A System for Tabled Constraint Logic Programming

Baoqiu Cui and David S. Warren

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
{cbaoqiu,warren}@cs.sunysb.edu

Abstract. As extensions to traditional logic programming, both tabling and Constraint Logic Programming (CLP) have proven powerful tools in many areas. They make logic programming more efficient and more declarative. However, combining the techniques of tabling and constraint solving is still a relatively new research area. In this paper, we show how to build a *Tabled Constraint Logic Programming* (*TCLP*) system based on XSB — a tabled logic programming system. We first discuss how to extend XSB with the fundamental mechanism of constraint solving, basically the introduction of attributed variables to XSB, and then present a general framework for building a TCLP system. An interface among the XSB tabling engine, the corresponding constraint solver, and the user's program is designed to fully utilize the power of tabling in TCLP programs.

1 Introduction

As two separate research directions within the area of Logic Programming (LP), tabling and Constraint Logic Programming (CLP) have long been studied. Both of them have proven to be powerful tools and have made logic programming more efficient and more declarative.

Since its introduction in logic programming [21], tabling (also called *memoing*) has been used in many areas [23]. It can not only avoid redundant computations and many infinite loops, but can also, through tabled aggregation [20], give an easy way to find the optimal solutions for some problems. Tabling for pure logic programming has been implemented in the XSB system [19,4].

CLP is a natural extension of LP, and has gained much success since the late 1980's. Stemming from LP, CLP is a new class of programming languages which applies efficient constraint solving techniques to increase the power and declarativity of LP. Just like a classic logic programming system, a CLP system can also benefit from the power of tabling — the ability to avoid redundant computations and infinite loops and to find the optimal solutions. In other words, tabling can further increase the declarativity of a CLP system. Some mostly theoretical work has been done to combine tabling and constraint solving. For example, tabling has been applied to the constraint extensions of Datalog in

J. Lloyd et al. (Eds.): CL 2000, LNAI 1861, pp. 478–492, 2000.

[22] and to CLP in [12], a general scheme for the evaluation of constraint logic programs based on a tabling mechanism has been given in [2], and a tabling algorithm for CLP has been designed in [14]. However, no practical general framework for building a *Tabled Constraint Logic Programming* (*TCLP*) system has been constructed.

Having the best tabling engine, XSB is a very good candidate system to be extended to a TCLP system. However, prior to version 2.0, XSB did not have the features necessary to incorporate constraint solving. This motivated to the introduction of *attributed variables* to XSB.

In the early stage of constraint logic programming, constraint solving was "hard-wired" into a built-in constraint solver over a specific constraint domain. This implementation strategy makes it difficult to modify an existent constraint solver to build a new solver over a new domain. To build a constraint solver over a new domain, one has to start everything from scratch. This situation changed with the introduction of attributed variables. Attributed variables [11,5,6] are a new logic programming data type that associates variables with arbitrary attributes and supports extensible unification [7]. Because of the ability to store attributes, attributed variables can be used to represent user-defined constraints on the variables (usually a whole constraint store can be represented by a set of attributed variables). Attributed variables can extend the default unification algorithm in that, when an attributed variable is to be unified with a term (which can be another attributed variable), a user-defined *unification handler* (in a high-level language, like Prolog) is called to process the two objects to be unified and possibly change the attributes of the involved attributed variable(s).

Attributed variables have proven to be a flexible and powerful mechanism to extend a classic logic programming system with the ability to solve constraints, and they have been implemented in many constraint logic programming systems, e.g. SICStus [13] and ECL^{i}PS^e [1]. Based on attributed variables, logic programming systems have been enhanced by constraint solvers over rational and real numbers [8] and feature trees [16]. Also, attributed variables have been recently used in the implementation of a high level language to write constraint solvers — Constraint Handling Rules (CHR) [9,10], where constraints are compiled into clauses and stored in attributed variables. Compared to CHR, using attributed variables to directly implement constraint solvers is as "constraint assembler" programming [10].

The flexibility of the attributed variable mechanism is the major reason why we chose it to introduce constraints to XSB. Another reason is that we want to make XSB a conservative extension of CLP, so that standard CLP programs run (reasonably) well on XSB. XSB is a conservative extension of Prolog in that it includes all functionality of Prolog: Prolog programs run well on XSB. For this reason we took relatively standard implementation techniques from CLP as a basis for our implementation of constraints in XSB. However, the interesting, important, and challenging aspect of the integration is the interaction between the tabling mechanisms and the constraint mechanisms, and whether this approach can result in a (reasonably) efficient TCLP system. This paper

concentrates on the interaction of the implementation of the two mechanisms: constraints and tabling: the representation of constraints through attributed variables and the representation of tables as tries.

To introduce attributed variables to XSB, a new data type and a new type of interrupt have to be added to XSB (see [3]). More importantly, in order to copy constraints into and out of tables, we need to modify parts of the basic data structure of the tabling engine, namely the *tabling tries* and the *substitution factor* [17,18], to support attributed variables. Tabling tries provide an efficient way to look up terms in a table or insert terms into a table, and the whole table space of XSB is divided into two parts: subgoal tables (a.k.a. *subgoal tries*) and answer tables (a.k.a. *answer tries*). Subgoal tables contain all the subgoal calls, while answer tables contain only the answer substitutions of the corresponding subgoal calls. (We call the answer substitutions the "substitution factor" since they are the only parts of the entire answer subgoal that need to be stored.) In a TCLP system, a subgoal call is associated with a set of constraints, and an answer is associated with a set of answer constraints. The two sets of constraints are normally represented by the same set of attributed variables, whose attributes in the call might be updated in the answer. Therefore, we have to keep the update information of attributed variables in the subgoal table and answer table. This requires the substitution factor to be extended to contain not only regular variables in the call, but also attributed variables in the call.

Having introduced attributed variables to XSB, it is possible to simply apply tabling to a CLP program in the same way we table a normal LP program: a subgoal (or answer), together with the constraints involved, is saved into or retrieved from the table as a regular XSB subgoal (or answer). The *identical* subgoal call (with the *identical* constraints) will never be computed twice. However there exist two drawbacks in this naive tabling. First, since only identical calls (or answers) are checked when they are saved into or retrieved from the table, in order to get any reasonable reuse of tables, constraints must be represented in a canonical form. Second, when a new call is made, only looking up the table for the equivalent call and then consuming the existing answers in the table cannot fully utilize the power of tabling, and the amount of table space required can be extremely large. In many cases, a new subgoal call can consume the answers of an old call in the table if the old call subsumes the new one. Allowing this kind of subsumption tabling can make more use of the tables and further reduce the amount of redundant computation [2].

In this paper, we shall present a general framework for building a TCLP system based on XSB using the idea of subsumption tabling. In this framework, an interface among XSB's tabling engine, the constraint solver, and the CLP programs is designed, which gives the user more control on how the tabling engine works on the tabled predicates. The interface is divided into two parts: one is at the point when a subgoal call is to be put into the subgoal table; the other is at the point when an answer is put into the answer table. In the first part, we can define what form of subgoals should be stored in the subgoal table: for example a goal more general than the specific call might be specified. In the

second part, we can define what kind of answers to a certain subgoal should be considered new and stored into the answer table. This is done in a similar way in which table aggregation [20] is implemented.

The remainder of the paper is organized as follows: In Sect. 2, we explain how to extend XSB with attributed variables. We concentrate more on the modifications of tabling tries and substitution factor to efficiently support attributed variables. In Sect. 3, we discuss some new issues when constraints and tabling are combined. Then, we present the general framework for building a TCLP system in Sect. 4. An example of the application of this framework is shown in Sect. 5. Finally, we give the conclusion and future work.

2 Extending XSB with Attributed Variables

2.1 Basic Changes to the System

Since attributed variables are a new data type in XSB, a new cell tag, ATTV, is added to the system. An attributed variable is represented as a pair of words (as a list): the first word is a free variable, which can be further bound to another term; the second word is a regular Prolog term, which is the attribute of this attributed variable. A new type of interrupt, *attributed variable interrupt*, is added to XSB, so that whenever an attributed variable is to be unified with a non-variable term, an attributed variable interrupt is triggered, and then the high-level user-defined unification handler is called to finish the unification.

In this paper, we focus on how to extend tabling tries and substitution factor to support attributed variables. Other more detailed information about the implementation can be found in [3].

2.2 Modifications of Tabling Engine for Attributed Variables

In XSB, tabling tries are used as the basic data structure of the table. They provide an efficient way for term lookup and insertion. As constraints are stored in attributed variables, we have to extend tabling tries to support attributed variables in order to copy constraints into and out of tables. Moreover, being variables, attributed variables must be stored in the substitution factor. Since they have certain patterns of use, optimizations can be done for them. Thus we treat them specially in the substitution factor.

As attributed variables are represented as lists, they can be copied into tries as lists: the first word (the free variable) can be copied as a regular variable, and the second word (the attribute) can be copied as a normal Prolog term. However, this representation could store the attributes of an attributed variable multiple times if it appeared in a term multiple time. This could waste a lot table space if the size of the attribute is very big (say this attributed variable is involved in many complicated constraints). Basically, to support attributed variables in tries, two problems have to be considered. First, since attributed variables are treated as variables, they (including their attributes) have to be

kept shared when copied into and out of tables. Second, because an attributed variable in the call might often not be updated in the answer, it is important not to construct its attribute again in the answer table. We need to find a way to share the unchanged attributed variables between the subgoal table and answer table.

In XSB without attributed variables, an array called *VarEnumerator* and a counter called *var_ctr* are used to keep track of all the variables encountered when we copy a term into a trie, so that variables are numbered and kept shared in tries. When a variable is encountered for the first time, it is bound to *VarEnumerator*[*var_ctr*] (and trailed) and *VarEnumerator*[*var_ctr*] itself is set to be a free variable. Then a node, $\overline{v_i}$ ($i = var\_ctr$), is put into the trie and *var_ctr* is increased by one. Later, if this variable is encountered again, it will be dereferenced to *VarEnumerator*[i]. Thus we can tell that it is an old variable, and a node, v_i, is inserted into the trie. Here, nodes $\overline{v_i}$ and v_i are two different types of trie nodes, which represent the first and a later occurrence of the ith variable in the term respectively.

Carefully designed, *VarEnumerator* and *var_ctr* can also be used efficiently to handle attributed variables in a similar way. The basic idea is that attributed variables and regular variables are numbered together and they share the use of *VarEnumerator*. To distinguish an attributed variable and regular variable in tries, a new type of trie node is added. When an attributed variable, `X` (see Fig. 1(a)), is encountered for the first time, its first word (the free variable) is bound to *VarEnumerator*[*var_ctr*] (and trailed) and *VarEnumerator*[*var_ctr*] is set to be a free variable (shown in Fig. 1(b)). Then a node, $\widehat{v_i}$ ($i = var\_ctr$), is put into the trie and *var_ctr* is increased by one. Node $\widehat{v_i}$ is the newly added type of trie node, which denotes the first occurrence of an attributed variable (the ith variable in the term). Following $\widehat{v_i}$, the attribute of `X` (pointed to by the CS cell) is copied into the trie as a normal term. Now, if this attributed variable is encountered again (from `X'` in Fig. 1(b)), it will be dereferenced to *VarEnumerator*[i] and treated as a later occurrence of a *regular* variable, so only one node, v_i, is inserted into the trie. The attribute of `X` is *not* copied into the trie again.

As we can see, the same type of trie node, v_i, is used for the later occurrence of a variable, no matter it is a regular variable or an attributed variable. This does not cause any confusion when a term is copied out of the table. We can tell whether a node v_i is a later occurrence of a regular variable or an attributed variable by the index i, because the first occurrence of this variable (saved in the trie as $\overline{v_i}$ or $\widehat{v_i}$) has been built in the heap and a tagged pointer to it has been saved in an array (similar to *VarEnumerator*).

The above described algorithm can be used directly to construct the subgoal trie. The numbering and sharing of attributed variables is shown in the following example.

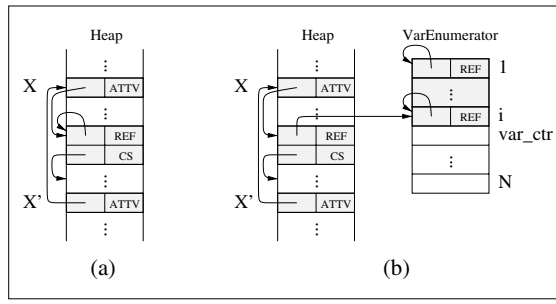


Fig. 1. How to number attributed variables and keep them shared in tries: (a) Before the new attributed variable `X` is processed; (b) After `X` is processed.

Example 1. Suppose we have a program and a query as shown in Fig. 2(a)[1]. After the query has been executed, the subgoal trie of `p/5` contains only one subgoal and is shown in Fig. 2(b). In this subgoal, attributed variable `A2` appears twice: the first occurrence is saved as $\widehat{v_3}$ (since it is the third variable in the call) followed by the attribute `a(2)`, while the second occurrence is saved as only one node, v_3.

Constructing the answer trie (e.g. the one shown in Fig. 2(c)) is more complex than constructing the subgoal trie. Because some attributed variables (e.g. `A3` in Fig. 2(a)) in a subgoal call might not be changed in the answer, there is no need to construct these attributed variables again in the answer trie. Instead, we want to share them between the subgoal trie and answer trie. This can be achieved by initializing the array *VarEnumerator* and *var_ctr* in a special way before we copy an answer into the answer trie (see [3]). We reserve the first k elements of *VarEnumerator* for all the attributed variables in the call (assuming there are a total of k attributed variables in the call), and number all the new variables in the answer starting from $k + 1$. By doing so, we can use only one node, say v_i $(1 \leq i \leq k)$, to represent an unchanged attributed variable in the call.

For example, in the program and query shown in Fig. 2, there are three attributed variables in the call of `p(X,A1,A2,A3,A2)`: `A1`, `A2`, `A3`, among which `A3` is not changed in the answer. Therefore, the first 3 elements of *VarEnumerator* are reserved for the three attributed variables, and *var_ctr* is initialized as 3. New variables in the answer, `A2` (= `A4`) and `New`, are numbered 4 and 5 respectively (see the nodes $\widehat{v_4}$ and $\widehat{v_5}$ in Figure 2(c)). The unchanged attributed variable `A3` is numbered 3 in the answer trie (since it is the third attributed variable in the call) and represented by a single node v_3.

[1] The predicate `put_attribute`(+*Var*,+*Attr*) is a newly added built-in predicate. It changes *Var* to an attributed variable with attribute *Attr* if *Var* is a regular variable, or updates the attribute of *Var* to *Attr* if *Var* is already an attributed variable.

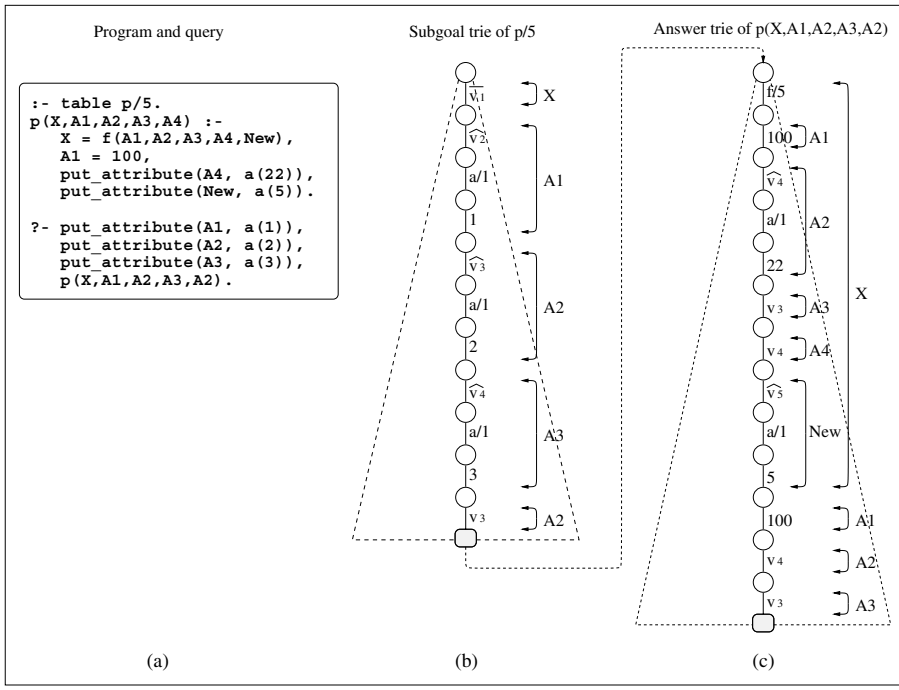


Fig. 2. An example of a subgoal trie and answer trie

3 Tabling and Constraints: Necessary Operations

To efficiently apply tabling to constraint logic programs, some general operations on constraints are needed from the constraint solver, e.g *projection* (or *approximate projection* [2]) and *entailment checking*, though they are not so important in a normal CLP system. Currently we only consider *ideal* CLP systems [14], i.e. we assume that complete algorithm are available for operations of satisfiability checking, projection, and entailment checking. For the constraint solvers in which projection operation is hard (or impossible) to get (e.g. the constraint solver over finite domains), approximate projection can be used but the completeness is not guaranteed [2].

3.1 Projection

In the execution of a query Q in a TCLP program, each call of a subgoal G (an atom) is associated with a set of constraints, a subset of constraints of the current constraint store which includes all the constraints accumulated so far since the beginning of the execution of Q. The basic idea of tabling is to try to avoid recomputing the subgoal G if it has been called in the same (or similar)

environment before, where the environment of G is the related constraint set. Each call of a subgoal and the associated constraint set are stored in the table and act as an index. Therefore, the need for the projection of a set of constraints onto a finite set of variables appears. This operation is indeed necessary whenever putting a call or putting an answer into the table. When putting a call into the table, we want to restrict the constraint set to contain only related constraints, the constraints which contain only the variables in the called subgoal; when putting an answer into the table, it is also necessary to project out variables and constraints introduced during the subcomputation.

3.2 Entailment Checking

In a TCLP system, the operation to check if a set of constraints is entailed by another set of constraints is required for several purposes. Firstly, before a call is put into the table, we have to make sure that no call in the table is equivalent to this call, i.e., their associated constraint sets are not equivalent. Two sets of constraints are equivalent if they can be entailed by each other[2].

Secondly, when a subgoal is called, only checking if it has been called in *exactly the same* environment as before cannot fully utilize the power of tabling. It not only requires more table space, but also forces some unnecessary redundant computation. This is because, if a previous call can be entailed by the current call (i.e., the previous call subsumes the current call), then the answer of this previous call can be consumed by the current call, and it is possible that there is no need to recompute the current call. For example, if a call $p(X) \wedge \{X > 5\}$ is already in the table and an answer, $X = 10$, has been returned, then a new call like $p(X) \wedge \{X > 7\}$ can immediately use the answer $X = 10$ in the table, since the constraint $\{X > 5\}$ is entailed by $\{X > 7\}$.

Thirdly, before a new answer of a call is saved into the table, it has to be guaranteed that no duplicate answers are stored in the table. More generally, if there is already an answer in the table and it is entailed by the new answer, then the new answer can be discarded.

4 Our Solution: A General Framework

Given the necessary operations on constraints, we construct a general framework for building a TCLP system. This framework is domain independent and is parameterized by the constraint operations. The implementations of the domain dependent operations themselves are left to the developers of the different constraint solvers.

Basically this framework sets up an interface among the tabling engine of XSB, the constraint solver, and the CLP programs, and the purpose is to give the user more control over the tabling engine. Generally, this interface can be

[2] In some constraint solvers which always keep constraints in a canonical form, this operation may not be required, since two equivalent constraints are always identical.

divided into two parts. In the first part, the user tells the XSB engine what kind of calls should be stored in the subgoal table. The user could generalize a call even if it has not been seen before. If a more general call has been called before, a new call which is subsumed by this old call should, in general, not be put into the subgoal table. Instead it should consume the answers for the more general call. In the second part, the constraint solver tells the XSB engine what kind of answers should be considered new and put into the answer table. This can be done in a similar way in which tabled aggregation is implemented (we assume some familiarity with the implementation of aggregation in XSB, see [20]).

The interface consists of three *interface predicates*. The first two of them are provided by the constraint solver:

1. `projection(+TargetVars)`
   This is the constraint projection operation. It projects the current constraint store over a set of variables, `TargetVars`.
2. `entail(+Answer1, +Answer2)`
   This predicate is defined using the entailment checking operation of the constraint solver. Given two answers, `Answer1` and `Answer2`, to the call of a tabled predicate, this predicate checks if the first one entails the second one. We assume that answer constraints are represented within the answers using attributed variables.

The third interface predicate is:

3. `abstract(+OrigCall,−NewCall,−Constraints)`

which is the most complicated one. It controls what kind of calls of a tabled predicate should be stored in the subgoal tables, i.e., it abstracts the call, `OrigCall`, of a tabled predicate *Pred* to a more general call, `NewCall` (which has a new predicate, *TPred*), and only stores `NewCall` in the subgoal tables. Basically `abstract/3` relaxes the constraints related to `OrigCall`, and stores the constraints that are not passed to `NewCall` into `Constraints`. In other words, `OrigCall` is equivalent to the conjunction of `NewCall` and `Constraints`.

Just as XSB performs variant tabling and subsumptive tabling on pure Prolog [20], our framework supports two different kinds of basic call abstractions. The first one is called *variant abstraction*, which is the default one and does not actually do any useful abstraction. In this case, the arguments of `NewCall` are the same as the arguments of `OrigCall`, and `Constraints` is an empty list `[]`. So every different call pattern of *Pred* is stored in the table. The second kind of abstraction is called *subsumptive abstraction*. In this case, whenever a call, `OrigCall`, of predicate *Pred* is made, `abstract/3` looks up the current subgoal table of *TPred* to see if a more general call than `OrigCall` has been called before (using the projection and entailment checking operations from the constraint solver). If there is, then the more general call is returned in `NewCall`. Otherwise, `NewCall` just takes the arguments of `OrigCall`, and `NewCall` is saved into the subgoal table.

As long as projection and entailment checking operations from the constraint solver are correct and complete, these two kinds of call abstraction guarantee the correctness and completeness of user's programs. However, they are not always the best call abstraction, and sometimes the user might want to overwrite the system defined `abstract/3` if she knows more about the call patterns of some predicates. This is allowed in our framework, but the user must be aware that it is the user's responsibility to keep the correct semantics and the correctness of the programs.

Having the three interface predicates defined, we can transform the user's program so that it can make more use of the table. The transformation and how the interface works can be explained by the following example:

Example 2. Given a tabled constraint logic program P as shown in Fig. 3, which contains only one tabled predicate `p/n` and has two clauses for it, we can transform it into the program shown in Fig. 4. Without losing generality, we assume `X1`, ..., `Xn` are (attributed) variables.

In the new program, we introduce a new tabled predicate `'_$tabled_p'`/n, and rewrite the clause of `p`/n, so that the call of `p`/n is abstracted by `abstract/3` first (line 04), and then the abstracted new call (of predicate `'_$tabled_p'`/n) is called. The constraints abstracted out by `abstract/3` are put back into the constraint store and solved (by `solve/1`, line 07) before any answer is returned to the original call of `p`/n.

The new predicate `'_$tabled_p'`/n is defined similar to the tabled aggregation predicate `bagPO/3` in XSB. Lines 10 and 11 are two internal predicates to get the return skeleton (the substitution factor) of the current call. The predicate `entail/2` is used as a partial order operator to keep only the most general answers in the answer table (some older answers might be reduced by the new answer). Also, before a new answer is put into the answer table, the projection operation is called (by `projection/1`, line 25) so that only related constraints are stored in the answer table.

We have to point out that, in some programs, especially those to search the optimal solution of some problems, there is only one (i.e. the best) answer for a call, which is returned by combining the old answer in the table with the new answer. In this case, the predicate `'_$tabled_p'`/n can be defined similar to `bagReduce/4` in XSB [20], and the interface predicate `entail/2` has to be substituted by a predicate like `reduce/3`.

5 A Real Example

Based on the above design, we have built a tabled constraint logic programming system over the domain of real numbers. The constraint solver is the partially ported version of clp(Q,R) written by Christian Holzbaur [8], which is implemented using attributed variables.

In this section, by giving the example of the shortest distance problem, we show how to write a tabled constraint program in this TCLP system, and how the user's program is transformed.

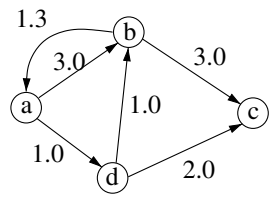
```
:- table p/n.

p(X1,...,Xn) :- Body1.
p(X1,...,Xn) :- Body2.
```

Fig. 3. Original program P

```
01  :- table '_$tabled_p'/n.
02
03  p(X1,...,Xn) :-
04          abstract(p(X1,...,Xn),
05                   '_$tabled_p'(NewX1,...,NewXn), Constraints),
06          '_$tabled_p'(NewX1,...,NewXn),
07          solve(Constraints).
08
09  '_$tabled_p'(X1,...,Xn) :-
10          '_$savecp'(Breg),
11          breg_retskel(Breg,n,Skel,Cs),
12          copy_term(p(X1,...,Xn,Skel), p(OldX1,...,OldXn,OldSkel)),
13          '_$orig_p'(X1,...,Xn),
14          ((get_returns(Cs,OldSkel,Leaf),
15            entail(p(OldX1,...,OldXn),p(X1,...,Xn)))
16           ->       fail
17           ;        (findall(t(Cs,OldX1,...,OldXn,Leaf),
18                             (get_returns(Cs,OldSkel,Leaf),
19                              entail(p(X1,...,Xn),p(OldX1,...,OldXn))),
20                             List),
21                    member(t(Cs,_,...,_,Leaf),List),
22                    delete_return(Cs,Leaf),
23                    fail
24                    ;
25                    projection([X1,...,Xn]),
26                    true
27                    )
28          ).
29
30  '_$orig_p'(X1,...,Xn) :- Body1.
31  '_$orig_p'(X1,...,Xn) :- Body2.
```

Fig. 4. New program transformed from the program in Fig. 3

Problem: *Given two nodes, X and Y, in the directed weighted graph of Fig. 5(a) (each edge is associated with a weight, the distance between the two nodes of the edge), find the shortest distance between X and Y.*



(a)

```
:- table sd/3.

sd(X,Y,D) :-
        edge(X,Y,D0),
        {D >= D0}.

sd(X,Y,D) :-
        sd(X,Z,D1),
        edge(Z,Y,D2),
        {D >= D1+D2}.
```

(b)

Fig. 5. Shortest distance problem

This problem can be solved by a TCLP program over the domain of real numbers shown in Fig. 5(b) (we omit the facts of `edge/3`). A call of `sd(+X,+Y,-Dist)` will return a sequence of answers, $D_1, \ldots, D_n$, for `Dist`, where each D_k ($1 \leq k \leq n$) is a constraint of the form `Dist >=` N_k. Each N_k is the *current achievable shortest* distance from `X` to `Y`, so we have $N_1 > N_2 > \ldots > N_n$, and finally N_n is the shortest distance from `X` to `Y`.

Since the call pattern of `sd/3` is general enough (see the second clause of Fig. 5(b), where the third argument `D1` is always a free variable), we can simply use the variant call abstraction. So the program shown in Fig. 5(b) is transformed to the program shown in Fig. 6 (the clause of `'_$tabled_sd'/3` is optimized for better performance).

Some running results of the transformed program are shown in Fig. 7. We can see that the shortest distance from `a` to `c` is 3.0, and the shortest distance from `a` to `b` is 2.0, all returned in the second answer. The shortest distance from `d` to `a` is 2.3, which is returned in the very first answer.

6 Conclusion and Future Work

As presented in this paper, we have introduced attributed variables into XSB and extended XSB with the basic mechanism to support constraint solving. By changing the data structure of subgoal table and answer table in XSB to support attributed variables, constraints can now be copied into and out of tables.

Based on such fundamental changes to the system, we constructed a general framework for building a Tabled Constraint Logic Programming System extending the tabling engine of XSB. This framework is domain independent, and it provides an interface among the XSB engine, the constraint solver, and the user's

```
:- table '_$tabled_sd'/3.

sd(X,Y,D) :-
        abstract(sd(X,Y,D),'_$tabled_sd'(X,Y,D), Constraints),
        % For variant abstraction, Constraints == []
        '_$tabled_sd'(X,Y,D),
        solve(Constraints).

'_$tabled_sd'(X,Y,D) :-
        '_$savecp'(Breg),
        breg_retskel(Breg,3,Skel,Cs),
        copy_term(p(X,Y,D,Skel),p(OldX,OldY,OldD,OldSkel)),
        '_$orig_sd'(X,Y,D),
        ((get_returns(Cs,OldSkel,Leaf), OldX == X, OldY == Y)
         ->     (entail(sd(X,Y,D), sd(X,Y,OldD))
                 ->     delete_return(Cs,Leaf)
                 ;      fail
                )
         ;      projection([D]),
                true
        ).

'_$orig_sd'(X,Y,D) :-
        edge(X,Y,D0),
        {D >= D0}.
'_$orig_sd'(X,Y,D) :-
        sd(X,Z,D1),
        edge(Z,Y,D2),
        {D >= D1+D2}.
```

Fig. 6. Transformed version of the program in Fig. 5(b)

`| ?- sd(a,c,Dist).`	`| ?- sd(a,b,Dist).`	`| ?- sd(d,a,Dist).`
`Dist >= 6.0000;`	`Dist >= 3.0000;`	`Dist >= 2.3000;`
`Dist >= 3.0000;`	`Dist >= 2.0000;`	`no`
`no`	`no`	

Fig. 7. Running results of the shortest distance program

programs. The user's programs are transformed at the source code level using the interface predicates, so that the transformed programs can make more use of the tables. Experiments have been done on the domain of real numbers (using the clp(Q,R) [8]), and this framework has been proven to work.

Future work includes:

1. Integrate the framework with constraint solvers over other domains;
2. Explore better ways to represent constraints using attributed variables over different domains, so that constraints can be stored more efficiently in tries;
3. Move some of the program transformation work into the engine level to improve the performance;
4. Apply our system to symbolic bisimulation of infinite-state systems, a formal verification problem requiring tabling and constraints [15].

Acknowledgements

This work is partially supported by NSF Grants CCR-9702681, CCR-9705998, and CCR-9711386.

References

1. P. Brisset, et al. *ECL^{i}PS^e 4.0 User Manual*. IC-Parc at Imperial College, London, July 1998.
2. Philippe Codognet. A tabulation method for constraint logic programs. In *Proceedings of INAP'95, 8th Symposium and Exhibition on Industrial Applications of Prolog*, Tokyo, Japan, October 1995.
3. Baoqiu Cui and David S. Warren. Attributed variables in XSB. In *Proceedings of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 61–74, Las Cruces, New Mexico, USA, December 1999.
4. The XSB Group. The XSB logic programming system, version 2.0, 1999. Available from `http://www.cs.sunysb.edu/~sbprolog`.
5. Christian Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, 1990.
6. Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *International Symposium on Programming Language Implementation and Logic Programming*, LNCS 631, pages 260–268. Springer Verlag, August 1992.
7. Christian Holzbaur. Extensible unification as basis for the implementation of CLP languages. In Baader F. and et al., editors, *Proceedings of the Sixth International Workshop on Unification*, TR-93-004, pages 56–60, Boston University, MA, 1993.
8. Christian Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.
9. Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, The Netherlands, 1998.

10. Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In *Proceedings of the 1999 International Conference on Principles and Pratice of Declarative Programming*, LNCS, Paris, France, September 1999. Springer Verlag.
11. S. L. Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Maluszyński, editors, *International Symposium on PLILP*, LNCS 456, pages 136–150. Springer, Berlin, Germany, August 1990.
12. Mark Johnson. Memoization in constraint logic programming. In *Proceedings of PPCP'93, First Workshop on Principles and Practice of Constraint Programming*, Rhode Island, Newport, April 1993.
13. The Intelligent Systems Laboratory. *SICStus Prolog User's Manual Version 3.7.1.* Swedish Institute of Computer Science, October 1998.
14. Fred Mesnard and Sèbastien Hoarau. A tabulation algorithm for CLP. In *Proceedings of International Workshop on Tabling in Logic Programming*, pages 13–24, 1997.
15. Madhavan Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and Rakesh Verma. Symbolic bisimulation using tabled constraint logic programming. Submitted to TAPD'2000. Available from `http://www.cs.sunysb.edu/~cram/papers/`.
16. Bernhard Pfahringer and Johannes Matiasek. A CLP schema to integrate specialized solvers and its application to natural language processing. Technical report, Austrian Research Institute for Artificial Intelligence, Vienna, 1992.
17. P. Rao, I. V. Ramakrishnan, K. Sagonas, T. Swift, and D. S. Warren. Efficient table access mechanisms for logic programs. In L. Sterling, editor, *ICLP*, pages 697–711, 1995.
18. Prasad Rao. *Efficient data structures for tabled resolution.* PhD thesis, SUNY at Stony Brook, 1997.
19. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference*, 1994.
20. K. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. *The XSB Programmer's Manual: version 1.9*, 1998.
21. H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings of the 3rd ICLP*, pages 84–98, 1986.
22. D. Toman. Top-down beats bottom-up for constraint extensions of datalog. In *Proceedings of International Logic Programming Symposium ILPS'95.* MIT Press, 1995.
23. D. S. Warren. Memoing for logic programs with applications to abstract interpretatino and partial deduction. *Communications of the ACM*, 1992.