc., giving Ciao its multi-paradigm flavor. However, our approach to the implementation of functions is actually also relatively straightforward to implement in any modern Prolog system [14]. We will highlight the Ciao features which make implementation in our view smoother and more orthogonal with the upfront intention of contributing if possible to the design of other logic programming systems, which is certainly an important objective of Ciao.

The rest of the paper is organized as follows: first, we discuss in Section 2 our general approach to integrating functional notation in a logic programming system. Section 3 presents how we implemented this approach in Ciao. Section 4 shows an example of the use of lazy evaluation, and how it is achieved by our implementation. Section 5 presents some experimental results. We come to a close in Section 6 with our conclusions.

2 Functional Notation in Ciao

2.1 Basic Concepts and Notation

Our notion of functional notation for logic programming departs in a number of ways from previous proposals. The fundamental one is that functional notation simply provides *syntactic sugar* for defining and using predicates as if they were functions, but they can still retain the power of predicates. In this model, any function definition is in fact defining a predicate, and any predicate can be used as a function. The predicate associated with a function has the same name and one more argument, meant as the place holder for the result of the function. This argument is by default added to the right, i.e., it is the last argument, but can be defined otherwise by using the appropriate declaration. The syntax extensions provided for functional notation are the following:

Function applications Any term preceded by the `~` operator is a function application, as can be seen in the goal `write(~arg(1,T))`, which is equivalent to the sequence `arg(1,T,A), write(A)`. To use a predicate argument other than the last as the return argument, a declaration like "`:- fun_return functor(~,_,_).`" can be used, so that `~functor(f,2)` is evaluated to `f(_,_)` (where `functor/3` is the standard ISO-Prolog builtin). This definition of the return argument can also be done on the fly in each invocation in the following way: `~functor(~,f,2)`. Functors can be declared as *evaluable* (i.e., being in calls in functional syntax) by using the declaration `function/1`. This allows avoiding the need to use the `~` operator. Thus, "`:- function arg/2.`" allows writing `write(arg(1,T))` instead of `write(~arg(1,T))` as above. This declaration can be combined with the previous one: "`:- function functor(~,_,_).`". Note that all these declarations, as is customary in Ciao, are local to the module where they are included. Finally, the `~` operator does not need to be used within a functional definition (see below) for the functor being defined.

Predefined evaluable functors In addition to functors declared with the declaration `function/1`, several functors are evaluable by default, those being:

- All the functors understood by `is/2`. Thus, it is possible to use `write(A+B)` to denote `T is A+B, write(T)`. This feature can be disabled by a declaration `:- function arith(false)` (and reverted by using `true` instead of `false`).
- The functors used for disjunctive and conditional expressions, `(|)/2` and `(?)/2`. A disjunctive expression has the form `(V1|V2)`, and its value when first evaluated is `V1`, and on backtracking `V2`. A conditional expression has the form `(Cond ? V1)`, or more commonly `(Cond ? V1 | V2)`, and its value, if the execution of `Cond` as a goal succeeds, is `V1`, otherwise in the first form it causes backtracking, and on the second form its value is `V2`. Due to operator precedences, a nested expression `(Cond1 ? V1 | Cond2 ? V2 | V3)` is evaluated as `(Cond1 ? V1 | (Cond2 ? V2 | V3))`.

Functional definitions A functional definition is composed of one of more functional clauses. A functional clause is written using the binary operator `:=`, as in

```
opposite(red) := green.
```

Functional clauses can also have a body, which is executed before the result value is computed. It can serve as a guard for the clause or to provide the equivalent of where-clauses in functional languages:

```
fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.
```

In a functional clause, the defined function does not need to be preceded by `~`. Note that guards can often be defined more compactly using conditional expressions:

```
fact(N) := N = 0 ? 1
         | N > 0 ? N * fact(--N).
```

The translation of functional clauses defining recursive predicates maintains the tail recursion of the equivalent predicate, thus allowing the usual compiler optimizations.

Quoting functors In clause heads (independently of whether they are defined as predicates or functions) functors can be prevented from being evaluated by using the `(^)/1` prefix operator (read as "quote"), as in

```
pair(A,B) := ^(A-B).
```

Note that this just prevents the evaluation of the principal functor of the enclosed term, not the possible occurrences of other evaluable functors inside.

Scoping When using function applications inside the goal arguments of meta-predicates, there is an ambiguity as they could be evaluated either in the scope of the outer execution or the in the scope of the inner execution. The chosen behavior is by default to evaluate function applications in the scope of the outer execution, and if they should be evaluated in the inner scope, the goal containing the function application needs to be escaped with the `(^^)/1` prefix operator, as in `findall(X, (d(Y), ^^(X = ~f(Y)+1)), L)` (which could also be written as `findall(X, ^^ (d(Y), X = ~f(Y)+1), L)`).

Laziness Lazy evaluation is a program evaluation technique used particularly in functional languages. When using lazy evaluation, an expression is not evaluated as soon as it is assigned, but rather when the evaluator is forced to produce the value of the expression. The `when`, `freeze`, or `block` control primitives present in many modern logic programming systems are

more powerful than lazy evaluation. However, they lack the simplicity of use and cleaner semantics of functional lazy evaluation. In our design, a function (or predicate) can be declared as lazy via the declarations: "`:- lazy function function_name/N.`". (or, equivalently in predicate version, "`:- lazy pred_name/M.`", where $M = N + 1$). In order to achieve the intended behavior, the execution of each function declared as lazy is suspended until the return value of the function is needed.

Definition of real functions In the previous scheme, functions are (at least by default) not forced to provide a single solution for their result, and, furthermore, they can be partial, producing a failure when no solution can be found. A predicate defined as a function can be declared to behave as a real function using the declaration "`:- funct name/N.`". Such predicates are then converted automatically to real functions by adding pruning operators and a number of Ciao assertions [23] which pose (and check) additional restrictions such as determinacy, modedness, etc., so that the semantics will be the same as in traditional functional programming.

2.2 Examples

We now illustrate the use of the functionality introduced above through examples. The following example defines a simple unary function `der(X)` which returns the derivative of a polynomial arithmetic expression:

```
der(x)          := 1.
der(C)          := 0                        :- number(C).
der(^(A + B))   := ^(der(A) + der(B)).
der(^(C * A))   := ^(C * der(A))            :- number(C).
der(^(x ** N)) := ^(N * ^(x ** (N - 1))) :- integer(N), N > 0.
```

Note that the code is a bit obfuscated because it uses frequently evaluable functors which need to be quoted. This quoting can be prevented by including the directive mentioned above to make functors understood by `is/2` not evaluable. Thus, the arithmetic evaluation in the last clause needs to be explicitly requested. The new program follows:

```
:- function(arith(false)).
der(x)      := 1.
der(C)      := 0                 :- number(C).
der(A + B)  := der(A) + der(B).
der(C * A)  := C * der(A)        :- number(C).
der(x ** N) := N * x ** ~(N - 1) :- integer(N), N > 0.
```

Both of the previous code fragments translate to the following code:

```
der(x, 1).
der(C, 0) :-
          number(C).
der(A + B, X + Y) :-
          der(A, X),
          der(B, Y).
der(C * A, C * X) :-
          number(C),
```

```
          der(A, X).
der(x ** N, N * x ** N1) :-
          integer(N),
          N > 0,
          N1 is N - 1.
```

Note that with our solution the programmer may use `der/2` as a function or as a predicate indistinctly.

As a simple example of the use of lazy evaluation consider the following definition of a function which returns the (potentially) infinite list of integers starting with a given one:

```
:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].
```

One of the nice features of functional notation is that it allows writing regular types (used in Ciao assertions [23] and checked by the preprocessor) in a very compact way:

```
color := red | blue | green.
list := [] | [_ | list].
list_of(T) := [] | [~T | list_of(T)].
```

which are equivalent to (note the use of higher-order in the third example, to which we will refer again later):

```
color(red). color(blue). color(green).
list([]). list([_|T]) :- list(T).
list_of(_, []). list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

3 Implementation Details

As mentioned previously, certain Ciao Prolog features have simplified its extension to handle functional notation. In the following we sketch the features of Ciao we used and then how we applied them for our needs.

3.1 Code Translations in Ciao

Traditionally, Prolog systems have included the possibility of changing the syntax of the source code by the use of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [24, 12]. This is usually how, e.g., definite clause grammars (DCG's) are typically implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and, specially, expansions are global, affecting a number of files. Furthermore, which files are affected cannot be determined statically, because these features are implemented as a side-effect, rather than a declaration, and they are meant to be active after they are read by the code processor (top-level, compiler, etc.) and remain active from then on. As a result, it is impossible by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees.

In order to solve these problems, these features were redesigned in Ciao, so that it is still possible to define source translations and operators, but such translations are local to the module or user

file defining them. Also, these features are implemented in a way that has a well-defined behavior in the context of a stand-alone compiler (and this has been applied in the Ciao compiler, `ciaoc` [8]). In particular, the directive `load_compilation_module/1` allows separating code that will be used at compilation time (e.g., the code used for translations) from code which will be used at run-time. It loads the module defined by its argument *into the compiler*.

In addition, in order to make the task of writing expansions easier, the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which, again, *affect only the current module*.

The solutions we provided to implement functions in Ciao are based on the possibility of defining these source translations. In particular, we have used the directives `add_sentence_trans/1` (to define *sentence translations*) and `add_goal_trans/1` (to define *goal translations*). A sentence translation is a predicate which will be called by the compiler to possibly convert each term read by the compiler to a new term, which will be used by the compiler in place of the original term. A goal translation is a predicate which will be called by the compiler to possibly convert each goal present in the clauses of the current text to another goal to replace the original one. Note the proposed model can be implemented in other Prolog systems using similarly using `term_expansion/2` and operator declarations, but that having operators and syntactic transformation predicates local to modules makes it scalable and amenable to combination with other packages and syntactic extensions.

3.2 Ciao Packages

Packages in Ciao are libraries which define extensions to the language, and have a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Any auxiliary code needed at compile-time (e.g., translations) is included in a separate module which is to be loaded into the compiler via a `load_compilation_module/1` directive placed in the main file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file.

In our implementation of functional notation in Ciao we have provided two packages: one for the bare function features without lazy evaluation, and an additional one to provide the lazy evaluation features. The reason for this is that in many cases the lazy evaluation features are not needed and thus the translation procedure is simplified.

3.3 The Ciao Implementation of Functional Extensions

To translate the functional definitions, we have used as mentioned above the `add_sentence_trans/1` directive to provide a translation procedure which transforms each functional clause to a predicate clause, adding to the function head the output argument, in order to convert it to the predicate head. This translation procedure also deals with functional applications in heads, as well as with `function` directives.

On the other hand, we have used the `add_goal_trans/1` directive to provide a translation procedure for dealing with function applications in bodies. The rationale to using a goal translation is that each function application inside a goal will be replaced by a variable, and the goal will be preceded by a call to the predicate which implements the function in order to provide a value for that variable.

An additional sentence translation is provided to handle the `lazy` directives. The translation of a lazy function into a predicate is done in two steps. First, the function is converted into a predicate using the procedure explained above. Then, the resulting predicate is transformed to suspend its execution until the value of the output variable is needed. This suspension is achieved by the use of the `freeze/1` control primitive that many modern logic programming systems implement quite efficiently [11] (`block` or `when` declarations can obviously also be used, but we explain the transformation in terms of `freeze` because it is more widespread). This translation will rename the original predicate to an internal name and add a *bridge predicate* with the original name which invokes the internal predicate through a call to `freeze/1`. This will delay the execution of the internal predicate until its result is required, which will be detected as a binding (i.e., demand) of its output variable. The following section will provide a detailed example of the translation of a lazy function. The implementation with `block` is even simpler since no bridge predicate is needed.

We show below, for reference, the main files for the Ciao library packages `functions`:

```
% functions.pl
:- include(library('functions/ops')).
:- load_compilation_module(library('functions/functionstr')).
:- add_sentence_trans(defunc/3).
:- add_goal_trans(defunc_goal/3).
```

and `lazy` (which will usually be used in conjunction with the first one):

```
% lazy.pl
:- include(library('lazy/ops')).
:- use_module(library(freeze)).
:- load_compilation_module(library('lazy/lazytr')).
:- add_sentence_trans(lazy_sentence_translation/3).
```

The Ciao system source provides the actual detailed coding along the lines described above.

4 Lazy Functions: an Example

In this section we show an example of the use of lazy evaluation, and how a lazy function is translated by our Ciao package.

Figure 1 shows in the first row the definition of a lazy function which returns the infinite list of Fibonacci numbers, in the second row its translation into a lazy predicate[2] (by the `functions` package) and in the third row the expansion of that predicate to emulate lazy evaluation (where `fiblist_$$lazy$$` stands for a fresh predicate name).

In the `fiblist` function defined, any element in the resulting *infinite* list of Fibonacci numbers can be referenced, as for example, `nth(X, ~fiblist, Value)`. The other functions used in the definition are `tail/2`, which is defined as lazy and returns the tail of a list; `zipWith/3`, which is also defined as lazy and returns a list whose elements are computed by a function having as arguments the successive elements in the lists provided as second and third argument;[3] and `add/2`, defined as `add(X,Y) := X + Y`.

Note that the `zipWith/3` function (respectively the `zipWith/4` predicate) is in fact a *higher-order* function (resp. predicate). Ciao provides in its standard library a package to support higher-order [7, 5, 10], which together with the `functions` and `lazy` packages (and, optionally, the type inference and checking provided by the Ciao preprocessor [18]) basically provide all the functionality present in modern functional languages.

[2] The `:- lazy function fiblist/0.` declaration is converted into a `:- lazy fiblist/1.` declaration.

[3] It has the same semantics as the `zipWith` function in Haskell.

```
:- lazy function fiblist/0.
fiblist := [0, 1 | ~zipWith(add, FibL, ~tail(FibL))]
        :- FibL = fiblist.
```

```
:- lazy fiblist/1.
fiblist([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

```
fiblist(X) :-
          freeze(X, 'fiblist_$$lazy$$'(X)).

'fiblist_$$lazy$$'([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

Figure 1: Code translation for a fibonacci function, to be evaluated lazily.

5 Some Performance Measurements

Since the functional extensions proposed simply provide a syntactic bridge between functions and predicates, there are only a limited number of performance issues worth discussing. For the case of *real* functions, it is well known that performance gains can be obtained from the knowledge of a certain function or predicate being moded (e.g., for a function all arguments being ground on input and the "designated output" argument being ground on output), determinate, non-failing, etc. [26, 17, 19]. In Ciao this information can in general (i.e., for any predicate or function) be inferred by the Ciao preprocessor or declared with Ciao assertions [18, 23]. As mentioned before, for declared "real" (`func`) functions, the corresponding information is added automatically. Some results on current Ciao performance when this information is available are presented in [19].

In the case of lazy evaluation of functions, the main goal of the technique presented herein is not really any increase in performance, but achieving new functionality and convenience through the use of code translations and delay declarations. However, while there have also been some studies of the overhead introduced by delay declarations, it is interesting to see how this overhead affects our implementation of lazy evaluation by observing its performance. Consider the `nat/2` function in Figure 2, a simple function which returns a list with the first N numbers from an (infinite) list of natural numbers.

```
:- function nat/1.
nat(N) := ~take(N, nums_from(0)).

:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].
```

```
:- function nat/1.
:- function nats/2.
nat(X) := nats(0, X).
nats(X, Max) := X > Max ? []
              | [X | nats(X+1, Max)].
```

Figure 2: Lazy and eager versions of function `nat(X)`.

Function `take/2` in turn returns the list of the first N elements in the input list. This `nat(N)` function cannot be directly executed eagerly due to the infinite list provided by the `nums_from(X)` function, so that, in order to compare time and memory results between lazy and eager evaluation, an equivalent version of that function is provided.

Table 1 reflects the time and memory overhead of the lazy evaluation version of `nat(X)` and that of the equivalent version executed eagerly. As a further example, Table 2 shows the results for a quicksort function executed lazily in comparison to the eager version of this algorithm. All the results were obtained by averaging ten runs on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 2.0, and with the translation of Figure 1.

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.030	1503.2	0.002	491.2
100 elements	0.276	10863.2	0.016	1211.2
1000 elements	3.584	104463.0	0.149	8411.2
2000 elements	6.105	208463.2	0.297	16411.2
5000 elements	17.836	520463.0	0.749	40411.2
10000 elements	33.698	1040463.0	1.277	80411.2

Table 1: Performance for `nat/2` (time in ms. and heap sizes in bytes).

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.091	3680.0	0.032	1640.0
100 elements	0.946	37420.0	0.322	17090.0
1000 elements	13.303	459420.0	5.032	253330.0
5000 elements	58.369	2525990.0	31.291	1600530.0
15000 elements	229.756	8273340.0	107.193	5436780.0
20000 elements	311.833	11344800.0	146.160	7395100.0

Table 2: Performance for `qsort/2` (time in ms. and heap sizes in bytes).

We can observe that there is certainly an impact on the execution time when functions are evaluated lazily, but even with this version the results are quite acceptable if we take into account that the execution of the predicate does really suspend. Related to memory consumption we show heap sizes, without garbage collection (in order to observe the raw memory consumption rate). Lazy evaluation implies as expected some memory overhead due to the need to copy (freeze) program goals into the heap. Also, while comparing with standard lazy functional programming implementations is beyond the scope of this paper, some simple tests done for sanity check purposes (with HUGS) show that the results are comparable, our implementation being for example slower on `nat` but faster on `qsort`, presumably due to the different optimizations being performed by the compilers.

An example when lazy evaluation can be a better option than eager evaluation in terms of performance and not only convenience is found in a concurrent or distributed system environment (such as, e.g., [13]), and in the case of Ciao also within the active modules framework [4, 6]. The example in Figure 3 uses a function, defined in an active module, which returns a big amount of data. Function `test/0` in module `module1` needs to execute function `squares/1`, in (active, i.e., remote)

module `module2`, which will return a very long list (which could be infinite for our purposes). If `squares/1` were executed eagerly then the entire list would be returned, to immediately execute the `takeWhile/2` function with the entire list. `takeWhile/2` returns the first elements of a (possibly infinite) list while the specified condition is true. But creating the entire initial list is very wasteful in terms of time and memory requirements. In order to solve this problem, the `squares/1` function could be moved to module `module1` and merged with `takeWhile/2` (or, also, they could exchange a size parameter). But rearranging the program is not always possible and may perhaps complicate other aspects of the overall program design.

But if the `squares/1` function is evaluated lazily, it is possible to keep the definitions unchanged and in different modules, so that there will be a smaller time and memory penalty for generating and storing the intermediate result. As more values are needed by the `takeWhile/2` function, more values in the list returned by `squares/1` are built (in this example, only while the new generated value is less than 10000), considerably reducing the time and memory consumption that the eager evaluation would take.

```
:- module(module1, [test/1], [functions, lazy, hiord,actmods]).
:- use_module(library('actmods/webbased_locate')).

:- use_active_module(module2, [squares/2]).

:- function takeWhile/2.
takeWhile(P, [H|T]) := P(H) ? [H | takeWhile(P, T)]
                            | [].
:- function test/0.
test := takeWhile(condition, squares).
condition(X) :- X < 10000.
```

```
:- module(module2, [squares/1], [functions, lazy, hiord]).

:- lazy function squares/0.
squares := map_lazy(take(1000000, nums_from(0)), square).

:- lazy function map_lazy/2.
map_lazy([], _)     := [].
map_lazy([X|Xs], P) := [~P(X) | map_lazy(Xs, P)].

:- function take/2.
take(0, _)     := [].
take(X, [H|T]) := [H | take(X-1, T)] :- X > 0.

:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].

:- function square/1.
square(X) := X * X.
```

Figure 3: A distributed (active module) application using lazy evaluation.

6 Conclusions

We have presented a functional extension of Prolog, which includes the possibility of evaluating functions lazily. The proposed approach has been implemented in Ciao and is used now throughout the libraries and other system code (including the preprocessor and documenter) as well as in a number of applications written by the users of the system.

The performance of the package has been tested with several examples. As expected, evaluating functions lazily implies some overhead (and also additional memory consumption) with respect to eager evaluation. This justifies letting the user choose via annotations for which functions or predicates lazy evaluation is desired. In any case, the main advantage of lazy evaluation is to make it easy to work with infinite (non-periodic) data structures in the manner that is familiar to functional programmers.

Acknowledgements

The authors would like to thank the anonymous referees for their useful comments. Manuel Hermenegildo and Amadeo Casas are supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. This work was also funded in part by the EC Future and Emerging Technologies program IST-2001-38059 *ASAP* project and by the Spanish MEC TIC 2002-0055 *CUBICO* project.

References

[1] S. Antoy. Lazy evaluation in logic. In *Symp. on Progr. Language Impl. and Logic Progr (PLILP'91)*, pages 371–382. Springer Verlag, 1991. LNCS 528.

[2] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *International Symposium on Logic Programming*, pages 160–168, Silver Spring, MD, February 1984. IEEE Computer Society.

[3] G. L. Bellia, M. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. P. (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[5] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[6] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.clip.dia.fi.upm.es/`.

[7] D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[8] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.

[9] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[10] D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

[11] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[12] M. Carlsson and J. Widen. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, April 1994.

[13] M. Carro and M. Hermenegildo. A simple approach to distributed objects in prolog. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[14] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[15] M. Hanus et al. Curry: An Integrated Functional Logic Language. `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

[16] S. Haridi and N. Franzén. *The Oz Tutorial*. DFKI, February 2000. Available from `http://www.mozart-oz.org`.

[17] F. Henderson et al. *The Mercury Language Reference Manual*. URL: `http://www.cs.mu.oz.au/research/mercury/information/doc/ reference_manual_toc.html`.

[18] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[19] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3507 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

[20] J. Moreno Navarro and M. Rodríguez-Artalejo. BABEL: A functional and logic programming language based on constructor discipline and narrowing. In *Conf. on Algebraic and Logic Programming (ALP)*, LNCS 343, pages 223–232, 1989.

[21] L. Naish. Adding equations to NU-Prolog. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, number 528 in Lecture Notes in Computer Science, pages 15–26, Passau, Germany, August 1991. Springer-Verlag.

[22] S. Narain. Lazy evaluation in logic programming. In *Proc. 1990 Int. Conference on Computer Languages*, pages 218–227, 1990.

[23] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[24] Quintus Computer Systems Inc., Mountain View CA 94041. *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[25] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[26] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

Views and Iterators for Generic Constraint Implementations

Christian Schulte[1] Guido Tack[2]

[1] IMIT, KTH - Royal Institute of Technology, Sweden

[2] PS Lab, Saarland University, Saarbrücken, Germany

schulte@imit.kth.se, tack@ps.uni-sb.de

Abstract

This paper introduces an architecture for generic constraint implementations based on variable views and range iterators. Views allow, for example, to scale, translate, and negate variables. The paper shows how to make constraint implementations generic and how to reuse a single generic implementation with different views for different constraints. Applications of views exemplify their usefulness and their potential for simplifying constraint implementations. We introduce domain operations compatible with views based on range iterators. The paper evaluates different implementation techniques for the presented architecture.

1 Introduction

A challenging aspect in developing and extending a constraint programming system is implementing a *comprehensive* set of constraints. Ideally, a system should provide simple, expressive, and efficient abstractions that ease development and reuse of constraint implementations.

This paper contributes a new architecture based on variable views and range iterators. The architecture comprises an additional level of abstraction to decouple variable implementations from propagators (as constraint implementations). Propagators compute generically with variable views instead of variables. Views support operations like scaling, translation, and negation of variables. For example, a simple generic propagator for linear equality $\sum_{i=1}^{k} x_i = c$ can be used with a scale-view $x_i = a_i \cdot y_i$ to obtain an implementation of $\sum_{i=1}^{k} a_i \cdot y_i = c$. Variable views assist in implementing propagators on a higher level of abstraction.

Range iterators support powerful and efficient domain operations on variables and variable views. The operations can access and modify multiple values of a variable domain simultaneously. Range iterators are efficient as they help avoiding temporary data structures. They simplify propagators by serving as adaptors between variables and propagator datastructures.

The architecture is carefully separated from its implementation. Two different implementation approaches are presented and evaluated. An implementation using parametric polymorphism (such as templates in C++) is shown to not incur any runtime cost. The architecture can be used for arbitrary constraint programming systems and has been fully implemented in Gecode [2].

Plan of the paper. The next section presents a model for finite domain constraint programming systems. Sect. 3 introduces variable views and exemplifies their use. Sect. 4 introduces iterator-based domain operations that are applied to views in the following section. Variable views for set constraints are discussed in Sect. 6. In Sect. 7 implementation approaches for views and iterators are presented, followed by their evaluation in Sect. 8. The last section concludes.

2 Constraint Programming Systems

This section introduces the model for finite domain constraint programming systems considered in this paper and relates it to existing systems.

Variables and propagators. Finite domain constraint programming systems offer services to support constraint propagation and search. In this paper we are only concerned with variables used for constraint propagation. We assume that a constraint is implemented by a *propagator*. A propagator maintains a collection of variables and performs constraint propagation by executing operations on them. In the following we consider finite domain variables and propagators. A finite domain variable x has an associated *domain* $\mathrm{dom}(x)$ being a subset of some finite subset of the integers.

Propagators do not manipulate variable domains directly but use operations provided by the variable. These operations return information about the domain or update the domain. In addition, they handle failure (the domain becomes empty) and control propagation.

Value operations. A *value operation* on a variable involves a single integer as result or argument. We assume that a variable x with $D = \mathrm{dom}(x)$ provides the following value operations: $x.\mathrm{getmin}()$ returns $\min D$, $x.\mathrm{getmax}()$ returns $\max D$, $x.\mathrm{adjmin}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \geq n\}$, $x.\mathrm{adjmax}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \leq n\}$, and $x.\mathrm{excval}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \neq n\}$. These operations are typical for finite domain constraint programming systems like Choco [6], ILOG Solver [9, 11, 4], Eclipse [1], Mozart [8], and Sicstus [5]. Some systems provide additional operations such as for assigning values.

Domain operations. A *domain operation* supports simultaneous access or update of multiple values of a variable domain. In many systems this is provided by supporting an abstract set-datatype for variable domains, as for example in Choco [6], Eclipse [1], Mozart [8], and Sicstus [5]. ILOG Solver [9, 11, 4] only allows access by iterating over the values of a variable domain.

Subscription. When a propagator p is created, it *subscribes* to its variables. Subscription guarantees that p is executed whenever the domain of one of its variables changes according to an event. An event describes when the propagator requires execution: $e = \mathrm{fix}$ when $\mathrm{dom}(x)$ becomes a singleton, $e = \mathrm{min}$ if $\min \mathrm{dom}(x)$ changes, $e = \mathrm{max}$ if $\max \mathrm{dom}(x)$ changes, and $e = \mathrm{any}$ if $\mathrm{dom}(x)$ changes. For more on events, see for example [13].

Range and value sequences. Range notation $[n \,..\, m]$ is used for the set of integers $\{l \in \mathbb{Z} \mid n \leq l \leq m\}$. A *range sequence* $\mathrm{ranges}(I)$ for a finite integer set $I \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle [n_1 \,..\, m_1], \ldots, [n_k \,..\, m_k] \rangle$ such that I is covered ($\mathrm{set}(s) = I$, where $\mathrm{set}(s)$ is defined as $\bigcup_{i=1}^{k} [n_i \,..\, m_i]$) and the ranges are ordered by their smallest elements ($n_i \leq n_{i+1}$ for $1 \leq i < k$). The above range sequence is also written as $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$. Clearly, a range sequence is unique, none of its ranges is empty, and $m_i + 1 < n_{i+1}$ for $1 \leq i < k$.

A *value sequence* $\mathrm{values}(I)$ for a finite set of integers $I \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle n_1, \ldots, n_k \rangle$ (also written as $\langle n_i \rangle_{i=1}^{k}$) such that I is covered ($\mathrm{set}(s) = I$, where $\mathrm{set}(s) = \bigcup_{i=1}^{k} n_i$) and that the values are ordered ($n_i \leq n_{i+1}$ for $1 \leq i < k$). Again, this sequence is unique and has no duplicate values.

3 Variable Views with Value Operations

This section introduces variable views with value operations. The full design with domain operations and a discussion of their properties follows in Sect. 5.

Example 1 (Smart n-Queens) Consider the well-known finite domain constraint model for n-Queens using three alldifferent constraints: each queen is represented by a variable x_i ($0 \leq i < n$) with domain $\{0,\ldots,n-1\}$. The constraints state that the values of all x_i, the values of all $x_i - i$, and the values of all $x_i + i$ must be pairwise different for $0 \leq i < n$.

If the used constraint programming system lacks versions of alldifferent supporting that the values of $x_i + c_i$ are different, the user must resort to using additional variables y_i and constraints $y_i = x_i + c_i$ and the single constraint that the y_i are different. This approach is clearly not very efficient: it triples the number of variables and requires additional $2n$ binary constraints.

Systems with this extension of alldifferent must implement two very similar versions of the same propagator. This is tedious and increases the amount of code that requires maintenance. In the following we make propagators *generic*: the same propagator can be reused for several variants.

To make a propagator generic, all its operations on variables are replaced by operations on variable views. A *variable view* (view for short) implements the same operations as a variable. A view stores a reference to a variable. Invoking an operation on the view executes the appropriate operation on the view's variable. Multiple variants of a propagator can be obtained by instantiating the single generic propagator with multiple different variable views.

Offset-views. For an *offset-view* $v = \text{voffset}(x,c)$ for a variable x and an integer c, performing an operation on v results in performing an operation on $x + c$. The operations on the offset-view are:

$$\begin{array}{ll} v.\text{getmin}() := x.\text{getmin}() + c & v.\text{getmax}() := x.\text{getmax}() + c \\ v.\text{adjmin}(n) := x.\text{adjmin}(n-c) & v.\text{adjmax}(n) := x.\text{adjmax}(n-c) \\ v.\text{excval}(n) := x.\text{excval}(n-c) & \end{array}$$

To obtain both alldifferent propagators required by Example 1, also an *identity-view* is needed. An operation on an identity-view $\text{vid}(x)$ for a variable x performs the same operation on x. That is, identity-views turn variables into views to comply with propagators now computing with views.

Obtaining the two variants of alldifferent is straightforward: the propagator is made generic with respect to which view it uses. Using the propagator with both an identity-view and an offset-view yields the required propagators.

Sect. 7 discusses how views can be implemented whereas this section focuses on the architecture only. However, to give some intuition, in C++ for example, propagators can be made generic by implementing them as templates with the used view as template argument. Instantiating the generic propagator then amounts to instantiating the corresponding template with a particular view.

Views are orthogonal to the propagator. In the above example, offset-views can be used for any implementation of alldifferent using value operations. This includes the naive version propagating when variables become assigned or the bounds-consistent version [10].

Scale-views. In the above example, views allow to reuse the same propagator for variants of a constraint, avoiding duplication of code and effort. In the following, views can also simplify the implementation of propagators.

Example 2 (Linear inequalities) A common constraint is linear inequality $\sum_{i=1}^{n} a_i \cdot x_i \leq c$ (equality and disequality is similar) with integers a_i and c and variables x_i. In the following we restrict the a_i to be positive.

A typical bounds-propagator executes for $1 \leq j \leq n$:

$$x_j.\text{adjmax}(\lceil (c - l_j)/a_j \rceil) \quad \text{with} \quad l_j = \textstyle\sum_{i=1, i\neq j}^{n} a_i \cdot x_i.\text{getmin}()$$

Quite often, models feature the special case $a_i = 1$ for $1 \leq i \leq n$. For this case, it is sufficient to execute for $1 \leq j \leq n$:

$$x_j.\text{adjmax}(c - l_j) \quad \text{with} \quad l_j = \textstyle\sum_{i=1, i \neq j}^{n} x_i.\text{getmin}()$$

As this case is common, a system should optimize it. An optimized version requires less space (no a_i required) and less time (no multiplication, division, and rounding). But, a more interesting question is: can one just implement the simple propagator and get the full version by using views?

With scale-views, the simple implementation can be used in both cases. A *scale-view* $v = \text{vscale}(a, x)$ for a positive integer $a > 0$ and a variable x defines operations for $a \cdot x$:

$$\begin{array}{ll} v.\text{getmin}() := a \cdot x.\text{getmin}() & v.\text{getmax}() := a \cdot x.\text{getmax}() \\ v.\text{adjmin}(n) := x.\text{adjmin}(\lceil n/a \rceil) & v.\text{adjmax}(n) := x.\text{adjmax}(\lfloor n/a \rfloor) \\ v.\text{excval}(n) := \textbf{if } n \bmod a = 0 \textbf{ then } x.\text{excval}(n/a) & \end{array}$$

From the simpler implementation the special case (identity-views) and the general case (scale-views) can be obtained. Multiplication, division, and rounding is separated from actually propagating the inequality constraint. Views hence support separation of concerns and can simplify the implementation of propagators. In particular, multiplication, division, and rounding need to be implemented only once for the scale-view: any generic propagator can use scale-views.

Minus-views. Another common optimization is to implement binary and ternary variants of commonly used constraints. This optimization reduces the overhead with respect to both time and memory as no array is needed.

Example 3 (Binary linear inequality) Consider a propagator for $v_1 + v_2 \leq c$ with views v_1 and v_2 propagating as described in Example 2. With scale-views $v_1 = \text{vscale}(a_1, x_1)$ and $v_2 = \text{vscale}(a_2, x_2)$ the propagator also implements $a_1 \cdot x_1 + a_2 \cdot x_2 \leq c$ provided that $a_1, a_2 > 0$. However, $x_1 - x_2 \leq c$ cannot be obtained with scale-views. Even if scale-views allowed negative constants, it would be inefficient to multiply, divide, and round to just achieve negation.

A *minus-view* $v = \text{vminus}(x)$ for a variable x provides operations such that v behaves as $-x$. Its operations reflect that the smallest possible value for x is the largest possible value for $-x$ and vice versa:

$$\begin{array}{ll} v.\text{getmin}() := -x.\text{getmax}() & v.\text{getmax}() := -x.\text{getmin}() \\ v.\text{adjmin}(n) := x.\text{adjmax}(-n) & v.\text{adjmax}(n) := x.\text{adjmin}(-n) \\ v.\text{excval}(n) := x.\text{excval}(-n) & \end{array}$$

With minus-views, $x_1 - x_2 \leq c$ can be obtained from an implementation of $v_1 + v_2 \leq c$ with $v_1 = \text{vid}(x_1)$ and $v_2 = \text{vminus}(x_2)$. With an offset-view it is actually sufficient to implement $v_1 + v_2 \leq 0$. Then $x_1 + x_2 \leq c$ can be implemented by an identity-view $\text{vid}(x_1)$ for v_1 and an offset-view $\text{voffset}(x_2, -c)$ for v_2. But again, given just $v_1 + v_2 \leq 0$, an implementation for $x_1 - x_2 \leq c$ with $c \neq 0$ cannot be obtained.

Derived views. It is unnecessarily restrictive to define views in terms of variables. The actual requirement for a view is that its variable provides the same operations. It is straightforward to make views generic themselves: views can be defined in terms of other views. The only exception are identity-views as they serve the very purpose of casting a variable into a view. Views such as offset, scale, and minus are called *derived views*: they are derived from some other view.

With derived views being defined in terms of views, the first step to use a derived view is to turn a variable into a view by an identity-view. For example, a minus-view v for the variable x is obtained from a minus-view and an identity-view: $v = \text{vminus}(\text{vid}(x))$.

Example 4 (Binary linear inequality reconsidered) With the help of offset-views, minus-views, and scale-views, all possible variants of binary linear inequalities can now be obtained from a propagator for $v_1 + v_2 \leq 0$. For example, $a \cdot x_1 - x_2 \leq c$ with $a > 0$ can be obtained with $v_1 = \text{vscale}(a, \text{vid}(x_1))$ and $v_2 = \text{vminus}(\text{voffset}(\text{vid}(x_2), c))$ or $v_2 = \text{voffset}(\text{vminus}(\text{vid}(x_2)), -c)$.

Scale-views reconsidered. The coefficient of a scale-view is restricted to be positive. Allowing arbitrary non-zero constants a in a scale-view $s = \text{vscale}(a,x)$ requires to take the signedness of a into account. This can be seen for the following two operations (the others are similar):

$$s.\text{getmin}() \;:= \textbf{if } a < 0 \textbf{ then } a \cdot x.\text{getmax}() \textbf{ else } a \cdot x.\text{getmin}()$$
$$s.\text{adjmax}(n) := \textbf{if } a < 0 \textbf{ then } x.\text{adjmin}(\lfloor n/a \rfloor) \textbf{ else } x.\text{adjmax}(\lfloor n/a \rfloor)$$

This extension might be inefficient. Consider Example 2: inside the loop implementing propagation on all views, the decision whether the coefficient in question is positive or negative must be made. For modern computers, conditionals — in particular in tight loops — can reduce performance considerably. A more efficient way is to restrict scale-views to positive coefficients and use an additional minus-view for cases where negative coefficients are required.

Example 5 (Linear inequalities reconsidered) An efficient way to implement a propagator for linear inequality distinguishes positive and negative variables as in $\sum_{i=1}^{n} x_i + \sum_{i=1}^{m} -y_i \leq c$.

The propagator is simple: it consists of two parts, one for the x_i and one for the y_i. Both parts share the same implementation used with different views. To propagate to the x_i, identity-views are used. To propagate to the y_i, minus-views are used. Arbitrary coefficients are obtained from scale-views as shown above.

The example shows that it can be useful to make parts of a propagator generic and reuse these parts with different views. Puget presents in [10] an algorithm for the bounds-consistent alldifferent. The paper presents only an algorithm for adjusting the upper bounds of the variables x_i and states that the lower bounds can be adjusted by using the same algorithm on variables y_i where $y_i = -x_i$. With views, this technique for simplifying the presentation of an algorithm readily carries over to its implementation: the implementation can be reused together with minus-views.

Another aspect in the design of scale-views is numerical precision: for large a the values occurring in computations with a scale-view $\text{vscale}(a,v)$ can exceed the range of normal integer computations. A system can provide several scale-views that differ in which numerical datatype is used for computation. For example, a system can provide a scale-view that computes with normal integers or with double-precision floating point numbers.

Constant-views. Derived views exploit that views do not need to be implemented in terms of variables. This can be taken to the extreme in that a view has no access at all to a variable. A constant-view $v = \text{vcon}(c)$ for an integer c provides operations such that v behaves as a variable x being equal to c:

$$v.\text{getmin}() \;:= c \qquad v.\text{getmax}() \;:= c$$
$$v.\text{adjmin}(n) := \textbf{if } n > c \textbf{ then fail} \qquad v.\text{adjmax}(n) := \textbf{if } n < c \textbf{ then fail}$$
$$v.\text{excval}(n) \;:= \textbf{if } n = c \textbf{ then fail}$$

Example 6 (Ternary linear inequalities) Another optimization for linear constraints are ternary variants. Given a propagator for $v_1 + v_2 + v_3 \leq c$ and using a constant-view $\text{vcon}(0)$ for one of the views v_i, all binary variants as discussed earlier can be obtained.

In summary, for linear inequalities (this carries over to linear equalities and disequalities), views support many optimized special cases from just two implementations (the general n-ary case and the ternary case). These implementations are simple as they do not need to consider coefficients.

Events. The handling of events in subscribing to a variable is straightforward. Clearly, for a constant-view subscription does nothing. For all but the minus-view the events on the variable and the events on the derived view coincide. If $v' = \text{vminus}(v)$, then a change of the minimum of v corresponds to a change of the maximum of v' and vice versa.

4 Domain Operations and Range Iterators

Today's constraint programming systems support domain operations either only for access or by means of an explicitly represented abstract datatype. In this paper, we propose domain operations based on range iterators. These operations are shown to be simple, expressive, and efficient. Additionally, range iterators are essential for views as presented in Sect. 5.

Range iterators. A *range iterator* r for a range sequence $s = \langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ allows to iterate over s: each of the $[n_i \,..\, m_i]$ can be obtained in sequential order but only one at a time. A range iterator r provides the following operations: $r.\text{done}()$ tests whether all ranges have been iterated, $r.\text{next}()$ moves to the next range, and $r.\text{min}()$ and $r.\text{max}()$ return the minimum and maximum value for the current range. By $\text{set}(r)$ we refer to the set defined by an iterator r (which must coincide with $\text{set}(s)$).

A possible implementation of a range iterator r for s maintains an index i_r which is initially $i_r = 1$, the operations can then be defined as:

$$\begin{array}{ll} r.\text{done}() := i_r > k & r.\text{next}() := (i_r \leftarrow i_r + 1) \\ r.\text{min}() := n_{i_r} & r.\text{max}() := m_{i_r} \end{array}$$

A range iterator hides its implementation. Iteration can be by position as above, but it can also be by traversing a list. The latter is particularly interesting if variable domains are implemented as lists of ranges themselves.

Iterators are consumed by iteration. Hence, if the same sequence needs to be iterated twice, a fresh iterator is needed. If iteration is cheap, a reset-operation for an iterator can be provided so that multiple iterations are supported by the same iterator. For more expensive iterators, a solution is discussed later.

Domain operations. Variables are extended with operations to access and modify their domains with range iterators. For a variable x, the operation $x.\text{getdom}()$ returns a range iterator for $\text{ranges}(\text{dom}(x))$. For a range iterator r the operation $x.\text{setdom}(r)$ updates $\text{dom}(x)$ to $\text{set}(r)$ provided that $\text{set}(r) \subseteq \text{dom}(x)$. The responsibility for ensuring that $\text{set}(r) \subseteq \text{dom}(x)$ is left to the programmer and hence requires careful consideration. Later richer (and safe) domain operations are introduced. The operation $x.\text{setdom}(r)$ is *generic* with respect to r: any range iterator can be used.

Domain operations can offer a substantial improvement over value operations, if many values need to be removed from a variable domain simultaneously. Assume a typical implementation of a variable domain D which organizes $\text{ranges}(D) = \langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ as a linked-list. Removing a single element from D takes $O(k)$ time and might increase the length of the linked-list by one (introducing an additional hole). Hence, in the worst case, removing l elements takes $O(l(k+l))$ time. With domain operations based on iterators, removal takes $O(k+l)$ time.

Range iterators serve as simplistic abstract datatype to describe finite sets of integers. However, they provide some essential advantages over an explicit set representation. First, any range iterator regardless of its implementation can be used to update the domain of a variable. This turns out to allow for simple, efficient, and expressive updates of variable domains. Second, no costly memory management is required to maintain a range iterator as it provides access to only one range at a time. Third, iterators are essential in providing domain operations on variable views as will be discussed in Sect. 5.

Intersection iterators. Let us consider intersection as an example for computing with range iterators. Intersection is computed by an intersection iterator r (returned by iinter(a,b)) taking two range iterators a and b as input where $\text{set}(r) = \text{set}(a) \cap \text{set}(b)$. The intersection iterator maintains integers n and m for storing the smallest and largest value of its current range. When initialized, the next operation is executed once. The operations for access are obvious: r.min() returns n and r.max() returns m. The other operations are:

r.done() := a.done() $\vee$ b.done()
r.next() := **if** a.done() $\vee$ b.done() **then return**
repeat
while $\neg a$.done() $\wedge$ (a.max() $<$ b.min()) **do** a.next()
if a.done() **then return**
while $\neg b$.done() $\wedge$ (b.max() $<$ a.min()) **do** b.next()
if b.done() **then return**
until a.max() $\geq$ b.min()
$n \leftarrow \max(a.\min(), b.\min())$; $m \leftarrow \min(a.\max(), b.\max())$
if a.max() $<$ b.max() **then** a.next() **else** b.next()

The **repeat**-loop iterates a and b until their ranges overlap. The tests whether a or b are done ensure that no operation is performed on a done iterator. The remainder computes the resulting range and prepares for computing a next range.

The iterators a and b can be arbitrary iterators (again, the intersection iterator is *generic*), so it is easy to obtain an iterator that computes the intersection of three iterators by using two intersection iterators. Intersection is but one example for a generic iterator, other useful iterators are for example: $r = \text{iunion}(a,b)$ for iterating the union of a and b ($\text{set}(r) = \text{set}(a) \cup \text{set}(b)$) and $r = \text{iminus}(a,b)$ for iterating the set difference of a and b ($\text{set}(r) = \text{set}(a) \setminus \text{set}(b)$).

Example 7 (Propagating equality) Consider a propagator that implements domain-consistent equality: $x = y$ (assuming that x and y are variables, views are discussed later). The propagator can be implemented as follows: get range iterators for x and y by $rx = x.\text{getdom}()$ and $ry = y.\text{getdom}()$, create an intersection iterator $ri = \text{iinter}(rx, ry)$, update one of the variable domains by $x.\text{setdom}(ri)$, and copy the domain from x to y by $y.\text{setdom}(x.\text{getdom}())$.

Cache-iterators. The above example suggests that for some propagators it is better to actually create an intermediate representation of the range sequence computed by an iterator. The intermediate representation can be reused as often as needed. This is achieved by a *cache-iterator*: it takes an arbitrary range iterator as input, iterates it completely, and stores the obtained ranges in an array. Its actual operations then use the array. The cache-iterator also implements a reset operation as discussed above. By this, the possibly costly input iterator is used only once, while the cache-iterator can be used as often as needed.

Richer domain operations. With the help of iterators, richer domain operations are effortless. For a variable x and a range iterator r, the operation $x.\text{adjdom}(r)$ replaces $\text{dom}(x)$ by $\text{dom}(x) \cap \text{set}(r)$, whereas $x.\text{excdom}(r)$ replaces $\text{dom}(x)$ by $\text{dom}(x) \setminus \text{set}(r)$:

$$x.\text{adjdom}(r) := x.\text{setdom}(\text{iinter}(x.\text{getdom}(), r))$$
$$x.\text{excdom}(r) := x.\text{setdom}(\text{iminus}(x.\text{getdom}(), r))$$

Value versus range iterators. Another design choice is to base domain operations on value iterators: iterate values rather than ranges of a set. This is not efficient: a value sequence is considerably longer than a range sequence (in particular for the common case of a singleton range sequence).

For implementing propagators, however, it can be simpler to iterate values. This can be achieved by a range-to-value iterator. A value iterator v has the operations $v.\text{done}()$, $v.\text{next}()$, and $v.\text{val}()$ to access the current value. A range-to-value iterator takes a range iterator as input and returns a value iterator iterating the values of the range sequence. The inverse is a value-to-range iterator: it takes as input a value iterator and returns the corresponding range iterator. For a value-to-range iterator it is helpful to allow duplicates in the increasing value sequence. This iterator has many applications, as is explained below.

Iterators as adaptors. Global constraints are typically implemented by a propagator computing over some involved data structure, such as for example a variable-value graph for domain-consistent all-distinct [12]. After propagation, the new variable domains must be transferred from the data structure to the variables. This can be achieved by using a range or value iterator as adaptor. The adaptor operates on the data structure and iterates the value or range sequence for a particular variable. The iterator (together with a value-to-range iterator for a value iterator) then can be passed to the appropriate domain operation.

5 Variable Views with Domain Operations

This section discusses domain operations for variable views using iterators.

Identity and constant views. Domain operations for identity-views and constant-views are straightforward. The domain operations for an identity-view $v = \text{vid}(x)$ use the domain operations on x: $v.\text{getdom}() := x.\text{getdom}()$ and $v.\text{setdom}(r) := x.\text{setdom}(r)$. For a constant-view $v = \text{vcon}(c)$, the operation $v.\text{getdom}()$ returns an iterator for the singleton range sequence $\langle [c \,..\, c] \rangle$. The operation $v.\text{setdom}(r)$ just checks whether the range sequence of r is empty.

Derived views. Domain operations for an offset-view $\text{voffset}(v,c)$ are provided by an offset-iterator. The operations of an offset-iterator o for a range iterator r and an integer c (created by $\text{ioffset}(r,c)$) are as follows:

$$o.\text{min}() := r.\text{min}() + c \qquad o.\text{max}() := r.\text{max}() + c$$
$$o.\text{done}() := r.\text{done}() \qquad o.\text{next}() := r.\text{next}()$$

The domain operations for an offset view $v = \text{voffset}(x,c)$ are as follows:

$$v.\text{getdom}() := \text{ioffset}(x.\text{getdom}(), c) \qquad v.\text{setdom}(r) := x.\text{setdom}(\text{ioffset}(r,-c))$$

For minus-views we just give the range sequence as iteration is obvious. For a given range sequence $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$, the negative sequence is obtained by reversal and sign change as $\langle [-m_{k-i+1} \,..\, -n_{k-i+1}] \rangle_{i=1}^{k}$. The same iterator for this sequence can be used both for setdom and getdom operations. Note that the iterator is quite complicated as it changes direction of the range sequence, possible implementations are discussed in Sect. 7.

Assume a scale-view $s = \text{vscale}(a,v)$ with $a > 0$ and $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ being a range sequence for v. If $a = 1$, the range sequence remains unchanged. Otherwise, the corresponding range sequence for s is $\langle \{a \cdot n_1\}, \{a \cdot (n_1+1)\}, \ldots, \{a \cdot m_1\}, \ldots, \{a \cdot n_k\}, \{a \cdot (n_k+1)\}, \ldots, \{a \cdot m_k\} \rangle$.

Assume that $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ is a range sequence for s. Then the ranges $[\lceil n_i/a \rceil \,..\, \lfloor m_i/a \rfloor]$ for $1 \le i \le k$ correspond to the required variable domain for v, however they do not necessarily form a range sequence as the ranges might be empty, overlapping, or adjacent. Iterating the range sequence is simple by skipping empty ranges and conjoining overlapping or adjacent ranges.

Consistency. An important issue is how views affect the consistency of a propagator. Let us first consider all views except scale-views. These views compute bijections on the values as well as on the ranges of a domain D. A bounds (domain) consistent propagator for a constraint C with variables $x_1,\ldots,x_n$ establishes bounds (domain) consistency for the constraint C with all the variables replaced by $v_k(x_k)$ (if v_k computes the view of x_k).

Scale-views only compute bijections on values: a range does not remain a range after multiplication. This implies that bounds consistent propagators do not establish bounds consistency on scale-views. Consider for example a bounds consistent propagator for alldifferent. With $x,y,z \in \{1,2\}$, alldifferent$(4x,4y,4z)$ cannot detect failure, while alldifferent(x,y,z) can. Note that this is not a limitation of our approach but a property of multiplication.

6 Views for Set Constraints

Views and iterators readily carry over to other constraint domains. This section shows how to apply them to finite sets.

Finite sets. Most systems approximate domains of finite set variables by a greatest lower and least upper bound [3]: $\mathrm{dom}(x) = (\mathrm{glb},\mathrm{lub})$. The fundamental operations are similar to domain operations on finite domain variables: x.glb() returns $\mathrm{glb}(x)$, x.lub() returns $\mathrm{lub}(x)$, x.adjglb(D) updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x) \cup D, \mathrm{lub}(x))$, and x.adjlub(D) updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x), \mathrm{lub}(x) \cap D)$.

All these operations take sets as arguments, so iterators play an important role here. In fact, range iterators provide exactly the operations that set propagators need: union, intersection, and complement. Most propagators thus do not require temporary datastructures.

As before, propagators now operate on views. In addition to the identity view, the following derived views make propagators more generic. As for finite domains, *constant-views* – like the empty set, the universe, or some arbitrary set – help derive binary propagators from ternary ones. For example, $s_1 \cap s_2 = s_3$ implements set disjointness if s_3 is the constant empty set. With a *complement-view*, $s_1 = s_2 \setminus s_3$ can be implemented as $s_1 = s_2 \cap \overline{s_3}$

Cross-domain views. With finite domain and set constraints in a single system, cross-domain views come into play. The most obvious cross-domain view is a finite domain variable viewed as singleton set. Using generic propagators, this immediately leads to domain-connecting constraints like $x \in s$ (using $\{x\} \subseteq s$), or $s = \{x_1,\ldots,x_k\}$ (using $s = \{x_1\} \uplus \cdots \uplus \{x_k\}$).

For cross-domain views, variable subscription handles the different sets of events. For instance, if a finite domain variable x has an any event, all propagators subscribed to upper-bound-change events of a singleton-view of x must be executed.

Cross-domain views can support more than one implementation for the same variable type. Set variables, for example, can be implemented with lower and upper bounds or with their full domain using ROBDDs [7]. A cross-domain view allows lower/upper bound propagators to operate on ROBDD-based sets, reusing propagators for which no efficient BDD representation exists.

7 Implementation

The presented architecture can be implemented as an orthogonal layer of abstraction for any constraint programming system. This section presents the fundamental mechanisms necessary for iterators and views.

Polymorphism. The implementation of generic propagators, views, and iterators requires *polymorphism*: propagators operate on different views, domain operations and iterators on different iterators. Both subtype polymorphism (through inheritance in Java, inheritance and virtual methods in C++) and parametric polymorphism (through templates in C++, generics in Java, polymorphic functions in ML or Haskell) can be used.

In C++, parametric polymorphism through templates is resolved at compile-time, and the generated code is monomorphic. This enables the compiler to perform aggressive optimizations, in particular inlining. The hope is that the additional layer of abstraction can be optimized away entirely. Some ML compilers also apply monomorphization, so similar results could be achieved. Java generics are compiled into casts and virtual method calls, probably not gaining much efficiency.

Achieving high efficiency in C++ with templates sacrifices expressiveness. Instantiation can *only* happen at compile-time. Hence, either C++ must be used for modeling, or all potentially required propagator variants must be provided by explicit instantiation. For n-ary constraints like linear inequality, this can be a real limitation, as all arrays must be monomorphic (only a single kind of view per array is allowed). The same holds true for n-ary set constraints, where especially constant-views for the empty set or the universe would come in handy.

A compromise is to use template-based polymorphism whenever possible and only resort to subtype polymorphism when necessary. This can be achieved by instantiating all propagators with special views that can be subclassed, and having an operation to construct such a view from an identity view. In Gecode, we currently only use template-based polymorphism. This makes the system very efficient, as will be analyzed in the next section.

For the instantiation of templates as well as for inlining, the code that should be instantiated or inlined must be available at compile time of the code that uses it. This is why most of the actual code in Gecode resides in C++ header files, slowing down compilation of the system. On the interface level however, no templates are used, such that the header files needed for *using* the library are reasonably small.

System requirements. Variable views and range iterators can be added as an orthogonal extension to existing systems. While value operations are not critical as discussed in Sect. 2, depending on which domain operations a system provides, efficiency can differ. In the worst case, domain operations need to be translated into value operations. This would decrease efficiency considerably, however intermediate computations on range iterators would still be carried out efficiently.

A particularly challenging aspect is reversal of range sequences required for the minus-iterator. One approach to implement reversal is to extend all iterators such that they can iterate both backwards and forwards. Another approach is similar to a cache-iterator: store the ranges generated from the input iterator in an array and iterate in reverse order from the array. In Gecode, we have chosen so far the latter approach due to its simplicity. We are going to explore also the former approach: as variable domains in Gecode are provided as doubly-linked lists, iteration in both directions can be provided efficiently.

8 Analysis and Evaluation

This section analyzes the impact different implementations of iterators and views have on efficiency. Two aspects are evaluated: compile-time polymorphism versus run-time polymorphism, and iterators versus temporary data structures.

The experiments use the Gecode C++ constraint programming library [2]. All tests were carried out on a Intel Pentium IV with 2.8GHz and 1GB of RAM, running Linux. Runtimes are the average of 25 runs, with a coefficient of deviation less than 2.5% for all benchmarks. The *optimized* column gives the time

Table 1: Runtime comparison

Benchmark	optimized	virtual	temporary
	time in ms	relative %	
Alpha	90.50	147.20	103.00
Donald	0.79	124.10	100.00
Golomb 10 (bound)	314.00	173.20	101.70
Golomb 10 (domain)	512.80	138.40	100.50
Magic Sequence 500	270.60	173.50	102.80
Magic Square 6	0.96	149.00	102.30
Partition 32	4 881.20	179.70	102.40
Photo	116.84	141.50	103.50
Queens 100	1.75	146.90	100.00
Crew	5.20	—	159.40
Golf 8-4-9	356.00	—	196.90
Hamming 20-3-32	1 539.60	—	163.00
Steiner 9	110.80	—	226.10

in milliseconds of the optimized system, the other columns are relative to *optimized*. The examples used are standard benchmarks, the first group using only integer constraints, the second group using mainly set constraints.

Code inspection. A thorough inspection of the code generated by several C++ compilers shows that they actually perform the optimizations we consider essential. Operations on both views and iterators are inlined entirely and thus implemented in the most efficient way. The abstractions do not impose a runtime penalty (compared to a system without views and iterators).

Templates versus virtual methods. As the previous section suggested, in C++, compile-time polymorphism using templates is far more efficient than virtual method calls. To evaluate this, we changed the basic operations of finite domain views into virtual methods. The required changes are rather involved, so we did not try the same for iterators and set views. The numbers give an idea of how much more efficient a template-based implementation is. Table 1 shows the results in column *virtual*. Virtual method calls cause a runtime overhead between 25% and 80%.

Temporary datastructures. One important claim is that iterators are advantageous because they avoid temporary datastructures. Table 1 shows in column *temporary* that computing temporary datastructures has limited impact on finite domain variables (about 3%), but considerable impact for set constraints (59% to 126% overhead). Temporary datastructures have been emulated by wrapping all iterators in a cache-iterator as described in Sect. 4.

9 Conclusion

The paper has introduced an architecture decoupling propagators from variables based on views and range iterators. We have argued how to make propagators generic, simpler, and reusable with views for different constraints. We have introduced range iterators as abstractions for efficient domain operations compatible with views. The architecture has been shown to be applicable to finite domain and finite set constraints. Using parametric polymorphism for views and iterators leads to an efficient implementation that incurs no runtime cost.

Acknowledgements

Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Guido Tack is partially funded by DAAD travel grant D/05/26003. Thanks to Patrick Pekczynski for help with the benchmarks, and to Mikael Lagerkvist and the anonymous reviewers for helpful comments.

References

[1] P. Brisset, H. El Sakkout, T. Frühwirth, W. Harvey, M. Meier, S. Novello, T. Le Provost, J. Schimpf, and M. Wallace. ECLiPSe Constraint Library Manual 5.8. User manual, IC Parc, London, UK, Feb. 2005.

[2] Gecode: Generic constraint development environment, 2005. Available upon request from the authors, www.gecode.org .

[3] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[4] ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, Aug. 2000.

[5] I. S. Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, Apr. 2005.

[6] F. Laburthe. CHOCO: implementing a CP kernel. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 71–85, 55 Science Drive 2, Singapore 117599, Sept. 2000.

[7] V. Lagoon and P. J. Stuckey. Set domain propagation using ROBDDs. In Wallace [14], pages 347–361.

[8] T. Müller. *Propagator-based Constraint Solving*. Doctoral dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Fachrichtung Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 2001.

[9] J.-F. Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, Nov. 1994.

[10] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.

[11] J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 513–527, Portland, OR, USA, Dec. 1995. The MIT Press.

[12] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[13] C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In Wallace [14], pages 619–633.

[14] M. Wallace, editor. *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*. Springer-Verlag, Toronto, Canada, Sept. 2004.

Closures for Closed Module Systems

Rémy Haemmerlé and François Fages
Projet Contraintes – INRIA Rocquencourt – France
FirstName.LastName@inria.fr

Abstract

In a classical paper of D.H.D. Warren, the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. Without disputing this argumentation in the context of a Logic Programming system without modules, we show that the situation is different in a closed module system. By closed we mean the property that the Module system is able to prevent any call to the private predicates of a module from the other modules, in particular through meta-programming predicates. We show that this property necessitates to distinguish the execution of a term (meta-programming predicate `call`) from the execution of a closure (higher-order). We propose a module system for Constraint Logic Programming with a notion of closures inspired from Linear Concurrent Constraint programming. This module system is quite simple and pretty independent of a precise language (it is currently implemented for GNU-Prolog). Although this system can be seen as a simple layer of syntactic sugar, it does provide a discipline for naming predicates and hiding code, making possible the development of libraries and facilitating the safe re-use of existing code. Furthermore we provide the module system with logical and operational semantics. This formal setting is used in the paper to compare our approach to the other module systems proposed for Prolog, which generally do not ensure full implementation hiding.

1 Introduction

It is often easier to work with a small number of concepts in mind. Thus to be written, large programs must be divided into small parts. Although this can be done in any language, a suitable module system helps the segmentation of programs by forcing the programmer to clearly define the limits and the interface of each block. Hence many errors can be avoided, the compiler being in charge of verifying that the calls between modules respect the declared interfaces. Moreover such divisions facilitate the understanding of a program by external programmers and improve the reusability of the code.

The context of our work is the design of a fully bootstrapped implementation of a Linear logic Concurrent Constraint (LCC) [11, 24] programming language called **SiLCC**, for "SiLCC is Linear Concurrent Constraint programming". The purpose of the present paper is to present the theoretical and practical choices we made for its module system. For the sake of simplicity and because we think, as Leroy [16], that "modular programming has little to do with the particulars of any programming language" we will present our module system in the formal setting of Constraint Logic Programs (CLP) [15] rather than in the more general setting of LCC.

The purpose of a module system is not unique and it is important to list the different purposes a module system may serve:

Implementation Hiding. A programmer should have the possibility to protect his code from the intrusion from other modules. This means that it should be possible to restrict the access of a module (i.e. calls, dynamic assertions/retractions, syntax modifications, global variable assignments etc ...) from extra-modular code. In the following, we will say that a module system that ensures *implementation hiding* is a *closed* module system, on the contrary a system that allows any call from a module to another will be called *open*.
Separation of name space. In large *flat* (i.e. without modules) system, name clashes are frequent and disturbing. Indeed, to be certain to avoid such clashes, the programmer is constrained to systematically prefix the name of his predicates. A module system has to avoid such clashes automatically.
Separate compilation. It should be possible to compile in an independent way each module of a program. In particular, standard libraries must be compiled once for all.
Bootstrapping. One of the most original aspects of SiLCC is the realization of a completely bootstrapped implementation of a constraint language from a small kernel (LCC with constraints over labeled graphs and linear logic constraints providing imperative features). Modules are essential to the bootstrap of such a complex language.

Furthermore, the design of a module system for Prolog needs to meet some other requirements:
Simplicity. The module system should be simple enough not to restrict the use of Prolog for rapid prototyping, in particular:

- the conciseness of Prolog should be preserved, avoiding too many module-related declarations;
- meta-programming predicates such as the `call` predicate should be fully supported;
- new concepts should be limited in order to be adopted by classical Prolog programmers.

Semantics. The module system should come with a formal semantics from which one can derive its properties.
Simple implementation. We want the module system to be easy to code and to port onto different logic languages, ranging from GNU-Prolog to SiLCC.

Modularity in the context of Logic Programming has been considerably studied, and there has been some standardization attempts [14]. In order to define a notion of module that captures important aspects such as import/export and implementation hiding, the *logical approaches* rely on an extension of the underlying logic. For example, one can cite extensions with nested implications [17], meta-logic [2] or second order predicates [8]. Some other approaches, such as contextual Logic Programming [19], or object-oriented extensions [20], go even further in the direction of fully dynamic module systems.

Other proposals add constructs for declaring and handling a notion of static modules. On the one hand, there are *algebraic approaches* defining module calculi on sets of program clauses, such as the module calculus of O'Keefe [21], or of Bugliesi, Lamma and Mello [4], or the calculus of Sanella and Wallen [23] inspired from functional programming. On the other hand, the so-called *syntactic approaches* deal mainly with the alphabet of symbols. This approach is criticized in [21, 23] for its lack of logical semantics. Nevertheless syntactic module systems are often chosen for their simplicity and compatibility with existing code. For instance, the existing code of OEFAI CLP(q,r) [12] or a Prolog implementation of CHR [25] should be ported as libraries in a modular system. Most of current modular Prolog systems, such as SICStus [26], SWI [28], ECLiPSe [1], XSB [22], Ciao [3, 6, 5], fall into this category. None of them however ensures every purpose we have exposed previously. For example, the popular system SICStus does not provide any kind of implementation hiding.

In this paper, we show that closures are needed to ensure code protection in a predicate-based syntactic module system. We propose a closed module system which distinguishes meta-programming predicates from closures. We give an operational semantics which is used to prove the properties of the module system, and to compare the different existing systems. Furthermore we provide a logical semantics with a translation of pure modular constraint logic programs into ordinary CLP programs with a simple module constraint system. The rest of the paper is organized as follows. The next section defines the syntax and operational semantics of modular constraint logic programs (MCLP) with meta-calls and closures, and shows that the module system satisfies the implementation hiding property. Section 3 presents an equivalent logical semantics. Then, in section 4, we discuss some pragmatic aspects of our module system. Finally we compare our proposal with existing syntactic module systems and conclude.

2 Modular Constraint Logic Programs

2.1 Syntax

In this section, we define the formal syntax of modular constraint logic programs, which is used in the following section to formally define their operational semantics. The simplified syntax (in `Typewritter letters` used by the programmer adopts some conventions explained in Sec. 4. In particular, the syntactic distinction between constraints, atoms and closures in goals, and the systematic prefixing of predicates are only used here for the sake of conceptual simplicity.

We thus consider the following disjoint alphabets of symbols given with their arity:

- V a countable set of variables (of arity 0) denoted by $x, y \ldots$;
- Σ_F a set of constant (of arity 0) and function symbols;
- Σ_C a set of constraint predicate symbols containing $= /2$ and $true/0$;
- Σ_P a set of program predicate symbols containing $call/2$, $closure/3$ and $apply/2$;
- Σ_M a set of module names (of arity 0), noted μ,...

Furthermore, we assume a coercion relation $\overset{P}{\rightsquigarrow}$: $\Sigma_F \times \Sigma_P$ to interpret function symbols as predicate symbols. In classical Prolog systems, where function symbols are not distinguished from predicate symbols, this relation is just the identity. We consider the usual sets of terms, formed over V and Σ_F, of atomic constraints, formed with predicate symbols in Σ_C and terms, and of atoms, formed with predicate symbols in Σ_P and terms. In addition, atoms prefixed by a module name, noted $\mu{:}A$, are called *prefixed atoms*. For the sake of simplicity, we do not describe here the conventions (given in Sect. 4) that are used for prefixing automatically the atoms given without module names in a clause or a goal.

Definition 2.1 *A **closure** is an atom of the form $closure(x, \mu{:}A, z)$ where x and z are variables, $\mu{:}A$ is a prefixed atom. The **application** of a closure associated to a variable z to an argument x is the atom $apply(z, x)$*

The closure $closure(x, \mu : A, z)$ associates to variable z a prefixed atom $\mu : A$ in which the variable x is abstracted. It is worth noticing that this notion of closure is simpler than the one of Warren [27], because only atoms are abstracted, and this is sufficient for the purpose of modular programming.

Definition 2.2 *A **MCLP clause** is a formula of the form*

$$A_0 \leftarrow c_1, \ldots, c_l | \kappa_1, \ldots, \kappa_n | \mu_1{:}A_1, \ldots, \mu_m{:}A_m.$$

where the κ_i's are closures, the $\mu_i : A_i$'s are prefixed atoms, and the c_i's are atomic constraints.

Definition 2.3 *A **MCLP module** is a tuple $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu)$ where $\mu \in \Sigma_M$ is the* name *of the module, $\mathcal{D}_\mu$ is a set of clauses, called its* implementation*, and $\mathcal{I}_\mu \subset \Sigma_P$ is the set of* public *predicates, called the* interface *of the module. The predicates not in I_μ are called* private *in μ. A **MCLP Program** $\mathcal{P}$ is a set of modules whose names are distinct.*

The import/export declarations described in Sect. 4 are just facilities for automatically prefixing the predicates of the used modules. However, it is important for the implementation hiding property that the visibility rules for predicates refer only to the module interface.

Definition 2.4 *A **MCLP goal** is a formula*

$$c_1, \ldots, c_l \,|\, \langle \nu_1 - \kappa_1 \rangle, \ldots, \langle \nu_n - \kappa_n \rangle \,|\, \langle \nu'_1 - \mu_1 : A_1 \rangle, \ldots, \langle \nu'_m - \mu_m : A_m \rangle$$

where the c_i's are atomic constraints, the κ_i are closures, the $(\mu_i : A_i)$'s are prefixed atoms and both the ν_i's and the ν'_i's are module names called calling contexts*.*

In the following, $\langle \nu - (\kappa_1, \ldots, \kappa_n) \rangle$ denotes the sequences of closures $(\langle \nu - \kappa_1 \rangle, \ldots, \langle \nu - \kappa_n \rangle)$ and similarly for sequence of atoms with context.

Definition 2.5 *A **pure** MCLP construct (goal, clause, module or program) is an MCLP construct containing no* call*,* closure *or* apply *predicates.*

The following example will illustrate the difference between a meta-call and the application of a closure in a module.

Example 2.6 *Let us consider the classical implementation of the* `findall/3` *predicate:*

```
findall(X,G,_) :- call(G), asserta(found(X)), fail.
findall(_,_,L) :- collect([],L).
collect(S,L) :- retract(found(X)), collect([X|S],L).
collect(L,L).
```

The use of `call` *in this implementation does not prevent intrusion from another module with a goal like* `findall(X,retract(found(X)),L)`*. On the other hand, an implementation of* `findall` *using closures and* `apply` *instead of* `call` *as below*

```
findall(C,_) :- apply(C,X), asserta(found(X)), fail.
findall(_,L) :- collect([],L).
```

ensures the protection of the code. For instance, the clauses retracted by the closure in the goal `(closure(X,retract(found(X)),C), findall(C,L))`*. will be safely retracted in the calling module, and not in the called module.*

2.2 Operational Semantics

Let $\mathcal{P}$ be a MCLP program defined over some constraint system $\mathcal{X}$. The transition relation $\longrightarrow$ on goals is defined as the least relation satisfying the rules in table 1, where θ is a renaming substitution with fresh variables. A successful derivation for a goal G is a finite sequence of transitions from G which ends with a goal containing no atom, only constraints (the computed answer) and closures.

The *modular CSLD* resolution rule is a restriction of the classical CSLD rule for CLP [15]. The additional condition $(\nu = \mu) \vee (p \in \mathcal{I}_\mu)$ imposes that $\mu : p(\vec{t})$ can be executed only if, either the call

Modular CSLD	$\dfrac{(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \qquad (\nu = \mu) \vee (p \in \mathcal{I}_\mu) \quad (p(\vec{s}) \leftarrow c' \mid k \mid \beta)\theta \in \mathcal{D}_\mu \qquad \mathcal{X} \models \exists (c \wedge \vec{s} = \vec{t} \wedge c')}{(c \mid K \mid \gamma, \langle \nu - \mu : p(\vec{t}) \rangle, \gamma') \longrightarrow (c, \vec{s} = \vec{t}, c' \mid K, \langle \mu - k \rangle \mid \gamma, \langle \mu - \beta \rangle, \gamma')}$
Call	$\dfrac{\mathcal{X} \models \exists (c \wedge t = f(\vec{x})) \qquad f \overset{P}{\rightsquigarrow} p}{(c \mid K \mid \gamma, \langle \nu - \mu : call(t) \rangle, \gamma') \longrightarrow (c, t = f(\vec{x}) \mid K \mid \gamma, \langle \nu - \mu : p(\vec{x}) \rangle, \gamma')}$
Apply	$\dfrac{\mathcal{X} \models c \Rightarrow z = y}{(c \mid \kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 \mid \gamma, \langle \nu - \nu : apply(y, t) \rangle, \gamma') \longrightarrow (c \mid \kappa_1, \langle \mu - closure(x, \mu' : A, z) \rangle, \kappa_2 \mid \gamma, \langle \mu - \mu' : A[x \backslash t] \rangle, \gamma')}$

Table 1: Transition relation for MCLP goals with calls and closures.

is made from inside the module (i.e. from the calling context μ), or the predicate p is a public predicate in μ. Moreover, this rule propagates the new calling context to the atoms and the closures of the body of the selected clause.

The *call* rule defines the operational semantics of meta-calls. It is worth noting that this rule does not change the calling context, which is necessary to guarantee the implementation hiding. This definition of *call* does not handle the meta-call of conjunctions of atoms nor the meta-call of a constraint. These calls can be emulated however, by supposing (','/2 $\overset{P}{\rightsquigarrow}$ *and*/2) and by adding the clause $(and(x, y) \leftarrow \mu : call(x), \mu : call(y))$ to the implementation of any module μ.

In this formal semantics of meta-calls, the goal $(c \mid \langle \mu - \nu : call(t, s) \rangle)$ may succeed if the argument of *call* is a free variable. Indeed in such a case, there will be one transition for each predicate visible in the calling context. A failure would not be a better solution however, as it would violate the independence of the selection strategy. This can be seen for instance with the goal (`X=true, call(X)`), where the left-to-right selection strategy would lead to the computed answer `X=true`, whereas a right-to-left strategy would lead to a failure. The proper way to handle such errors would be to raise an exception, which is not formalized here.

The *apply* rule allows the invocation of a closure collected by a previous predicate call. In practice, it looks for the closure associated to the closure variable (formally checks the equality of variables $z = y$), and applies the closure to the argument in the closure context.

Example 2.7 *Closures can be used to define general iterator predicates for data structures. In a module defining some data structure, it is possible to define the binary predicates* `forall/2` *that check that every element of a data structure passed in the first argument, verifies a property passed as a closure in the second argument. For instance, in a library for lists, such iterators can be defined as follows:*

```
:-module(lists, [..., forall/2]).
forall([], C).
forall([X|T], C) :- apply(C, X), forall(T, Z).
```

On the other hand, if we employed a meta-call instead of a closure, as below, the called predicate would fail as it is not visible in the lists module.

```
forall([], P).
```

```
forall([H| T], P):- G=..[P, H], call(G), forall(T, P).
```

2.3 Implementation Hiding

Intuitively, the implementation hiding property states that in order to execute the predicate of a module, one has to pass through a public predicate of this module. Formally, this property can be stated as follows:

Definition 2.8 (Implementation Hiding) *The operational semantics of MCLP programs satisfies the implementation hiding property if, whenever an atom or a closure with context $\langle \nu - X \rangle$ resolves in a goal $\langle \mu - Y \rangle$, then either $\mu = \nu$, or X is of the form $\mu{:}p(\vec{t})$ and p is public in μ.*

Proposition 2.9 *The operational semantics of MCLP programs satisfies the implementation hiding property.*

3 Logical Semantics

3.1 Modules as a Constraint System $\mathcal{M}$

To a given MCLP program $\mathcal{P}$, one can associate a simple module constraint system $\mathcal{M}$, in which the constraint $allow(\nu, \mu, p)$ that states that the predicate p of module μ can be called in module ν, is defined by the following axiom schemas:

$$\frac{\nu \in \Sigma_M \quad p \in \Sigma_P}{\mathcal{M} \models allow(\nu, \nu, p)} \qquad \frac{\nu, \mu \in \Sigma_M \quad (\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P} \quad p \in \mathcal{I}_\mu}{\mathcal{M} \models allow(\nu, \mu, p)}$$

This constraint system depends solely on the interface of the different modules that composes the program $\mathcal{P}$, and not on its implementation.

3.2 Pure MCLP programs

Pure MCLP programs can be given a logical semantics equivalent to their operational semantics, obtained by a simple translation of pure MCLP($\mathcal{X}$) programs into ordinary CLP($\mathcal{M}, \mathcal{X}$) programs. This translation can be used for the implementation, and shows that the module system can be viewed as simple syntactic sugar. The alphabet $\dot{\Sigma}_P$ of the associated CLP($\mathcal{M}, \mathcal{X}$) program, is constructed by associating one and only one predicate symbol $\dot{p} \in \dot{\Sigma}_P$ of arity $n + 2$ to each predicate symbol $p \in \Sigma_P$ of arity n. Let us consider the translations Π and $\Pi^{\langle\rangle}$ of MCLP programs and MCLP goals respectively, given in table 2.

Proposition 3.1 (Soundness) *Let $\mathcal{P}$ and $(c|\gamma)$ be a pure MCLP program and a pure MCLP goal*

$$if \ \left((c|\gamma) \xrightarrow{\mathcal{P}} (d|\gamma')\right) \ then \ \left((c|\Pi^{\langle\rangle}(\gamma)) \xrightarrow{\Pi(\mathcal{P})} (d, allow(y, \mu, p), y = \nu|\Pi^{\langle\rangle}(\gamma'))\right)$$

for some ν, μ, p and such that y is not free in d.

Proposition 3.2 (Completeness) *Let $\mathcal{P}$ and $(c|\gamma)$ be pure MCLP program and goal*

$$if \ \left((c|\Pi^{\langle\rangle}(\gamma')) \xrightarrow{\Pi(\mathcal{P})} (d|\alpha)\right) then \left((c|\gamma) \xrightarrow{\mathcal{P}} (d'|\gamma'')\right)$$

where $\Pi^{\langle\rangle}(\gamma'') = \gamma'$ and $d' = (d, allow(y, \mu, p), y = \nu)$ for some ν, μ, p and such that y is not free in d.

$$\Pi\left(\bigcup\{(\mu,\mathcal{D}_\mu,\mathcal{I}_\mu)\}\right)=\bigcup\{\Pi_\mu(\mathcal{D}_\mu)\}$$

$$\Pi_\mu\left(\bigcup\{(A\leftarrow c|\alpha)\}\right)=\bigcup\{\Pi_\mu(A\leftarrow c|\alpha)\}$$
$$\Pi_\mu\left(p_0(\vec{t})\leftarrow c|\alpha\right)=\dot{p_0}(y,\mu,\vec{t})\leftarrow allow(\mu,y,p_0),c|\Pi_\mu(\alpha)$$
$$\Pi_\mu(A,A')=\Pi_\mu(A),\Pi_\mu(A')$$
$$\Pi_\mu(\nu{:}p(\vec{t}))=\dot{p}(\mu,\nu,\vec{t})$$

$$\Pi^{\langle\rangle}(\gamma,\gamma')=\Pi^{\langle\rangle}(\gamma),\Pi^{\langle\rangle}(\gamma')$$
$$\Pi^{\langle\rangle}\left(\langle\nu-\mu{:}p(\vec{t})\rangle\right)=\dot{p}(\nu,\mu,\vec{t})$$

Table 2: Formal translation of MCLP($\mathcal{X}$) to CLP($\mathcal{M},\mathcal{X}$)

The simplicity of this logical semantics for our module system is due to the fact that we do not impose (as in Java) to import explicitly a module in order to use it. Hence every module can be translated in a way independent of each other. This advocates the use of public/private directives in modules instead of import directives for defining visibility rules (the import directive described in Sec 4.3 is only used for the automatic prefixing of predicate names).

3.3 Closures

The logical semantics of closures is obviously beyond the scope of first-order CLP programs. A natural setting for giving a logical account to closures is higher-order logic, as used higher-order extensions of Prolog such as lambda-Prolog with modules [18], or more recently Hiord [7].

Nevertheless, the restricted form of closures we use in this paper can also be defined in Linear logic Concurrent Constraint (LCC) programming languages [11] as mere syntactic sugar, as follows:

$$closure(x,A,z)\equiv !\forall x.((arg(z,x))\rightarrow A)\qquad\qquad apply(t,z)\equiv arg(z,t)$$

In this definition, a closure is an LCC agent which waits for its argument given in a token (linear logic constraint) $arg(z,x)$ that is posted by the apply agent. The logical semantics of LCC in Linear Logic thus provides a logical semantics for MCLP programs with closures, in which the semantics of the module system is given by the module constraint system defined above. This is the logical semantics we use for SiLCC and its module system.

4 Pragmatic Aspects

In practice, we deal with a set of *atoms* defined as in ISO Prolog [10] to represent function symbols, predicate symbols and module names. In this chapter, the term atom has thus a different meaning than in the previous sections, and we will use the term *predicate* to refer to the atomic propositions as defined in Sect. 2.

4.1 Defining a Module

A module is a set of clauses and directives contained in a single file named with the name of the module and a `.pl` extension. The file must begin with a `module/2` declaration which specifies the

name of the module and its interface. The name of the module is an atom, whereas its interface must be either a list of predicate specifications of the form `Functor/Arity` for representing the set of public predicates, or a '_' denoting the fact that all the predicates of the module are public.

4.2 Visibility

The formal semantics proposed in Sect. 2 (or 3) clearly define the visibility of predicates from a particular module. The predicates which can be called from a module are the predicates defined in that module plus all public predicates of any module. A call is either a module-prefixed predicate (in the form `module:p(X1,..., XN)`) or a non-prefixed predicate. If a not fully prefixed predicate occurs, the compiler assumes that this call is implicitly made inside the module. In other words, in the module `module` the predicate `p(X_1, ..., X_n)` stands for `module:p(X_1, ..., X_n)`.

4.3 Using a Module

The *use of a module* consists in the import of some publmic predicates of a module to the current one. The programmer can use a module by means of the `use_module/2` declaration where the first argument is the name of the module used, and the second argument is the list of imported predicates, `[Functor/Arity, ...]`. The purpose of this argument is to consider these predicates as predicates of the current module, without needing to prefix them. `use_module(m, [p/1])` is nothing but a shortcut to force the loading of a module `m` and to avoid typing clauses such as `p(x):-m:p(x)` for each predicate `p` imported from the module `m`.

4.4 The Default Module

In our module system we allow the compilation of classical Prolog files without module declaration. The code belonging to such files is considered to belong to the special module `user`. This module is public and access to this module is never controlled by the system. Our system is thus able to execute old GNU-Prolog files. Moreover this module is the default module of every call made from the top level.

4.5 Packages

Despite the fact that the module system we have defined strongly limits conflicts of predicate names, its distributed use leads quickly to a new conflict problem: the conflict of module names. To avoid that, a notion of package similar to the Java language has been introduced. A package is defined as a sequence of atoms separate by slashes. The notion of module name is extended to an atom or a pair package - atom separated by a slash.

For sake of conciseness, the directive `import/1` has been added to import module names. For example, the use of `:-import(oefai/clpr)` indicates to the system that the atom `clpr` refers to the module `clpr` belonging to the package `oefai`. In the same way, with `:-import(oefai/_)`, the user can import all modules of the package `oefai`. The import of two modules with the same name is however forbidden.

4.6 About the Implementation

The actual implementation of our module system consists of a layer, written in Prolog, on the top of GNU Prolog RH [9], a version of GNU Prolog extended with coroutines and attributed variables. This layer is composed of:

a **preprocessor** that converts modular code into non-modular code ;

a **library** to handle dynamic calls and dynamical module loading;

a **new top-level**, that interprets modular programs and is able to dynamically load modules. Attributed variables are used to represent closures, and are useful to protect them from a "reverse engineering" approach, as they forbid the user to hack the internal representation of closures. This is the only reason to use attributed variables in the implementation of the module system.

As explained in [27], application of closures can be compiled slightly more efficently than dynamic calls. Indeed a closure can be represented by a pointer to the code of the corresponding predicate, whereas a dynamic call needs to access a hash table. Our prototype does not include this optimization yet.

5 Comparison with other Module Systems

5.1 SICStus Prolog.

In SICStus [26], all predicates are public (formally for all module $(\mu, \mathcal{D}_\mu, \mathcal{I}_\mu) \in \mathcal{P}$ we have $I_\mu = \Sigma_P$). On the other hand, this system does not provide any kind of implementation hiding. Thanks to the `meta_predicate` directive, that allows to explicitly declare meta-predicates, the system is able to implicitly add the calling context to meta-data. The principal defect of this approach is thus the abandonment of the principle of implementation hiding.

5.2 ECLiPSe.

ECLiPSe [1] proposes an intermediary solution. In fact it provides two different predicates to invoke meta-calls. The first one `:/2` behaves as `call` for a prefixed atom according to our operational semantics, whereas the second one `@/2` does not make any visibility test. It provides also a directive `tool/1`, (analogous to the `meta_predicate` directive) that allows the implicit addition of the calling context as argument of the meta-predicate. Hence the programmer employs `:/2` for common use and `@/2` for meta-predicates. This solution has the advantage of limiting the unauthorized calls made in a unconscious way. Nonetheless, as beforehand, it is not possible to ensure the hiding of any predicate.

5.3 SWI Prolog.

For meta programming, SWI Prolog [28] uses a slightly different semantics from the one proposed in Sect 2.2. In fact, SWI handles two distinct calling contexts. The former, that we call *static context*, plays the role of the calling context in a modular CSLD transition. The latter, that we call *dynamic context*, plays the role of the calling context in a call transition. By default the dynamic context is the same that the static context. However, for a meta-predicate, the dynamic context is the context of the goal that calls it. In SWI, predicates with this behavior are called "module transparent" and are declared with the directive `module_transparent/1`. By declaring `forall/2` as "module transparent", SWI-Prolog executes the example 2.7 as expected.

Nonetheless, a dynamic call does not behave as a static one. Therefore with a "module transparent" predicate, the two goals `p(x)` and `(G=p(x), call(G))` do not call `p/1` in the same module. The former makes the call in the static context, whereas the latter makes it in the dynamic context.

Moreover, these conventions do not provide a code protection as strong as we want. For instance, the use of the implementation of the `findall/3` predicate (defined in the example 2.6) does not

work better either in SWI, all assertion/retraction being made in the calling module instead of the called module.

5.4 Ciao Prolog.

Conceptually, Ciao Prolog [3] adopts a closed module system. In order to allow the manipulation of meta-data through the module system, it provides an advanced version of the directive `meta_predicate/1`. Before calling the meta-predicates, the system compiles on the fly meta-data following a method analoguous to the one presented in Sect. 3. Since this compilation is done before the call of the meta-predicate, the system is knows the calling context in which the meta-data must be called, to obtain the expected result. The meta-predicate handles then a compiled version of the meta-data that is possible to invoke with the predicate `call/1`. As far as the system does not document any predicate (except `call/1`) able to create or manipulate such compiled meta-data, the implementation hiding property is preserved.

Although not clearly stated in the manual, Ciao Prolog does make a distinction between terms and higher-order data (i.e. compiled versions of goal). However, instead of hiding these objects behind a directive, we propose to manipulate closures as first-class citizens.

5.5 XSB

The XSB system [22] is also considered as a syntactic system, but follows a quite different approach. In fact, this system belongs to the class of *atom-based* systems, rather than *predicate-based*. The main difference is that XSB modularizes function symbols too. Hence two compound terms constructed in two different modules can not be unified. In a module, it is possible however to import public symbols from another module, the system considers then that the two modules share the same symbols. The semantics of the `call/1` predicate is hence very simple: the meta-call of a term corresponds to the call of the predicate of the same symbol and arity as the module where the term has been created.

However, this solution mainly moves the problem to the construction of the terms. Indeed in XSB, the terms constructed with `=../2`, `functor/2` and `read/1` are supposed to belong to the module `user`. As a consequence, the system does not respect anymore the independence of the selection strategy. For instance, in a module different form `user`, the goal `(functor(X,foo,1), X=foo(_))` fails, whereas `(X=foo(_), functor(X,foo,1))` succeeds.

6 Conclusion

In a classical paper of D.H.D. Warren [27], the higher-order extensions of Prolog were questioned as they do not really provide more expressive power than meta-programming predicates. We have shown here that the situation is different in the context of logic programs with modules, and that the protection of private module predicates necessitates to distinguish between term calls (meta-programming) and closures (higher-order). The module system we propose is close to the one of Ciao Prolog in its implementation, but has the advantage of revealing the need for closures, and of positioning them w.r.t. meta-programming predicates in the framework of formal operational semantics. Furthermore, an equivalent logical semantics has been provided for pure modular programs.

Our module system has been implemented in GNU-Prolog. Some existing code has been ported as libraries using the module system, such as for instance an implementation of CHR [25]. This

modularization of CHR provides an example of intensive use of the module system allowing the development of several layers of constraint solvers in CHR. Other libraries have been developed with this closed module system for the development of SiLCC. One can quote a dynamic lexer-parser allowing modular redefinitions of the syntax with a powerful generalization of the directive `op/3`. This library belongs to the bootstrap libraries of our on-going implementation of SiLCC.

As for future work, we can mention the possible implementation of so-called *functors* [16] allowing the static instantiation of parameterized modules, and the investigation of object-oriented programming features with modules and global variables in CLP.

Acknowledgements.

We are grateful to Sumit Kumar for a preliminary work he did on this topic, during his summer 2003 internship at INRIA. We thank also Emmanuel Coquery and Sylvain Soliman for valuable discussion on the subject. and to Daniel de Rauglaudre and Olivier Bouissou for their comments and help in the implementation.

References

[1] A. Aggoun and al. *ECLiPSe User Manual Release 5.2*, 1993 – 2001.

[2] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In *META-92: Third International Workshop on Meta Programming in Logic*, pages 105–119, Berlin, Heidelberg, 1992. Springer-Verlag.

[3] F. Bueno, D. C. Gras, M. Carro, M. V. Hermenegildo, P. Lopez-Garca, and G. Puebla. The ciao Prolog system. reference manual. Technical Report CLIP 3/97-1.10#5, University of Madrid, 1997-2004.

[4] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programmming*, 19/20:443–502, 1994.

[5] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid, Aug. 2004.

[6] D. Cabeza and M. Hermenegildo. A new module system for Prolog. In *First International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 131–148. Springer-Verlag, July 2000.

[7] D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A type-free higher-order logic programming langugae with predicate abstraction. In *Proceedings of ASIAN'04, Asian Computing Science Conference*, pages 93–108, Chiang Mai, 2004. Springer-Verlag.

[8] W. Chen. A theory of modules based on second-order logic. In *The fourth IEEE. Internatal Symposium on Logic Programming*, pages 24–33, 1987.

[9] D. Diaz and R. Haemmerlé. *GNU Prolog RH user's manual*, 1999–2004.

[10] P. D. A. Ed-Dbali and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[11] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, Feb. 2001.

[12] C. Holzbaur. Oefai clp(q,r) manual rev. 1.3.2. Technical Report TR-95-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1995.

[13] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 1: General core*, 1995. ISO/IEC 13211-1.

[14] International Organization for Standardiztion. *Information technology – Programming languages – Prolog – Part 2: Modules*, 2000. ISO/IEC 13211-2.

[15] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, Jan. 1987.

[16] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[17] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[18] D. Miller. A proposal for modules in lambda prolog. In *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, volume 798 of *Lecture Notes in Computer Science*, pages 206–221, 1994.

[19] L. Monterio and A. Porto. Contextual logic programming. In *Proceedings of ICLP'1989,International Conference on Logic Programming*, pages 284–299, 1989.

[20] P. Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Informatics, University of Beira Interior, Portugal, Sept. 2003.

[21] R. A. O'Keefe. Towards an algebra for constructing logic programs. In *Symposium on Logic Programming*, pages 152–160. IEEE, 1985.

[22] K. Sagonas and al. *The XSB System Version 2.5 - Volume 1: Programmer's Manual*, 1993 – 2003.

[23] D. T. Sannella and L. A. Wallen. a calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, pages 147–177, 1992.

[24] V. A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[25] T. Schrijvers and D. S. Warren. Constraint handling rules and table execution. In *Proceedings of ICLP'04, International Conference on Logic Programming*, pages 120–136, Saint-Malo, 2004. Springer-Verlag.

[26] Swedish Institute of Computer Science. *SICStus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 1991–2004.

[27] D. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence*, volume 10 of *Lecture Notes in Mathematics*, pages 441–454. 1982.

[28] J. Wielemaker. *SWI Prolog 5.4.1 Reference Manual*, 1990– 2004.

Speeding up constrained path solvers with a reachability propagator

Luis Quesada, Peter Van Roy, and Yves Deville
Université catholique de Louvain
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{luque, pvr, yde}@info.ucl.ac.be

Abstract

Constrained path problems have to do with finding paths in graphs subject to constraints. One way of constraining the graph is by enforcing reachability on nodes. For instance, it may be required that a node reaches a particular set of nodes by respecting some restrictions like visiting a particular set of nodes or edges and using less than a certain amount of resources. The reachability constraints of this paper were suggested by a practical problem regarding mission planning in the context of an industrial project.

We deal with this problem by using concurrent constraint programming where the problem is solved by interleaving Propagation and Labeling. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a directed graph g. Given a source node $source$ in g, we can identify three parts in the *Reachability* constraint: (1) the relation between each node of g and the set of nodes that it reaches, (2) the association of each pair of nodes $\langle source, i\rangle$ with its set of cut nodes, and (3) the association of each pair of nodes $\langle source, i\rangle$ with its set of bridges.

We show the effectiveness of our *Reachability* propagator by applying it to the Simple Path problem with mandatory nodes. We do an experimental evaluation of *Reachability* that shows that it provides strong pruning, obtaining solutions with very little search. Furthermore, we show that *Reachability* is also useful for defining a good labeling strategy and dealing with ordering constraints among mandatory nodes. These experimental results give evidence that *Reachability* is a useful primitive for solving constrained path problems over graphs.

1 Introduction

Constrained path problems have to do with finding paths in graphs subject to constraints. One way of constraining the graph is by enforcing reachability on nodes. For instance, it may be required that a node reaches a particular set of nodes by respecting some restrictions like visiting a particular set of nodes or edges and using less than a certain amount of resources. We have instances of this problem in Vehicle routing [14, 4, 9] and Bioinformatics[7].

An approach to solve this problem is by using Concurrent Constraint Programming (CCP) [18, 13]. In CCP, we solve the problem by interleaving two processes: propagation and labeling. In Propagation, we are interested in filtering the domains of a set of finite domain variables according to the semantics of the constraints that have to be respected. In Labeling, we are interested in specifying which alternative should be selected when searching for the solution.

Our goal is to implement so-called *Constrained Path Propagators (CPPs)* for achieving global consistency [6]. In this paper, we define a propagator which we call *Reachability* that implements a generalized reachability constraint on a directed graph g. Given a source node $source$ in g, we can identify three parts in the *Reachability* constraint: (1) the relation between each node of g and the set of nodes that it reaches, (2) the association of each pair of nodes $\langle source, i\rangle$ with its set of cut nodes, and (3) the association of each pair of nodes $\langle source, i\rangle$ with its set of bridges.

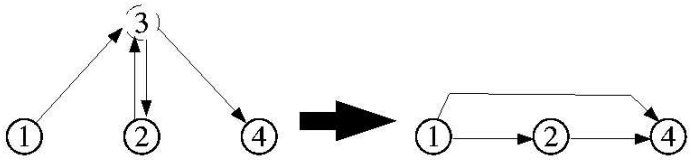

Figure 1: Relaxing Simple Path with mandatory nodes by eliminating the optional nodes

Our contribution is a propagator that is suitable for solving the Simple Path problem with mandatory nodes [19, 3]. This problem consists in finding a simple path in a directed graph containing a set of mandatory nodes. A simple path is a path where each node is visited once. Certainly, this problem can be trivially solved if the graph has no cycles since in that case there is only one order in which we can visit the mandatory nodes [19]. However, if the graph has cycles the problem is NP complete since we can easily reduce the Hamiltonian Path problem [10, 5] to this problem.

Notice however that we can not trivially reduce Simple Path with mandatory nodes to Hamiltonian path. One could think that optional nodes (i.e. nodes that are not mandatory) can be eliminated in favor of new edges as a preprocessing step, which finds a path between each pair of mandatory nodes. However, the problem is that the paths that are precomputed may share nodes. This may lead to violations of the requirement that a node should be visited only once.

In figure 1, we illustrate this situation. Mandatory nodes are in solid lines. In the second graph we have eliminated the optional nodes by connecting each pair of mandatory nodes depending on whether there is a path between them. However, we observe that the second graph has a simple path going from node 1 to node 4 (visiting all the mandatory nodes) while the first one does not. Indeed, the simple path in the second graph is not a valid solution to the original problem since it implies that node 3 is visited twice.

The other reason that makes the elimination of optional nodes difficult is that finding k pairwise disjoint paths between k pairs of nodes $\langle s_1, d_1\rangle$,$\langle s_2, d_2\rangle$,...,$\langle s_k, d_k\rangle$ is NP complete [20].

In general, we can say that the set of optional nodes that can be used when going from a mandatory node a to a mandatory node b depends on the path that has been traversed before reaching a. This is because the optional nodes used in the path going from the source to a can not be used in the path going from a to b.

From our experimental measurements in Section 4, we observe that the suitability of *Reachability* for dealing with Simple Path with mandatory nodes is based on the following aspects:

- The strong pruning that *Reachability* performs. Due to the computation of cut nodes and bridges (i.e., nodes and edges that are present in all the paths going from a given node to another), *Reachability* is able to discover non-viable successors early on.

- The information that *Reachability* provides for implementing smart labeling strategies. By labeling strategy we mean the way the search tree is created, i.e., which constraint is used for branching. *Reachability* associates each node with the set of nodes that it reaches. This information can be used to guide the search in a smart way. For instance, one of our observations is that, when choosing first the node i that reaches the most nodes and selecting as a successor of i first a node that i reaches, we obtain paths that minimize the use of optional nodes.

An additional feature of *Reachability* is its suitability for imposing ordering constraints among mandatory nodes (which is a common issue in routing problems). In fact, it might be the case that we have to visit the nodes of the graph in a particular (partial) order. Taking into account that a node a reaches a node b if

there is a path going from node a to node b, we force a node i to be visited before a node j by imposing that i reaches j and j does not reach i. We have performed experiments that show that *Reachability* takes the most advantage of this information to avoid branches in the search tree with no solution.

The structure of the paper is as follows: first, we introduce *Reachability* by presenting its semantics and deriving pruning rules in a systematic way. Then, we show how we can model Simple Path with mandatory nodes in terms of *Reachability*. Finally, we show examples that demonstrate the performance of *Reachability* for this type of problem, and elaborate on some works that are related to our approach.

2 The reachability propagator

2.1 Reachability constraint

The Reachability Constraint is defined as follows:

$$Reachability(g, source, rn, cn, be) \equiv \forall_{i \in N}. \begin{array}{l} rn(i) = Reach(g, i) \wedge \\ cn(i) = CutNodes(g, source, i) \wedge \\ be(i) = Bridges(g, source, i) \end{array} \quad (1)$$

Where:

- g is a graph whose set of nodes is a subset of N.
- $source$ is a node of g.
- $rn(i)$ is the set of nodes that i reaches.
- $cn(i)$ is the set of nodes appearing in all paths going from $source$ to i.
- $be(i)$ is the set of edges appearing in all paths going from $source$ to i.
- *Reach*, *Paths*, *CutNodes* and *Bridges* are functions that can be formally defined as follows:

$$j \in Reach(g, i) \leftrightarrow \exists_p . p \in Paths(g, i, j) \quad (2)$$

$$p \in Paths(g, i, j) \leftrightarrow \begin{array}{l} p = \langle k_1, ..., k_h \rangle \in nodes(g)^h \wedge k_1 = i \wedge k_h = j \wedge \\ \forall_{1 \leq f < h} . \langle k_f, k_{f+1} \rangle \in edges(g) \end{array} \quad (3)$$

$$k \in CutNodes(g, i, j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . k \in nodes(p) \quad (4)$$

$$e \in Bridges(g, i, j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . e \in edges(p) \quad (5)$$

The above definition of *Reachability* implies the following properties which are crucial for the pruning that *Reachability* performs. These properties define relations between the functions rn, cn, be, nodes and edges. These relations can then be used for pruning, as we show in section 2.2.

1. If $\langle i, j \rangle$ is an edge of g, then i reaches j.

$$\forall_{\langle i,j \rangle \in edges(g)} . j \in rn(i) \quad (6)$$

2. If i reaches j, then i reaches all the nodes that j reaches.

$$\forall_{i,j,k \in N}.j \in rn(i) \wedge k \in rn(j) \rightarrow k \in rn(i) \tag{7}$$

3. If $source$ reaches i and j is a cut node between $source$ and i in g, then j is reached from $source$ and j reaches i:

$$\forall_{i,j \in N}.i \in rn(source) \wedge j \in cn(i) \rightarrow j \in rn(source) \wedge i \in rn(j) \tag{8}$$

4. Reached nodes, cut nodes and bridges are nodes and edges of g:

$$\forall_{i \in N}.rn(i) \subseteq nodes(g) \quad (9) \qquad \forall_{i \in N}.cn(i) \subseteq nodes(g) \quad (10) \qquad \forall_{i \in N}.be(i) \subseteq edges(g) \quad (11)$$

2.2 Pruning rules

We implement the constraint in Equation 1 with the propagator

$$Reachability(G, Source, RN, CN, BE) \tag{12}$$

In this propagator we have that:

- G is a graph variable (i.e., a variable whose domain is a set of graphs [8]). The upper bound of G ($max(G)$) is the greatest graph to which G can be instantiated, and its lower bound ($min(G)$) is the smallest graph to which G can be instantiated. So, $i \in nodes(G)$ means $i \in nodes(min(G))$ and $i \notin nodes(G)$ means $i \notin nodes(max(G))$ (the same applies for edges). In what follows, $\{\langle N_1, E_1\rangle \# \langle N_2, E_2\rangle\}$ will denote a graph variable whose lower bound is $\langle N_1, E_1\rangle$ and upper bound is $\langle N_2, E_2\rangle$. I.e., if $g = \langle n, e\rangle$ is the graph that G approximates, then $N_1 \subseteq n \subseteq N_2$ and $E_1 \subseteq e \subseteq E_2$.
- $Source$ is an integer representing the source in the graph.
- $RN(i)$ is a Finite Integer Set (FS) [11] variable associated with the set of nodes that can be reached from node i. The upper bound of this variable ($max(RN(i))$) is the set of nodes that could be reached from node i (i.e., nodes that are not in the upper bound are nodes that are known to be unreachable from i). The lower bound ($min(RN(i))$) is the set of nodes that are known to be reachable from node i. In what follows $\{S_1 \# S_2\}$ will denote a FS variable whose lower bound is the set S_1 and upper bound is the set S_2.
- $CN(i)$ is a FS variable associated with the set of nodes that are included in every path going from $Source$ to i.
- $BE(i)$ is a FS variable associated with the set of edges that are included in every path going from $Source$ to i.

The definition of *Reachability* and its derived properties give place to a set of propagation rules. We show here the most representative ones. The others are given in [16]. A propagation rule is defined as $\frac{C}{A}$ where C is a condition and A is an action. If C is true, the pruning defined by A can be performed.

- From (6) $\forall_{\langle i,j\rangle \in edges(g)}.j \in rn(i)$ we obtain:

$$\frac{\langle i, j\rangle \in edges(min(G))}{j \in min(RN(i))} \tag{13}$$

- From (7) $\forall_{i,j,k \in N}.j \in rn(i) \wedge k \in rn(j) \rightarrow k \in rn(i)$ we obtain:

$$\frac{j \in min(RN(i)) \wedge k \in min(RN(j))}{k \in min(RN(i))} \tag{14}$$

- From (8)$\forall_{i,j \in N}.i \in rn(source) \wedge j \in cn(i) \rightarrow j \in rn(source) \wedge i \in rn(j)$ we obtain:

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{j \in min(RN(Source))} \tag{15}$$

$$\frac{i \in min(RN(Source)) \wedge j \in min(CN(i))}{i \in min(RN(j))} \tag{16}$$

- From (1)$\forall_{i \in N}.rn(i) = Reach(g, i)$ we obtain:

$$\frac{j \notin Reach(max(G), i)}{j \notin max(RN(i))} \tag{17}$$

- From (1)$\forall_{i \in rn(source)}.cn(i) = CutNodes(g, source, i)$ we obtain:

$$\frac{j \in CutNodes(max(G), Source, i)}{j \in min(CN(i))} \tag{18}$$

- From (1)$\forall_{i \in rn(source)}.be(i) = Bridges(g, source, i)$ we obtain:

$$\frac{e \in Bridges(max(G), Source, i)}{e \in min(BE(i))} \tag{19}$$

- From (9) $\forall_{i \in N}.rn(i) \subseteq nodes(g)$, (10)$\forall_{i \in N}.cn(i) \subseteq nodes(g)$ and (11)$\forall_{i \in N}.be(i) \subseteq edges(g)$ we obtain:

$$\frac{k \in min(RN(i))}{k \in nodes(min(G))} \quad (20) \qquad \frac{k \in min(CN(i))}{k \in nodes(min(G))} \quad (21) \qquad \frac{e \in min(BE(i))}{e \in edges(min(G))} \quad (22)$$

2.3 Implementation of *Reachability*

Reachability has been implemented using a message passing approach [21] on top of the multi-paradigm programming language Oz [12]. In [15], we discuss the implementation of *Reachability* in detail. In this section we will simply refer to some of the functions that are used in the implementatin of *Reachability*:

In our pruning rules we have three functions:

- *Reach* that is $O(N + E)$ since it is basically a call to *DFS* [5].
- *CutNodes* whose algorithm is based on the following definition:

$$k \in CutNodes(g, i, j) \leftrightarrow j \notin Reach(RemoveNode(g, k), i) \tag{23}$$

So, checking whether a node is a cut node is $O(N + E)$. Notice that we assume that $RemoveNode$ returns the same graph when $k \notin nodes(g)$.

- *Bridges* whose algorithm is based on the following definition:

$$e \in Bridges(g, i, j) \leftrightarrow j \notin Reach(RemoveEdge(g, e), i) \tag{24}$$

So, checking whether an edge is a bridge is $O(N + E)$. Notice that we assume that $RemoveEdge$ returns the same graph when $e \notin edges(g)$.

The computation of cut nodes and bridges in each propagation step implies running *DFS* per each potential cut node and per each potential bridge. However, we take advantage of the fact that the cut nodes and the bridges are part of the tree returned by *DFS* (assuming that the destination node is reached from the source). In fact, this means that both the computation of cut nodes and the computation of bridges have the same complexity since the number of edges in the DFS tree is proportional to the number of nodes. I.e., updating CN /BE after removing a set of edges from the upper bound of G is $O(N * (N + E))$ [1].

As explained in [15], we do not compute cut nodes and bridges each time an edge is removed since this certainly leads to a considerably amount of unnecessary computation. This is due to the fact that the set of cut nodes/bridges evolves monotonically. What we actually do is to consider all the removals at once and make one computation of cut nodes and bridges per set of edges removed.

3 Solving *Simple Path with mandatory nodes* with *Reachability*

In this section we will elaborate on the important role that *Reachability* can play in solving Simple Path with mandatory nodes. This problem consists in finding a simple path in a directed graph containing a set of mandatory nodes. A simple path is a path where each node is visited once. I.e., given a directed graph g, a source node $source$, a destination node $dest$, and a set of mandatory nodes $mandnodes$, we want to find a path in g that goes from $source$ to $dest$, going through $mandnodes$ and visiting each node only once.

The contribution of *Reachability* consists in discovering nodes/edges that are part of the path early on. This information is obtained by computing the cut nodes and bridges in each labeling step. Let us consider the following two cases [2]:

- Consider the graph variable on the left of Figure 2. Assume that node 1 reaches node 9. This information is enough to infer that node 5 belongs to the graph, node 1 reaches node 5, and node 5 reaches node 9.

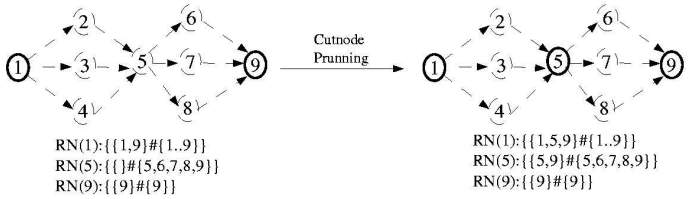

Figure 2: Discovering cut nodes

[1] Notice that the notion of cut node that we have presented does not correspond to the notion of articulation point. An articulation point is a node whose removal increases the number of strongly connected components. A cut node, in our definition, does not necessarily have this property.

[2] In Figures 2 and 3, nodes and edges that belong to the lower bound of the graph variable are in solid line. For instance, the graph variable on the left side of Figure 2 is a graph variable whose lower bound is the graph $\langle\{1, 5\}, \emptyset\rangle$, and whose upper bound is the graph $\langle\{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, 5\rangle, \langle 3, 5\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle, \langle 5, 7\rangle, \langle 5, 8\rangle, \langle 6, 9\rangle, \langle 7, 9\rangle, \langle 8, 9\rangle\}\rangle$.

- Consider the graph variable on the left of Figure 3. Assume that node 1 reaches node 5. This information is enough to infer that edges $\langle 1, 2\rangle$, $\langle 2, 3\rangle$,$\langle 3, 4\rangle$ and $\langle 4, 5\rangle$ are in the graph, which implies that node 1 reaches nodes 1,2,3,4,5, node 2 at least reaches nodes 2,3,4,5, node 3 at least reaches nodes 3,4,5 and node 4 at least reaches nodes 4,5.

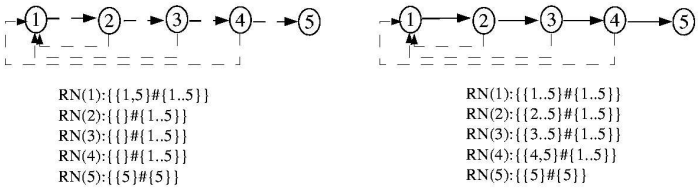

Figure 3: Discovering bridges

Notice that Hamiltonian Path (i.e., the problem where we have to find a simple path between two nodes containing all the nodes of the graph [10, 5]) can be reduced to Simple Path with mandatory nodes by defining the set of mandatory nodes as $nodes(g) - \{source, dest\}$.

The above definition of Simple Path with mandatory nodes can be formally defined as follows:

$$SPMN(g, source, dest, mandnodes, p) \leftrightarrow \begin{array}{l} p \in Paths(g, source, dest) \wedge \\ NoCycle(p) \wedge \\ mandnodes \subset nodes(p) \end{array} \quad (25)$$

$SPMN$ stands for "Simple Path with mandatory nodes". $NoCycle(p)$ states that p is a simple path (i.e., a path where no node is visited twice). This definition of Simple Path with mandatory nodes implies the following property:

$$Reachability(p, source, rn, cn, be) \wedge dest \in rn(source) \wedge cn(dest) \supseteq mandnodes \quad (26)$$

This is because the destination is reached by the source and the path contains the mandatory nodes. This derived property and the fact that we can implement $SPMN$ in terms of the *AllDiff* constraint [17] and the *NoCycle* constraint [4] suggest the two approaches for Simple Path with mandatory nodes summarized in Table 1 (which are compared in the next section). In the first approach, we basically consider *AllDiff* and *NoCycle*. In the second approach we additionally consider *Reachability*.

Notice that, even though the computation of bridges plays a crucial role in the pruning that *Reachability* performs, we do not use the be argument in the second approach. In fact be can play an important role in solving Constrained Euler Path problems (i.e., problems where the objective is to find a path visiting a set of edges by respecting some additional constraints).

Approach 1	**Approach 2**
$SPMN(g, source, dest, mandnodes, p)$	$SPMN(g, source, dest, mandnodes, p)$ $Reachability(p, source, rn, cn, be)$ $dest \in rn(source)$ $cn(dest) \supseteq mandnodes$

Table 1: Two approaches for solving Simple Path with mandatory nodes

4 Experimental results

In this section we present a set of experiments that show that *Reachability* is suitable for Simple Path with mandatory nodes. I.e., in our experiments *Approach 2* (in Table 1) outperforms *Approach 1*. These experiments also show that Simple Path with mandatory nodes tends to be harder when the number of optional nodes increases if they are uniformly distributed in the graph. We have also observed that the labeling strategy that we implement with *Reachability* tends to minimize the use of optional nodes (which is a common need when the resources are limited).

In Table 2, we define the instances on which we made the tests of Table 4. The id of the destination is also the size of the graph. The column Order is true for the instances whose mandatory nodes are visited in the order given. Notice that SPMN_52Order_b has no solution. The time, in Table 4, is measured in seconds. The number of failures means the number of failed alternatives tried before getting the solution.

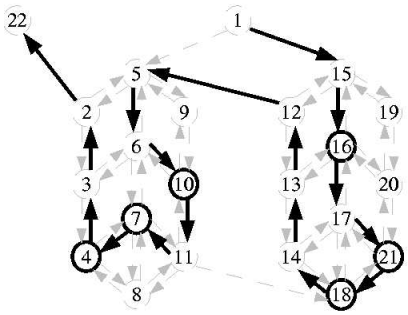

Figure 4: SPMN_22:A path from 1 to 22 visiting 4 7 10 16 18 21

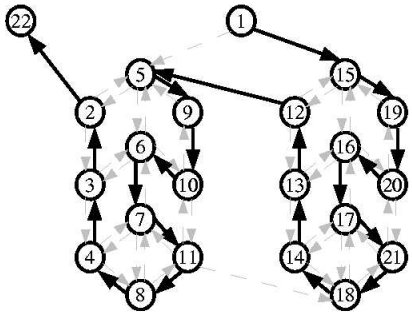

Figure 5: SPMN_22full:A path from 1 to 22 visiting all the nodes

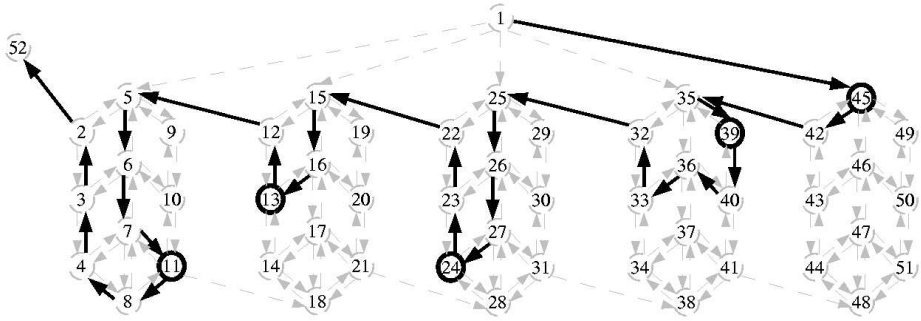

Figure 6: SPMN_52a:A path from 1 to 52 visiting 11 13 24 39 45

In our experiments, we have made five types of tests: using *SPMN* without *Reachability* (column "SPMN"), using *SPMN* and *Reachability* but without computing cut nodes nor bridges (column "SPMN+R"), using *SPMN* and *Reachability* but without computing bridges (column "SPMN+R+CN"), using *SPMN* and *Reachability* but without computing cut nodes (column "SPMN+R+BE"), and using *SPMN* and *Reachability* (column "SPMN+R+CN+BE").

As it can be observed in Table 4, we were not able to get a solution for SPMN_22 in less than 30 minutes

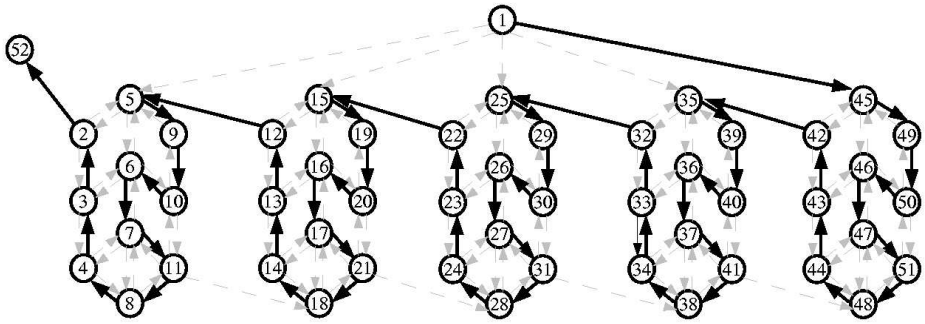

Figure 7: SPMN_52full:A path from 1 to 52 visiting all the nodes

Name	Figure	Source	Destination	Mand. Nodes	Order
SPMN_22	4	1	22	4 7 10 16 18 21	false
SPMN_22full	5	1	22	all	false
SPMN_52a	6	1	52	11 13 24 39 45	false
SPMN_52b	6	1	52	4 5 7 13 16 19 22 24 29 33 36 39 44 45 49	false
SPMN_52full	7	1	52	all	false
SPMN_52Order_a	6	1	52	45 39 24 13 11	true
SPMN_52Order_b	6	1	52	11 13 24 39 45	true

Table 2: Simple Path with mandatory nodes instances

Opt. Nodes	Failures	Time
5	30	89
10	42	129
15	158	514
20	210	693
25	330	1152
32	101	399
37	100	402
42	731	3518
47	598	3046

Table 3: Performance with respect to optional nodes

without using *Reachability*. In fact, we did not even try to solve SPMN_52b without it. However, even though the number of failures is still inferior, the use of *Reachability* does not save too much time when dealing with mandatory nodes only. This is due to the fact that we are basing our implementation of *SPMN* on two things: the use *AllDiff* [17] (that lets us efficiently remove branches when there is no possibility of associating different successors to the nodes) and the use *NoCycle* [4] (that avoids re-visiting nodes).

The reason why *SPMN* does not perform well with optional nodes is because we are no longer able to impose the global *AllDiff* constrain on the successors of the nodes since we do not know a priori which nodes are going to be used. In fact, one thing that we observed is that the problem tends to be harder to solve when the number of optional nodes increases. In Table 3, all the tests were performed using *Reachability* on the graph of 52 nodes.

Even though, in SPMN_22, the benefit caused by the computation of bridges is not that significant, we were not able to obtain a solution for SPMN_52b in less than 30 minutes, while we obtained a solution in 402 seconds by computing bridges. So, even though the computation of bridges is costly, that computation pays off in most of the cases. Nevertheless, cut nodes should be computed in order to profit from the computation of bridges.

Problem		SPMN		SPMN+R		SPMN+R+CN		SPMN+R+BE		SPMN+R+CN+BE	
Instance	Figure	Failures	Time	Failures	Time	Failures	Time	Failures	Time	Failures	Time
SPMN_22	4	+130000	+1800	91	6.81	40	6.55	70	13.76	13	4.45
SPMN_22full	5	213	1.44	19	0.95	0	0.42	19	2.76	0	1.22
SPMN_52b	-	-	-	+900	+1800	+700	+1800	+1000	+1800	100	402
SPMN_52full	7	3012	143	774	765	3	8.51	+700	1800	3	45.03
SPMN_52Order_a	6	+12000	+1800	51	46.33	55	81	27	97	16	57.07
SPMN_52Order_b	-	+12000	+1800	+1500	+1800	81	157	+400	+1800	41	117

Table 4: Simple Path with mandatory nodes tests

4.1 Labeling strategy

Reachability provides interesting information for implementing smart labeling strategies due to the fact that it associates each node with the set of nodes that it reaches. This information can be used to guide the search in a smart way. For instance, we observed that, when choosing first the node that reaches the most nodes, we obtain paths that minimize the use of optional nodes (as it can be observed in 6).

4.2 Imposing order on nodes

An additional feature of *Reachability* is the suitability for imposing dependencies on nodes (which is a common issue in routing problems). In fact, it might be the case that we have to visit the nodes of the graph in a particular (partial) order.

Our way of forcing a node i to be visited before a node j is by imposing that i reaches j and j does not reach i. The tests on the instances SPMN_52Order_a and SPMN_52Order_b show that *Reachability* takes the most advantage of this information to avoid branches in the search tree with no solution. Notice that we are able to solve SPMN_52Order_a (which is an extension of SPMN_52a) in 57.07 seconds. We are also able to detect the inconsistency of SPMN_52Order_b in 117 seconds.

5 Related work

- The cycle constraint of CHIP [1, 2] $cycle(N, [S_1, ..., S_n])$ models the problem of finding N distinct circuits in a directed graph in such a way that each node is visited exactly once. Certainly, Hamiltonian Path can be implemented using this constraint. In fact, [2] shows how this constraint can be used to deal with the Euler knight problem (which is an application of Hamiltonian Path). However, this constraint only covers the case where we are to visit all the nodes of the graph, which is a specific case of Simple Path with mandatory nodes.

- [19] suggests some algorithms for discovering mandatory nodes and non-viable edges in directed acyclic graphs. These algorithms are extended by [3] in order to address directed graphs in general with the notion of strongly connected components and condensed graphs. Nevertheless, examples like SPMN_52a represent tough scenarios for this approach since almost all the nodes are in the same strongly connected component.

- CP(Graph) introduces a new computation domain focussed on graphs including a new type of variable, graph domain variables, as well as constraints over these variables and their propagators [7, 8]. CP(Graph) also introduces node variables and edge variables, and is integrated with the finite

domain and finite set computation domain. Consistency techniques have been developed, graph constraints have been built over the kernel constraints and global constraints have been proposed. $Path(p, s, d, maxlength)$ is one of these global constraints. This constraint is satisfied if p is a simple path from s to d of length at most $maxlength$. Certainly, Simple Path with mandatory nodes can be implemented in terms of *Path*. However, we still have to compare the performance of *Reachability* with respect to this approach.

6 Conclusion and future work

We presented *Reachability*: a constrained path propagator that can be used for speeding up constrained path solver. After introducing its semantics and pruning rules, we showed how the use of *Reachability* can speed up a standard approach for dealing with Simple Path with mandatory nodes.

Our experiments show that the gain is increased with the presence of optional nodes. This is basically because we are no longer able to apply the global *AllDiff* since we do not know a priori which nodes participate in the path.

From our observations, we infer that the suitability of *Reachability* is based on the strong pruning that it performs and the information that it provides for implementing smart labeling strategies. We also found that *Reachability* is appropriate for imposing dependencies on nodes. Certainly, we still have to see whether our conclusions apply to other types of graphs.

It is important to remark that both the computation of cut nodes and the computation of bridges play an essential role in the performance of *Reachability*. The reason is that each one is able to prune when the other can not. Notice that Figure 2 is a context where the computation of bridges cannot infer anything since there is no bridge. Similarly, Figure 3 represents a context where the computation of bridges discovers more information than the computation of cut nodes.

A drawback of our approach is that each time we compute cut nodes and bridges from scratch, so one of our next tasks is to overcome this limitation. I.e., given a graph g, how can we use the fact that the set of cut nodes between i and j is s for recomputing the set of cut nodes between i and j after the removal of some edges? We believe that a dynamic algorithm for computing cut nodes and bridges will improve our performance in a radical way.

As mentioned before, the implementation of *Reachability* was suggested by a practical problem regarding mission planning in the context of an industrial project. Our future work will concentrate on making propagators like *Reachability* suitable for non-monotonic environments (i.e., environments where constraints can be removed). Instead of starting from scratch when such changes take place, the pruning previously performed can be used to repair the current pruning.

References

[1] N. Beldiceanu and E. Contejean. Introducing global constraints in chip, 1994.

[2] E. Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes*. Doctoral dissertation, Université Paris, Paris, France, 1999.

[3] H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. In *10e Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'04)*, pages 107–121, Angers, France, June 2004.

[4] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In *International Conference on Logic Programming*, pages 316–330, 1997.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[7] G. Dooms, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *5èmes Journées Ouvertes Biologie Informatique Mathématiques*, 2004.

[8] G. Dooms, Y. Deville, and P. Dupont. CP(Graph):introducing a graph computation domain in constraint programming. In *CP2005 Proceedings*, 2005.

[9] F. Focacci, A. Lodi, and M. Milano. Solving tsp with time windows with constraints. In *CLP'99 International Conference on Logic Programming Proceedings*, 1999.

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the The Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *CONSTRAINTS journal*, 1(3):191–244, 1997.

[12] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2004. Available at *http://www.mozart-oz.org/*.

[13] T. Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[14] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows, 1996.

[15] L. Quesada, P. Van Roy, and Y. Deville. Reachability: a constrained path propagator implemented as a multi-agent system. In *CLEI2005 Proceedings*, 2005.

[16] L. Quesada, P. Van Roy, and Y. Deville. The reachability propagator. Research Report INFO-2005-07, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.

[17] J. C. Régin. A filtering algorithm for constraints of difference in csps. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, 1994.

[18] C. Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[19] M. Sellmann. *Reduction Techniques in Constraint Programming and Combinatorial Optimization*. Doctoral dissertation, University of Paderborn, Paderborn, Germany, 2002.

[20] Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 1978.

[21] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.