The Use of Mercury for the Implementation of a Finite Domain Solver

Henk Vandecasteele
Bart Demoen
Joachim Van Der Auwera
Department of Computer Science
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
{henk.vandecasteele, bart.demoen}@cs.kuleuven.ac.be

September 9, 1996

Keywords: Logic Programming, Constraint Logic Programming.

abstract

Mercury [SHC94]is a recent phenomenon in the field of logic programming: it is 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 concepts and technology used in the implementation of Mercury are rather new and not yet evaluated thoroughly outside its implementors group. This paper reports on an evaluation of Mercury, by describing the porting from the medium-sized constraint solving tool ROPE written originally in Prolog, to Mercury. At first our aim and hope, was only to arrive at a faster implementation of the constraint solving tool, but in the course of the porting, it became clear that the main feature necessary for efficiency and missing from the current implementation of Mercury, is efficient backtrackable destructive assignment, at least for our application. We report on a naive port of the Prolog implementation of ROPE in Mercury, a redesigned version using more efficient data-structures and finally a version which uses our own hacked together implementation of backtrackable destructive assignment: a future version of Mercury will hopefully support this functionality through user-declarations or analysis.

1. Motivation

In the past years, we have developed a prototype finite domain solver ROPE on top of Prolog [VDS94]. This implementation will further we 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 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. On the other hand, exactly because of these declarations the port was non-trivial. For example the mode-declarations do not allow some partially instantiated structures. As a result open-ended data structures are not possible. The prototype of ROPE in Prolog heavily depends on such partially instantiated data structures. Still, we were encouraged to make the experiment as the designers of Mercury claim there exists alternative methods to cure these problems. In the next section we report on the Prolog-implementation of ROPE. This is followed by a description of the initial, pure Mercury implementation of the finite domain solver ROPE. The implementation of the finite domain solver on Mercury will be called mrope in the rest of the paper. This first pure version of mrope did not fulfil our performance expectations and we have experimented with different data-structures. Finally we added backtrackable destructive assignment to the Mercury-system and report on this in section 4. We end with some conclusions.

2. The finite domain solver ROPE

2.1. Finite domain solvers in general

The finite domain solver ROPE [VDS94] is a system within the paradigm of constraint logic programming, an extension of logic programming. In constraint logic programming languages unification is substituted with a more general constraint solving method. This way connections other than sharing can be build between variables where the involved variables do not have to be instantiated. This allows the solver to prune the search space before choices are made. Typical for a finite domain CLP-language is that the possible instantiations of a variable is restricted to a finite set of values.

The working method of such a solver is usually by consistency techniques. Other names for these techniques are a-priori pruning and constraint propagation. This consist in keeping with each variable a set of values which are still candidates for assignment and which are still consistent with the constraints. These sets are usually called the domains. The use of a constraint consists in removing inconsistent values from these domains. A value is inconsistent if there exist a constraint (or in general a set of constraints) such that given the domains of the involved variables, assignment of this value to the corresponding variable violates the constraint (or set of constraints). The removal of inconsistent values from a domain can lead to new inconsistent values for other variables. A fixpoint algorithm like AC3 [Mac77] can be used to accomplish this task.

2.2. Features of ROPE

The finite domain solver ROPE allows the user to express a wide range of different pruning strategies for each constraint. These pruning strategies include forward checking, partial lookahead and full lookahead [Hen89], but also contain many other pruning behaviours in between forward checking and lookahead. The existence of such pruning strategies was suggested in [Nad89]. In [VDS94] these pruning strategies were presented and discussed.

Another feature of the finite domain language is a powerful enumeration primitive [Van94]: consistency techniques (removal of inconsistent values) are not enough to solve a finite domain problem. Most problems also need sophisticated enumeration. This means that a variable is chosen from the list of uninstantiated finite domain variables and that this variable is assigned a value from it's corresponding domain. If such an assignment fails, or if the algorithm searches another solution, this assignment is undone and another assignment is tried. The enumeration strategy (choice of the variable and the value from its domain) is decisive for the efficiency of the search. ROPE contains a primitive which allows to direct the enumeration with three parameters. The first parameter determines a heuristic to select a variable among a list of uninstantiated variables. The second parameter determines which value from the domain is selected for assignment, while the last parameter concerns the backtrack behaviour. This last parameter could express for example that on backtracking a new variable must be chosen for assignment, while in classical methods the same variable is assigned another value.

The language also contains a version of the cardinality constraint [HD91] and the possibility to express an optimization function.

2.3. Implementation of ROPE on top of Prolog

2.3.1. Data representation

A domain in ROPE is a finite set of natural numbers which is represented as an union of disjunct intervals. For example the set {1,2,3,8,9,12} is represented as 1..3:8..9:12..12 . The operator ../2 denotes an interval, the operator :/2 denotes the union of the two operands.

A constraint is usually transformed to a set of "in"-constraints, as done in [HSD93]. Each constraint can lead to a different set of "in"-constraints depending on the desired pruning behaviour. For example the constraint X = Y + 1 could result in the set of constraints { X in dom(Y) + 1, Y in dom(X) -1}, {X in int(Y) + 1, Y in int(X) - 1} or { ask(ground(X), Y in X - 1), ask(ground(Y), X in Y + 1) }. The functor dom/1 takes the domain of its argument. The functor int/1 results in approximating the domain of its argument to the smallest interval containing the domain. The construction ask(ground(X), Constraint) leads to the delay of the constraint until the variable X becomes instantiated. As the reader may have noticed the first set of constraints expresses lookahead. The second set of constraints leads to partial lookahead behaviour and the last set of constraints results into forward checking. Cardinality constraints and optimization functions lead to other primitives but we will not discuss these here.

Since our implementation of ROPE does not use any extension to Prolog, the domains and constraints have to be represented as ordinary Prolog data structures. Such data structures must be very easily accessible and also very easy to update. One alternative is to add to every clause an extra input and output parameter. The data-structure in these extra parameters contains information on the domains and the constraints connected to the variables. 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 where 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. 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. We have to change the domain of the variable by further instantiating it. For this an open-ended data structure is used with logarithmic access depending on the number of updates to this domain. Updating shows the same complexity behaviour. Concerning the constraints, four pieces of information are attached to a constrained variable: a list of constraints to be activated when the variable becomes instantiated, a list of constraints to be activated when the lower-bound of the domain changes, a list of constraints to be activated when the upper-bound of the variables changes and finally a list of constraints to be activated with every change to the domain of the variables. For some types of constraints, for example cardinality, we have to be able to remove constraints from these lists. For this purpose we keep a free variable with each constraint. To remove a constraint from a list, this variable is instantiated.

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.

2.3.2. Algorithm

Given these data structures, the main loop of the solver algorithm performs the following actions: (1) take a constraint from the queue, (2) use the constraint for removal of inconsistent values, (3) if the domain of a constrained variable changed, take the constraint(s) to be checked from the constraint store and add them to the queue, (4) if the queue is not empty restart with (1) otherwise stop. If a domain becomes empty in step (2) then the algorithm fails.

The solver is incremental. Whenever a new constraint is added to the system, the constraint is transformed to internal constraints. These new constraints are added to the constraint store and to the queue of the fixpoint algorithm. Subsequently the fixpoint algorithm described above is activated.

After all constraints are added to the constraint store and the queue is empty enumeration is usually started by the user program.

Enumeration consist of assigning a value to a variable and then activate the propagation mechanism explained above. When the propagation finishes the process is restarted with another variable. On failure the program returns to the most recent choice-point and takes the appropriate actions.

Besides this fixpoint and enumeration code there are some Prolog clauses that compute the intersection, union, addition, multiplication, ... of domains. Also subset properties are computed.

3. A pure implementation of ROPE in Mercury

In this first experiment we restricted the language to “in”-constraints. This implementation allowed to experiment with some small examples. From this we could draw some first conclusions.

3.1. Data-representation

Concerning the representations of the domain only some small changes were needed for the typing system:

```
:- type mrope_srange ---> ..(int, int).

:- type mrope_range
        ---> simple(mrope_srange)
        ; :(mrope_srange, mrope_range).
```

The example-set {1,2,3,8,9,12} is then represented as :(..(1,3), :(..(8,9), simple(..(12,12)))) instead of 1..3:8..9:12..12 .

Also for the "in"-constraints similar constructs were used as in ROPE. X in int(Y) + 1 is now represented as in(X, int(Y) + val(1)) . The reason for the val/1 functor, is that Mercury does not support subtypes. The type of such constraints look as follows:

```
:- type mrope_bound
        ---> val(int)
        ; lo(mrope_var)
        ; up(mrope_var)
        ; mrope_bound * mrope_bound
        ; mrope_bound / mrope_bound
        ; mrope_bound // mrope_bound
        ; mrope_bound + mrope_bound
        ; mrope_bound - mrope_bound.

:- type mrope_domain
        ---> :(mrope_domain,mrope_domain)
        ; ..(mrope_bound,mrope_bound)
        ; int(mrope_var)
        ; dom(mrope_var)
        ; val(int)
        ; compl(mrope_domain)
        ; mrope_domain * mrope_domain
        ; mrope_domain / mrope_domain
        ; mrope_domain + mrope_domain
        ; mrope_domain - mrope_domain.

% Each  constraint contains a unique number as first element.
% The maximum number used so far is stored in the mrope_system.

:- type mrope_constraint
        ---> in(int,mrope_var,mrope_domain)
        ; askgroundin(int,mrope_var,mrope_var,mrope_domain).
```

In Prolog to connect a domain to a variable we instantiated the finite domain variable to a structure with an open end. By further instantiating this structure the domain could be changed. Such a data structure is not possible in Mercury. The only possibility left is to use an extra in and out argument in the clauses of the program and most of the clauses of the solver. This arguments then contain a global data-structure with information on all finite domain variables in the user-program at hand. As argued in the previous section, updating such a data structure is usually very time consuming and dependent on the number of finite domain variables in the user-program. In case the update time is logarithmic in the number of finite domain variables, also the access to the domains becomes dependent on the number of finite domain variables. In a first attempt, this global data structure was implemented with the array-library in Mercury. The implementation of this array is based on a balanced tree. Here are the type definition of these extra arguments together with the type-definition of the finite domain variables:

```
:- import_module array.
:- import_module int.
:- import_module queue.

:- type mrope_queues --->
        mrope_queues(queue(mrope_constraint),queue(mrope_constraint),
                queue(mrope_constraint),queue(mrope_constraint)).

:- type finitevar --->
        finitevar(mrope_range,mrope_queues).

:- type variables == array(finitevar).

% mrope_system/3 takes two numbers as input, each being the highest unique
% number used so far for mrope_constraint objects and mrope_var objects.

:- type mrope_system
        ---> mrope_system(int,int,variables).

:- type mrope_var==int.
```

This subject on representation of constraint-store and changing domains is further elaborated in the subsection on results.

Also the constraints connected to a finite domain variable are represented in this global data structure. As we want the finite domain system to be incremental, also these data structures are subject to change when new constraints are added to the constraint store. One list of constraints connected with a finite domain variable is represented with the queue data type in the Mercury library. Also the queue of constraints still to be consumed in the fixpoint algorithm uses this queue data type from the library.

3.2. Primitives in this first version

Because Mercury is a moded system it is not possible to initialise a finite domain variable implicitly at its first occurrence. As a result every finite domain variable has to be initialised explicitly before use. For these purposes the following primitive can be used:

```
:- pred domain(mrope_var, mrope_domain, mrope_system, mrope_system).
:- mode domain(out,in,in,out) is semidet.
```

Then the “in”-constraints can be formulated with the following primitive:

```
:- pred in(mrope_var, mrope_domain, mrope_system, mrope_system).
:- mode in(in,in,in,out) is semidet.
```

These primitives are used in a small example:

```
% there is a field with both feazants and rabbits
% there are 9 animals and 24 legs
% how many feazants and rabbits are there ?

mrope_main -->
        % initialise the variables.
        domain( F, ..(val(0),val(100)) ),
        domain( R, ..(val(0),val(100)) ),

        % constraint : F+R=9
        in( F, val(9)-int(R) ),
        in( R, val(9)-int(F) ),

        % constraint : 2*F + 4*R = 24
        in( F, (val(24)-val(4)*int(R))/val(2) ),
        in( R, (val(24)-val(2)*int(F))/val(4) ),

        % display result
        output([F,R]).
```

In general a problem cannot be solved with constraint propagation only, so a simple enumeration primitive was added to the implementation for experimental purposes. It concerns a predicate that enumerates one variable, with a limited number of heuristics to select a value from the domain of a variable. Also a limited number of backtrack behaviours can be specified

```
:- type selectvalue ---> up ; down ; middle ; split.
:- type backtrackmethod ---> standard ; standard_ex.

:- pred enum(mrope_var, selectvalue, backtrackmethod, mrope_system, mrope_system).
:- mode enum(in,in,in,in,out) is nondet.
```

3.3. Results

A first observation concerns the order in which solutions are generated. As it happens, the Mercury version of ROPE generates solutions in the opposite order as the Prolog version. Since Mercury is intended as a pure logic language and logic makes abstraction of the order in which solutions are found, Mercury does not care which solutions are generated first. We do, especially in the enumeration phase.

The first results show that this first pure implementation in Mercury is slower than the implementation on top of Prolog using proLog by BIM. Two versions were tried, one with garbage collection and one without. Mercury uses garbage collection at the level of C [BW88]. Timing is given for finding all solutions.

test	mrope	ROPE
queens(10), with gc	95s	39s
queens(10), without gc	47s	

The main difference between the Prolog version and the Mercury version is the different data structure for the domains of the variables and the constraint store. As Mercury is known to be faster than Prolog, for Mercury programs that Prolog can execute, the reason for this slow-down must be because of this different data structure. First we wanted to measure the speedup we got in the parts of the program where the code did not change. A small program was written which computed intersections of random generated domains. Both programs were run on Mercury and ProLog by BIM on a Sparccenter-1000. About 40.000 intersections were computed.

	proLog by BIM	Mercury
40.000 intersections	3.7s	0.36s

This 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.

In another experiment we wanted to measure the results of changes to the representation of the data in the extra input and output arguments. The original data-structure was an array from the Mercury library.

```
:- import_module array.
:- type variables == array(finitevar).
:- type mrope_system
        ---> mrope_system(int,int,variables).
```

Every element of this array corresponds with information on a finite domain variable. This array is implemented as a 234-tree. For testing purposes a simplified special purpose balanced tree was written.

```
:- type mybintree(V)    --->    empty
                     ; tree( int, V, mybintree(V), mybintree(V)).

:- type variables == mybintree(finitevar).
:- type mrope_system
        ---> mrope_system(int,int,variables).
```

The last representation concerns the functor. In the last case every argument of the functor is information on one variable. If the information of a variable changes a new functor is created where all arguments but one, the one that changed, are copied. Then the changed information is put in the corresponding argument.

```
:- type variables --->
    array(finitevar,finitevar,finitevar,finitevar,finitevar,
          finitevar,finitevar,finitevar,finitevar,finitevar,
          finitevar,finitevar,finitevar,finitevar,finitevar,
          finitevar,finitevar,finitevar,finitevar,finitevar).
:- type mrope_system
      ---> mrope_system(int,int,variables).
```

This test was again on the queens(10) program. The Mercury program was compiled without garbage collection. For the first time we got timings better than the version in Prolog.

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

From this we can see clearly that a different data structure has a large effect on the efficiency of the program.

In another version of the Mercury program checking for duplicates in the constraint-queue was added. This was implemented with the help of the same binary tree used in the previous experiment. Also another example was added to the test-examples. It concerns the bridge problem described in [Hen89].

	mrope	ROPE
queens(10)	65s	39s
bridge1(200)	5s	27s

With the checking for duplicates the queens program is slowed down again, while with the new example we get an interesting speedup. So, it is possible that some examples run 5 times faster in this pure implementation of Mercury. As we will see in the next section better results can be obtained when using backtrackable destructive assignment.

4. ROPE in Mercury with backtrackable destructive assignment

The previous section reports on the results of a straightforward port from ROPE to Mercury : the results were not so positive for Mercury. The main reason for this seems the absence of appropriate data representations because there is no good alternative to an open-ended data structure in pure Mercury. This section describes the addition of backtrackable destructive assignment to Mercury, and the effect on the efficiency of our constraint solver. In the pure version of the previous section only "in"-constraints were implemented. It was also very difficult to implement for example the cardinality constraint [HD91]. Such a constraint causes the need to remove constraints from the dependency lists connected to a variable. In Prolog this is realized by including a free variable with each constraint. To remove a constraint from a list this variable is instantiated. This is not possible in Mercury. We did not see any good and efficient alternative in Mercury than backtrackable destructive assignment to implement this feature.

4.1. Preview on the gains of backtrackable destructive assignment

Backtrackable destructive assignment is currently not part of the Mercury system. In version v0.5 the modes *unique* and *almost unique* were implemented, but this can only be used at the top-node of a data structure. Allthough the modes are there, destructive assignment is not yet there, so we had to do it ourself. Before jumping into the experiment we wanted to have an idea of the efficiency gain we could get. For this purpose we used SICStus Prolog v2.1 which contains a variant of backtrackable destructive assignment in the from of the builtin predicate setarg/3. The Mercury code for mrope was with some minor changes ported to SICStus v2.1 Prolog. We then again ran the queens(10) problem using different data-representations for the constraint-store and the domains of the variables. Still all variants use two extra arguments, an in and out argument. But in case setarg is used, the in and out arguments are always unified and the update is done locally. We used the same two data structures as before: a functor and a binary tree. Also in both representations a number of artificial variables were introduced. This should indicate how speed scales when larger examples are tackled.

Queens(10) problem in SICStus Prolog	normal	with setarg
functor/20	120s	94s
functor/100	1000s	94s
mybintree	152s	120s
mybintree with more variables	247s	163s

We concluded that it was worthwhile to try to implement a variant of setarg in Mercury. As the aim is to store the information on a finite domain variable in the variable itself we can assume constant access and update. This was the case in the 94s timing.

4.2. Implementation issues

To insert the code in C that implements backtrackable destructive assignment we introduced a dummy module which contains the interface for the code. The C-file generated by Mercury is then replaced with a file which contains the actual code for the backtrackable destructive assignment.

```
% dummy module to generate template for support routines
% to allow backtrackable destructive update in Mercury

:- module mrope_du.

:- interface.

:- import_module std_util.

% announce a backtrack-point
% places a marker on the trail
:- pred du__backtrackpoint_anounce is det.

% untrail all the changes after the last backtrack-point
:- pred du__backtrack is det.

% remove a backtrack-point - will remove marker from trail
% du__backtrack has to be called beforehand to make
% sure that the marker is the last element on the trail
:- pred du__backtrackpoint_remove is det.

% set argument (1), of predicate (2) to given value (3)
% this can be restored by du__backtrack when needed
:- pred du__setarg(int::in, TP::in, TV::in) is det.

:- implementation.

du__backtrackpoint_anounce.
du__backtrackpoint_remove.
```

```
du__backtrack.
du__setarg(_,_,_).
```

4.3. Results

Given backtrackable destructive assignment also the cardinality constraint [HD91] was added to the system. With the additional cardinality constraints a wider range of examples was possible. A new version of the bridge problem was added. In this version all disjunctions are added to the system before any choices are made. The old version of the bridge problem is called “bridge1”, the new version “bridge2”. A variant of the perfect square [AB92] was added with small dimensions. Also a Japanese puzzle called suudoku.

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 (ProLog by 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	

This table 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 optimization: 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 we should not take into account the queens results; then we can deduce a speed-up ranging from a factor 5 to almost 15 from the table compared to ROPE in Prolog.

5. Conclusions and Future work

This work reports on the use of Mercury for the implementation of a finite domain solver. This implementation is based on an implementation of such a solver in Prolog. Not all features of the original finite domain solver were implemented. For example transformation of high-level constraints to low-level constraints was not included in the new system. This resulted in the absence of some optimisations which are based on the high level constraints. This explains the bad results for queens. For a usable finite domain solving package these missing parts are essential but they will not change much the obtained results and conclusions.

The main result of the experiment concerns data representation: our experiment suggests at least that a straightforward port from Prolog code to Mercury code, will not always lead to a high performance gain. The use of alternative data structures can help, but it seems paramount that because of the lack of open ended data structures, data structures can be locally updated without being completely copied. This seems to be the intention of the Mercury team, with the introduction of support for the modes *unique* and *mostly unique* in release 0.5 (we started this work at the time of release 0.4). However, currently this information is checked, but not yet used and the occurrence of these modes is still restricted to the top-level of data structures. It is not clear whether the intended support is enough for our purposes neither whether in general automatic support is possible within the context of our needs.

References

[AB92] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *JFPL*, pages 51–66, 1992.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[HD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *proceedings of ICLP*, 1991.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989.

[HSD93] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(fd). Technical report, Brown University, 1993.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Nad89] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5(4):188–224, November 1989.

[SHC94] Zoltan Somogy, Fergus Henderson, and Thomas Conway. The implemenatation of mercury: an efficient declarative logic programming language. In *Proceedings of the ILPS'p4 Post-conference Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

[Van94] Henk Vandecasteele. On backtracking in finite domain problems. In *Proceedings of the Fifth Benelux Workshop on Logic Programming*, September 1994.

[VDS94] Henk Vandecasteele and Danny 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.