A High-Level Intermediate Language and the Algorithms for Compiling Finite-Domain Constraints[1]

Neng-Fa Zhou
Faculty of Computer Science and Systems Engineering
Kyushu Institute of Technology
680-4 Kawazu, Iizuka, Fukuoka 820, Japan
zhou@mse.kyutech.ac.jp
http://www.cad.mse.kyutech.ac.jp/people/zhou

Abstract

In this paper, we present a high-level intermediate language, called *delay-Prolog*, for implementing CLP(FD), the constraint language over finite-domains. Delay-Prolog has a much stronger description power than *indexical*, the intermediate language widely used in current finite-domain constraint compilers. It is powerful enough for describing all kinds of constraint propagation procedures, which by now have been mostly written in low level languages. And, most importantly, delay-Prolog opens new ways to compiling constraints. Besides the *indexical* algorithm that compiles constraints into indexicals, we implemented two new algorithms: the *wait* algorithm that compiles a constraint as whole into a propagation procedure without splitting it into small pieces, and the *incremental simplification* algorithm that incrementally simplifies constraints into smaller ones. The results are noteworthy: The wait algorithm generates the fastest code, while the widely used indexical algorithm generates code that is about 3 times slower than that generated by the wait algorithm. The wait algorithm is adopted in the latest version of B-Prolog. The experimental results show that B-Prolog is comparable in performance with the fastest CLP(FD) systems, although it is an emulated system.

1 Introduction

CLP(FD) [7][2], the constraint logic programming language over finite domains, has been proved effective for solving a large number of real-life optimization problems. The key operation employed in CLP(FD) is called *constraint propagation*, which uses constraints actively to prune search spaces as follows: whenever a variable changes, i.e., the variable has been instantiated or its domain has been updated, the domains of all the remaining variables are filtered to contain only those values that are consistent with this variable. Like compiling unification is important for improving the performance of Prolog, compiling constraint propagation is a key to improving the performance of CLP(FD).

In early CLP(FD) systems, such as in the CHIP system [5], constraints are interpreted rather than compiled: constraints are first transformed into canonical-form

[1] This research was supported by a grant (No.09780298) from the Ministry of Education, Science, and Technology of Japan.

[2] In this paper, we use `CLP(FD)` as the name for the class of constraint languages over finite domains and `clp(FD)` to refer to the system developed at INRIA.

terms and are then executed by an interpreter that performs, among other things, constraint propagation. The propagation procedure adopted is general enough for handling all types of constraints. Learning from the experience of compiling Prolog programs into the Warren Abstract Machine (WAM), several researchers have extended the WAM for compiling CLP(FD). In the CHIP compiler [1], constraints are compiled into low level instructions such that different specialized propagation procedures can be used for different types of constraints. This compiler is called a *CISC* and a *black-box* compiler because it is based on a complex abstract machine and the constraint solver is not open to the users.

Recently, an intermediate language, called *indexical*, has been widely used for compiling finite-domain constraints [8, 2]. An indexical is a primitive constraint in the form of X *in* r, where X is a domain variable and r specifies the range for X. For each indexical, a propagation procedure specific to it is adopted. In contrast to the CHIP compiler, these compilers are called *RISC* and *glass-box* compilers because the underlying abstract machines are much simpler than that adopted in the CHIP compiler and the intermediate language is open to the users. However, indexical is very limited in description power because the delay mechanism is embedded in the language. It can be used to compile arithmetic as well as Boolean constraints[3], but is too week to be used to implement other types of constraints. In addition, the compilation from constraints into indexicals is so straightforward that it is impossible to try some new compilation algorithms and optimization techniques.

In this paper, we present a high-level intermediate language, called *delay-Prolog*, for implementing finite-domain constraints. Delay-Prolog extends Prolog in that it supports *domain variables* and a delay mechanism, called *delay clause*, that has the functionality of describing delay conditions on goals, triggering times for delayed goals, and actions taken after goals are delayed. Delay-Prolog will be defined in Section 3.

Delay-Prolog is a nice implementation language for programming constraint propagation. We will show in Section 4 how to program constraint propagation for various constraints including *reification* and the global constraint `all_distinct(L)`. Delay-Prolog has a much stronger description power than indexical, and opens new ways to compiling constraints. Besides the *indexical* algorithm, we will describe in Section 5 two new algorithms: the *wait* algorithm that compiles each constraint as a whole into a propagation procedure without splitting it into small pieces, and the *incremental simplification* algorithm that incrementally simplifies constraints into smaller ones. We will also analyze these algorithms and clarify their advantages and disadvantages.

We have implemented all the algorithms in B-Prolog and compared their performance for several benchmark programs. The results, which will be given in Section 6, are noteworthy: The wait algorithm is the best algorithm which generates code that is 3 times faster than that generated by the indexical algorithm. The wait algorithm is used in the latest version of B-Prolog (Version 3.0). The experimental results also show that B-Prolog is comparable in performance with, and sometimes faster than SICStus-Prolog and clp(FD), although it is an emulated system.

The advantages of delay-Prolog have not yet been explored thoroughly. In Section 7, we will propose two important optimization techniques.

The reader is assumed to be familiar with logic programming and constraint satisfaction, but no knowledge about abstract machines is assumed. In Section 2, we define some preliminary terms about CLP(FD) and constraint propagation. For detailed description, the reader may consult [7] and [9].

2 Preliminaries

2.1 CLP(FD)

In general, a CLP(FD) program is composed of three parts: the first part, called *variable generation*, generates variables and specifies their domains; the second part, called *constraint generation*, specifies constraints over the variables; and the final part, called *labeling*, instantiates the variables by doing enumeration.

The domain of a variable is defined as follows:

```
Vars in D
```

where `Vars` is a variable or a list of variables, and `D` is a list of integers or a range between two integers $l..u$ which denotes the set of integers $\{l, l+1, \ldots, u\}$.

A basic constraint is a call in the form $E_1\ R\ E_2$ where E_1 and E_2 are two arithmetic expressions and R is a relation symbol in {`#=`, `#\=`, `#>`, `#>=`, `#<`, `#=<`} which is self-explainable. There are several other primitives available for describing other types of constraints, choosing variables, instantiating variables, and finding optimal solutions with respect to some criteria.

2.2 Constraint propagation

Constraint Propagation (CP) is a basic operation employed in CLP(FD) systems for solving constraints. It uses constraints actively to exclude no-good values from the domains of variables. Many algorithms can be derived from this idea depending on when and to what level CP is applied. The following table illustrates how domains in `X#=Y+1` ($X, Y \in 1..5$) are updated by four different algorithms. The *consistency checking* algorithm performs CP when constraints are generated. The *forward checking* algorithm, besides doing consistency checking, also invokes CP when a domain variable is instantiated. The *partial look-ahead* algorithm, besides doing forward checking, also invokes CP when a bound of a domain is updated. The *full look-ahead* algorithm, besides doing partial look-ahead, also invokes CP when an intermediate element is excluded from a domain.

Algorithm	Time	Effect
Consistency checking	generated	$X \in 2..5, Y \in 1..4$
Forward checking	$X = 3$	$X = 3, Y = 2$
Partial look-ahead	$X \neq 5$	$X \in 2..4, Y \in 1..3$
Full look-ahead	$X \neq 4$	$X \in [2,3,5], Y \in [1,2,4]$

3 Delay-Prolog

3.1 Domain variables

A domain variable is represented internally just like a suspending variable of Prolog, but there is more information associated with it (see the table below).

type	type of the domain
min	minimum element
max	maximum element
size	number of elements in the domain
ins_cs	list of constraints to be executed when the variable is instantiated
min_cs	list of constraints to be executed when the low bound is updated
max_cs	list of constraints to be executed when the upper bound is updated
dom_cs	list of constraints to be executed when any element is excluded
elms	data structure that stores the elements

The information about a domain variable can be accessed and/or updated through a group of primitives. The expression `min(X)` denotes the minimum element, `max(X)` the maximum element, `dom(X)` the list of elements, and `size(X)` the size of the domain of `X`. The primitive `exclude(X,E)` excludes the element `E` from the domain of `X`, and the primitive `elm(X,E)` checks whether `E` is an element in the domain of `X`.

A failure occurs when the domain of a variable becomes empty. When the domain of a variable becomes a singleton, then the variable will be bound to the element.

An update of a domain variable is called a *trigger* and the variable is called a *triggering variable*. There are four kinds of triggers. The trigger *ins(X)* is on when `X` is instantiated; *min(X)* is on when the low bound of `X` is updated; `max(X)` is on when the upper bound of `X` is updated; and `dom(X)` is on when any change happens to `X`. There are four lists of constraints associated with a domain variable, each of which stores the constraints to be executed when the corresponding trigger is turned on. Notice that the triggers are not completely independent each other. If the trigger `ins(X)` is on, all the other three triggers will be on too. If either `min(X)` or `max(X)` is on, then `dom(X)` will be on too.

3.2 Delay clauses

In this subsection, we describe the syntax and semantics of delay clauses. Examples that illustrate their usage will be given in the next Section.

A delay clause takes the following form:

```
delay Head :- Condition : [{Triggers}] Action.
```

where `Condition` is a sequence of in-line tests, `Triggers` is a sequence of trigger declarations, and `Action` is a sequence of arbitrary calls.

For any call to the predicate of `Head`, if it matches `Head` (*i.e.,* the call is an instance of `Head`) and `Condition` is satisfied, then the call delays and `Action` is executed. Since one-directional matching rather than full-unification is used to match the call against `Head` and `Condition` is composed of only in-line tests, the call will be kept unchanged before `Action` is executed. If the call does not match `Head` or `Condition` fails, then the next clause will be tried. If `Action` fails, then the original predicate call will fail.

The sequence of trigger declarations `Triggers`, which is optional, declares when the clause should be re-executed. The trigger declaration `ins(X)` means the time when the variable `X` is instantiated; `min(X)` (`max(X)`) means the time when the the low (upper) bound of `X` is updated; and `dom(X)` means the time when any change happens to `X`. When no trigger declaration is given, the compiler automatically generates a trigger `ins(X)` for each variable occurring in `Condition`.

A delay clause is executed in a event-driven manner. At the entry and exit points of every predicate, the system checks to see whether or not there is trigger that has been turned on. If so, then the current procedure is interrupted and control is moved to the lists of constraints associated with the trigger. When a delay clause is re-entered, the execution starts from different points depending on the type of the trigger. If the trigger is `ins(X)`, then the execution starts from the condition; otherwise, the execution starts from the action. This optimization technique avoids testing of the condition redundantly because the triggers `min(X)`, `max(X)`, and `dom(X)` do no affect the satisfiability of the condition. After the constraints finish their execution, the interrupted procedure will resume its execution. The implementation tails can be found in [13].

Delay clauses defined here are much more powerful than other delay constructs available in different Prolog systems. It extends delay clauses proposed by Meier [10] which are unable to describe triggers and actions. All other constructs, such as *freeze* in Prolog-II and *when declarations* in NU-Prolog, can be implemented very easily by using delay clauses.

4 Programming Constraint Propagation

In this section, we show four examples that illustrate how to programming constraint propagation in delay-Prolog.

4.1 An FD-constraint interpreter

After we have delay clauses, it becomes a trivial work for us to write a constraint interpreter. The following shows such an interpreter:

```
delay interp_constr(C):-no_dvars_gt(C,0) :
   {min(C),max(C)}, reduce_domains(C).
interp_constr(C):-true : test_constr(C).
```

The condition `no_dvars_gt(T,N)` succeeds if there are more than `N` domain variables in the term `T`. For a constraint `C`, if there are at least one variable in $\vec{C}$, the interpreter delays the constraint and invokes the procedure `reduce_domains` which excludes no-good values from the variables in `C` to ensure the consistency of the constraint. The two triggers, `min(C)` and `max(C)`, ensure that the constraint will be reconsidered whenever either bound of any variable in `C` is updated.

4.2 Indexical

Indexical, which is adopted by many CLP(FD) compilers for compiling constraints, can be implemented easily with delay-Prolog. Consider the indexical

```
X in min(Y)+min(Z)..max(Y)+max(Z).
```

This indexical is a primitive constraint that excludes no-good values from the domain of `X` that do not lie in the range of `min(Y)+min(Z)..max(Y)+max(Z)`. This constraint is reconsidered whenever a bound of `Y` or `Z` is updated. The following procedure implements the indexical:

```
delay 'V in V+V'(X,Y,Z):-dvar(X) :
      {min(Y),max(Y),min(Z),max(Z)},
      X in min(Y)+min(Z)..max(Y)+max(Z).
'V in V+V'(X,Y,Z):-true : X >= min(Y)+min(Z), X=< max(Y)+max(Z).
```

The predicate call `'V in V+V'(X,Y,Z)` propagates changes from `Y` and/or `Z` to `X`. This implementation is inefficient because the action taken is the same regardless of the instantiation states of `Y` and `Z`.

To improve the efficiency, we rewrite the first clause into four clauses, each one taking care of one instantiation state of the constraint:

```
delay 'V in V+V'(X,Y,Z):-
      dvar(X),dvar(Y),dvar(Z) :
      {min(Y),max(Y),min(Z),max(Z)},
      X in min(Y)+min(Z)..max(Y)+max(Z).
delay 'V in V+V'(X,Y,Z):-
      dvar(X),dvar(Y):
      {min(Y),max(Y)},
      X in min(Y)+Z..max(Y)+Z.
delay 'V in V+V'(X,Y,Z):-
      dvar(X),dvar(Z):
      {min(Z),max(Z)},
      X in Y+min(Z)..Y+max(Z).
'V in V+V'(X,Y,Z):-
      dvar(X):
      X in Y+Z.
```

We can do the same thing to the second clause in the previous definition of 'V in V+V', but cannot expect as much gain in efficiency because non-delay clauses are executed only once while delay clauses may be executed many times.

4.3 Reification

One well used technique in FD constraint programming is called *reification*, which uses a new Boolean variable `B` to indicate the satisfiability of a constraint `C`. `C` must be satisfied if and only if `B` is equal to `1`. This relationship is denoted as:

```
C <=> (B #= 1)
```

Reification is useful for implementing various global constraints, such as cardinality constraints [8].

It is straightforward to implement reification in delay-Prolog. Consider, as an example, the reification:

```
(X #= Y) <=> (B #= 1)
```

where `X` and `Y` are domain variables, and `B` is a Boolean variable. The following procedure describes the relationship:

```
delay reification(X,Y,B):-
      dvar(B),dvar(X),X\==Y : {ins(X),ins(Y),ins(B)}.
delay reification(X,Y,B):-
      dvar(B),dvar(Y),X\==Y : {ins(Y),ins(B)}.
reification(X,Y,B):-dvar(B) : (X==Y -> B=1; B=0).
reification(X,Y,B):-true : (B==0 -> X #\= Y; X #= Y).
```

Curious readers might have noticed that `ins(Y)` is a trigger in the first clause but `ins(X)` is not a trigger in the second clause. The reason is simply that `ins(Y)` affects the condition of the first clause but `ins(X)` has no effect on the condition

of the second clause. When `Y` is bound after the first clause is executed, it may be bound to be the same variable as `X`. In this case, the test `X\==Y` in the condition will fail. In contrast, `X` is guaranteed to be an integer after the constraint is delayed by the second clause. Therefore, declaring `ins(X)` as a trigger is a nonsense.

4.4 all_distinct(L)

Besides arithmetic constraints, a constraint system usually offers the functionality for describing and solving symbolic constraints. Delay-Prolog is also a nice language for implementing symbolic constraints. We consider, as an example, how to implement `all_distinct(L)`.

The constraint `all_distinct(L)` holds if variables in `L` are pair-wisely different. One naive way of implementing this constraint is to generate binary inequality constraints between all pairs of variables in `L`. This implementation has two problems: First, the space required to store the constraint is quadratic in the number of variables in `L`; Second, splitting the constraint into small granularity ones may lose possible propagation opportunities. The second problem has been pointed out before by other researchers (e.g., [11], [12]).

To solve the space problem, we define `all_distinct(L)` in the following way:

```
all_distinct(L):-all_distinct(L,[]).

all_distinct([],Left).
all_distinct([X|Right],Left):-
      outof(X,Left,Right),
      all_distinct(Right,[X|Left]).

delay outof(X,Left,Right):- dvar(X) : {ins(X)}.
outof(X,Left,Right):-true : outof(X,Left),outof(X,Right).
```

For each variable `X`, let `Left` be the list of variables to the left of `X` and `Right` be the list of variables to the right of `X`. The predicate call `outof(X,Left,Right)` holds if `X` appears in neither `Left` nor `Right`. Instead of generating inequality constraints between `X` and all the variables in `Left` and `Right`, the call `outof(X,Left,Right)` delays until `X` is instantiated. After `X` becomes an integer, the calls `outof(X,Left)` and `outof(X,Right)` exclude `X` from the domains of the variables in `Left` and `Right`. It is not difficult to understand that this implementation only consumes linear space.

In terms of the propagation ability, the second implementation is the same as the first one. In some systems, consistency checks are done to detect failure as early as possible. It is very easy to introduce consistency checks into the implementation. To do so, we only need to define `outof(X,Left,Right)` as follows:

```
delay outof(X,Left,Right):- dvar(X) : {dom(X)},
      consistency_check(X,Left,Right).
outof(X,Left,Right):-true : outof(X,Left),outof(X,Right).
```

where `consistency_check(X,Left,Right)` does the consistency check. There are many possible algorithms. The one implemented in B-Prolog is as follows: Let n be the size of the domain of `X`, and m be the number of variables in `Left` and `Right` whose domains are subsets of that of `X`. If $m+1 > n$, then fails; otherwise, if $m+1 = n$, then for each value v in `X`'s domain, exclude v from the domains of all the variables whose domains are not subsets of that of `X`. The soundness of the

algorithm is obvious: If $m+1 > n$, then it is impossible to assign n different values to $m+1$ variables, and if $m+1 = n$, then no value in the domain of `X` can be assigned to other variables except `X` and the m variables.

5 Compiling Arithmetic Constraints

In this section, we show that delay-Prolog can serve as an excellent intermediate language for compiling constraints. For each constraint, the compiler generates a constraint propagator tailored to the source constraint. We only consider linear arithmetic constraints.

Definition 1 (Canonical-form) *A constraint E_1 R E_2 is said to be in canonical-form if E_1 and E_2 are expressions in the following form: $T_1+T_2+\ldots+T_n$ where $T_i(i = 1,\ldots,n)$ is either a domain variable or a domain variable proceeded by a positive coefficient.*

For each constraint, the compiler first translates it into a canonical-form and then generates a constraint propagator for it. It is usually impossible to know the run-time properties of the variables in an expression at compilation time. A variable may be a constant, a domain variable, or even a complex expression at execution time. To invoke the propagator, the variables must be first checked at runtime. If they are domain variables or integers, then execute the specialized propagator; otherwise, call the constraint interpreter.

5.1 Compiling low-degree constraints

We first consider how to compile constraints that contains three or less variables. As *unary* constraints, i.e., constraints containing only one variable, can be translated into appropriate primitives on domain variables, we only need to consider constraints that contains two or three variables.

5.1.1 Compiling constraints into indexicals

We have shown in the previous section how to implement indexical. With indexical, we can compile constraints in the same way as many other compilers do. Consider, for example, the constraint `X #= Y+Z`. This constraint is replaced by the following three indexicals:

```
'V in V+V'(X,Y,Z),
'V in V-V'(Y,X,Z),
'V in V-V'(Z,X,Y).
```

This algorithm has both advantages and disadvantages. On one hand, indexicals themselves are specialized code that contains no redundancy. For example, when a bound of `X` is updated, the indexicals `'V in V-V'(Y,X,Z)` and `'V in V-V'(Z,X,Y)` will be triggered but the indexical `'V in V+V'(X,Y,Z)` will not be triggered. On the other hand, the granularity of indexicals is very small and the cost for executing them is usually high because space is required for storing the constraints and the execution context, i.e., some of the registers of the abstract machine, has to be switched frequently.

5.1.2 Compiling constraints into delay predicates

Another algorithm is to compile each constraint into one delay predicate without splitting it into smaller pieces. The delay predicate contains clauses for different modes of the constraints, starting from the mode where all the variables are free to the mode where all the variables are instantiated. The delay predicates for *binary constraints* (i.e., constraints having two variables) look like:

```
delay p(X,Y,C):-dvar(X),dvar(Y) : {Triggers}, Actions.
p(X,Y,C):-dvar(X) : solve the unary constraint over X.
p(X,Y,C):-dvar(Y) : solve the unary constraint over Y.
p(X,Y,C):-true : test the ground constraint.
```

The contents of `Triggers` and `Actions` are determined based on the type of the constraint.

For example, the constraint `X #= Y` is compiled into the following predicate:

```
delay neq(X,Y):-dvar(X),dvar(Y): {ins(X),ins(Y)}.
neq(X,Y):-dvar(X) : exclude(X,Y).
neq(X,Y):-dvar(Y) : exclude(Y,X).
neq(X,Y):-true : X=\=Y.
```

No action will be taken when both `X` and `Y` are uninstantiated. As another example, consider the equality constraint `X #= Y+C`. The compiled predicate is as follows:

```
delay 'X=Y+C'(X,Y,C):-dvar(X),dvar(Y) :
      {min(X),max(X),min(Y),max(Y)},
      X in min(Y)+C..max(Y)+C.
      Y in min(X)-C..max(X)-C.
'X=Y+C'(X,Y,C):-dvar(X) : X is  Y+C.
'X=Y+C'(X,Y,C):-true : Y is X-C.
```

This predicate is a little different from the skeleton we mentioned above: The last clause takes care of both of the cases when `Y` is a variable and when `Y` is an integer.

Compared with the scheme of compiling constraints into indexicals, this scheme does not have the small granularity problem. One constraint is compiled into only one predicate and thus only one frame is enough for storing the constraint. On the other hand, the code for reducing domains is not as specialized as in indexicals. Executing `X in min(Y)+C..max(Y)+C` is redundant if the constraint is triggered by changes to `X`. It is difficult to judge which scheme is better than the other without doing experimental comparison. In Section 6, we will give the experimental results.

5.2 Compiling high-degree constraints

There are many ways for compiling n-ary constraints where n is greater than 3. The first algorithm is to compile all constraints in the same way as that for compiling low-degree constraints: generate a clause for each mode of a constraint. We call this the *explosive examination* algorithm. This algorithm has the advantage that specific code for reducing domains can be generated for different modes of a constraint. However, for an n-ary constraint, there are 2^n clauses in the generated predicate. This algorithm is obviously not practical for compiling constraints with large n.

The second algorithm is to split an n-nary constraint into a sequence of three-tuple constraints each of which contains at most three variables. We call this algorithm *split* algorithm. This algorithm is adopted by many compilers, such as

SICStus-Prolog and clp(FD). It does not have the size-explosion problem of the explosive examination algorithm and can take advantage of the constraint library which is usually implemented in low-level languages and are well tuned. In addition, this algorithm, as described in [2], can restrict value propagation within small constraints as long as the shared variables do not change. The disadvantages of this algorithm are that new domain variables have to be introduced and the granularity of constraints becomes smaller.

The third algorithm is to incrementally simplify n-ary constraints to lower degree ones. The generated predicate for an n-ary constraint looks like:

```
delay c(X1,X2,...,Xn,I):-
    dvar(X1),...,dvar(Xn) :
    {Triggers}
    reduce domains of X1,...,Xn.
c(X1,X2,...,Xn,I):-
    integer(X1) :
    I1 is a new constant,
    c1(X2,...,Xn,I1).
    ...
c(X1,X2,...,Xn,I):-
    integer(Xn) :
    In is a new constant,
    cn(X1,...,Xn-1,In).
```

When the reduced constraints c_1, c_2, ..., and c_n have the same structure, then only one predicate is adequate to define the propagation procedure for them. The simplification process is repeated until the simplified constraints have no more than three variables. We call this algorithm *incremental simplification* algorithm. This algorithm has the advantage that specific code is used to reduce domains. The disadvantages are that: there are a lot of argument passing involved in the execution, and the compiled code takes at least $O(n^3)$ space and thus cannot be applied to large constraints.

The fourth algorithm is to compile an n-ary constraint into:

```
delay c(X1,X2,...,Xn):-
    (dvar(X1);dvar(X2);...;dvar(Xn)) :
    {Triggers},
    reduce domains of X1,...,Xn.
c(X1,X2,...,Xn):-true : do the test.
```

The constraint is delayed until it becomes ground. We call this *wait* algorithm. This algorithm is simple and generates linear-size code. However, the code for reducing domains does not take the modes of constraints into account and is thus inefficient.

In the current implementation of delaying, no constraint can be delayed after it becomes ground. In other words, ground constraints will possibly become garbage to be collected by the garbage collector. By taking advantage of this feature, we can simplify the predicate to:

```
delay c(X1,X2,...,Xn):-
    true : {Triggers},
    reduce domains of X1,...,Xn.
```

It does not test the instantiation state of the constraint, but relies on the system to do the test.

Table 1: Comparison of four compilation algorithms (SPARC-2, milliseconds)

Program	Wait	Split1	Split2	Simplify
magic4	33 (1)	83 (2.52)	66 (2.00)	33 (1.00)
eq10	100 (1)	700 (7.00)	665 (6.65)	116 (1.16)
eq20	256 (1)	1466 (5.73)	1950 (7.61)	333 (1.30)
crypta	83 (1)	300 (3.61)	300 (3.61)	83 (1.00)
alpha-ff	167 (1)	400 (2.40)	483 (2.89)	200 (1.20)

.

6 Experimental Results

All the algorithms described in this paper have been implemented in B-Prolog. In this section, we first compare the performance of the four compilation algorithms and then compare the best one with other three CLP(FD) systems.

6.1 Comparing different compilation algorithms

Table 1 compares the execution time (on a SPARC-2) of the code generated by the four compilation algorithms: `Wait` is the algorithm that delays constraints until they become ground; `Split` is the algorithm that splits n-ary constraints into three-tuple ones; and `Simplify` is the algorithm that incrementally simplifies n-ary constraints into lower degree ones. For `Split`, the two algorithms for compiling small constraints are tested: `Split1` compiles constraints into indexicals and `Split2` compiles constraints into delay predicates. Each program was run five times and the minimal time was taken. There is no big difference between `Split1` and `Split2`. Based on the results, we can rank the algorithms as follows:

$$\text{Wait} > \text{Simplify} > \text{Split}$$

The `Wait` algorithm is the best. The generated code by `Wait` is 2 to 7 times as fast as that generated by the `Split` algorithm.

The cost of executing a constraint program comes mainly from three sources: delay-condition tests, domain reduction, and context switching. The `Wait` algorithm is the best in terms of the costs of delay-condition tests and context switching, but the worst in terms of the cost for domain reduction. The `Split` algorithm is the best in terms of the cost for reducing domains because value propagation can be restricted to small constraints as long as the temporary variables that connect them with others do not change. Nevertheless, the `Split` algorithm is the worst in terms of the cost for context switching. For an n-ary constraint, a change to one variable in it may trigger as many as $n-1$ three-tuple constraints in the worst case. The `Simplify` algorithm is better than `Wait` in terms of the cost for reducing domains, but has to pay the cost for passing arguments.

In B-Prolog, all propagation procedures are encoded in Prolog which are compiled to byte code that is interpreted by an emulator. In many systems, propagation procedures for small granularity constraints are written directly in C and are well tuned. If small constraints are handled in the same way, then the results for the `Split` algorithm will certainly become better. However, the same thing can be done to the `Wait` and `Simplify` algorithms. We have noticed that the code for reducing domains involves a lot of arithmetic computations. If it is compiled into C, then a

Table 2: Comparison of four systems (SPARC-2)

Program	BP	SICS	clp(FD)	Eclipse
magic4	33 (1)	50 (1.52)	45 (1.36)	233 (7.06)
eq-10	100 (1)	120 (1.80)	80 (0.80)	416 (4.16)
eq-20	256 (1)	200 (0.78)	120 (0.50)	867 (3.39)
crypta	83 (1)	110 (1.33)	80 (0.96)	366 (4.41)
alpha	24,550 (1)	34,600 (1.41)	8,045 (0.33)	201,433 (8.21)
queens-16	914 (1)	2,290 (2.50)	1,553 (1.70)	12,084 (13.22)

great speed-up is expectable because the overhead of interpreting byte code is gone and the data involved in the computation can be untagged.

Through this experiment, we found that the `Wait` algorithm should be chosen. As a side result, we found that compiling constraints into small granularity ones is not a good idea.

6.2 Comparing B-Prolog with three other constraint systems

Table 2 compares the CPU times required to execute the benchmark programs by four systems: B-Prolog[14] version 3.0 (BP), SICStus-Prolog[11] version 3.0 #6 (SICS), clp(FD)[4] version 2.21, and Eclipse [6] version 3.5.2. For all the programs, variables are instantiated in the given order from left to right.

The constraints in `queens-16` are compiled by hand into efficient forms for running in `BP`, `SICS`, and `clp(FD)`. Therefore, this comparison is a little unfare to `Eclipse`. On average, `clp(FD)` is the fastest and `Eclipse` is the slowest system. `BP` and `SICS` lie in between, but `BP` is a little faster than `SICS`.

The systems compared are very different. `clp(FD)` compiles programs into C, `SICS` compiles programs into native code, and `BP` and `Eclipse` are emulated. In addition, `Eclipse` uses a general data structure, called meta-terms, to represent domain variables, while the other three systems use a special data structure for representing domain variables.

The results are very encouraging for us. Except for `eq-20` and `alpha`, `BP` has a comparable performance with `clp(FD)`, the fastest finite-domain constraint solver available now. If constraint propagation procedures are translated into C as done in `clp(FD)`, then a great speed-up is expectable because the overhead of byte-code interpretation and many low-level operations such as tagging and untagging can be eliminated. Therefore, delay-Prolog is not only a good programming language for programming constraint propagation, but also an efficient intermediate language for compiling constraints.

7 Improvements

As we mentioned before, the cost of executing a constraint program comes from mainly three sources: delay-condition tests, domain reduction, and context switching. All these three operations can be optimized.

7.1 Improving domain reduction

The code for reducing domains can become much faster if value propagation is done only when necessary. By profiling program executions of the benchmark programs, we found that more than 80% of the primitives of `X in L..U` are fruitless because the range `L..U` is bigger than or equal to the domain of `X`. To avoid executing this kind of useless primitives, we can do similar things done in the indexical compilation algorithm and split constraints into three-tuple ones by introducing temporary variables and connect these constraints into a loop chain. As long as the connecting variable changes, the code for the next three-tuple constraint in the chain will be executed. For each triggering variable in the constraint, there is a re-entry point that indicates the code of the three-tuple constraint to be executed first when the constraint is triggered by the variable. When registering the constraint into the corresponding constraint list of the triggering variable, we also register the re-entry point.

Let us take the constraint `X+Y+Z #= U` as an example to illustrate the idea. The constraint is split into two small ones: `X+Y #= T` and `T+Z #= U`. Notice here that the temporary variable `T` is not a domain variable. No event happens when temporary variables are updated, and for a temporary variable, we only need to store its low and upper bounds. The code for reducing domains looks like:

```
X_updated: Y in min(T)-max(X)..max(T)-min(X), %X+Y = T
           T in min(X)+min(Y)..max(X)+max(Y),
           if (updated(T)) goto T1_updated else continue,
Y_updated: X in min(T)-max(Y)..max(T)-min(Y),
           T in min(X)+min(Y)..max(X)+max(Y),
           if (updated(T)) goto T1_updated else continue,
T2_updated:X in min(T)-max(Y)..max(T)-min(Y),
           Y in min(T)-max(X)..max(T)-min(X),
           continue,
T1_updated:U in min(T)+min(Z)..max(T)+max(Z), %T+Z = U
           Z in min(U)-max(T)..max(U)-min(T),
           continue,
Z_updated: U in min(T)+min(Z)..max(T)+max(Z),
           T in min(U)-max(Z)..max(U)-min(Z),
           if (updated(T)) goto T2_updated else continue,
U_updated: T in min(U)-max(Z)..max(U)-min(Z),
           Z in min(U)-max(T)..max(U)-min(T),
           if (updated(T)) goto T2_updated else continue.
```

For each variable `V` in the original constraint, the label `V_updated` indicates the re-entry point from where the execution starts when the constraint is triggered by an update of `V`. For each temporary variable `T`, there are two labels that indicate, respectively, the two points to go when the two occurrences of the temporary variable is updated. The code has the advantages of indixicals but does not have the problems because no new domain variables are introduced and the granularity of constraints never become smaller.

7.2 Increasing the granularity of constraints

Increasing the granularity of constraints is an important technique for reducing the number of delay-condition tests and context switching. To do so, we combine multiple constraints and compile them into a bigger propagation procedure.

For several constraints, if they share the same set of variables, then combing them is quite straightforward. Consider, for example, the three constraints taken from the queens program:

```
X #\= Y, X+C #\= Y, X-C #\= Y
```

where `C` is an integer, and `X` and `Y` are domain variables. As the constraints share the same set of variables, they can be compiled as a whole into the following propagation procedure:

```
delay noattack(X,Y,C):-dvar(X),dvar(Y) : {ins(X),ins(Y)}.
noattack(X,Y,C):-dvar(X) :
      exclude(X,Y), exclude(X,Y-C), exclude(X,Y+C).
noattack(X,Y,C):-dvar(Y) :
      exclude(Y,X), exclude(Y,X-C), exclude(Y,X+C).
noattack(X,Y,C):-X=\=Y, X+C=\=Y,X-C=\=Y.
```

When `X` or `Y` is instantiated, context switching is done only once rather than three times, and thus the delay-condition tests against `X` and `Y` need not be done separately three times.

For constraints that do not have the same set of variables but share some variables, we can also increase the granularity by combining them. Let $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ be three disjoint sets of variables. Suppose there are two constraints $c_1(\overline{X},\overline{Y})$ and $c_2(\overline{X},\overline{Z})$. We generate three propagation procedures, one called $p(\overline{X},\overline{Y})$ for c_1 which is triggered only when some change happens to variables in $\overline{Y}$, one called $q(\overline{X},\overline{Z})$ for c_2 which is triggered only when some change happens to variables in $\overline{Z}$, and the third called $pq(\overline{X},\overline{Y},\overline{Z})$ for both c_1 and c_2 which is triggered only when some shared variable in $\overline{X}$ changes. The two constraints is translated into the following three calls:

$$p(\overline{X},\overline{Y}), q(\overline{X},\overline{Z}), pq(\overline{X},\overline{Y},\overline{Z})$$

The procedure $p(\overline{X},\overline{Y})$ is defined as follows:

```
delay p(X̄,Ȳ) :-
    delay condition tests over variables in Ȳ :
    declare variables in Ȳ as triggering variables,
    reduce domains of the variables in X̄ and Ȳ.
    ...
```

The procedure $q(\overline{X},\overline{Z})$ is similar, but only variables in $\overline{Z}$ become triggering variables. The procedure $pq(\overline{X},\overline{Y},\overline{Z})$ is defined as follows:

```
delay pq(X̄,Ȳ,Z̄) :-
    delay condition tests over variables in X̄ :
    declare variables in X̄ as triggering variables,
    reduce domains of the variables in X̄, Ȳ, and Z̄.
    ...
```

When a shared variable in $\overline{X}$ changes, only the propagator pq will be triggered.

8 Concluding Remarks

we presented a high-level intermediate language, called *delay-Prolog*, for programming constraint propagation and compiling finite-domain constraints. Delay-Prolog has a much stronger description power than indexical and opens new ways to compiling constraints. We tried four compilation algorithms and found that the wait algorithm is the best algorithm and that compiling constraints into indexicals has the worst performance. We also showed that B-Prolog, which employs the wait compilation algorithm, is already comparable in performance with the fastest finite-domain constraint systems available now. We further proposed two techniques for improving the compilation algorithm. A great speed-up of the generated code is expected after the techniques are implemented.

References

[1] Abderrahmane Aggoun and Nicolas Beldiceanu: Overview of the CHIP Compiler System, In *Proceedings of the 8th International Conference on Logic Programming*, pp.775-789, MIT Press, 1991.

[2] Philippe Codognet and Daniel Diaz: Compiling Constraints in clp(FD), *Journal of Logic Programming*, pp.185-226, 1996.

[3] Philippe Codognet and Daniel Diaz: A Simple and Efficient Boolean Solver for Constraint Logic Programming, *Journal of Automated Reasoning*, 17, pp.97-128, 1996.

[4] Daniel Diaz: *clp(FD) 2.21 User's Manual*, INRIA Rocquencourt, 1994.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier: The Constraint Logic Programming Language CHIP, In *Proceedings of the Fifth Generation Computer Systems*, pp.693-702, ICOT, 1988.

[6] Eclipse 3.5 Extensions User Manual, ECRC Common Logic Programming System, 1995.

[7] Pascal Van Hentenryck: *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[8] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville: Design, Implementation, and Evaluation of the Constraint Language cc(FD), Technical Report No.CS-93-02, Brown University, 1993.

[9] Vipin Kumar: Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 32–44, 1992,.

[10] Micha Meier: Better Late Than Never, *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds., Kluwer Academic Publishers, 1994.

[11] *SICStus Prolog User's Manual*, Programming Systems Group, Swedish Institute of Computer Science, 1997.

[12] Gert Smolka: Constraint Programming in Oz, *Proc. 1997 International Symposium on Logic Programming*, pp.37-38.

[13] Neng-Fa Zhou: A Novel Implementation Method of Delay. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pp.97-111, MIT Press, 1996.

[14] Neng-Fa Zhou: *B-Prolog User's Manual, Version 3.0*, Kyushu Institute of Technology, 1998.