A Methodology for Managing Hard Constraints in CLP Systems

Joxan Jaffar
IBM T.J. Watson Research Ctr.
P.O. Box 704
Yorktown Heights, NY 10598
(joxan@ibm.com)

Spiro Michaylov
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(spiro@cs.cmu.edu)

Roland H.C. Yap
IBM T.J. Watson Research Ctr.
P.O. Box 704
Yorktown Heights, NY 10598
(rhc@ibm.com)

Abstract

In constraint logic programming (CLP) systems, the standard technique for dealing with hard constraints is to delay solving them until additional constraints reduce them to a simpler form. For example, the CLP($\mathcal{R}$) system delays the solving of nonlinear equations until they become linear, when certain variables become ground. In a naive implementation, the overhead of delaying and awakening constraints could render a CLP system impractical.

In this paper, a framework is developed for the specification of *wakeup degrees* which indicate how far a hard constraint is from being awoken. This framework is then used to specify a runtime structure for the delaying and awakening of hard constraints. The primary implementation problem is the timely awakening of delayed constraints in the context of temporal backtracking, which requires changes to internal data structures be reversible. This problem is resolved efficiently in our structure.

1 Introduction

The Constraint Logic Programming scheme [7] prescribes the use of a constraint solver, over a specific structure, for determining the solvability of constraints. In practice, it is difficult to construct efficient solvers for most useful structures. A standard compromise approach has been to design a partial solver, that is, one that solves only a subclass of constraints, the *directly solvable* ones. The remaining constraints, the *hard* ones, are simply delayed from consideration when they are first encountered; a hard constraint is reconsidered only when the constraint store contains sufficient information to reduce it into a directly solvable form. In the real-arithmetic-based CLP($\mathcal{R}$) system [8, 9], for example, nonlinear arithmetic constraints are classified as hard constraints, and they are delayed until they become linear.

The key implementation issue is how to efficiently process just those delayed constraints that are affected as a result of a new input constraint. Specifically, the cost of processing a change to the current collection of delayed constraints should be related to the delayed constraints affected by the change, and not to all the delayed constraints. The following two items seem necessary to achieve this end.

First is a notion which indicates how far a delayed constraint is from being awoken. For example, it is useful to distinguish the delayed CLP($\mathcal{R}$) constraint $X = max(Y, Z)$, which awaits the grounding of Y and Z, from the constraint $X = max(5, Z)$, which awaits the grounding of Z. This is because, in general, a delayed constraint is awoken by not one but a conjunction of several input constraints. When a subset of such input constraints has already been encountered, the runtime structure should relate the delayed constraint to just the remaining kind of constraints which will awaken it.

The other item is some data structure, call it the *access structure*, which allows immediate access to just the delayed constraints affected as the result of a new input constraint. The main challenge is how to maintain such a structure in the presence of backtracking. For example, if changes to the structure were trailed using some adaptation of PROLOG techniques [14], then a cost proportional to the number of entries can be incurred even though no delayed constraints are affected.

There are two main elements in this paper. First is a framework for the specification of *wakeup degrees* which indicate how far a hard constraint is from being awoken. Such a formalism makes explicit the various steps a CLP system takes in reducing a hard constraint into a directly solvable one. The second element is a runtime structure which involves a global stack representing the delayed constraints. This stack also contains all changes made to each delayed constraint when a new input constraint makes progress towards the awakening of the delayed constraint. A secondary data structure is the access structure. Dealing with backtracking is straightforward in the case of the global structure simply because it is a stack. For the access structure, no trailing/saving of entries is performed; instead, they are *reconstructed* upon backtracking. Such reconstruction requires a significant amount of interconnection between the global stack and access structure. In this runtime structure, the overhead cost of managing an operation on the delayed constraints is proportional to the size of the delayed constraints *affected* by the operation, as opposed to all the delayed constraints.

2 Background and Related Work

In this section we first review some early ideas of dataflow and local propagation, and the notion of flexible atom selection rules in logic programming systems. We then briefly review the basics of CLP, discuss the issue of delaying constraints in CLP, and mention some delay mechanisms in various CLP systems.

2.1 Data Flow and Local Propagation

The idea of dataflow computation, see e.g. [1], is perhaps the simplest form of a delay mechanism since program operations can be seen as directional constraints with fixed inputs and outputs. In its pure form, a dataflow graph is a specification of the data dependencies required by such an operation before it can proceed. Extensions, such as the I-structures of [2], are used to provide a delay mechanism for lazy functions and complex data structures such as arrays in the context of dataflow.

In local propagation, see e.g. [11], the solving of a constraint is delayed until enough of its variables have known values in order that the remaining values can be directly computed. Solving a constraint can then cause other constraints to have their values locally propagated, etc. The concept and its implementation are logical extensions of data flow – in essence, a data flow graph is one possible local propagation path through a constraint network. In other words, directionality is eliminated.

2.2 Delay Mechanisms in PROLOG

In PROLOG, the notion of delaying has been mainly applied to goals (procedure calls), and implemented by the use of a dynamically changeable atom selection rule. The main uses of delaying were to handle safe negation [10], and also to attempt to regain some of the completeness lost due to PROLOG's depth first search. There are similarities between implementing delay in PROLOG and implementing a data flow system, except in one fundamental aspect: temporal backtracking[1]. Further complications are related to the saving of machine states while a woken goal is being executed.

Some PROLOGs allow the user to specify delay annotations. One kind is used on subgoals. For example, the annotation *freeze(X, G)* in PROLOG-II [4] defers the execution of the goal G until X is instantiated. Another kind of annotation is applied to relations. For example, the *wait* declarations of MU-PROLOG [10] and the *when* declarations of NU-PROLOG [13] cause all calls to a procedure to delay until some instantiation condition is met.

Carlsson [3] describes an implementation technique for *freeze(X, G)*. While an implementation of *freeze* can be used to implement more sophisticated annotations like *wait* and *when*, this will be at the expense of efficiency. This is mainly because the annotations can involve complicated conjunctive and disjunctive conditions. Since *freeze* takes just one variable as an argument, it is used in a complicated manner in order to simulate the behavior of a more complex wakeup condition.

In general, the present implementation techniques used for PROLOG systems have some common features:

- The delay mechanism relies on a modification of the unification algorithm [3]. This entails a minimal change to the underlying PROLOG engine. In CLP systems, this approach is not directly applicable since there is, in general, no notion of unification.
- Goals are woken by variable bindings. Each binding is easily detectable (during unification). In CLP systems, however, detecting when a delayed constraint should awaken is far more complicated in general. In this paper, this problem is addressed using wakeup degrees, described in the next section.
- The number of delayed goals is not large in general. This can render acceptable, implementations in which the cost of awakening a goal is related to the number of delayed goals [3], as opposed to the number of awakened goals. In a CLP system, the number of delayed constraints can be very large, and so such a cost is unacceptable.

2.3 Delaying Hard Constraints in CLP

Before describing the notion of delaying constraints, we briefly recall some main elements of CLP. At the heart of a CLP language is a structure $\mathcal{D}$ which specifies the underlying domain of computation, the constant, function and constraint symbols, and the corresponding constants, functions and constraints. *Terms* are constructed using the constant and function symbols, and a *constraint* is constructed using a constraint symbol whose arguments are terms. An *atom* is a term of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and the t_i are terms. A CLP *program* is a finite collection of *rules*, each of which is of the form

$$A_0 :- \alpha_1, \alpha_2, \ldots, \alpha_k$$

where each α_i, $1 \leq i \leq k$, is either a constraint or an atom, that is, a term of the form $p(t_1, \ldots, t_n)$ where p is a user-defined predicate symbol and the t_i are terms. The language CLP($\mathcal{R}$) for example, involves arithmetic terms, e.g. $X + 3 * Y + Z$ and constraints, e.g. $X + 3 * Y + Z \geq 0$.

The essence of the CLP operational model is that it starts with an empty collection $\mathcal{CS}_0$ of constraints as its *constraint store*, and successively augments this store with a new *input* constraint. That is, each primitive step in the operational model obtains a new store $\mathcal{CS}_{i+1}$ by adding an input constraint to the previous store $\mathcal{CS}_i$, $i \geq 0$. The conjunction of the constraints in each store is satisfiable. If every attempt to generate a collection $\mathcal{CS}_{i+1}$ from $\mathcal{CS}_i$ results in an unsatisfiable collection, then the store may be reset to some previous

[1] Committed choice logic programming languages [12] use delaying for process synchronization. However there is no backtracking here.

store $\mathcal{CS}_j$, $j < i$, that is, *backtracking* occurs. The full details of the operational model, not needed in this paper, can be obtained from [7].

In principle, a CLP system requires a decision procedure for determining whether a satisfiable constraint store can be augmented with an input constraint such that the resulting store is also satisfiable. In practice, this procedure can be prohibitively expensive. An incomplete system, but which is often still very useful, can be obtained by partitioning the class of constraints into the *directly solvable* ones, and the *hard* ones. Upon encountering a hard constraint, the system simply defers the consideration of this constraint until the store contains enough information to reduce the hard constraint into a directly solvable form[2].

There are a number of CLP systems with delayed constraints. One is PROLOG-II [4] where the hard constraints are disequations over terms, and these constraints awaken when their arguments become sufficiently instantiated. In CLP($\mathcal{R}$) , the hard constraints are the nonlinear arithmetic ones, and these delay until they become linear. In CHIP [6], hard constraints include those over natural numbers, and these awaken when both an upper and lower bound is known for at least all but one of the variables. In PROLOG-III [5], some hard constraints are word equations and these awaken when the lengths of all but the rightmost variables in the two constituent expressions become known.

3 Wakeup Systems

Presented here is a conceptual framework for the specification of operations for the delaying and awakening of constraints. We note that the formalism below is not designed just for logic programming systems.

Let the *meta-constants* be a new class of symbols, and hereafter, these symbols are denoted by α and β. A meta-constant is used as a template for a (regular) constant. Define that a *meta-constraint* is just like a constraint except that meta-constants may be written in place of constants. A meta-constraint is used as a template for a (regular) constraint.

To indicate how far a hard constraint is from being awoken, associate with each constraint symbol Ψ a finite number of *wakeup degrees*. Such a degree $\mathcal{D}$ is a template representing a collection of Ψ-constraints[3]. It is defined[4] to be either the special symbol *woken*, or a pair $(t, \mathcal{C})$ where

- t is a term of the form $\Psi(t_1, \ldots, t_n)$ where each t_i is either a variable, constant or meta-constant, and
- $\mathcal{C}$, is a conjunction of meta-constraints which contain no variables and whose meta-constants, if any, appear in t.

Let θ be a mapping from variables into variables and meta-constants into constants. An *instance* of a wakeup degree $\mathcal{D} = (t, \mathcal{C})$ is the constraint obtained by applying to t such a mapping θ which evaluates $\mathcal{C}$ into *true*. The instance is denoted $\mathcal{D}\theta$. In CLP($\mathcal{R}$) for example, a subset of the constraints involving the constraint symbol *pow* (where $pow(X, Y, Z)$ means $X = Y^Z$) may be represented by the degree $pow(A, B, \alpha)$, $\alpha \neq 0$. This subset contains all the constraints of the form $pow(X, Y, c)$ where X and Y are not necessarily distinct variables and c is a nonzero real number.

Associated with each wakeup degree $\mathcal{D}$ is a collection of pairs, each of which contains a *generic wakeup condition* and a wakeup degree called the *new degree*. Each wakeup condition is a conjunction of meta-constraints all of whose meta-constants appear in $\mathcal{D}$. Any variable in a wakeup condition which does not appear the in associated degree is called *existential*. An *instance* $\mathcal{W}\theta$ of a generic wakeup condition $\mathcal{W}$ is the constraint

[2] It is possible that hard constraints remained indefinitely deferred.

[3] These are constraints written using the symbol Ψ.

[4] This specific definition is but one way to represent a set of expressions. It may be adapted without affecting what follows.

obtained by applying the mapping θ which maps non-existential variables into variables and meta-constants into constants.

Like wakeup degrees, a wakeup condition represents a collection of constraints. Intuitively, a wakeup condition specifies when a hard constraint changes degree to the new degree. More precisely, suppose that $\mathcal{D}$ is a wakeup degree and that $\mathcal{W}$ is one of its wakeup conditions with the new degree $\mathcal{D}'$. Let $\mathcal{C}$ be a hard constraint in $\mathcal{D}$, that is, $\mathcal{C}$ is an instance $\mathcal{D}\theta$ of $\mathcal{D}$. Further suppose that the constraint store implies the corresponding instance of $\mathcal{W}$, that is, the store implies

$$\exists X_1 \ldots X_n(\mathcal{W}\theta)$$

where the X_i denote the existential variables of $\mathcal{W}$. Let $\mathcal{C}'$ denote a constraint equivalent to $\mathcal{C} \wedge \exists X_1 \ldots X_n(\mathcal{W}\theta)$. We then say the constraint $\mathcal{C}$ *reduces* to the constraint $\mathcal{C}'$ *via* $\mathcal{W}$.

Consider once again the CLP($\mathcal{R}$) constraints involving *pow*. These constraints may be partitioned into classes represented by the wakeup degrees $pow(A, B, C)$, $pow(\alpha, B, C)$, $pow(A, \alpha, C)$, $pow(A, B, \alpha)$ and *woken*. For the degree $pow(A, B, C)$, which represents constraints of the form $pow(X, Y, Z)$ where X, Y and Z are variables, an example wakeup condition is $C = \alpha$. This indicates that when a constraint, e.g. $Z = 4$, is entailed by the constraint store, a delayed constraint such as $pow(X, Y, Z)$ is reduced to $pow(X, Y, 4)$. This reduced constraint may have the new degree $pow(A, B, \alpha)$. Another example wakeup condition is $A = 1$, indicating that when a constraint such as $X = 1$ is entailed, a delayed constraint of the form $pow(X, Y, Z)$ can be reduced to $pow(1, Y, Z)$. This reduced constraint, which is in fact equivalent to the directly solvable constraint $Y = 1 \vee (Y \neq 0 \wedge Z = 0)$, may be in the degree *woken*. We exemplify some other uses of wakeup conditions below.

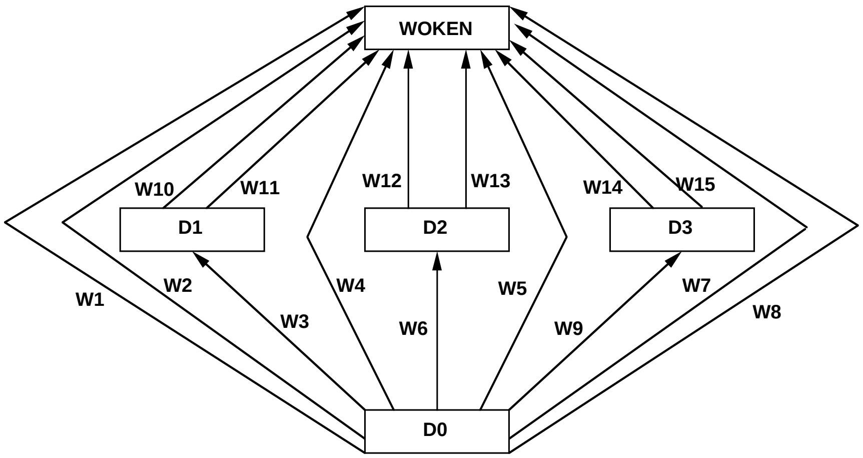
A *wakeup system* for a constraint symbol Ψ is a finite collection S of wakeup degrees for Ψ satisfying:

- S contains the special degree called *woken* which represents a subset of all the directly solvable Ψ-constraints.
- No two degrees in S contain the same Ψ-constraint.
- Let the Ψ-constraint $\mathcal{C}$ have a degree $\mathcal{D}$ which has a wakeup condition $\mathcal{W}$ and new degree $\mathcal{D}'$. Then every reduced constraint $\mathcal{C}'$ of $\mathcal{C}$ via $\mathcal{W}$ is contained in $\mathcal{D}'$.

The illustration in figure 1 contains an example wakeup system for the CLP($\mathcal{R}$) constraint symbol *pow*. A wakeup degree is represented by a node, a wakeup condition is represented by an edge label, and the new degree of a reduced constraint is represented by the target node of the edge labelled with the wakeup condition which caused the reduction.

Generic wakeup conditions can be used to specify the operation of many existing systems which delay constraints. In PROLOG-like systems whose constraints are over terms, awaiting the instantiation of a variable X to a ground term can be represented by the wakeup condition $X = \alpha$. Awaiting the instantiation of X to a term of the form $f(\ldots)$, on the other hand, can be represented by $X = f(Y)$ where Y is an existential variable. We now give some examples on arithmetic constraints. In PROLOG-III, for example, the wakeup condition $X \leq \alpha$ could specify that the length of word must be bounded from above before further processing of the word equation at hand. For CHIP, where combinatorial problems are the primary application, an example wakeup condition could be $\alpha \leq X \wedge X \leq \beta \wedge \beta - \alpha \leq 4$ which requires that X be bounded within a small range. For CLP($\mathcal{R}$) , an example wakeup condition could be $X = \alpha * Y + \beta$, which requires that a linear relationship hold between X and Y.

Summarizing, each constraint symbol in a CLP system which gives rise to hard constraints can be associated with a finite collection of wakeup degrees each of which indicate how far the constituent constraints are from being awoken. These degrees can be organized into a wakeup system, that is, a graph



Legend:

D0 : $pow(A,B,C)$ D1 : $pow(\alpha,B,C), \alpha \neq 0 \wedge \alpha \neq 1$
D2 : $pow(A,\alpha,C), \alpha \neq 0 \wedge \alpha \neq 1$ D3 : $pow(A,B,\alpha), \alpha \neq 0 \wedge \alpha \neq 1$

$w_1 : A = 0$ $w_2 : A = 1$ $w_3 : A = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_4 : B = 0$ $w_5 : B = 1$ $w_6 : B = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_7 : C = 0$ $w_8 : C = 1$ $w_9 : C = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_{10} : B = \alpha$ $w_{11} : C = \alpha$ $w_{12} : A = \alpha$
$w_{13} : C = \alpha$ $w_{14} : A = \alpha$ $w_{15} : B = \alpha$

Figure 1: Wakeup degrees for *pow*/3

whose nodes represent degrees and whose edges are represent the wakeup condition/new degree relation between two degrees. Such a wakeup system can be viewed as a deterministic transition system, and can be used to specify the organization of a constraint solver: the degrees discriminate among constraints so that the solver is able to treat them differently, while the wakeup conditions specify degree transitions of hard constraints with respect to new input constraints.

An important design criterion is that the entailed constraints corresponding to the wakeup conditions be efficiently recognizable by the constraint solver. This problem, being dependent on a specific solver, is not addressed in this paper. It is difficult in general[5]. In CLP($\mathcal{R}$) , for example, it is relatively inexpensive to perform a check if an equation like $X = 5$ is entailed whenever the constraint store is changed. The situation for an inequality like $X \leq 5$ is quite different.

4 The Runtime Structure

Here we present an implementational framework in the context of a given wakeup system. There are three major operations with hard constraints which correspond to the actions of delaying, awakening and backtracking:

1. adding a hard constraint to the collection of delayed constraints;
2. awakening delayed constraints as the result of inputting a directly solvable constraint, and
3. restoring the entire runtime structure to a previous state, that is, restoring the collection of delayed constraints to some earlier collection, and restoring all auxiliary structures accordingly.

The first of our two major structures is a stack[6] containing the delayed constraints. Thus imple-

[5] In PROLOG, this problem reduces to the easy check of whether a variable is bound.

[6] Hereafter, the term stack refers to this structure.

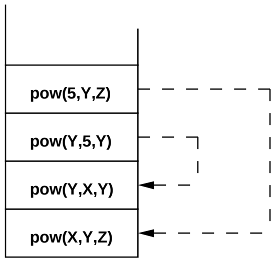


Figure 2: The stack

menting operation 1, delaying a hard constraint, simply requires a push on this stack. Additionally, the stack contains hard constraints which are reduced forms of constraints deeper in the stack. For example, if the hard constraint $pow(X, Y, Z)$ were in the stack, and if the input constraint $Y = 3$ were encountered, then the new hard constraint $pow(X, 3, Z)$ would be pushed, together with a pointer from the latter constraint to the former. In general, the collection of delayed constraints contained in the system is described by the subcollection of stacked constraints which have no inbound pointers.

Figure 2 illustrates the stack after storing the hard constraint $pow(X, Y, Z)$, then storing $pow(Y, X, Y)$, and then encountering the entailed constraint $X = 5$. Note that this one equation caused the pushing of two more elements, these being the reduced forms of the original two. The top two constraints now represent the current collection of delayed constraints.

The stack operations can be more precisely described in terms of the degrees of the hard constraint at hand. This description is given during the definition of the access structure below.

Now consider operation 2. In order to implement this efficiently, it is necessary to have some access structure mapping an entailed constraint $\mathcal{C}$ to just those delayed constraints affected by $\mathcal{C}$. Since there are in general an infinite number of entailed constraints, a finite classification of them is required. We define this classification below, but we assume that the constraint solver, having detected an entailed constraint, can provide access to precisely the classes of delayed constraints which change degree.

A *dynamic wakeup condition* is an instance of a generic wakeup condition $\mathcal{W}$ obtained by (a) renaming all the non-existential variables in $\mathcal{W}$ into runtime variables[7], and (b) instantiating any number of the meta-constants in $\mathcal{W}$ into constants. An *instance* of a dynamic wakeup condition is obtained by mapping all its meta-constants into constants.

A dynamic wakeup condition is used as a template for describing the collection of entailed constraints (its instances) which affect the same sub-collection of delayed constraints. For example, suppose that the only delayed constraint is $pow(5, Y, Z)$ whose degree is $pow(\alpha, B, C)$ with generic wakeup conditions $B = \alpha$ and $C = \alpha$. Then only two dynamic wakeup conditions need be considered: $Y = \alpha$ and $Z = \alpha$. In general, only the dynamic wakeup conditions whose non-existential variables appear in the stack need be considered.

We now specify an access structure which maps a dynamic wakeup condition into a doubly linked list of nodes. Each node contains a pointer to a stack element containing a delayed constraint[8]. Corresponding to each occurrence node is a reverse pointer from the stack element to the occurrence node. Call the list associated with a dynamic wakeup condition $\mathcal{DW}$ a *$\mathcal{DW}$-list*, and call each node in the list a *$\mathcal{DW}$-occurrence node*.

Initially the access structure is empty. The following specifies what is done for the basic operations. It is assumed, without loss of generality, that the variables in the wakeup system are disjoint from runtime variables, and that no existential variable appears in more than one generic wakeup condition.

[7] These are the variables the CLP system may encounter.

[8] The total number of occurrence nodes is generally larger than the number of delayed constraints.

4.1 Delaying a new hard constraint

To delay a new hard constraint $\mathcal{C}$, first push a new stack element for $\mathcal{C}$. Let $\mathcal{D}$ denote its wakeup degree and $\mathcal{W}_1$, ..., $\mathcal{W}_n$ denote the generic wakeup conditions of $\mathcal{D}$. Thus $\mathcal{C}$ is $\mathcal{D}\theta$ for some θ. Then:

- For each $\mathcal{W}_i$, compute the dynamic wakeup condition $\mathcal{DW}_i$ corresponding to $\mathcal{C}$ and $\mathcal{W}_i$, that is, $\mathcal{DW}_i$ is $\mathcal{W}_i\theta$.
- For each $\mathcal{DW}_i$, insert into the $\mathcal{DW}_i$-list of the access structure a new occurrence node pointing to the stack element $\mathcal{C}$.
- Set up reverse pointers from $\mathcal{C}$ to the new occurrence nodes.

4.2 Processing an Entailed Constraint

Suppose there is a new entailed constraint, say $X = 5$. Then:

- Obtain the dynamic wakeup conditions in the access structure whose instances are implied by $X = 5$. If no such conditions exist (i.e. no delayed constraint is affected by $X = 5$ being entailed), nothing more needs to be done.
- Consider the lists L associated with the above conditions. Then consider in turn each delayed constraint pointed to by the occurrence nodes in L. For each such constraint $\mathcal{C}$, perform the following.
  - ○ Delete all occurrence nodes pointed to by $\mathcal{C}$.
  - ○ Construct the reduced form $\mathcal{C}'$ of $\mathcal{C}$ by replacing all occurrences of X by 5. (Recall that in general, $\mathcal{C}'$ is obtained by conjoining $\mathcal{C}$ with the entailed constraint.) Now push $\mathcal{C}'$ onto the stack, set up a pointer from $\mathcal{C}'$ to $\mathcal{C}$, and then perform the modifications to the access structure as described above when a new delayed constraint is pushed.

Figure 3 illustrates the entire runtime structure after the two hard constraints $pow(X, Y, Z)$ and $pow(Y, X, Y)$ were stored, in this order. Figure 4 illustrates the structure after a new input constraint makes $X = 5$ entailed.

4.3 Backtracking

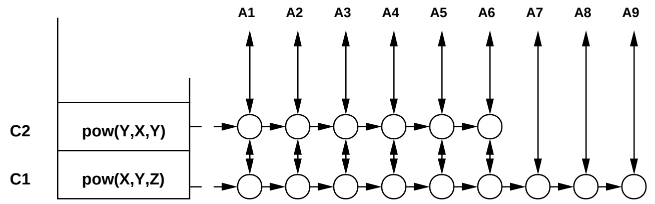
Restoring the stack during backtracking is easy because it only requires a series of pops. Restoring the access structure, however, is not so straightforward because no trailing/saving of the changes was performed. In more detail, the primitive operation of backtracking is the following:

- Pop the stack, and let $\mathcal{C}$ denote the constraint just popped.
- Delete all occurrence nodes pointed to by $\mathcal{C}$.
- If there is no pointer from $\mathcal{C}$ (and so it was a hard constraint that was newly delayed) to another constraint deeper in the stack, then nothing more need be done.
- If there is a pointer from $\mathcal{C}$ to another constraint $\mathcal{C}'$ (and so $\mathcal{C}$ is the reduced form of $\mathcal{C}'$), then perform the modifications to the access structure *as though* $\mathcal{C}'$ were being pushed onto the stack. These modifications, described above, involve computing the dynamic wakeup conditions pertinent to $\mathcal{C}'$, inserting occurrence nodes, and setting up reverse pointers.

Note that the access structure obtained in backtracking may not be structurally the same as that of the previous state. What is important, however, is that it depicts the same *logical* structure as that of the previous state.

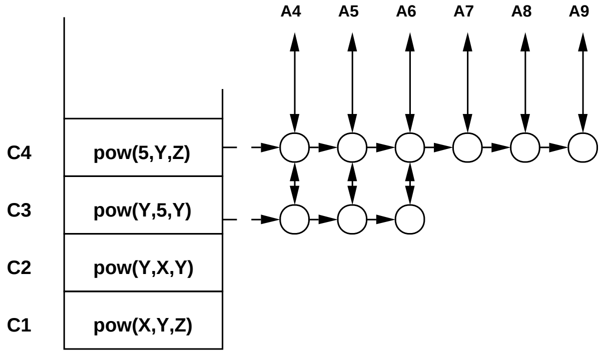
4.4 Optimizations

Additional efficiency can be obtained by not creating a new stack element for a reduced constraint if there is no choice point (backtrack point) between



$$\text{Legend: } \begin{cases} A_1 : X = 0 & A_2 : X = 1 & A_3 : X = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \\ A_4 : Y = 0 & A_5 : Y = 1 & A_6 : Y = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \\ A_7 : Z = 0 & A_8 : Z = 1 & A_9 : Z = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \end{cases}$$

Figure 3: The access structure



$$\text{Legend: } \begin{cases} A_4 : Y = 0 & A_5 : Y = 1 & A_6 : Y = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \\ A_7 : Z = 0 & A_8 : Z = 1 & A_9 : Z = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1 \end{cases}$$

Figure 4: The new access structure

the changed degrees in question. This saves space, saves pops, and makes updates to the access structure more efficient.

Another optimization is to save the sublist of occurrence nodes deleted as a result of changing the degree of a constraint. Upon backtracking, such sublists can be inserted into the access structure in constant time. This optimization, however, sacrifices space for time.

A third optimization is to merge $\mathcal{DW}$-lists. Let there be lists corresponding to the dynamic wakeup conditions $\mathcal{DW}_1$, ..., $\mathcal{DW}_n$. These lists can be merged into one list with the condition $\mathcal{DW}$ if

- the push of any delayed constraint $\mathcal{C}$ results in either (a) no change in any of the n lists, or (b) every list has a new occurrence node pointing to $\mathcal{C}$;
- for every constraint $\mathcal{C}$ and for every mapping θ of meta-constants into constants, $\mathcal{C}$ implies $\mathcal{DW}\theta$ iff $\mathcal{C}$ implies $\mathcal{DW}_i\theta$ for some $1 \leq i \leq n$.

In the example of figure 3, the three lists involving X can be merged into one list which is associated with the dynamic wakeup conditions $X = \alpha$. Similarly for Y and Z.

4.5 Summary of the Runtime Structure

A stack is used to store delayed constraints and their reduced forms. An access structure maps a finite number of dynamic wakeup constraints to lists of delayed constraints. The constraint solver is assumed to identify those conditions for which an entailed constraint is an instance. The basic operations are then implemented as follows.

1. Adding a new constraint $\mathcal{C}$ simply involves a push on the stack, creating new occurrence nodes corresponding to $\mathcal{C}$ and the setting of pointers between the new stack and occurrence nodes. The cost here is bounded by the number of generic wakeup conditions associated with (the degree of) $\mathcal{C}$.
2. Changing the degree of a constraint $\mathcal{C}$ involves a push of a new constraint $\mathcal{C}'$, deleting and inserting a number of occurrence nodes. Since the occurrence nodes are doubly-linked, each such insertion and deletion can be done in constant time. Therefore the total cost here is bounded by the number of generic wakeup conditions associated with $\mathcal{C}$ and $\mathcal{C}'$.
3. Similarly, the cost in backtracking of popping a node $\mathcal{C}$, which may be the reduced form of another constraint $\mathcal{C}'$, involves deleting and inserting a number of occurrence nodes. The cost here is again bounded by the number of generic wakeup conditions associated with $\mathcal{C}$ and $\mathcal{C}'$.

In short, the cost of one primitive operation on delayed constraints (delaying a new hard constraint, upgrading the degree of one delayed constraint, including awakening the constraint, and undoing the delay/upgrade of one hard constraint) is bounded by the (fixed) size of the underlying wakeup system. The total cost of an operation (delaying a new hard constraint, processing an entailed constraint, backtracking) on delayed constraints is proportional to the size of the delayed constraints affected by the operation.

5 Concluding Discussion

A framework of wakeup degrees is developed to specify the organization of a constraint solver. These degrees represent the various different cases of a delayed constraint which should be treated differently for efficiency reasons. Associated with each degree is a number of wakeup conditions which specify when an input constraint changes the degree of a hard constraint. What is intended is that the wakeup conditions represent all the situations in which the constraint solver can efficiently update its knowledge about how far each delayed constraint is from being fully awoken.

The second part of this paper described a runtime structure for managing delayed constraints. A stack is used to represent the current collection of delayed constraints. It is organized so that it also records the chronological order of all changes made to this collection. These changes appear in the form of inserting a new delayed constraint, as well as changing the degree of an existing delayed constraint. An access structure is designed to quickly locate all delayed constraints affected by an entailed constraint. By an appropriate interconnection of pointers between the stack and the table, there is no need to save/trail changes made in the structure. Instead, a simple process of inserting or deleting nodes, and of redirecting pointers, is all that is required in the event of backtracking. By adopting this technique of performing minimal trailing, the speed of forward execution is enhanced, and space is saved, at the expense of some reconstruction in the event of backtracking. Even so, the overall overhead cost of the runtime structure for managing an operation to the delayed constraints is, in some sense, minimal.

Finally we remark that the implementation technique described has been used in the CLP($\mathcal{R}$) system for delaying hard constraints such as *multiply* and *pow*.

Acknowledgement. We thank Nevin Heintze and Michael Maher for their comments.

References

[1] Arvind and D.E. Culler, "Dataflow Architectures", in *Annual Reviews in Computer Science*, Vol. 1, Annual Reviews Inc., Palo Alto (1986), pp 225–253.

[2] Arvind, R.S. Nikhil and K.K. Pingali, "I-Structures: Data Structures for Parallel Computing", *ACM Transactions on Programming Languages and Systems*, 11(4) (October 1989) pp 598–632.

[3] M. Carlsson, "Freeze, Indexing and other Implementation Issues in the WAM", *Proceedings* 4^{th} *International Conference on Logic Programming*, MIT Press (June 1987), 40–58.

[4] A. Colmerauer, "PROLOG II Reference Manual & Theoretical Model," Internal Report, Groupe Intelligence Artificielle, Université Aix - Marseille II (October 1982).

[5] A. Colmerauer, "PROLOG-III Reference and Users Manual, Version 1.1", PrologIA, Marseilles (1990). [See also "Opening the PROLOG-III Universe", BYTE Magazine (August 1987).]

[6] M. Dincbas, P. Van Hentenryck, H. Simonis and A. Aggoun, "The Constraint Logic Programming Language CHIP", *Proceedings of the* 2^{nd} *International Conference on Fifth Generation Computer Systems*, Tokyo (November 1988), pp 249–264.

[7] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proceedings* 14^{th} *ACM Symposium on Principles of Programming Languages*, Munich (January 1987), pp 111–119. [Full version: Technical Report 86/73, Dept. of Computer Science, Monash University, June 1986]

[8] J. Jaffar and S. Michaylov, "Methodology and Implementation of a CLP System", *Proceedings* 4^{th} *International Conference on Logic Programming*, MIT Press (June 1987), pp 196–218.

[9] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap, "The CLP($\mathcal{R}$) Language and System: an Overview", IBM Research Report, RC 16292 (#72336), (November 1990).

[10] L. Naish, "Negation and Control in Prolog", Technical Report 85/12, Department of Computer Science, University of Melbourne (1985).

[11] G.L. Steele, "The Implementation and Definition of a Computer Programming Language Based on Constraints", Ph.D. Dissertation (MIT-AI TR 595), Dept. of Electrical Engineering and Computer Science, M.I.T.

[12] E.Y. Shapiro, "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys*, 21(3) (September 1989), pp 412–510.

[13] J.A. Thom and J. Zobel (Eds), "NU-PROLOG Reference Manual - Version 1.3" Technical Report 86/10, Dept. of Computer Science, University of Melbourne (1986) [revised in 1988].

[14] D.H.D. Warren, "An Abstract PROLOG Instruction Set", Technical note 309, AI Center, SRI International, Menlo Park (October 1983).