MROPE II: A Finite Domain Solver on top of Mercury

Henk Vandecasteele
K.U.Leuven, departement computerwetenschappen,
Celestijnenlaan 200A
B-3001 Heverlee
henk.vandecasteele@cs.kuleuven.ac.be

Abstract

In this paper we describe a new implementation of the Finite Domain solver ROPE [Van99], called MROPE II. This new version was preceded by an implementation on top of Prolog [VD94] and a version using an early version of Mercury [VDV96]. In the previous implementation Mercury was chosen for it's speed and, compile-time checking properties and fast reliable development. This previous experiment with Mercury was already a success, still there were some problems. For example for an efficient execution backtrackable destructive assignment was needed. Later on, this (backtrackable destructive assignment) and other features like impure declarations were added to the Mercury system. All we needed for a new attempt on a new implementation of ROPE: MROPE II.

1 The Finite Domain CLP System MROPE II

The constraints handled in our new implementation, can easily be deduced from the Mercury types in the interface of the solver:

A constraint has the following type:

```
:- type expression ---> num(int)
                      ; var(finite_var)
                      ; +(expression, expression)
                      ; -(expression, expression)
                      ; *(expression, expression).
:- type constraint ---> <>(expression, expression)
                      ; =(expression, expression)
                      ; =<(expression, expression)
                      ; >=(expression, expression)
                      ; >(expression, expression)
                      ; <(expression, expression)
                            /* A constraint encapsulated into an equiv
                               constraint should not be an equiv again */
                      ; equiv(finite_var, constraint) .
```

In contrast with the previous solver, *indexical* constraints are not part of the solver yet. Depending on the need of the applications such constraints can be brought into the system very easily. *Reification* constraints on the other hand are part of the language in the format of equivalence. *equiv(finite_var, constraint)* is a true constraint if the first parameter is a boolean variable expressing the truth value of the constraint in the second parameter. This second parameter cannot be an equiv/2 constraint again. This is not really a restriction on the language and could be overcome with a small transformation. Reification constraints are quite powerful, they can be used as a more flexible stand-in for the cardinality constraint [VD91].

The type range,which is used to specify a domain of a variable is a bit awkward, but it allows expressing the domains very easily. Also infinite domains are allowed.

```
:- type range —> num(int)  /* A singleton */
        ; int(int, int)        /* An interval the bounds included */
        ; unn(int, range)      /* A union of an integer and a remaining range */
        ; uni(int, int, range) /* A union of an interval made by the
                                  first two arguments and another range */
        ; inf                  /* An infinite domain open two both ends */
        ; unl(int, range)      /* A union of an interval from minus infinity up to the
                                  first argument with the range in the second */
        ; opl(int)             /* An interval from -infinity up to the argument */
        ; opr(int).            /* An interval from the argument up to plus infinity */
```

The remainder of the interface defines method to create new variables, adding constraints, find solutions with an enumeration predicate. Also a method is provided to retrieve the current value/domain of Finite Domain variables.

The solver has an explicit current state, containing all finite domain variables and constraints. It has type *fd_store*. This store is initialised with the predicate *init/1*.

```
:- pred init(fd_store).
:- mode init(fd_store_muo) is det.
```

Every finite domain variable within the system needs to be created before use. This can be done with the predicate `new_var/4`.

```
:- pred new_var(finite_var, range, fd_store, fd_store).
:- mode new_var(out, in, fd_store_mdi, fd_store_muo) is det.
```

A new constraint is added to the system by calling *add_constraint/3*.

```
:- pred add_constraint(constraint, fd_store, fd_store).
:- mode add_constraint(in, fd_store_mdi, fd_store_muo) is semidet.
```

Finally enumeration can be started with *enum/6*.

```
:- pred enum(list(finite_var), var_selection_mode, value_selection_mode,
             backtrack_mode, fd_store, fd_store).
:- mode enum(in, in, in, in, fd_store_mdi, fd_store_muo) is nondet.
```

Other variants of the enumeration exist: one in case of optimisation; one with preferred values for the variables; and one combining optimisation and preferred values. An example where enumeration with preferred values can be used is rescheduling.

One can also request for the current domain of a finite domain variable with the predicate *value_var*.

```
:- pred value_var(finite_var, list(int), fd_store).
:- mode value_var(in, out, fd_store_mui) is det.
```

The reader may have noticed that the predicates making up the interface of the solver use the modes *fd_store_muo*, *fd_store_mdi* and *fd_store_mui*. These modes use *mostly_unique* and *mostly_dead* instantiation patterns. *mostly_unique* and *mostly_dead* allow to specify when there is only one reference to a particular value, and when there will be no more references to that value. Mercury defines some standard modes for manipulating ”mostly unique” values:

```
% mostly unique output
:- mode muo :: free -> mostly_unique.

% mostly unique input
:- mode mui :: mostly_unique -> mostly_unique.

% mostly destructive input
:- mode mdi :: mostly_unique -> mostly_dead.
```

fd_store_mdi is equivalent to *mdi* but specific for the type *fd_store*. *fd_store_muo* relates to *muo* in the same way and *fd_store_mui* relates to *mui*. In our case it means that when a store has been used as a parameter with mode *fd_store_mdi*, the variable referring to this store can not be used in the remainder of the current predicate. A parameter used in a call as parameter with mode *fd_store_muo* is known to be the only reference to the current store. The mode *fd_store_mui* means that the corresponding parameter is the only reference to that data, and the code of the called predicate will keep it that way. An important property of these *mostly_unique* and *mostly_dead* instantiation patterns is the possibility to use them in non-deterministic code. This is what *mostly* stands for. A parameter which is *mostly_dead* cannot be used in future code, and is therefore dead, but it could come alive again after backtracking. An example of a program using this MROPEII module can be found in Section 4.

2 A log of previous experiments

2.1 Implementing a finite domain solver in Prolog

In Logic Programming, it is standard practice to implement *enhanced* Logic Programming languages in Prolog through meta-interpretation. The advantage of such an approach is that those features of the *enhanced* language that coincide with corresponding features of Prolog can be implemented through downward reflection. Only features of the language that require a treatment different from that in Prolog are reified in the meta-interpreter. However, it is well-known that this type of implementation may produce significant overhead. One of the main motivations of LP work on partial evaluation has been to remove this overhead through transformation. We refer to [BD92] for a more extensive discussion on these issues.

It has frequently been observed that an alternative solution, avoiding the need for partial evaluation, is to implement the enhanced language by means of a transformer which maps the enhanced language to Prolog. Denoting a meta-interpreter for the enhanced language as M and a partial evaluator for Prolog as PE, conceptually such a transformer can be defined as a program T, such that for every program P in the enhanced language: T(P) $\equiv$ PE(M(P)). The point raised in discussions on this topic is that writing T from scratch is often not significantly more difficult than writing M and should therefore be considered as a valid (and more economic) approach to implementing the enhanced language.

In this version of the ROPE language such a "transformation approach" was taken. The Finite Domain CLP program is then transformed to Prolog program which contains calls to a Finite Domain Library.

2.1.1 Passing information around

In case there is no extra information to be passed around such a transformation T is quite simple: wrapping the special features of the enhanced language in calls to predefined library predicates will do the job. Given the program:

Example 2.1

a(X, Y):- r(X), c(X, Y).
a(X, Y):- s(X, Y, Z), a(Y, Z).
with r/1 and s/3 special calls.

Then this would be transformed to:

Example 2.2

a(X, Y):- takeCareOfFeature(r(X)), c(X, Y).
a(X, Y):- takeCareOfFeature(s(X, Y, Z)), a(Y, Z).

If there is need for extra information to be passed around from one call to takeCareOfFeature to another then every clause of the program needs an extra parameter, and the clause must be renamed. For every predicate definition in the program a new clause is added with the original name of the clause which calls the transformed clause with the initialised extra parameters. Applying this technique to the example results in:

Example 2.3

```
a(X, Y):- initialise(ParameterIn),
        a_1(X, Y, ParameterIn, ParameterOut),
        results(ParameterOut).
a_1(X, Y, ParamIn, ParamOut):-
        takeCareOfFeature(r(X), ParamIn, Param1),
        c(X, Y, Param1, ParamOut).
a_1(X, Y, ParamIn, ParamOut):-
        takeCareOfFeature(s(X, Y, Z), ParamIn, Param1),
        a_1(Y, Z, Param1, ParamOut).
```

In some cases such parameter passing can be avoided by instantiating the variables where the special features of the language act on, with the information to be passed. This technique is not general but is applicable in our case. Then we also have to rename the predicates, as we have to intercept the output of the program and reconstruct the output that the original program was supposed to produce. The compiled program then looks like:

Example 2.4

```
a(X, Y):-
        getFreeVariables(a(X, Y), Free),
        a_1(X, Y),
        printResults(Free).
a_1(X, Y):- takeCareOfFeature(r(X)), c_1(X, Y).
a_1(X, Y):- takeCareOfFeature(s(X, Y, Z)), a_1(Y, Z).
```

First the free variables are extracted from the query. After successful completion of the program we print the results of the original program. The user is responsible for preventing Prolog from printing the internal representation of X and Y. This can be done by adding a *fail* to the query: *a(X, Y), !, fail* in case only one solution is needed. *a(X, Y), fail* if all solution are wanted.

Of course with this kind of technique special care must be taken. Normal Prolog unification of the special variables must be prevented while transforming the program. Also the use of builtins on these special variables must be taken care of.

2.1.2 The finite domain library

Before execution of such programs above a library must be provided. More specific for a finite domain constraint solver a predicate must be defined to handle a constraint to replace to calls to *takeCareOfFeature/1*. There are three predicates in the finite domain library:

- getFreeVariables/2. This predicates searches for the free variables in first arguments and stores them in a list in the second argument.
- printResults/1 prints out the list of Prolog expressions while substituting the finite domain variable with their semantic value.
- constraint_/1 (takeCareOfFeature/1) is by far the most interesting predicate of the finite domain library. This predicate instantiates domain variables to a specific structure, fulfilling the purpose of passing on relevant information for the solver, adds the constraint to the constraint store and activates the solver for checking the constraint.

In the next subsections we explain how the store, containing constraints and domains, is kept.

2.1.3 Representation of the store

Within the constraint solver constraints and domains of Finite Domain variables must be stored in some way. These domains and constraints should be easily accessible. Domains are updated and used very

frequently. Whenever a domain changes, constraints affected must be activated. As stated before, one alternative is to add to every clause an extra input and output parameter. The data structure in these extra parameters contain information on the constraints connected to the variables and the domains as well. Every finite domain variable then refers to this global data structure with a unique number. When information is changed concerning a finite domain variable, a smaller domain for example, then the out parameter reflects these changes while it still contains the old information on the other variables. Such a working method results in efficiency problems as updates to this global store are dependent on the number of finite domain variables in the system. An example of such a data structure is a flat term where each argument contains information on one finite domain variable (in the sequel, we refer to this representation as "functor"). The unique number connected to each finite domain variable is the position of this argument in the functor. If the information on one variable changes then a new functor must be created where all arguments but one must be copied from the old functor. Time and place complexity of this operation is O(N), where N is the number of finite domain variables in the program. A better alternative is a tree structure. In this case the complexity of copying in case of changes is logarithmic in the number of existing finite domain variables. Unfortunately, also access without modification becomes logarithmic in the number of finite domain variables.

Therefore we choose to instantiate each finite domain variable to a structure that contains both the domain of the variable and the constraints in which this variable is involved. Then extra arguments for every clause in the program containing information on the current state of constraints and domains are not needed. The access to the data on one variable, having the variable available, becomes independent on the number of variables. There exist several references to one finite domain variable, namely each constraint in which the variable occurs. As a result we cannot replace the variable with a new variable when the domain changes. For this purpose we will use open ended data structures[1]. Passing on information on constraints and domains through the domain variable itself is particularly interesting for our application, since we need to propagate constraints as soon as some type of changes occurs to the domain of a variable. Thus, by instantiating a domain variable to a structure *finiteVar(Domain, Constraint_Store)*, where *Domain* is a representation of the domain of the variable and *Constraint_Store* of all constraints in the store that contain the variable, easy access for propagation is guaranteed. Actually there are only four classes of constraint to be activated.

- Constraints to be activated after every change to the domain of a variable.
- Constraints to be activated after a variable becomes ground.
- Constraints to be activated after the lower bound of a variable's domain changes.
- Constraints to be activated after the upper bound of a variable's domain changes.

Note that some constraint belong at the same time to different classes. The structure for the constraint store is then easily chosen: *store(Always, Ground, LowerBound, UpperBound)*. With *Always* the constraints that need to be checked whenever the domain of the associated variable changes, *Ground* the constraints that need to be checked as soon as the finite domain variable becomes ground and *Lowerbound* and *UpperBound* whenever the lower bound (resp the upper bound) of the domain changes. If we handle a new constraint we check for the operators used in the expression. An expression *int(X), min(X)* will lead to storing the constraint in the *LowerBound* constraint list of the variable X. *dom(Y)* tells the system to put the constraint in the *Always* constraint list of variable Y. *ask(ground(Z), Constraint)* will put the constraint in the *Ground* list of the variable Z.

As an example, after treating the 4 low-level constraints of the rabbit program we obtain the following instantiation of the finite domain variables P and R:

Example 2.5

P=finiteVar(Domain1,store([],[],[Constr2,Constr4],[Constr2,Constr4])),
R=finiteVar(Domain2,store([],[],[Constr1,Constr3],[Constr1,Constr3])),
with

[1] we come back later to this matter, together with the time complexity issue involved.

Constr1=P in 9 - int(R),
Constr2=R in 9 - int(P),
*Constr3=P in (24 - 4*int(R)) div 2,*
*Constr4=R in (24 - 2*int(P)) div 4.*
(the values for the domains will be discussed in the next subsection)

These data structures are generated while the predicate *constraint_/1* from the finite domain library handles a new constraint. Note that the representation we choose here for the list of constraints is too simplistic. Since constraints can be added at any time of the execution we have to use open ended data structures, discussed below.

2.1.4 Representation of the domains

As we are going to reason on the bounds of the domains, add and subtract domains we need a representation which is convenient for this kind of computations. Therefore we choose a union of intervals. For example a domain with the values 1, 2, 3, 4, 8, 10, 11, 12 will be represented as $1..4 \vee 8..8 \vee 10..12$ using the same operators as defined in the low-level language. A fresh finite domain variable is initialised with the domain 0..infinity. As we have to update the domains of the finite domain variables we also need a structure which we can update by further instantiating. Using an open ended list for explaining the principle, every finite domain variable then starts with the domain representation: [0..infinity| _]

For the rabbit problem: after adding the two constraints *constraint1* and *constraint2* both finite domain variables are instantiated as

finiteVar([0..infinity, 0..9 | _], Constraint_Store).

after adding the two other constraints *constraint3* and *constraint4* the data structure becomes

P = finiteVar([0..infinity, 0..9, 3..7, 5..6, 6..6| _], Store1)
R = finiteVar([0..infinity, 0..9, 2..6, 3..4, 3..3| _], Store2)

As a result of adding the two constraints to the constraint store and letting them propagate the two variables become ground. How this result is achieved is explained below.

2.1.5 Open ended data structures

This subject has already been mentioned several times above: the part of the store connected to one variable cannot be replaced but should always be changed by further instantiating it. In case of the constraints a simple open ended list can be used. When adding a new constraint the end of the list is further instantiated with a new element, the new constraint, and a fresh variable becomes the new end of the list. Also, for some optimisations in the finite domain library and for implementing constraints like cardinality, there is need for removing constraints from the constraint store. This is solved by adding a free variable to every constraint. A constraint can be deactivated by instantiating this free variable to the atom 'old'. When activating constraints, after a domain of a finite domain variable has changed, such constraints must be skipped. Similar to adding new constraints to the store, the domain of the Finite Domain variable should be changed during computation. In principle this could again be done with an open ended list. The annoying thing here is that access to the domain, and updating as well, becomes linear in the number of updates to the finite domain variable. Indeed when fetching the current domain, the whole list of old domains must be traversed until the last element before the open end is reached. In our implementation we used an open ended data structure based on trees, which we will call open ended trees. The access and update complexity of this open ended tree is logarithmic in the number of updates to the domain. Moreover an extra variable is reserved which is instantiated when the domain becomes a singleton, hence fetching the domain of a Finite Domain variable which has been assigned can be done in constant time. For the curious reader interested in this topic the code handling this feature is added:

Program 2.1

```
getDomain(domain(Tree, Var), Range):- % Var is ground for
    ( atomic(Var) →     % a singleton domain
            Range = ..(Var, Var)          ; findRange(Tree, Range)
    ).
```

```
findRange(Tree, Range):-
        ( nonvar(Tree) →
                lookup(Tree, D)
        ;        upperbound(Limit),
                Range = ..(0, Limit) % still a free variable
        ).
putDomain(Info, Domain):-
        ( Domain = ..(Value, Value) →
                arg(2, Info, Value)
        ;        arg(1, Info, Tree),
                insert(Tree, Domain)
        ).
/* An implementation of an open ended tree. It has an update and retrieval complexity
   logarithmic in the number of updates. The last value in the tree is always the
   right-most nonvar element in the tree. When updating the tree grows from left to right,
   while subtrees grow larger, such that logarithmic access is assured. */
lookup(tree(Left, El, Right), Value):-
        ( var(Right) →
                ( var(Left) →
                        El = Value
                ;        lookup(Left, Value)
                )
        ;
                lookup(Right, Value)
        ).
insert(Tree, Value):-
        ( nonvar(Tree) →
                insert1(Tree, Value, 1)
        ;        Tree = tree(_, Value, _)
        ).
insert1(tree(Left, _, Right), Value, Depth):-
        ( var(Right) →
                insert2(Left, Value, Depth, Right)
        ;        Depthplus1 is Depth + 1,
                insert1(Right, Value, Depthplus1)
        ).
insert2(tree(Left, El, Right), Value, Depth, Back):-
        ( var(El) →
                El = Value
        ;        ( Depth = 0 →
                        Back = tree(_, Value, _)
                ;        Depthmin1 is Depth - 1,
                        ( var(Right) →
                                insert2(Left, Value, Depthmin1, Right)
                        ;        insert2(Right, Value, Depthmin1, Back)
                        )
                )
        ).
```

2.1.6 Queue of constraints

A final data structure concerns the queue of constraints which were activated, but not used yet. This is an open ended list where new constraints are added at the end of the list. A desirable feature of such a queue is that a constraint can only appear once in the queue. For this purpose an unbalanced binary tree is used in

the pure Prolog version. In an impure version, we have used a record database. Every constraint contains a unique number. For every constraint in the queue this number is stored in the tree/database.

2.2 Early Mercury as a platform for a solver

At the time of this first experiment, Mercury [SHC94] was a recent phenomenon in the field of logic programming: it was faster than other logic language implementations (e.g. Prolog) and better suited for the development of large applications because of its compile time error detection capabilities. The implementation described in the previous subsection is a prototype implemented in pure Prolog [VD94]. This implementation will further be referred to as ROPE. This prototype lacks efficiency because its implementation doesn't rely on any non-standard support of the Prolog implementation. On the other hand, this helped the development of the ideas of ROPE quite a bit. We believe that the reason for this inefficiency is partly the generality of Prolog (reversible, non-typed predicates) and partly the lack of support for data structures that can be updated at constant cost. When the new logic programming language Mercury emerged, it looked very promising to port our prototype to this new system. One of the advantages of Mercury is faster execution: type, mode and determinacy declarations allow to generate more efficient code. In this subsection we summarise the work presented in [VDV96] and [VDV99].

2.2.1 A preview on efficiency gain

As an initial experiment we ported the code computing intersections of domains. Then we compared the speed of the Prolog version on ProLog_by_BIM and the Mercury version on a Sparccenter-1000.

About 40.000 intersections were computed.

	ProLog_by_BIM	Mercury
40.000 intersections	3.7s	0.36s

Table 1: A preview on speedup

The table 1 shows that it is reasonable to expect a 10-fold speedup when going from Prolog to Mercury, on the assumption that Mercury offers efficient pure alternatives for the tricks in the Prolog program.

2.2.2 Replacing open ended data-structures

Replacing the open ended data-structures, heavily used in the Prolog version, was a hard job. Without using the C-interface for implementing other data-structures is was unavoidable to add extra parameters to the program passing information around about the state of the finite domain variables and the involved constraints. For representing this global state a Mercury array was used. To our surprise the program was slower than the Prolog version. In a attempt to find the reason for this inefficiency, different data-structures were tried in this global store: the arrays were replaced by binary trees and a plain functor.

It turned out that this data-structure was significant as can be seen in table 2

	queens(10)
Mercury array	47s
bintree	25s
functor /20	21s

Table 2: Changing the data-structure

The execution time of the Prolog version being 39 seconds, the two new versions were faster than the Prolog execution.

2.2.3 Using backtrackable destructive assignment

Finally we decided to hack the Mercury system and add our own backtrackable destructive assignment. Then we could change the data on finite domain variables in-place, having a constant time access and update time-complexity. Next to this in-place modification, we could also add a cardinality constraint to the system. With the additional cardinality constraints a wider range of examples was possible.

For comparison the same code was executed in SICStus v2.1 using setarg because the ROPE system on ProLog_by_BIM and the implementation in Mercury now use totally different data structures.

	MROPE(Mercury)	MROPE(SICSTUS)	ROPE(BIM)	pure MROPE
queens	21s	109s	39s	65s
bridge1	1.9s	10s	27s	5s
bridge2	1.7s	11s	14s	
perfect	23s	125s	126s	
suudoku	1.2s	5.5s	6.8s	

Table 3: The effect of backtrackable destructive assignment

Table 3 shows a bad result for the queens problem. In the version of ROPE in Prolog *the different from* constraint is handled at a higher level, which allows some optimisation: a constraint X <> Y can be removed from the constraint store as soon as one variable is instantiated. This is still not done in the Mercury version.

So, ignoring the queens result, we observe that MROPE is 5 to 15 times faster than the original ROPE in Prolog.

3 Using a More Mature Mercury

As the Mercury system is a constant evolving system, the Mercury system has become a lot more mature since the experiment reported in the previous section. In the current version (0.8.1) building stones for backtrackable destructive assignment and other features, like impure declarations, were included. This makes it possible to create an efficient but still high level implementation of our Finite Domain solver.

3.1 Data structures and implementation

A finite domains solver has three main data structures:

- domains attached to the finite domain variables,
- constraints and
- a queue of constraints to be checked in the fixpoint algorithm.

For all operations on these data structures: creation, retrieval and update, it is important to achieve constant time operations. In this section we will show that Mercury allows to define and use such data structure with a minimum of low-level programming. The low-level programming concerns the use of backtrackable destructive assignment. A small module *mutable* defines such operations:

```
:- module mutable.
:- interface.

:- type mutable(T). /* a polymorfic mutable object */

:- pred mutable_init(mutable(T), T).
:- mode mutable_init(out, in) is det.
```

```
                /* create a new mutable object with the initial contents */

:- pred mutable_overwrite(mutable(T), T).
:- mode mutable_overwrite(in, in) is det.
                /* Overwrite the current contents of the mutable object with new data. When
                   the system backtracks over this operation the old data will be restored */

:- pred mutable_get(mutable(T), T).
:- mode mutable_get(in, out) is det.
                /* Get the current contents of the mutable object */
```

These operations are implemented using the C-interface of the mercury system:

```
:- pragma(c_code, mutable_init(Mutable::out, Value::in),
                        will_not_call_mercury, "{
                Word *mutable;
                mutable = make(Word);
                (*mutable) = Value;
                Mutable = (Word) mutable;
}").

:- pragma(c_code,mutable_overwrite(Mutable::in,Value::in),
                        will_not_call_mercury, "{
                Word *mutable;
                mutable = (Word *) Mutable;
                MR_trail_current_value(mutable);
                (*mutable) = Value;
}").

:- pragma(c_code, mutable_get(Mutable::in, Value::out),
                        will_not_call_mercury, "{
                Word *mutable;
                mutable = (Word *) Mutable;
                Value = *mutable;
}").
```

Using the mutable object a whole new range of new data structures can be used: doubly linked lists, difference lists, ... On creation of a new variable, the implementation creates a number of slots containing mutable objects.

- A slot for the current domain.
- Five slots for lists of constraints. Each of those five slots contain constraints with a different propagation scheme. The constraints in the first slot are only propagated whenever the domain of the variable becomes a singleton, the second list of constraints becomes active when the lower-bound of the domain changes, the third when the upper bound changes. A fourth list of constraints is activated when the upper bound or the lower bound changes[2]. The last list of constraints is propagated whatever change to the domain is made.
- An extra slot can be added containing a string, describing a problem specific meaning of the finite domain variable (can be used for debugging or output purposes).

Since the information in those slots will be updated by backtrackable destructive assignment, there is need to store these slots in a global data-structure. When storing directly them as the contents of the

[2] In previous version this slot was not available, all constraints to be activated as soon as the lower bound or upper bound changed were added twice to the constraint store for the variable. Once in the list to be activated when the lower bound changed, and once to the list to be activated when he upper bound changed.

finite domain variable, the infomation will be visable and updateable from anywhere in the program at constant cost. For storing the constraints in one of these five slots holding constraints a doubly linked list is used. This way new constraints can be added, old constraints deleted and constraints retrieved at constant time. The queue of constraints maintained by the fixpoint algorithm is a difference list. The only way to implement a difference list in the current version of Mercury is using backtrackable destructive assignment. This way constraints can be added and removed in constant time. An important property is the ability to concatenate queues in constant time. Of course special attention has to be paid when using these mutable data structures, bugs in these parts of the program will never be found by the compiler. Fortunately the size of the non-declarative code is rather small, and the advantage of using Mercury is gained for the remainder of the code.

3.2 Comparison

A small comparison was made between this new implementation (MROPE II), the clp(fd) library [COC97] from SICStus, the old implementation of ROPE in Prolog on top of SICStus, and CHIP, one of the leading commercial products in the area having a low-level implementation.

benchmark	MROPE II	SICStus	ROPE(SICStus)	CHIP
queens 8 (all solutions)	0.12	0.25	1.06	0.08
queens 25 (first solution)	5.37	13.45	45.27	2.77
bridge1	3.35	1.96	8.79	0.82
bridge2	2.03	2.19	4.44	1.13
suudoku	0.31	0.16	0.2	0.18

Table 4: Comparing the solver in Mercury with other systems on a SparcII

Five small benchmarks were selected. The classic queens problem, finding all solutions for queens(8) and finding the first solution for queens(25). Two implementation of the bridge problem [Van89], the version *bridge1* backtracks over the disjunctions for finding the optimal solution, the version *bridge2* first expresses the disjunctions connecting a boolean variable to each of the disjunctions. In the end the boolean variables are enumerated for finding the optimal solution. In principle the last version should be superior, since the disjunctive constraints are brought into the store while delaying the choice. These constraints can start pruning before they are decided on. The last benchmark is *suudoku*, a Japanese puzzle.

The benchmarks were performed on both a Solaris machine with a SparcII processor running at 167 Mhz and a Linux machine having a Pentium II 233Mhz. On the Solaris machine the SICStus compiler generates “fastcode”, on the Linux machine the SICStus compiler generates “bytecode”, this explains the slower execution on the linux box. From table 4 we can see the Mercury implementation of our solver approaches the timing of the solver [COC97] delivered with SICStus. The Mercury implementation outperforms the Prolog implementation ROPE with grace. Our system is still two or three times slower than CHIP, and of course does not have the global constraints of CHIP. The results of running the same benchmarks on Linux, are shown in Table 5. Here the Mercury implementation is superior to the SICStus implementation, because the last one is using “bytecode” on Linux.

benchmark	MROPE II	SICStus	ROPE(SICStus)
queens 8 (all solutions)	0.05	0.42	1.79
queens 25 (first solution)	2.7	15.7	78.69
bridge1	1.57	2.57	14.11
bridge2	0.85	1.73	7.79
suudoku	0.06	0.18	0.35

Table 5: Comparing the solver in Mercury with other systems on a Pentium II

4 An example using MROPE II

Here an example using the primitives above. It is the implementation of the classic queens program.

Program 4.1

```
:- pred queens(io__state::di, io__state::uo) is det.
queens –>
        io__write_string(”number of queens?”),
        {non_logical_io__read_int(NQ)},
        ({
                init(Store1),
                generate(Q, NQ, NQ, Store1, Store2),
                safe(Q, Store2, Store3),
                enum(Q, up, up, standard_ex, Store3, Store4),
                non_logical_io__write_string(”[”),
                output_list(Q , Store4), fail} -> {true};
                io__write_string(”no_solutions”)
        ).
:- pred generate(list(finite_var), int, int, fd_store, fd_store).
:- mode generate(out, in, in , fd_store_mdi, fd_store_muo) is det.
generate(Q, N, M) –>
        ( {N =0} ->
                {Q = []};
                new_var(X, int(1, M)),
                {N1 is N - 1},
                {Q = [X|Q1]},
                generate(Q1, N1, M)
        ).
:- pred safe(list(finite_var), fd_store, fd_store).
:- mode safe(in, fd_store_mdi, fd_store_muo) is semidet.
safe(Q) –>
        ( {Q = []} ->
                {true}
        ;
                {Q = [X—T]},
                noAttack(X,1,T),
                safe(T)
        ).
:- pred noAttack(finite_var, int, list(finite_var), fd_store, fd_store).
:- mode noAttack(in, in, in, fd_store_mdi, fd_store_muo) is semidet.
noAttack(X, N, Q) –>
        ( {Q = []} ->
                {true}
        ;
                {Q = [Y|Z]},
                add_constraint(<>(var(X), +(var(Y), num(N)))),
                add_constraint(<>(var(X), var(Y))),
                add_constraint(<>(var(Y), +(var(X),num(N)))),
                {S is N + 1},
                noAttack(X, S, Z)
        ).
:- pred output_list(list(finite_var), fd_store).
:- mode output_list(in, fd_store_mui).
output_list(Q, Store):-
```

```
(
Q = [X],
        value_var(X, [V|_], Store),
        non_logical_io__write_int(V),
        non_logical_io__write_string("]")
;
        Q = [X, Y|Q1],
        value_var(X, [V|_], Store),
        non_logical_io__write_int(V),
        non_logical_io__write_string(", "),
        output_list([Y|Q1], Store)
).
```

5 Contribution

In this paper we show how a Finite Domain solver written on top of Mercury, with very little low-level programming, can be competitive with other well-known systems.

The system is available from http://www.cs.kuleuven.ac.be/ henkv/mropeII0.1.tar.gz

References

[BD92] M. Bruynooghe and D. De Schreye. Meta-interpretation. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 939–940. John Wiley & Sons, Inc, 1992.

[COC97] M. Carlsson, G. Ottosson, and B Carlson. An open-ended finite domain constraint solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.

[SHC94] Z. Somogyi, F. Henderson, and T. Conway. The implemenatation of mercury: an efficient declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989.

[Van99] H. Vandecasteele. *Constraint Logic Programming: Applications and Implementation*. PhD thesis, Department of Computer Science, K.U.Leuven, 1999.

[VD91] P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In *proceedings of ICLP*, pages 745–759, 1991.

[VD94] H. Vandecasteele and D. De Schreye. Implementing a finite-domain CLP-language on top of Prolog : a transformational approach. In Frank Pfenning, editor, *Proceedings of Logic Programming and Automated Reasoning*, number 822 in Lecture Notes in Artificial Intelligence, pages 84–98. Springer-Verlag, 1994.

[VDV96] H. Vandecasteele, B. Demoen, and J. Van Der Auwera. The use of mercury for the implementation of a finite domain solver. In *Proceedings of JICSLP'96 Post-conference Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*, 1996.

[VDV99] H. Vandecasteele, B. Demoen, and J. Van Der Auwera. The use of Mercury for the implementation of a finite domain solver. In I. de Castro Dutra, M. Carro, V. Santos Costa, G. Gupta, E. Pontellia, and Silva F, editors, *Nova Science Special Volume on Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science Publishers Inc, 1999.