Combining Partial Evaluation and Constraint Solving: A New Approach to Constraint Logic Programming

Jin-kao Hao † and Jean-Jacques Chabrier
Centre de Recherche en Informatique de Dijon
B.P.138
21004 Dijon Cédex, France

Abstract
Constraint logic programming (CLP) is a powerful paradigm combining constraint and logic programming. Existing CLP systems are built in a "tightly coupled" manner. We argue that CLP systems can be built "loosely" by separating a system into independent parts to avoid some repeated processing, and thus to increase further the efficiency of CLP systems. An architecture scheme is presented. System based on the architecture have desirable modularity. The system Conslog is described as an instance of the scheme. Conslog consists of two independent parts: a partial evaluator to carry out logical inferences and a solver in finite domains. The efficiency and behavior of the system are shown on a cryptarithmetic problem. The system is extensible.

1. Introduction
Over the past four years, considerable effort has been devoted to Constraint Logic Programming (CLP). CLP is a powerful paradigm combining constraint and logic programming. A formal framework for CLP systems has been identified [18]. Several operational prototypes, for example, Prolog III [2], CLP(3) [19] and Chip [6,7], have been reported. All of them introduce constraint solving techniques into the logic programming paradigm. The most significant feature of these systems is the greater expressive power and great increase in efficiency for solving some Constraint Satisfaction Problems (CSPs) [24]. At first view, we may consider these systems as some simple extensions of the Prolog system, because they employ almost the same syntax as Prolog. But in reality, there exist some fundamental differences between these systems and Prolog. In fact, the Robinson's very unification used by Prolog, a special case of constraint solving techniques, is replaced by other more general constraint solving techniques.

Constraint logic programming systems must deal with logical inferences and constraint solving. The last can be probably further separated into static and dynamic solving of constraints. In all reported systems, these parts intertwine. We call this kind of systems "tightly coupled". Philosophically, grouping a set of functionalities into a single unit often causes problems such as complex implementation, unextensibility and overhead. Specializing the functionalities into different parts is more desirable. In this paper, we propose a new approach to constraint logic programming which separates explicitly logical inferences and constraint solving. We call our approach "loosely coupled". Associated to the approach is an architecture scheme. The architecture scheme is totally open. Any system built following the architecture has desirable modularity, i.e., different parts of the system can be built and enhanced independently. This feature makes the system easily extensible. A system called Conslog has been implemented following this approach. The actual system consists of a domain independent partial evaluator and a constraint solver in finite domains. For a given (constraint) logic program, the partial evaluator is in charge of carrying out logical inferences, unfolding predicate calling, propagating instantiated values and collecting constraints. For the given program, this phase needs to be executed only once. Thanks to the partial evaluator, the remaining work for the solver becomes a pure constraint solving problem. At this level, many techniques developed in constraint programming can be investigated and used. Besides, other solvers can be easily added to the system. The partial evaluator builds a "bridge" between logic programming and constraint solving. The results on some well-known examples are very promising.

2. Constraint Logic Programming
2.1. Constraint Satisfaction Problems
A Constraint Satisfaction Problem, also called Labelling Problem or Network of Constraints in some early literatures [16,23] is defined by a set of variables, a set of constraints on the variables and a collection of sets of possible values for the variables. More formally, a CSP is specified by a 3-tuple \( (V,D,C) \) where:
- \( V = \{V_1,V_2,\ldots,V_n\} \) variable set
- \( D = \{D_1,D_2,\ldots,D_n\} \) collection of possible value sets with
- \( D_i = \{d_{ij} \mid d_{ij} \text{ is a possible value of } V_i \} \) (1 ≤ i ≤ n) domains of variables \( V_i \)
- \( C = \{(C_{i_1,\ldots,i_k}) \mid C_i \text{ is a relation on } V_{i_1,\ldots,i_k} \subseteq V\} \) solving a CSP means to determine all possible assignments of values to variables such that all constraints involved in \( C \) are satisfied at the same time.

In this paper, we will not restrict ourself to binary CSPs, in which constraints contain at most two variables. Instead, we will consider the general case where each constraint can be n-ary relations. Specially, we will concentrate on a class of CSPs where variables of constraints take their values from finite subsets of positive integers. Many problems ranging

‡ partially supported by the ALF project (Esprit No. 1520)
from puzzles to integer programming such as cryptarithmetic problems, map coloring, resource allocation, scheduling and so on, fall into this class.

There are several techniques to solve CSPs. The simplest one, also the most inefficient one in most cases, is "generate and test" combining backtracking. We know that this solving strategy has some pathological behavior [23]. Alternative methods concern consistency checking techniques. Algorithms like Waltz filtering [30], forward checking [17], synthesizing algorithm [8] and three level (node, arc and path) consistency in constraint network [23] are all examples of exploiting these techniques. The basic idea behind these algorithms is to use constraints to eliminate inconsistent values of variables. Using constraints in this manner is also called active use of constraints [11]. However, since many CSPs are known NP-complete, there is no hope to solve these problems in polynomial time. Backtracking algorithms are unavoidable. The best we can do is using above pre-processing or filter techniques to reduce a priori the most the later search space.

2.2. Combining Constraint and Logic Programming

There exist several programming paradigms in practice such as imperative, logic, functional, object-oriented and constraint. Each paradigm has strong and weak points. Considerable effort trying to combine some of them has been done in order to give more powerful paradigms. Combining logic programming and constraint solving gives Constraint Logic Programming (CLP). The power of CLP has been demonstrated by the systems like Prolog III, Chip, CLP(9), and Conslog.

Constraint programming is a powerful paradigm to solve CSPs. Its power comes from the following features. Firstly, many problems can be naturally formulated by a set of constraints as CSPs. Secondly, there exist "efficient" algorithms for constraint solving. While there are many successful constraint programming systems, for example, [4,12,22], few of them are really general programming languages. On the other hand, logic programming systems like Prolog are very general declarative programming languages and can be used to state CSPs as well as many other applications. Logic programming supports not only the possibility to define elementary constraints, but also the possibility to group these constraints into a larger one by means of clauses. Recursive definition of clauses is another example of its strengths. While it is easy to encode CSPs into logic programs, the solving of the problems is inefficient. In fact, the very techniques employed by Prolog-like systems, which implement a brute-force "generate and test" strategy, are too primitive.

The analysis of these two programming paradigms leads to the need of combining them. One way to do this is to integrate some constraint solving techniques into the kernel of logic programming systems like Prolog. This is the approach followed by Prolog III, Chip and CLP(9). Our work investigates another way of their combination via the partial evaluation technique.

3. Partial Evaluation

Partial evaluation is a technique specializing a general program into a semantically equivalent program by taking into account some available, but often partial information [9]. The specialized program is usually more efficient because some computation has been done during the specialization, and the specialized program treats fewer cases. The program that carries out the partial evaluation is called a partial evaluator. The result P* of partial evaluation of a given program P with respect to some partially known information is called residual program or program partially evaluated. The counterpart of partial evaluation is complete evaluation or computation.

In logic programming, computations involved in a program are logical inferences (by resolution) and instantiation of variables (by unification). The available information for partial evaluation is a logic program P and a partially instantiated goal G. The partial computation carried out by a partial evaluator is mainly the logical inferences and the propagation of instantiated values through P. The residual program P* is a specialized version of P with respect to G and can be expected to be more efficient. A lot of work has been reported in this field [3,10,25,29]. Subsequently, we will use Prolog as a standard reference of logic programming systems.

Partial evaluation of Prolog programs is very flexible, and natural because of its unification mechanism. A partial evaluator for logic programs can be relatively easily built via a meta-interpreter. Three techniques are largely used. They are: 1) to propagate instantiated parameters of the top-level goal through programs, 2) to carry out all possible logical inferences by unfolding predicate calls, 3) to evaluate built-in predicates whenever possible. A partial evaluator can end up with success or failure. In the case of success, a residual program, is produced. In the case of failure, nothing will be produced. The failure means there exists some incompatibility between the goal and the program.

The partial evaluation technique has been successfully used in many applications [20,25,26,29]. Its potential role in constraint logic programming will be demonstrated in the next section.

4. A New Approach to Constraint Logic Programming

In this section, we propose first a general architecture for constructing constraint logic programming systems [14]. Then a system, called Conslog, is presented as an instance of the architecture. The system combines partial evaluation and constraint solving techniques.

4.1. Architecture Scheme

Constraint logic programming systems must deal with logical inferences, and constraint solving which may be further divided into static and dynamic solving of constraints. The logical inferences can be separated from static and dynamic solving of constraints since they play completely different roles. The part for logical inferences, called "pre-compilation", is executed only once for a given problem.
By static solving of constraints, we mean that from a set of constraints, we can deduce some intermediate results, which we call constraint results. This part of constraint solving, like the part for logical inferences, need to be done only once for a given problem. The dynamic solving of constraints is the part of constraint solving which combines some backtrack techniques. It is this part that solves finally the given problem. Further, the logical inferences part and static processing part together can be conceptually considered as a compilation phase. We can formalize the above description as follows.

A constraint logic program for a CSP can be characterized as three parts.

\( V = \{ V_0, V_1, \ldots, V_{n-1} \} \) variable set

(I) \( D = \{ D_0, D_1, \ldots, D_{n-1} \} \) collection of possible value sets

C local constraints scattered in predicate definitions

The program can be "pre-compiled" into a set of constraints by, for example, a partial evaluator. Logical inferences are carried out. The pre-compilation gives a residual program which can be represented by a 3-tuple \( (V, D, C) \).

\( V = \{ V_0, V_1, \ldots, V_{n-1} \} \) variable set

(II) \( D = \{ D_0, D_1, \ldots, D_{n-1} \} \) collection of possible value sets

C = \( \{ C_0, C_1, \ldots, C_{m-1} \} \) constraint set

From (II), some static processing such as simplification can be applied causing that 1) some \( V_i \in V \) may be instantiated, 2) some \( D_k \in D \) may be reduced, 3) some \( C \in C \) may be solved. This gives constraint results. Constraint results produced by this phase is correct in the sense that instantiated values are the only possible values, reduced domains can never get bigger, and finally, solved constraints can never come back. Constraint results can be characterized as follows:

\( V = \{ V_{i,0}, V_{i,1}, V_{i,n-1} \} \ 0 \leq i \leq n-1 \)

(III) \( D = \{ (d_{i,0}), (d_{i,1}), \ldots, (d_{i,n-1}) \} \ 0 \leq i \leq n-1 \)

C = \( \{ C_{i,j}, \ldots, C_{i,j} \} \)

where each \( V_{i,j} \) is a variable already instantiated by its only consistent value \( d_{i,j} \). \( V_{i,j} = p+1, \ldots, n-1 \), are variables not yet instantiated. \( C_{i,j} \in C \) \( j \neq 0, \ldots, m-1 \) are constraints yet unsolved. Two particular situations may occur at this stage. If some \( D_{i,j} = 0, \ldots, n-1 \), becomes empty, there will be no need to continue, because the initial CSP has no solution. If all the variables are instantiated, the unique solution is found. The processing ends up.

For those \( V_{i,j} = p+1, \ldots, n-1 \) waiting instantiation in (III), some constraint solving techniques combining backtrack procedures will be used leading to the following state (or failure):

\( V = \{ V_{i,0}, V_{i,1}, \ldots, V_{i,n-1} \} \ 0 \leq i \leq n-1 \)

(IV) \( D = \{ (d_{i,0}), (d_{i,1}), \ldots, (d_{i,n-1}) \} \ 0 \leq i \leq n-1 \)

C = \( \emptyset \)

where each \( D_{i,j} = p+1, \ldots, n-1 \) contains at least one consistent element. Each \( V_{i,j} = p+1, \ldots, n-1 \) takes its value from \( D_{i,j} \) and all \( V_{i,j} = 0, \ldots, n-1 \) satisfy all the constraints. Backtracking will only be possible on such \( D_{i,j} = p+1, \ldots, n-1 \).

According to the discussion, we have the architecture scheme of Figure 1.

This architecture has several advantages. Firstly, any system based on the architecture has the desirable modularity. From an implementor's viewpoint, organizing different functionalities into different parts facilitates implementations. A system as such is extensible by adding other solvers. Existing parts of the system can be improved or replaced independently. Secondly, efficiency can be gained by avoiding some repeated computation and reusing the results obtained in previous steps.

The system Conslog has been built by using this approach. The architecture suggests that there may exist a set of constraint solvers, each of which is specialized to a specific domain. At the moment, Conslog consists of two parts: a domain independent partial evaluator for logic programs and a constraint solver in finite domains. Other solvers are under development. Given a CSP specified by a constraint logic program P and a goal G, the partial evaluator first evaluates P with respect to G. This gives a residual program, which is essentially a set of constraints yet unsolved. The logical inferences of the initial program P are effected (once for all). The constraint solver takes this set of constraints as input data and uses the algorithm integrated in it to solve the constraints. It is important to note that a program is partially evaluated only once. The resolution for the rest of the problem begins each time from the solver, not from the partial evaluation.

4.2. The Language

Conslog is defined by its syntax, its declarative semantics and its procedural interpretation. The description in this section will be very informal.

The computation domain of Conslog's actual solver is
the finite subset of positive integers. The domain concept was introduced in [27]. For example, a variable X ranges over the domain \{2,4,7,8,10\}. This information is declared by a predefined predicate domain/2.

A Conslog program consists of a finite set of clauses and a domain predicate. A clause can be either a rule or a fact. A rule takes the following form:

\[ \text{head: - body.} \]

The head of the rule is of the form \( p(t_1, \ldots, t_n) \) with \( p \) predicate symbol in the sense of Prolog and \( t_1, \ldots, t_n \) are terms. The body is composed of elements \( q(t_1, \ldots, t_n) \) where \( q \) are constraints. Constraints are constructed from constraint symbols \( \phi \), \( \phi \in \{=, \neq, <, \leq, >, \geq \} \), constants, variables and arithmetic operators \(+, -, \ldots\). For example, \( X = a + 2 \), \( R4 + N + R \equiv E + 10 \times R \), \( X + Y + 2 > Z + 5 + V \) are all well formed constraints. The syntax for \( q(t_1, \ldots, t_n) \) is the same as \( p(t_1, \ldots, t_n) \).

A fact is a rule without the right part. A goal has the same form as the right part of a rule.

The informal declarative semantics is the same as that of Prolog. The procedural semantics consists of two parts: the inferences on constraint predicates and the constraint solving. The first part is realized during the partial evaluation, while the constraint solving is done by the solver. Suppose we have a resolvent at some moment during partial evaluation:

\[ \langle q_1(t_1), q_2(t_2), \ldots, q_n(t_n) \rangle. \]

The computation rule of the partial evaluator selects always the left-most atom. Two cases appear. If the selected atom is a predicate, a derivation step like that of Prolog is taken. That means the partial evaluator tries to match the selected atom with the left part of a clause in the program and to replace the selected atom by the right part of the matched clause. If the selected atom is a constraint, then this constraint is moved to the end of the resolvent to join other constraints accumulated up to now. The partial evaluation is ended when the resolvent contains only constraints or nothing. The set of constraints constitutes the input of the constraint solver.

Note that unlike Prolog III, Chip or CLP(\( \mathcal{P} \)), the solvability of constraints accumulated is not tested here. This is done during the constraint solving. So the procedural semantics is different from that of above systems.

4.3. The Partial Evaluator

The partial evaluator in Conslog system is a general tool for constraint logic programs. It follows the philosophy of any partial evaluator, i.e. "do as much as you can do, put aside what you can't". The reasoning process is based on the SLD-resolution principle. The left-most computation rule and the depth-first search rule are used. Many techniques presented in the literatures on partial evaluation for logic programs have been investigated and integrated in the partial evaluator. The partial evaluator has sufficient knowledge to recognize evaluable system predicates and evaluate them whenever possible. For example, X is 4+6 makes X instantiated to 10. And this leads to propagation of the value to others occurrences of X. For non evaluable predicates, it stores them as part of residual program. Constraints are treated as non evaluable predicates, except for the constraints sufficiently instantiated. The partial evaluator tries to evaluate recursive predicates while avoiding infinite loops. The partial evaluator is powerful enough to treat complete logic (possibly open) programs containing multiple clauses, recursive predicates, built-in predicates and constraints. For a more complete discussion of the partial evaluator, the reader is referred to [13]. Here, a small example from [29] explains the behavior of the partial evaluator.

The following program is a definition of the predicate app_list/3 for the concatenation of two lists from a point of view of abstract data type.

\begin{verbatim}
app_list(L1, L2, L3):- append(L1, L2, L3).
empty([]).
append([], L2, L2):- empty(L1).
append([H|L1], L2, [H|L3]):- cons(H, T, L1), append(T, L2, T1), cons(T, T1, L3).
\end{verbatim}

Partially evaluating the program with respect to the goal (not instantiated at all) ?- app_list(X, Y, Z) gives the following residual program:

\begin{verbatim}
app_list(L1, L2, L3):- append(L1, L2, L3).
append([], L, L).
append([H|L1], L2, [H|L3]):- append(L1, L2, L3).
\end{verbatim}

which is the normal definition of concatenation of two lists. The same question given to Prolog leads to an infinite loop.

If we give the partial evaluator a constraint logic program and a possibly partially instantiated goal, the evaluator will work in the same manner as described above. At the same time, it collects the constraints encountered, which constitutes the residual program. Like the residual program of a logic program is executed by a logic programming system like Prolog, the residual program of a constraint logic program will be executed by a constraint solver which is the topic of the following section.

The partial evaluator is implemented by means of a meta-interpreter written in Prolog.

4.4 The Solver

Conslog's architecture makes it possible that different solvers exist for different computation domains. We present now our actual solver which works in discrete finite domains. The solver's input data comes from the outputs of the partial evaluator, which is a set of well-formed constraints. The outputs of the solver are of two kinds. If the set of constraints has solutions, all of them will be found by backtracking. If there is no solution at all, the solver will answer "no solution". The solver is therefore sound and complete.
The algorithm integrated in the solver uses a Waltz-like algorithm [30,5] combining different refining procedures (see below) and backtracking. The Waltz algorithm concerns essentially a control structure which allows to refine the variables of a set of constraints repeatedly until no further refinement can be achieved. The Waltz algorithm alone, like any consistency algorithm, can not solve the set of constraints in the general case. When the Waltz algorithm can go no further, the value for a variable is generated. The effects of this value are propagated to other domains of variables by running again the refining procedures. This leads to logical deletion of inconsistent values of other variables and probably instantiate some variables. The refining procedures are concerned with interval calculus [1,6,15,21] and forward-checking [17]. The solver works in such a way that constraints are used to efficiently eliminate inconsistent values of variables from their domains and hence to greatly reduce the search space in a a priori manner.

A constraint is called active, if the domains of variables involved in it can be possibly reduced by this constraint. To refine a variable by a constraint is to reduce the domain of the variable by deleting the inconsistent values with respect to the constraint. To refine an (active) constraint means to use the constraint to refine all its variables in it. The refining of variables and constraints is carried out via refining procedures. A constraint is said solved if it is sufficiently instantiated and its truth value can be verified.

General Algorithm:
CONST← Set of constraints yet unsolved
WHILE CONST ≠ Empty DO
BEGIN
S1: WHILE ∃ active constraints in CONST DO
BEGIN
a. Choose an active constraint C of CONST
b. Refine C */ if the domain of a variable is reduced to empty, there is some inconsistency. Backtrack or stop with failure */
c. IF C is solved THEN Throw away C
ELSE Replace old C by refined C in CONST
END (while)
S2: Choose a variable VAR
Generate a value belonging to its domain for VAR
END (while)
S3: IF CONST = Empty THEN End with success.

In the algorithm, S1.a, called constraint selection rule (CSR), corresponds to the computation rule in logic programming systems. In our system, a dynamic constraint selection rule is employed. The solver chooses always the constraint which has more chance to reduce the domains of variables. For example, a constraint like E+B+C=5*A will be selected before E=A=B+2, while E=A=7 will be chosen before E+B+C=5*A.

S1.b corresponds to different refining procedures which reduce the domains of variables from a given constraint. Consistency-enforcing happens here, i.e. inconsistent values for constraints are eliminated by the refining procedures. For disequality (≠=) constraints, forward-checking is used. For the binary constraints the reducing procedures are trivial. For example, the constraint X<Y with Xe [2,...,10], Ye [0,...,8] will reset X to [2,...,7], Y to [3,...,8] because X<Y<8 gives X<8; 2≤X<Y gives Y>2. For general constraints of the form ∑C X ∏ D Y with C, D ≥ 0,0≤e (||=,<,≤,>,||), and X, Y variables, interval calculus is systematically used. For example, from X+2*Y+5*Z=21 with Xe [1,...,10], Ye [1,...,10], Ze [1,...,10], the interval calculus reduces Y to [1,...,7], Z to [1,...,3] with X unchanged. However, two points need to be noted. First, variables involved in an initial constraint are refined directly; no new constraints are produced and added to the set of constraints, which was not the case, for example, in Alice. Secondly, a domain after refinement (RD) is not a simple replacement by the resulted interval (RI) from interval calculus, rather it is the intersection between the initial domain (ID) and the resulted interval, i.e. RD={vi ∩ riv ID & riv RI}. If RD becomes a singleton, the value is assigned directly to the variable and propagated to other constraints. If RD is empty, backtracking occurs on generated values by S2. If there are no more untried values, a failure occurs.

S2 is a generating procedure defining a variable selection rule (VSR). When no active constraints can be chosen, the algorithm will select a variable and generates for it a value from its domain. The VSR chooses the variable to be instantiated the one which has the smallest domain. This corresponds to the smallest commitment principle. The generated value is propagated to all occurrences of the variable. This makes some constraints active and stimulates again the consistency-enforcing in S1. Note that if the CSR makes the constraint order in a program irrelevant, the VSR makes the variable order unimportant.

S3 corresponds to the successful ending of the algorithm when all constraints are solved.

From the algorithm we note that a variable can obtain its values in two ways. One concerns the generated value in S2.b. The other possibility is that the domain is reduced to a single element by the refining procedures in S1.b. These two ways of instantiating variables are very different in nature. The first kind of instantiation consists of giving a value to a variable and then verifying the value by constraints. On the contrary, an instantiation in the second way is a logical consequence deduced from constraints. Note also that the part for the static solving of constraints discussed in § 4.1 corresponds to the first phase of S1, i.e., before intervenes the generating procedure. The partial solution obtained at this phase will not be questioned by the backtracking in the future as a result of wrongly generated values.

The solver is implemented in Prolog. More details concerning the solver and the interval calculus based refining procedures can be found in [15].

498
4.5. An example

In this section, we illustrate our approach with an example. We take the cryptarithmic problem SEND+MORE=ONEY, which consists of finding for each letter a different number ranging from 0 to 9 such that the equation remains true.

The problem is naturally formulated in Conslog by the following constraint logic program containing equality constraints, disequality constraints and domain constraints.

\[
\begin{align*}
\text{sendmory}((S,E,N,D,M,O,R,Y,R1,R2,R3,R4)):
\quad & \text{domain}([S,E,N,D,M,O,R,Y],[0,...,9]),
\quad \text{domain}([R1,R2,R3,R4],[0,1]),
\quad M \equiv 0, S \equiv 0, R1 \equiv M,
\quad R2+S+M \equiv O+10\times R1,
\quad R3+E+O \equiv N+10\times R2,
\quad R4+N+R \equiv E+10\times R3,
\quad D+Y \equiv Y+10\times R4,
\quad \text{alldiff}([S,E,N,D,M,O,R,Y]).
\end{align*}
\]

\[
\begin{align*}
alldiff([X1,X2]): & \text{ outof}(X1,X2), \text{alldiff}(X2).
\end{align*}
\]

\[
\begin{align*}
\text{alldiff}([]).
\end{align*}
\]

\[
\begin{align*}
\text{outof}(X1,[Y1,Y2]): & \text{ X1} = Y1, \text{outof}(X1,Y2).
\end{align*}
\]

\[
\begin{align*}
\text{outof}([X1]).
\end{align*}
\]

The execution of the program is divided into two independent parts: the pre-compilation of the initial program and the proper constraint solving. Firstly the program is partially evaluated (once and for all) with respect to the goal sendmory((S,E,N,D,M,O,R,Y,R1,R2,R3,R4)) by the partial evaluator. During the partial evaluation, the predicate callings unfolded, and the constraints collected giving in an intermediate file the collection of constraints \( \{ M = 0, S = 0, R1 = M, R2 + S + M = O + 10 \times R1, R3 + E + O = N + 10 \times R2, R4 + N + R = E + 10 \times R3, D + E = Y + 10 \times R4, S = E, S \neq N, S = D, S = M, S = O, S = R, S = Y, E = N, E = D, \ldots \} \) with domain variables. Note that the variables after the partial evaluation have been renamed. We continue to use the initial variables for the reason of readability. As we will see in the following neither the constraint order nor the variable order is important, because dynamic CSR and VSR are used.

The solver can be run now with the goal conslog(sendmory(([S,E,N,D,M,O,R,Y,R1,R2,R3,R4]))).

Let’s trace the constraint solving. \( M = 0 \) and \( S = 0 \) are selected first by the CSR and solved immediately, i.e. 0 is removed from the domains of M and S. Then \( R1 = M \) is chosen. This instantiates R1 and M to 1 because the refining procedures reduce the two domains to the single element 1. This entails eliminating 1 from other domains because the CSR selects successively all the constraints of the type \( M = 0 \times X \in \{ S,E,N,D,O,R,Y \} \). Next, \( R2+S+M = O+10\times R1 \) is chosen and refined reducing S to \{8,9\} and instantiation O to 0. All the disequality constraints containing O are then chosen and solved removing 0 from the related domains. The next chosen constraint \( R3+e+o = n+10\times r2 \) leads to instantiating R2 to 0. When \( R2+S+M = O+10\times R1 \) (with \( M = 1, O = 0, R1 = 1, R2 = 0 \)) is reconsidered, the domain of S is reduced to 9. This causes the removing of 9 from other domains by the \( \equiv \) constraints.

At this stage the refining process can not go further, i.e. there is no more active constraint. A variable will be chosen and forced to have a value. The R3 is chosen by the VSR and given the value 0 belonging to its domain. This makes some constraints active. However, \( e = n \) (\( R3 + e + O = N + 10\times R2 \)) and \( R4 + N + R = E + 10\times R3 \) lead to a contradiction because the domain of R4, N must be empty. The first backtrack occurs giving the other value 1 to R3. Refining \( R4 + N + R = E + 10\times 1 \) rules out 8 from E, 2 from N and R. No constraints can be further reduced now. Now R4 is chosen and given the value 0 leading to again a contradiction. So R4 must takes the single possible value 1. The algorithm reduces E to \{4,5,6,7\}, N,D,R to \{5,6,7,8\}, Y to \{2,3,4,5\} using \( 1 + E = N, 1 + N + R = E + 10\times 1, D + E = Y + 10\times 1 \) and can not go further. Forcing E takes 4 lead to a backtrack. 5 is now given to E. The disequality constraints remove 5 from the domains of N,D,R and Y. Refining \( D + 5 = Y + 10\times 1 \) removes 6 from D’s domain. Refining \( 1 + E = N \) solves the constraint and instantiates N to 6. This value is removed immediately from R’s domain. Refining \( 1 + 6 + R = 5 + 10\times 1 \) solves the constraints and gives 8 to R. Removing 8 from D’s domain instantiates D to its possible single value 7. The only unsolved constraint \( 7 + 5 = Y + 10\times 1 \) is solved with Y instantiated to 2. In total, three backtracks are needed. The problem is solved in 3 seconds on a Sun3/210. Note that the solver works at a meta level.

Other problems such as N-queens, zebra, magic square and map coloring problems can be also specified easily and solved very efficiently. The performance on N-queens (15xN<30) beats largely Bratko’s algorithm—-the fastest specific algorithm for this problem. We observed that the bigger N is, the greater the speedup is. We measured a speedup of 3 to 40 times (130 times faster for N=22).

5. Related Work

Our work is related to CLP systems such as Prolog III, CLP(\$), especially Chip. However, there are some differences. The most significant one is the philosophical one discussed in the first section. Our system is based on a modular, easily extensible architecture. For example, other solves for other domains can be independently connected to the system. Besides, while Prolog III and CLP(\$) work in numerical domains such as rational and real with the Simplex algorithm, Conslog’s actual solver works in finite domain with specific techniques such as interval calculus which is first investigated in Alice system. Forward-checking technique which is used in Chip [28] is equally integrated to deal with disequality constraint. Not like Chip, the procedures like forward-checking are totally transparent to the user. No declaration is needed.
6. Conclusions

In this paper, a new approach to constraint logic programming is presented. The system Conslog based on this approach is described. Our contribution can be considered at two levels. Firstly, an architecture is proposed for building constraint logic programming systems. Systems based on the architecture will be very efficient and extensible. Secondly, the implemented system Conslog constitutes another contribution. Two well known techniques, the partial evaluation and constraint solving, are harmoniously combined in the system. The partial evaluator is used as a tool for pre-compilation of constraint logic programs into a set of constraints which is fed to the constraint solver. At the solver level, many techniques were investigated. The actual solver integrates consistency-enforcing procedures into a Waltz-like algorithm combining backtracking. Interval calculus is systematically used to largely reduce the search space. The dynamic constraint selection rule and the variable selection rule make almost irrelevant the constraint and variable order in a program. Thanks to the modularity feature, the system can be improved and extended without affecting the existing parts. Horizontally, other solvers for other domains, for example, boolean, real and set solvers can be built and connected to the system. Vertically, alternative techniques can be investigated and integrated into different parts of the system. Our experiences with the system show that one can prototype different ideas about, and build one's experimental laboratories for constraint logic programming with existing logic programming systems like Prolog.

Acknowledgments

We would like to thank the members of Crid for their support and encouragement. Thanks to Jacqueline Chabrier for her comments on an earlier version of this paper. Special thanks go to Mehmet Dincbas of ECRC for his valuable comments. Discussion with him helped clarify some points.

Bibliography