RISC-CLP(Tree(Δ))
A Constraint Logic Programming System with Parametric Domain*

Hoon Hong and Stefan Ratschan

May 11, 1995

Abstract

Current implementations of constraint logic programming languages (like CLP($\Re$), CHIP or RISC-CLP(Real) support constraint solving over a certain fixed domain. In this paper a system is presented which gives the possibility to instantiate a constraint logic programming language with an arbitrary constraint domain. The interface between the system and such a constraint domain is given and the extensions to the Warren Abstract Machine (WAM) for this implementation are presented.

1 Introduction

There already exist several implementations of constraint logic programming languages which are highly optimized for a certain constraint domain (for example PROLOG III [9], CHIP [1], CLP($\Re$) [16], CLP(FD) [23], LIFE [4]), RISC-CLP(Real) [12], RISC-CLP(R-Trees) [7] and RISC-CLP(CF) [13]). An alternative approach would be a system which does not involve a fixed constraint domain but is parametric in that it can be instantiated with an arbitrary constraint domain. There already exist several systems which provide such funcionality, for example CLPS ([18]), ECLiPSe ([10], [5]) or CAL ([22]).

In this paper we report another system that we developed. The name RISC-CLP(Tree(Δ)) indicates that constraint solving over symbolic trees over some arbitrary user-defined constraint domain (Δ stands for domain) is supported.

The RISC-CLP(Tree(Δ)) system provides interfaces to constraint domains (= constraint compiler + constraint solver) on several abstraction levels such that the user can first implement a simple but inefficient prototype and then refine it to more complicated but gefficient implementation.

Common methods for implementing constraint logic programming systems include Meta-interpreters (like [8]), interpreters (like presented in [16]) and compilers (CLP($\Re$) [15] or CHIP [2]) which extend the WAM for handling constraints over certain domains.

Existing systems with exchangeable constraint solvers are either interpreter systems and/or they use a predefined syntax for the constraints whereas we gain efficiency by using a compiler/emulator system and flexibility by allowing user- defined constraint syntax and internal constraint representation.

In [18] a constraint logic programming shell (CLPS) is described, which also is a CLP system which can be instantiated by arbitrary constraint domains. CLPS is more designed to be a rapid prototyping system for pre- existing constraint solvers like the ones in computer algebra systems, whereas our system aims more at efficiency using constraint solvers written in traditional programming languages and computer algebra libraries (like SACLIB [6] and Sturm [14]). CLPS also provides abstraction levels (3 levels in one dimension vs. 6 levels in two dimensions in our system) and constraints are stored as normal PROLOG structures and

*This research was done in the framework of the ACCLAIM project sponsored by ESPRIT, BRA and FWF.

have to be transformed into the data structure of the constraint solver each time it is called, whereas our system cleanly separates the logic and the constraint domain and also allows the user to design one's own optimal byte- code for representing the constraints. Furthermore in our system one is not restricted to the predefined class of PROLOG constants but can define a class of constants on ones's own.

Another method has been implemented in ECLiPSe ([10], [5]), with the concept of *meta-terms*, which can be looked at as a variable with (one or more) attached attributes. Metaterms behave in some respects like variables, but in most respects their behaviour is user-definable (e.g. what happens with them on unification, etc.) ([20], [19] and [11]). By this method one can integrate constraint solvers into a logic programming system. The drawback of this method is that one is still restricted to PROLOG syntax (including constants). Furthermore the concept is very general and provides no clear interface designed especially for constraint solvers.

CAL is an example of a CLP system which allows the user to include one's own constraint solver but is an interpreter whereas our system uses a compiler ([22]).

The paper is structured as follows: The next section gives an introductory example, the third section defines the interface by which constraint domains can be linked to the system and the last section gives the internal design including the extensions introduced to the WAM.

2 Introductory Example

In this section, we give an overview of the whole cycle of instantiating RISC-CLP with a particular domain. In order to help intuition we work with the simple example domain *Bool* which allows us to formulate equations and disequations over the booleans with a function symbol defining negation. This domain allows us to formulate programs like the one of figure 1, describing the toy problem of discussing with four kids whether they want to go out, when Tom wants to stay with Maria, Michael does not want to be with John, Michael does not want to stay with Tom if John goes with Maria and Tom definitely wants to go.

```
onlyif(X, Y, A, B):- X=Y, A=B.

onlyif(X, Y, A, B):- X!=Y, A!=B.

?- Tom = Maria, Michael != John, onlyif(Michael, not(Tom), John, Maria),
   Tom = true.
```

Figure 1: Example Program

So if one instantiates RISC-CLP(Tree(Δ)) with a description of the constraint domain *Bool* to get RISC-CLP(Tree(*Bool*)) and then runs this program one gets the answer that Tom and Maria and either one of John and Michael go out.

To get the desired CLP system one first defines the syntax of *Bool* (as shown in Figure 2).

```
Relation Symbols:
"!=" NOTEQ 2

Function Symbols:
"not" NOT 1

Constants:
(true|false)
```

Figure 2: Constraint Domain Syntax Definition

The first field in the relation and function symbol definitions is the external representation, the second field is the internal symbolic name for it and the third field is the arity of the symbol. The equation symbol is predefined by the system and given the name `EQUALS`. Constants are defined by giving a regular expression.

What remains to be defined now is the constraint solver itself. This can be done on several abstraction levels, depending on whether one wants a prototype which is fast to program but slow in runtime or a more complex but faster implementation. As we want to show an easy example here we chose the first option.

A constraint solver has to be defined as a `C++` class. Figure 3 gives the class definition (a simplified version of the `C++` header file).

```
class Bool {
  Bool();                                    constructor (initialization)

  DeltaTerm buildConst(int c);               constraint construction
  DeltaTerm buildNewVar();
  DeltaTerm buildFunc1(int funcSym, DeltaTerm arg1);

  solve2(int relSym, DeltaTerm lhs, DeltaTerm rhs);
  int isConsistent();                        constraint solving

  char *dumpTerm(DeltaTerm t);               textual representation
  char *dumpConstraints(void);

  copy(Bool source, int isNewActive);        backtracking
  int hasChanged();
  changePoint();
};

typedef BTDelta<Bool> _DeltaCS;
```

Figure 3: Constraint Solver Header File

The `build` functions construct constants, new variables and function symbols respectively. The `solve2` function adds a 2-ary constraint to the constraint store, `isConsistent` returns its consistency. The `dumpTerm` function returns a character string representing one single term (if there is one), while `dumpConstraints` returns a character string representing the whole constraint store. The `copy` function copies the state of the constraint solver, `hasChanged` returns a non-zero value if the constraint solver state has changed since the last call to `changePoint`. These functions are used by the system to do backtracking. They are translated into lowlevel calls by the `BTDelta` template class (see last line of figure 3).

Figure 4 gives the implementation (all the member functions of the `Bool` class). Its idea is as follows: The constraint solver maintains an equivalence relation on the variables. Whenever two variables occur in a constraint this says that they are in the same equivalence class. Every equivalence class is divided into two disjoint sets (say every member of a class has one of two colors). The equality constraint says that two variables have the same color, the disequality constraint says that they have different colors. The constraint store gets inconsistent as soon as a variable is colored with two different colors. There is a predefined equivalence class with the two members `TRUE` and `FALSE`, which have different color.

The equivalence classes are implemented as doubly linked lists (all elements are stored in the array `var`). A list element contains two array indices pointing to the equivalent variables and two fields which are `0` if the according linked variable has the same color and `1` if it has different color.

After telling the RISC-CLP system which abstraction level has been used one just has to type in `make all` which recompiles the whole system. After this has been finished one has a

```
#include "deltaCS.h"

Bool::Bool()
```
set up predefined equivalence class with "false" and "true"

```
DeltaTerm Bool::buildConst(int c)
```
*return 1 if (char *)c points to "true", 0 otherwise*

```
DeltaTerm Bool::buildNewVar()
```
return number of a new variable, which is in no current equivalence class

```
DeltaTerm Bool::buildFunc1(int funcSym, DeltaTerm t)
```
build new var in same equivalence class as t which has different color from t

```
Bool::solve2(int relSym, DeltaTerm arg1, DeltaTerm arg2)
```
tells that arg1 and arg2 are in the same equivalence class.
If(relSym==EQUALS) they have the same color, otherwise different color
If they are already in the same class but have different color
than required then the constraint store gets inconsistent.

```
int Bool::isConsistent()
```
return consistency of constraint store

```
char *Bool::dumpTerm(DeltaTerm t)
```
if t is equivalent to true and false return "TRUE" or "FALSE" according
to negation along doubly linked list

```
char *Bool::dumpConstraints()
```
return string describing all equivalence classes and their colorings

```
Bool::copy(Bool source, int isNewActive)
```
copy source into this object (isNewActive can be ignored here)

```
Bool::changePoint()
```
see hasChanged()

```
int Bool::hasChanged()
```
return a non-zero integer if the state of the constraint store has
not changed sinze the last call to changePoint

Figure 4: Constraint Solver Implementation

ready CLP system which uses the defined constraint domain.

So if one runs the program from figure 1 one gets the output of figure 5.

```
RISC-CLP(Bool):
Compilation started
Successful compilation (CodeSize: 204)

Program terminated normally:
Tom: TRUE
Maria: TRUE
Michael: FALSE
John: TRUE
Delta- Constraints:

More? (Y/N)y

Program terminated normally:
Tom: TRUE
Maria: TRUE
Michael: TRUE
John: FALSE
Delta- Constraints:

More? (Y/N)y

Failed
```

Figure 5: Example Session

We have implemented further constraint domains, for example a domain using Involution method ([25]), which uses a special type of Groebner bases for deciding the consistency of systems of linear and nonlinear equations over the complex numbers and computing a normal form. This constraint solver has been implemented by Andreas Neubacher as a part of the Sturm project ([14]).

3 Interface to Constraint Domain

In this section we describe the interface between the RISC-CLP system and constraint domain descriptions. An instantiated RISC-CLP system has two parts:

Compiler: Gets a RISC-CLP source file as input and produces intermediate code.

Emulator: Executes the code produced by the compiler.

This means that a constraint domain has to be defined in two parts (see figure 6):

Constraint compiler: Translates constraints into bytecode. Consists of a file defining the syntax of the constraint domain and a function for compiling constants or terms (this function has not been shown in the introductory example because it is trivial there).

Constraint solver: Is called by the emulator and does the constraint solving. It gets the translated constraints as input.

The system was designed to achieve the following aims:

- Simple/easy to use: The interface between the RISC-CLP system and user-defined constraint domain should be so simple that even users that do not know anything about implementation of CLP- systems can understand and use it within a few minutes.

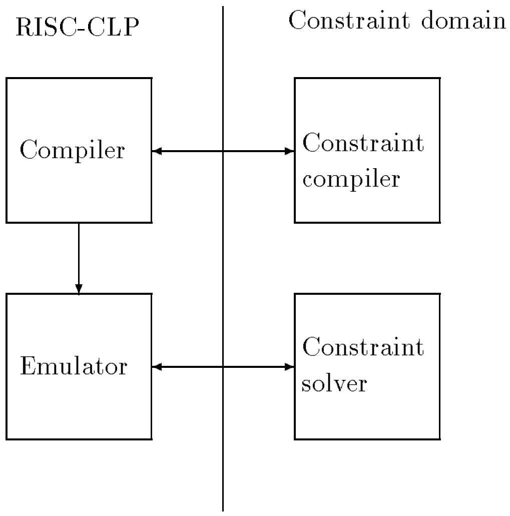

Figure 6: RISC-CLP Overall Structure

- Efficient: The runtime cost of the interface should be as small as possible.

In order to achieve simple usability the system was designed such that the code for the constraint domain does not depend on any internal of the controlling system, which in turn does not assume anything about implementation details of the constraint domain. This concept is supported very well by the class concept of the object oriented language C++ which we used to implement the system.

As the number of variables is critical especially for the efficiency of non-linear solvers (which is especially important for us, see [12]), different from most CLP systems we do not do transformations of the following kind:

`f(N-1).` $\Rightarrow$ `f(_T):- _T=N-1.`

This saves some temporary variables, but introduces difficulties for the implementation. The implication of that concept is that our grain (for compilation and WAM- instructions) is not constraints but terms.

There is a trade- off between simple usage and efficiency. In order to deal with this the system provides several interface levels of different abstraction. These give the user the possibility to start with a simple (but inefficient) prototype implementation of a constraint domain which can then be refined up to a level where the whole system runs almost as fast as a compiler with a fully- integrated constraint solver (like the systems presented in [1], [2] or [15]).

The abstraction level can be changed independently on two different aspects of the interface, which means that one can move arbitrarily on a 2- dimensional grid. The following aspects of the interface can be changed.

- Constraint construction: Should the constraint data structure for the constraint solver be built at runtime, before runtime, or at compile time? The earlier this is done, the more information about how to do this has to be provided by the user (for example the binary representation of constraints), which in turn gives more flexibility and more optimization possibilities and efficiency.

- Backtracking: Should backtracking be automatically done by the system?

On each abstraction level three files have to be provided for defining the constraint domain:

Syntax definition: A file in which all the relation symbols, function symbols and constants have to be defined. The structure of this file is shown the example of figure 7 (the equation symbol is predefined).

```
Relation Symbols:
"!=" NOTEQ 2

Function Symbols:
"+" PLUS 2 left
"-" MINUS 2 left <
"^" EXP 2 right
"sin" SIN 1

Constants:
[1-9]*
```

Figure 7: Syntax definition file

After the self- explanatory keywords the symbols can be defined. Relation and function symbol definitions consist of a string in the first column defining their representation and then the arity of the symbol. Two- ary function symbols can have an additional field with the keywords `left`, `right` or `nonassoc` which define the associativity of the symbols. If the last symbol on a line defining a function symbols is <, this means that the following symbols have higher precedence.

The generated parser parses constraints in functional prefix notation and in infix notation for 2-ary symbols (both prefix and infix can be used).

For constants only one line can be given. It is a `lex` compatible regular expression ([17]) which scans a constant.

Constant/Term compilation: Here a function has to be defined which compiles a constant (or term if compile time constraint construction is used) into bytecode. For constants, a string matching the regular expression given in the syntax definition file, is passed as a parameter - terms are represented by their syntax tree.

```
void deltaConstGenerate(char *text, int &length, char *&buf);
```

or

```
void deltaTermGenerate(DeltaTerm *term, int &length, char *&buf);
```

Constraint solver: A `C++` class containing the following kinds of functions:

Constructor: If necessary, this function initializes the constraint solver.

Constraint solving: The following functions should be provided for adding constraints to the constraint store:

```
void solve0(int relSym);
void solve1(int relSym, DeltaTerm arg1);
void solve2(int relSym, DeltaTerm arg1, DeltaTerm arg2);
void solve3(int relSym, DeltaTerm arg1, DeltaTerm arg2, DeltaTerm arg3);
```

Only those functions for which there are relation symbols of according arity need be provided.

The function

```
int isConsistent(void);
```

should return 0 if the constraint store is known to be inconsistent. It should return a non-zero value otherwise.

Textual constraint representation: The function `dumpTerm` should return the character representation of a single term (if there is one). The function `dumpConstraints` should return the textual representation of the whole constraint store.

Constraint construction:

The following functions are used:

```
DeltaTerm buildFunc0(int funcSym);
DeltaTerm buildFunc1(int funcSym, DeltaTerm arg1);
DeltaTerm buildFunc2(int funcSym, DeltaTerm arg1, DeltaTerm arg2);
DeltaTerm buildFunc3(int funcSym, DeltaTerm arg1, DeltaTerm arg2, DeltaTerm
arg3;)

DeltaTerm buildNewVar(void);
DeltaTerm buildVar(int v);
DeltaTerm buildConst(int c);

int computeVariant (void *term, unsigned int *bindingTable);
```

Depending on which constraint construction aspect is used, a certain subset of the above functions has to be defined (only those buildFunc functions have to be defined for which there are function symbols of according arity).

At Runtime: The functions `buildConst`, `buildNewVar` and `buildFunc` are used to build up the constraints at runtime. If constraints should be simplified or transformed before code generation then the user can (optionally) define a function `simplify()` (in a seperate file), which can transform the syntax tree of a constraint (e.g. transform polynomials into distributive notation for a Groebner basis constraint solver).

Before Runtime: The functions `buildConst`, `buildVar` and `buildFunc` are used to build up the terms of the constraints before runtime (with unbound variables). The actual variants of these terms are then built at runtime by the `computeVariant` and `buildNewVar` functions. The second function is called for every unbound variable in a constraint.
The `bindingTable` argument to the `computeVariant` function gives the actual instantiation of each variable in a constraint term (can be the result of an earlier `computeVariant` call or a variable over the constraint domain, which has been computed by `buildNewVar`). The position of each variable in the binding table is given as an argument to the `buildVar` function when the templates are built. This has the following advantages and disadvantages:

- Copying the template into a new data structure can sometimes be costly (recursion).
- The constraint solver can use the fastest internal representation at runtime (e.g. flat polynomial representation).

At Compile Time: The `computeVariant` and `buildNewVar` functions are used to build the term variants of the byte code produced by the `deltaTermGenerate` function. This has the following advantages and disadvantages:

- Optimal bytecode for terms can be used (highly efficient).
- Transformation of terms at compile time possible (this can also be done with the simplification function of the last section).
- The user has to care about byte code.

Backtracking:

Automatic Backtracking: The following functions have to be defined:

```
void copy(const Bool& dummy, int isNewActive);
int hasChanged(void);
void changePoint(void);
```

The `copy` function copies the state of dummy into the constraint solver. The argument `isNewActive` is non-zero of it is the new active constraint solver. The `hasChanged` function returns a non-zero value if the constraint solver state has changed since the last call to `changePoint`.

User-Defined Backtracking: In order to to implement backtracking by oneself, one has to provide the following three member functions of the constraint solver class, create and remove choicepoints and backtrack to the last choicepoint:

```
void createCP(void);
void removeCP(void);
void backtrack(void);
```

This concept of choicepoints is more complicated than the one used in [18] but also more efficient.

4 Design

In this section we describe the internal design of the RISC-CLP system. As shown by figure 6 it consists of a compiler and an emulator. The compiler builds up a syntax tree of the whole program, where some of the leaves call the functions for compiling constants (or terms when using compile time constraint construction).

The emulator is an implementation of an extension of the Warren Abstract Machine (see [24] or [3]), where the WAM has been extended with:

Constraint construction stack: A stack on which constraints are built. The abstract data type operations `push` for pushing a new object onto the stack and `pop` for removing the last pushed object from the stack are used in this section.

Term table: An array for storing references to terms. It is needed because term representing values returned by the constraint solver have to be tagged. An alternative approach would be to restrict the term identifiers returned by the constraint solver to not to use the tag bits.

New tag: An additional tag `deltaTerm` is used. Storage cells using this tag store an index into the term table.

The unification algorithm has to be extended according to the following table:

X / Y	*unbound*	Δ-*term*	otherwise
unbound	bind higher to lower	bind X to Y	
Δ-*term*	bind Y to X	if X$\neq$Y, solve X=Y	fail
otherwise		fail	compare

Solving an equation means that the equation is passed to the constraint solver which should then consider the two terms as equal (by calling `deltaCS.solve2(0, termTable[unTag(X)], termTable[unTag(Y)])`).

The constraint construction stack and the term table have to be registered to the constraint solver if the solver used garbage collection and returns pointers to internal objects in the `build` functions.

Alternative approaches would be to let the constraint construction stack be in the constraint solver (no garbage collection problems) or to construct constraints not on a stack but in registers (could be faster for native code compilation).

4.1 Extension of the WAM for runtime constraint construction

The instructions of figure 8 build up the constraints on the constraint construction stack (the operation tag tags the first argument with the second argument, deltaCS is the object representing the constraint solver):

The `build_variable` instruction is used instead of `build_value` whenever it is sure that the argument variable is unbound. The instructions have to be generated in postfix sequence. For example the term `2*U+V` is compiled like that:

```
build_const constant

       stack.push(deltaCS.buildConst(constant));

build_value Vi

       if(unbound(Vi)) {
         termTable[numTerms]= deltaCS.buildNewVar();
         stack.push(termTable[numTerms]);
         Vi= tag(numTerms++, deltaTerm);
       } else if(isDeltaTerm(Vi)) {
         stack.push(termTable[unTag(Vi)]);
       } else backtrack();

build_variable Vi

       termTable[numTerms]= deltaCS.buildNewVar();
       stack.push(termTable[numTerms]);
       Vi= tag(numTerms++, deltaTerm);

build_func/arity funcSym

       argi= stack.pop();
       stack.push(deltaCS.buildFunc(funcSym, arg1, arg2, ...));
```

Figure 8: WAM- Instructions for Constraint Building

```
build_const1 2
build_value U
build_func/2 *
build_value V
build_func/2 +
```

The `solve` instruction takes some constraint terms from the stack and passes them as a constraint to the constraint solver. The `put_term_s`, `get_term_s` and `unify_term_s` instructions do unification of the terms (see figure 9).

For example the clause `h(2).` is compiled as follows:

```
build_const1 2
get_term A1
```

Equations are compiled as if the left side would occur in a head/goal matching (which means that a constraint is only passed to the constraint solver if both sides consist of a term over the constraint domain which is not a single unbound variable, which implies that less constraint variables are generated as if equations would be handled directly by the constraint solver).

4.2 Extensions for constraint construction before runtime

Here templates representing the terms are built by the `build` instructions of the last chapter except that variables are built by the following instruction:

```
build_varnum num

       stack.push(deltaCS.buildVar(num));
```

```
solve_s/arity relSym

       argi= stack.pop();
       if(!deltaCS.solve(relSym, arg1, arg2, ...)) backtrack();

put_term_s Ai

       termTable[numTerms]= stack.pop();
       Ai= tag(numTerms++, deltaTerm);

get_term_s Ai

       termTable[numTerms]= stack.pop();
       if(!unify(Ai, tag(numTerms++, deltaTerm))) backtrack();

unify_term_s 1

       termTable[numTerms]= stack.pop();
       if(!unify(structPtr++, tag(numTerms++, deltaTerm))) backtrack();
```

Figure 9: WAM- Instructions for Term Unification

These templates are then instantiated at runtime using a *binding table* which stores the actual variable bindings, where the argument of the `build_varnum` instruction is the address where this binding can be found in the table. The instructions of figure 10 are used for instantiating the terms:

Here `<term>` is a memory address which refers to the template representing that term.

For every variable occuring in a term in a `get/put/unify_term` instruction either a `pass_value` or a `pass_variable` instruction is generated before in order to build up the binding table. The `pass_variable` instruction has to be used instead of `pass_value` if the variable occurence in the term is the first occurence of this variable in the clause.

The terms of constraints (which can be full terms or single variables) are stored in an *argument table*, which is done by the instructions of figure 11.

4.3 Extensions for constraint construction at compile-time

Here no templates are built but user- defined code is used. This means that no `build` instructions are generated but all the other instructions are used in a similar way. The term in the `get/put/unify/arg_term` instructions now is a pointer to a memory block containing the user- defined bytecode.

4.4 Choicepoint Handling

The `try` instructions have to be extended to call the `createCP()` member function of the constraint solver. The `trust` instructions call the `removeCP()` instruction and when backtracking, also the `backtrack()` member function has to be called.

5 Conclusion and Further Work

The constraint logic programming system RISC-CLP(Tree(Δ)) allows easy and efficient instantiation of a CLP system over user- defined domains. This domain can be defined on several abstraction levels depending on how much control the user wants to have and how efficient the solution should be. We gave the extension to the WAM which are needed to support our concept.

```
pass_value Vn, i

       if(unbound(Vi)) {
         termTable[numTerms]= deltaCS.buildNewVar();
         bindingTable[i]= termTable[numTerms];
         Vi= tag(numTerms++, deltaTerm);
       } else if(isDeltaTerm(Vn)) {
         bindingTable[i]=termTable[unTag(Vi)];
       } else backtrack();

pass_variable Vn, i

       termTable[numTerms]= deltaCS.buildNewVar();
       bindingTable[i]= termTable[numTerms];
       Vn= tag(numTerms++, deltaTerm);

put_term <term>, Ai

       termTable[numTerms]= deltaCS.computeVariant(<term>, bindingTable);
       Ai= tag(numTerm++, deltaTerm);

get_term <term>, Ai

       termTable[numTerms]= deltaCS.computeVariant(<term>, bindingTable);
       if(!unify(Ai, tag(numTerms++, deltaTerm))) backtrack();

unify_term <term>

       termTable[numTerms]= deltaCS.computeVariant(<term>, bindingTable);
       if(!unify(structPtr++, tag(numTerms++, deltaTerm))) backtrack();
```

Figure 10: WAM- Instructions for Term Instantiation

```
arg_value Vn, i

       if(unbound(Vi)) {
         termTable[numTerms]= deltaCS.buildNewVar();
         argTable[i]= termTable[numTerms];
         Vi= tag(numTerms++, deltaTerm);
       } else if(isDeltaTerm(Vn)) {
         bindingTable[i]=termTable[unTag(Vi)];
       } else backtrack();

arg_variable Vn, i

       termTable[numTerms]= deltaCS.buildNewVar();
       argTable[i]= termTable[numTerms];
       Vn= tag(numTerms++, deltaTerm);


arg_term <term>, i

       argTable[i]= deltaCS.computeVariant(<term>, bindingTable);

solve/arity relSym

       if(!deltaCS.solve(relSym, argTable[0], ..., argTable[arity-1]))
         backtrack();
```

Figure 11: WAM- Instructions for Argument Table

Possible future work includes:

- Automatize the interface further (like automatic constraint delaying).
- Compile a library of byte-code representations (the `deltaConstGenerate` and especially the `deltaTermGenerate` functions), such that new constraint solvers can be linked without having to define and implement a new byte-code representation.
- Like proposed in [15] the instruction set could be extended, such that:
  - The constraint solver is informed if a variable will not be needed again after its occurence and can therefore be dropped.
  - Constraints that get redundant in the future and act simply as a test will be passed to the constraint solver by a special instruction.
- If a variable occurs in two Δ- terms in the goals or head of the same clause then only one pass instruction needs to be generated, if the same binding table location is used in one clause. If different locations are used then a new instruction would just copy the values.

References

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP compiler system. In Koichi Furukawa, editor, *Proc. 8th International Conference on Logic Programming*, pages 775–789, Paris, France, June 1991. MIT Press.

[2] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP compiler system. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming - Selected Research*, pages 412–435. MIT Press, 1993.

[3] H. Ait-Kacï. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. MIT- Press, Cambridge, MA, 1991.

[4] H. Aït-Kaci and A. Podelski. Logic programming with functions over order-sorted feature terms. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming; Third International Workshop, ELP '92; Proceedings*, pages 100–119, Berlin, Germany, 1993. Springer-Verlag.

[5] Pascal Brisset and Thom Fruehwirth. *ECLiPSe Extensions User Manual*. ECRC, ECRC Gmbh., Arabellastr. 17, D-81925 Munich, Germany, 1994.

[6] Bruno Buchberger, George Collins, Hoon Hong, et al. Saclib 1.1. user's guide. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[7] Olga Caprotti. RISC-CLP(R-Trees): RISC-CLP(Real) handles symbolic functions. In *Proceedings DISCO*, 1993.

[8] Jacques Cohen, Pascal Koiran, and Catherine Perrin. Meta-level interpretation of CLP(lists)*. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming - Selected Research*, pages 457–481. MIT Press, 1993.

[9] A. Colmerauer. An introduction to PROLOG-III. *Commun. ACM*, 33(7):69–90, July 1990.

[10] ECRC, ECRC Gmbh., Arabellastr. 17, D-81925 Munich, Germany. *ECLiPSe User Manual*, 1993.

[11] Christian Holzbaur. A high-level approach to the realization of CLP languages. In *Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems*, Washington D.C., 1992.

[12] Hoon Hong. RISC-CLP(Real): Logic programming with non-linear constraints over the reals. In *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[13] Hoon Hong. RISC-CLP(CF): Constraint Logic Programming over Functions. In *Logic Programming and Automated Reasoning (LPAR'94)*, July 1994.

[14] Hoon Hong, Andreas Neubacher, Volker Stahl, et al. The Sturm library manual - a C++ library for symbolic computation. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1994.

[15] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

[16] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[17] M. E. Lesk and E. Schmidt. *LEX - Lexical Analyzer Generator*.

[18] Pierre Lim and Peter J. Stuckey. A constraint logic programming shell. In *Proc. 1990 International PLILP Workshop, Sweden*. Springer Verlag LNCS, 456, 1990.

[19] Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *3rd International Workshop on Extensions of Logic Programming*, pages 319–338, Bologna, 1992.

[20] U. Neumerkel. Extensible unification by metastructures. In *META90*, 1990.

[21] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):367–412, 1992.

[22] K. Sakai and A. Aiba. CAL: A theoretical background of CLP and its applications. *Journal of Symbolic Computation*, 8(6):589–603, December 1989.

[23] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Brown University, 1992.

[24] D.H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, October 1983.

[25] A. Yu. Zharkov and Yu. A. Blinkov. An involution approach to solving systems of algebraic equations. In G. Jacob, N.E. Oussous, and S. Steinberg, editors, *International IMACS Symposion on Symbolic Computation*, Lille, France, 1993.