A General Computational Scheme for Constraint Logic Programming

John Darlington Yike Guo Qian Wu
Department of Computing
Imperial College
180 Queen's Gate London SW7 2BZ U.K.
e-mail: {jd,yg,wq}@doc.ic.ac.uk

July 29, 1991

Abstract

In this paper we propose a novel computational model for constraint logic programming (CLP) languages. The model provides an efficient mechanism for executing CLP programs by exploiting constraint satisfaction as a means for both solving constraints and controlling the whole computation. In the model, we separate constraint solving from the deduction procedure. Deductions over constraints are extracted from the source program and represented as a *context-free grammar* that encodes the way in which deduction will generate constraints to be solved. Therefore, deduction is performed abstractly at compile time. Executing the grammar generates all the constraints that need to be solved at run time. A very flexible control mechanism is therefore provided by the model in terms of the information fed back from the constraint solving procedure. It is shown that the model provides a general scheme for investigating an efficient computational model for implementing constraint logic programming systems.

1 Introduction

In recent years, there has been a flurry of interest in constraint languages. In constraint programming constraints provide a means to specify the component conditions of a problem. The whole problem can then be represented as a program which is organized by putting constraints together using logical connectives. To enhance the expressive power of the constraint programming paradigm, extensive work has been carried out within the declarative programming framework, particularly in functional programming and logic programming. In logic programming, research has led to a new logic programming paradigm which has been given the name "Constraint Logic Programming (CLP)". CLP views a logic programming language as a constraint language on the domain of discourse. Such new logic programming languages as PrologII [2], PrologIII [3], CLP(R) [14] et.al. fall within this framework. In [12], the authors proposed a general framework for constraint logic programming by defining the *Assertional Programming* paradigm. From the assertional programming point of view, programming in any declarative language can be viewed simply as *making assertions*. These assertions, or sentences, are axioms that the programmer believes to be true in a well-understood mathematical system acting as the semantic foundation of the language. These assertions can be entered into the computer as a program and then used as the premises for *deductive inference* when a problem (query, goal or constraint) is submitted to the system. These assertions are then invoked automatically in deducing the submitted problem as a consequence. Therefore, a CLP system can be viewed as an assertional programming system in which symbolic logical deduction and constraint solving in the domain of discourse are integrated. Programming thus becomes equivalent to defining new constraints over the domain of discourse. The authors developed several new CLP systems, such as constraint equational logic [10] following this approach as well as a new programming paradigm named *Constraint Functional Logic Programming* which was proposed [8] as a uniform language framework to integrate typed higher order functional programming systems with constraint logic programming systems.

The constraint model of computation, which was first introduced by Steele [19], is based on constraint solving which is a constructive procedure to solve the following constraint satisfaction problem: **Given a constraint, do there exist values for all variables in the constraint such that the constraint holds?** In[19], a technique called **local propagation** was used for solving constraints. Local propagation attempts to satisfy a large collection of constraints piece by piece. A local propagation step occurs when enough variables in a constraint become ground for other variables to be instantiated. A collection of constraints is solved by local propagation if all the variables in the system become determined after a finite number of local propagation steps. This technique is also used in some logic programming systems such as CP [17] in which constraints are represented by the predicates and solved in a sequence indicated by the form of the guards of the rules. In CLP(R), algorithmic procedures, such as Gaussian Elimination and the Simplex method, are used for solving some arithmetic constraints on the real number domain. A delay/wakeup condition is defined for those constraints whose satisfiability problem is intractable or even undecidable (e.g. non-linear equations). That is, constraint solving will be delayed until a sufficient number of other constraints, which are selected from the subset defined by the wakeup condition, entail that the delayed constraint may now become eligible for solving. All these mechanisms only solve a static primitive constraint set. However, in any CLP system, constraints are dynamically created by deduction steps such as resolution or narrowing [13]. The impact on the design issues of the computational model for CLP systems is two-fold:

1. How do we design a proper interface for constraint generation (by constraint deduction) and constraint solving?
2. How do we design a proper solver which can incrementally solve the dynamically generated constraints?

Both problems are related and, fortunately, many constraint solvers from the mathematical world possess the incremental property. But, to our knowledge, there is no satisfactory solution yet to the first problem. In the CLP(R) implementations, an inference engine was used to perform logic deduction to generate constraints and the solver was responsible for solving non-trivial constraints. A distinct interface between the deduction engine and the solver was designed to do some simple solving work and transform constraints to a canonical form acceptable to the solver. The inference engine was actually the extension of the traditional Prolog system in the sense that the unification procedure was replaced by a procedure to generate constraints which were then solved by a built-in constraint solver over free terms and real numbers. There are many problems with this design. The most important drawback is that everything is performed at run-time, including resolution (for logic deduction), generating constraints, constraint solving itself and backtracking. This is by no means efficient. An important observation is that in any CLP system, the deduction procedure is mainly responsible for generating constraints and is always separated from constraint solving itself. The major computational task is actually shifted from logical deduction to constraint solving. Therefore, as we will claim in the paper, the idea of separating deduction from constraint solving and doing much of the deduction at compile time is an important issue in the efficient implementation of constraint logic programming systems. What we propose in this paper is a novel computational model which does all that can be done at compile time. The resolution step, which is only used to generate constraints, isn't performed at run time but compiled into a **context-free grammar** that represents the way constraints will be generated. At run time, grammar rewriting is executed to generate **solving plans**, denoting collections of constraints to be solved. The solver decodes the solving plans and solves the corresponding constraints. Since all the information about every collection of constraints of each computational path is maintained by the grammar, the control strategy for non-deterministic computation is completely flexible. There is no sequential restriction, no "built-in" control strategy and, of course, no backtracking. This work originated in the connection graph approach to theorem proving[1]. In the next section, we will recapitulate the notion of constraint logic programming and its operational model. An overview of the abstract machine is presented in section 3 together with a compilation scheme. In section 4, we present the organization of the machine focusing on the control strategy. Conclusions and related work appear in section 5.

2 Constraint Logic Programming

2.1 Constraint Systems

Constraint systems are motivated by the desire to perform computation over some well understood domains such as boolean algebra, integers, rational numbers or lists. These domains come equipped with natural algebraic operations such as boolean conjunction and disjunction, rational addition and multiplication, they also have associated privileged predicates such as equality and various forms of inequalities. Such a computational domain together with its operators can be regarded as an algebra. The logical formulae of the algebra can be abstractly regarded as *constraints*. Constraints provide a way of defining objects implicitly by stating the logical relations that must hold between them. That is, a constraint denotes a set of objects which realise the relation. This notion is captured as follows:

Definition 2.1 (Constraint System) *Given a computational domain $\mathcal{A}$ and a set of variables $\mathcal{V}$. We define a constraint system, C, formally as a tuple: $< \mathcal{A}, \mathcal{V}, \Phi, \mathcal{I} >$ where Φ is a decidable set of constraints over $\mathcal{A}$. $\mathcal{I}$ is a solution-mapping $[\![]\!]^{\mathcal{I}}$ which maps every constraint $\phi \in \Phi$ to $[\![\phi]\!]^{\mathcal{I}}$, a set of $\mathcal{A}-$valuations, α, which are called the solutions of ϕ. Let Val_A be the set of all $\mathcal{A}$-valuations. A constraint is satisfiable in C iff its solution set is non-empty. A constraint ϕ is valid in I iff $[\![\phi]\!]^{I} = Val_A$. For a set of constraints Φ, I is a model of Φ if all the constraints in Φ are valid in I. Assuming a set W of variables, the W-solutions of a constraint ϕ in $\mathcal{I}$ are the set of solutions: $[\![\phi]\!]^{I}_{|W} := \{\alpha_{|W} \mid \alpha \in [\![\phi]\!]^{I}\}$. A constraint ϕ is equivalent to a constraint ϕ' iff $[\![\phi]\!]^{I} = [\![\phi']\!]^{I}$.*

Definition 2.2 (Closed under Logical Connectives) *A constraint system: $\mathcal{S}_{\mathcal{A}} :< \mathcal{A}, \mathcal{V}, \Phi, \mathcal{I} >$, is closed under logical conjunction (resp. disjunction, implication, existential quantification) if for any constraints $\phi_1, \phi_2 \in \Phi$, there exists a constraint $\phi_1 \wedge \phi_2$ (resp. $\phi_1 \vee \phi_2, \phi_1 :- \phi_2, \exists x.\phi_1) \in \Phi$ and:*

$$[\![\phi_1 \wedge \phi_2]\!]^{I} = [\![\phi_1]\!]^{I} \cap [\![\phi_2]\!]^{I}$$

$$[\![\phi_1 \vee \phi_2]\!]^{I} = [\![\phi_1]\!]^{I} \cup [\![\phi_2]\!]^{I}$$

$$[\![\phi_1 :- \phi_2]\!]^{I} = [\![\phi_1]\!]^{I} \cup \{ Val_A - [\![\phi_2]\!]^{I} \}$$

$$[\![\exists x.\phi]\!]^{I} = \{\alpha \in Val_A \mid \alpha_{|W-x} = \beta_{|W-x}, \beta \in [\![\phi]\!]^{I}\}$$

where W is the set of variables in ϕ. Constraints which do not contain logical connectives are called atomic constraints.

In this paper we assume that all constraint systems are closed under these logical connectives and contain equality constraints. Closure under conjunction permits systems of atomic constraints to be solved simultaneously. Closure under disjunction is required to represent alternative solutions to a goal constraint. Implication expresses the deduction relation between constraints, specifically it may be used as a programming construct. The existential quantifier permits the use of bound variables in constraints, i.e. variables that do not occur in the solution set.

To solve constraints, we require that a constraint system comes with a set of *solved forms* such that every constraint in solved form is satisfiable. For every satisfiable constraint G, there exists a complete set of solved forms Sol_G such that the disjunction of all constraints in the set is equivalent to G. That is, for all $G'_i \in Sol_G$ and $G' = \bigvee_{i=1}^{n} G'_i$, $[\![G]\!]^{I} = [\![G']\!]^{I} = \bigcup_{i=1}^{n} [\![G'_i]\!]^{I}$. Solved forms are introduced because of the ease with which their satisfiability may be verified, and with which solutions may be derived from them. A procedure which computes solved forms in a constraint system is called *a constraint solver*. We use $\longrightarrow_c$ to denote the derivation relation of a constraint solver C.

Definition 2.3 (Soundness and Completeness) *Let $< \mathcal{A}, \mathcal{V}, \Phi, \mathcal{I} >$ be a constraint system and C a constraint solver for the system.*

Soundness: *C is sound iff for any constraint G: $G \longrightarrow_c G' \Longrightarrow [\![G']\!]^{I} \subseteq [\![G]\!]^{I}$*

Completeness: *C is complete iff for any constraint G, $\forall \alpha \in [\![G]\!]^{I}$, there exists a constraint G' which is in solved form and: $G \longrightarrow_c G'$ and $\alpha \in [\![G']\!]^{I}$.*

We present two constraint systems as examples.

Example 2.3.1 *Let $\mathcal{R}$ be the real number domain equipped with the arithmetic operations $F :< +, -, * >$ and Φ_R be a set of linear equations. $\mathcal{S}_{Real} =< \mathcal{R}, \mathcal{V}, \Phi_R, \mathcal{I}_R >$ is a constraint system over real numbers where the interpretation $\mathcal{I}_R$ maps each linear equation to the (possibly infinite) set of its solutions. A set (or conjunction) of linear equations:*

$$
\begin{aligned}
y_1 &= a_{1,1}x_1 + \ldots + a_{1,n}x_n + c_1 \\
&\ldots \\
y_m &= a_{m,1}x_1 + \ldots + a_{m,n}x_n + c_m
\end{aligned}
$$

is in solved form if the variables $y_1, \ldots, y_m$ and $x_1, \ldots, x_n$ are all distinct. The variables $y_1, \ldots, y_m$ are **eliminable variables** *and $x_1, \ldots, x_n$ are* **parameters**. *Algorithms for solving linear equations, such as Gaussian elimination, are sound and complete constraint solvers for this system. Linear arithmetic constraints have been studied extensively by Lassez and Jaffer el.al. and used as the predefined component of a constraint logic programming system [14].*

Example 2.3.2 *Let T_Σ be the set of all ground Σ−terms for a given signature Σ, V be a set of variables and Φ_Σ be the set of equations over $T_\Sigma(V)$. $\mathcal{S}_\Sigma =< T_\Sigma, \emptyset, V, \Phi_\Sigma, I_\Sigma >$ is a constraint system over the first order Σ-terms where the interpretation I_Σ maps each term equation to the (possibly infinite) set of its ground unifiers. A conjunction of term equations ϕ is usually called a system of equations in the literature. As shown in [15], for any satisfiable system of equations, its unique solved form is a new system of equations of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ where the x_i's are distinct variables which do not occur in any of the t_j's. Therefore, as with the constraint system $\mathcal{S}_{Real}$, the variables $x_1, \ldots, x_n$ are eliminable variables and variables in the t_j's are parameters. Traditionally, the notion of the most general unifier (mgu) is used to represent solutions. The relation between the solved form and the mgu of a system of equations can be established by following theorem [15].*

Let $\alpha = \{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ be an idempotent substitution, α is the mgu of a system of equations ϕ iff the equation system: $\{x_1 = t_1, \ldots, x_n = t_n\}$ is the solved form of ϕ.

Unification algorithms, such as the following Martelli and Montanari algorithm [16], are complete and sound constraint solvers, transforming a system of equational constraints to its solved forms.

Trivial:

$$\frac{G_0 : \{t = t\} \wedge S}{G_1 : S}$$

Term Decomposition:

$$\frac{G_0 : \{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \wedge S}{G_1 : \{t_1 = s_1, \ldots, t_n = s_n\} \wedge S}$$

Variable Elimination:

$$\frac{G_0 : \{x = t\} \wedge S}{G_1 : \{x = t\} \wedge \sigma S}$$

where $x \notin var(t)$ and $\sigma = t/x$.

2.2 Constraint Horn Clause Logic Programming

A constraint system can easily be integrated into a logic programming system by exploiting the semantic information of the abstract symbols. The resulting *constraint logic programming* system is a logic assertional system with a well-defined intended model. In [12], the authors presented a procedure for constructing a CLP system by defining general logic assertions over a constraint system. We focus here on the constraint Horn clause logic programming paradigm. Let $C :< \mathcal{A}, \mathcal{V}, \Phi_C, I_C >$ be a constraint system with Φ_C as a set of primitive constraints over A.

A constraint (Horn clause) logic program Γ over C is a set of constrained defining rules of the form:

$$p(e_1, \ldots, e_n) :- \; c_1, \ldots, c_n, B_1, \ldots, B_m$$

where $p(e_1, \ldots, e_n)$ is an atom, i.e., $p \in \Pi$ is an n-ary user defined predicate and $e_1, \ldots, e_n$ are expressions over $\mathcal{A}$ where Π is the signature of user-defined predicates in Γ. The $c_i \in \Phi_C$ and the B_i are atoms. An interpretation I of Γ over C is defined by a function $[\![]\!]^I$ interpreting any predicate symbol $p \in \Pi$ as a relation p^I over $\mathcal{A}$. That is,

$$\forall c \in \Phi_C . [\![c]\!]^I = [\![c]\!]^{I_C}$$

$$[\![p(x_1,\ldots,x_n)]\!]^I = \{\alpha \in Val_A^n \mid \alpha(x_1,\ldots,x_n) \in p^I\}$$

Since the underlying constraint system is assumed to contain equality constraints, the interpretation function can interpret an atom $p(e_1,\ldots,e_n)$ as:

$$[\![p(e_1,\ldots,e_n)]\!]^I = [\![p(x_1,\ldots,x_n)]\!]^I \cap \bigcap_{i=1}^{n} [\![x_i = e_i]\!]^{I_C}$$

A model of Γ over C is an interpretation which satisfies all the rules of the program. We use Mod_Γ to denote all the models of the program. Models of a CLP program can be ordered in terms of the set inclusion ordering. The minimal model M_Γ of Γ over C may be constructed as the least fixed point of the traditional "bottom-up" iteration which the computes ground relations entailed by a program[18]. That is, M_Γ is the limit $\bigcup_{i\geq 0} M_\Gamma^i$ of the sequence of interpretations:

$$p^{M_\Gamma^0} = \emptyset$$

$$p^{M_\Gamma^{i+1}} = \{\alpha(x_1,\ldots,x_n) \mid \alpha \in \bigcap_{i=1}^{n} [\![x_i = e_i]\!]^{I_C} \cap \bigcap_{i=1}^{j} [\![c_i]\!]^{I_C} \cap \bigcap_{i=1}^{m} [\![B_i]\!]^{M_\Gamma^i}\}$$

where $p(e_1,\ldots,e_n) :- \ c_1,\ldots,c_j, B_1,\ldots,B_m \in \Gamma$

Theorem 2.3.1 *Given a constraint system $C :< \mathcal{A}, \mathcal{V}, \Phi_C, I_C >$ and Γ, a constraint logic program over C. The sequence of interpretations M_Γ^i constitutes a chain in a cpo of interpretations of Γ ordered by the set inclusion relation. The limit of the chain is the minimal model of Γ over C.*

The following theorem shows that a CLP program extends its underlying constraint system by defining new relational constraints.

Theorem 2.3.2 *Let Γ be a constraint logic program and Π be the signature of the user-defined predicates in Γ. Γ is a constraint system:*

$$\Pi(C) :< \mathcal{A}, \mathcal{V}, \Phi_C \cup \Phi_\Pi, I_C \cup I_\Pi >$$

which is a **relational extension** *of the underlying constraint system*

$$C :< \mathcal{A}, \mathcal{V}, \Phi_C, I_C >$$

constructed by extending Φ_C to include user-defined relations over $\mathcal{A}$. That is, Φ_Π contains all Π-atoms of the form $p(x_1, ..., x_n)$ The interpretation of all C-constraints remains unchanged in $\Pi(C)$. Each user-defined predicate is interpreted by I_Π which maps the predicate to a relation over $\mathcal{A}$ defined by the minimal model of Γ over C.

Therefore, for a given constraint system C, a constraint logic program defines a **relational extension** of C by interpreting the user-defined predicates through the minimal model of the program over C.

It is clear that the traditional Horn clause logic programming system (Prolog) is a special case of this CLP paradigm where the underlying constraint system is an equational system over the free term algebra (Herbrand space) (see example 2.3.2). The *constrained SLD-resolution* procedure for computing the solved forms of query constraints can be defined as a non-deterministic algorithm consisting of following three deduction rules[1]:

Semantic Resolution

$$\frac{G : \exists X < \Pi \cup \{p(s_1,\ldots,s_n)\} \ [\!] \ c >}{G' : \exists X \cup Y < \Pi \cup \{B_1,\ldots,B_m\} \ [\!] \ c \cup \{c_1,\ldots,c_k\} \cup \{e_1 = s_1,\ldots,e_n = s_n\} >}$$

where $\forall Y . p(e_1 \ldots e_n) :- \ c_1,\ldots,c_k, B_1,\ldots,B_m$ is a variant of a clause in a program Γ.

Constraint Simplification:

$$\frac{G : \exists X < \Pi \ [\!] \ c >}{G' : \exists X < \Pi \ [\!] \ c' >}$$

if $\exists X . c \longrightarrow_c \exists X . c'$.

[1]We represent a goal, which is a conjunction of relational constraints (in Φ_Π) and primitive constraints (in Φ_C), by a multiset. Π denotes the multiset of all relational constraints and C denotes the multiset of all primitive constraints. The symbol $[\!]$ is used only to emphasise the distinguished components of a constraint set and should be read as multiset union

Finite Failure:

$$\frac{G : \exists X \Pi \; [] \; c}{\bot}$$

if $\exists X.c \longrightarrow_c \bot$, where $\bot$ denotes finite failure.

In constrained SLD resolution, semantic resolution generates a new set of constraints whenever a particular program rule is applied. The key point is that the unification component of SLD-resolution is replaced by solving a set of constraints over the computational domain. The constraints are accumulated during deduction and then simplified to their solved form. Whenever it can be established that the set of constraints is unsolvable, finite failure results.

The following theorem proved in [18] and [9] shows that constrained SLD-resolution is a sound and complete solver for a relationally extended constraint solver.

Theorem 2.3.3 *Given a constraint system L and its relational extension $P(L)$, constrained SLD-resolution is sound and complete when solving constraints with a $P(L)-$constraint Horn logic program.*

Applied to a given query (goal), let $\longrightarrow^{R*}\longrightarrow^{C*}$, $\longrightarrow^{R,C*}$ stand for derivations, semantic resolution, constraint simplification and constrained resolution respectively. Soundness and completeness of the calculus means that a goal constraint can be computed to its solved forms by enumerating constrained resolution derivations. For example, the following CLP program [3]:

```
InstalmentsCapital ([], 0);
InstalmentsCapital (i::x, c)  :- InstalmentsCapital (x, 1.1*c - i);
```

can be used to compute a series of instalments which will repay capital borrowed at a 10% interest rate. The first rules states that there is no need to pay instalments to repay zero capital. The second rule states that the sequence of `N+1` instalments needed to repay capital `c` consists of an instalment `i` followed by the sequence of `N` instalments which repay the capital increased by 10% interest but reduced by the instalment `i`. When we use the program to compute the value of `m` required to repay `$1000` in the sequence `(m, 2m, 3m)`, we compute the solved form of the goal constraint: `InstalmentsCapital ([m, 2m, 3m], 1000)`. One execution sequence is illustrated as:

```
InstalmentsCapital ([m, 2m, 3m], 1000)
--->R InstalmentsCapital (x,1.1c-i), x=[2m, 3m], i=m, c=1000
--->R InstalmentsCapital (x',1.1c'-i'), x=i'::x', c'=1.1c-i, x=[2m, 3m],
i=m, c=1000
--->C InstalmentsCapital (x',1.1c'-i'), i'=2m, x'=[3m],i=m, c'=1100-m
--->R InstalmentsCapital (x'',1.1c''-i''), x'=i''::x'',1.1c'- i'=c'', i'=2m,
x'=[3m],i=m,c'=1100-m
--->C InstalmentsCapital (x'',1.1c''-i''), x''=[], i''=3m,i'=2m,i=m,x'=[3m],
c'=1100-m, c''=1210-3.1m
--->R x''=[],1.1c''-i''=0, i''=3m, i'=2m, i=m, x'=3m, c'=1100-m,
c''=1210-3.1m
--->C 1.1(1210-3.1m)=3m
--->C m=207+413/641
```

As shown by this example, the semantic resolution step dynamically generates constraints which are then solved via constraint simplification. The interaction between these two steps is essential to the whole computation procedure. Constraints should be generated incrementally and then solved efficiently. More importantly, the constraint solving procedure should be able to control the computation effectively. That is, whenever finite failure is reached, the corresponding computational branch should be pruned promptly and no more constraints generated along that path. Thus, as mentioned in section 1, a proper treatment of the communication between deduction and constraint solving is crucial for the implementation of any constraint based deduction. In the next section we present a general computational scheme that provides a promising solution to this problem.

3 An Overview of the Scheme

In this section, we present a computational scheme for CLP systems. The scheme can be viewed as an abstract execution model (or abstract machine). Therefore, we will overview the scheme by outlining its instruction set, compilation scheme and computational mechanism.

3.1 The Instruction Set

The instruction set of the scheme is a context-free grammar $M = (\Sigma, T, P, I)$ where

1. $\Sigma : \{S, S_1, S_2, \ldots\}$ is a finite set of non-terminals;
2. $T : \{\alpha_1, \alpha_2, \ldots\}$ is a finite set (disjoint from Σ) of terminals;
3. P is a finite set of production rules, which are of the form:

   $$S \longleftarrow \prod_{i=0}^{k} \alpha_i \prod_{j=0}^{m} S_j$$

   where

   $$\prod_{i=0}^{k} \alpha_i : \alpha_1 \times \alpha_2 \times \ldots \times \alpha_k \text{ is a string of terminals}$$

   $$\prod_{j=0}^{m} S_j : S_1 \times S_2 \times \ldots \times S_m \text{ is a string of non-terminals}$$

   If there are n production rules having the same LHS:

   $$S_k \longleftarrow \prod_{i_1=0}^{h_1} \alpha_{i_1} \prod_{j_1=0}^{m_1} S_{j_1} \in P$$

   $$\ldots$$

   $$S_k \longleftarrow \prod_{in=0}^{hn} \alpha_{in} \prod_{jn=0}^{mn} S_{jn} \in P$$

   then we can combine these rules into one rule:

   $$S_k \longleftarrow \sum_{l=1}^{n} \prod_{i_l=0}^{h_l} \alpha_{il} \prod_{j_l=0}^{h_l} S_{jl} \in P$$

   where $+$ is an alternative operation
4. I is a special non-terminal, called the start symbol.

The derivation relation $\longrightarrow$ is defined as a rewriting relation over $(\Sigma \cup T)^*$ which contains all the strings consisting of terminals and non-terminals produced by regarding the set of rules P as a rewriting system. That is, $L_1 A L_2 \longrightarrow L_1 e L_2$ for any $L_1, L_2 \in (\Sigma \cup T)^*$ iff there exists $A \longrightarrow e \in P$. We use $\longrightarrow^*$ as the reflexive and transitive closure of $\longrightarrow$. A sentence generated by a grammar M is a string which contains only terminals and can be derived from I using the production rules. We call the set of all M-generated sentences Γ_M, a M-generated language. That is,

$$\Gamma_M = \{\alpha^* \in T^* \mid I \longrightarrow^* \alpha^*\}$$

In our scheme, a CLP program is compiled into a context free grammar (see section 3.2). The semantics of a compiled grammar can be defined by the semantic function $\mathcal{S}[\![]\!]$, mapping each string in $(\Sigma \cup T)^*$ to primitive constraints:

1. Each terminal α_i corresponds to a primitive constraint, i.e.

   $$\mathcal{S}[\![\alpha_i]\!] = \phi_C$$

2. Each sentence $\alpha^* = \alpha_1, \ldots, \alpha_i$ is a conjunction of constraints:

   $$\mathcal{S}[\![\alpha_1 \times \ldots \times \alpha_i]\!] = \mathcal{S}[\![\alpha_1]\!] \wedge \mathcal{S}[\![\alpha_2]\!] \wedge \ldots \& \mathcal{S}[\![\alpha_i]\!]$$

3. Each non-terminal S is a set of primitive constraints which correspond to the sentences derived from S, i.e.

$$\mathcal{S}[\![S]\!] = \{\mathcal{S}[\![\alpha^*]\!] \mid S \longrightarrow^* \alpha^*\}$$

Two associative -commutative operators, $\times$ and $+$ over strings are interpreted as the conjunction and disjunction operations on the constraint sets. Following this semantics, the language generated by a grammar is a set of constraints to be solved by the underlying constraint system. Computation in the scheme is performed by executing the grammar to generate constraints (simulating the resolution steps) and then solving the generated constraints. Therefore, it is reasonable to call a compiled grammar M the **instructions** of the scheme. Each sentence generated by the grammar is called a **solving plan**.

3.2 The Compilation Scheme

In order to obtain the grammar, we have to compile the source CLP program using the following compilation rules.

Definition 3.1 *For a CLP program*[2] Γ *with clauses of the form* $p(e_1, \ldots, e_n) :- \Phi$ *and a goal* G*, the function* $\mathcal{C}[\![\Phi]\!] : \Gamma \longrightarrow M$ *compiles program* Γ *to grammar* M *by:*

Compiling constraint conjunctions: *Conjunctions of constraints are compiled by compiling each component:*

$$\mathcal{C}[\![\Phi_1 \wedge \Phi_2]\!] = \mathcal{C}[\![\Phi_1]\!] \times \mathcal{C}[\![\Phi_2]\!]$$

Compiling primitive constraints: *Primitive constraints are directly compiled to terminals which means that primitive constraint solving is static, it will not involve resolution.*

$$\mathcal{C}[\![\{\}]\!] = \varepsilon$$

$$\mathcal{C}[\![\Phi]\!] = \alpha$$

where ε *is a special terminal denoting the empty constraint and* α *is a terminal denoting the primitive constraint* Φ*.*

Compiling defined relations: *If there are* n *rules for a predicate* p*, the relational constraint* $p(e)$ *will be solved by generating new constraints using resolution. Therefore, we use:*

$$\mathcal{C}[\![p(e)]\!] = \sum_{i=1}^{n} \alpha_i S_p(i) \;\; \textit{if there are n rules for p} : \begin{cases} p(e_1) :- \Phi_1 \\ p(e_2) :- \Phi_2 \\ \ldots \\ p(e_n) :- \Phi_n \end{cases}$$

where $\alpha_i = \{e = e_i\}$ *and* $S_p(i)$ *is a non-terminal associated with the ith rule for* p*.*

Compiling program rules: *Rules are compiled as production rules in the grammar and will be responsible for generating new primitive constraints:*

$$\mathcal{C}[\![p(e_i) :- \Phi_i]\!] = S_p(i) \longleftarrow \mathcal{C}[\![\Phi_i]\!]$$

Compiling the goal: *The goal is compiled into a rewrite rule for the start symbol* I*:*

$$\mathcal{C}[\![:- G]\!] = I \longleftarrow \mathcal{C}[\![G]\!]$$

By this compilation scheme, the following CLP program :

```
InstalmentsCapital ([], 0);
InstalmentsCapital (i::x, c)  :- InstalmentsCapital (x, 1.1*c - i);
```

with the goal constraint: `InstalmentsCapital ([m, 2m, 3m], 1000)` will be compiled into the grammar:

[2]As we described before, a CLP program defines a relationally extended constraint system $\Pi(C)$ over its underlying constraint system C. Since we assume any constraint system is closed under conjunction, it is easy to write a program rule as $p(e_1, \ldots, e_n) :- \Phi$ where Φ is the conjunction of primitive constraints and relational atoms, and therefore itself is a constraint in $\Pi(C)$.

$$I \longleftarrow \alpha_1 \times S_{ic}(1) + \alpha_2 \times S_{ic}(2)$$
$$S_{ic}(1) \longleftarrow \varepsilon$$
$$S_{ic}(2) \longleftarrow \alpha_3 \times S_{ic}(1) + \alpha_4 \times S_{ic}(2)$$

where $\alpha_1 = \{[m, 2m, 3m] = [], 1000 = 0\}$, $\alpha_2 = \{[m, 2m, 3m] = i :: x, 1000 = c\}$, $\alpha_3 = \{x = [], 1.1 * c - i = 0\}$ and $\alpha_4 = \{x = i' :: x', 1.1 * c - i = c'\}$ are primitive constraints in the underlying constraint system consisting of a unification procedure together with a constraint solver for linear equations over the real numbers [14]. Simplifying the production rules, we get the grammar:

$$I \longleftarrow \alpha_1 + \alpha_2 \times S_{ic}(2)$$
$$S_{ic}(2) \longleftarrow \alpha_3 + \alpha_4 \times S_{ic}(2)$$

3.3 The Computational Mechanism

For a compiled grammar, $M = (\Sigma, T, P, I)$, computation starts with the start symbol I and generates solving plans. Each solving plan denotes a conjunction of primitive constraints which are exactly the constraints that would be generated by a complete deduction path. Therefore, the language generated by M enumerates all constrained resolution derivations $\longrightarrow^{R,C*}$. A constraint solver can now be used to check whether the plan denotes a satisfiable constraint and to convert all satisfiable constraints to their solved forms. For example, the above grammar for the `InstalmentsCapital` program can be executed to generate the solving plans:

$$\{\alpha_1, \alpha_2\alpha_3^1, \alpha_2\alpha_4^1\alpha_3^2, \alpha_2\alpha_4^1\alpha_4^2\alpha_3^3, \alpha_2\alpha_4^1\alpha_4^2\alpha_4^3\alpha_3^4, \ldots,\}$$

where a superscript on a terminal distinguishes the different invocations of a grammar rule. This information is necessary for correct variable renaming. We will not discuss this issue in this paper because of space limitations. We assume all renamings are correctly performed. Using a linear unification procedure (e.g. the Martelli & Montanari algorithm [16]) together with a solver for linear equations over real numbers as the constraint solver of the underlying constraint system we get the solution:

$\{\perp\}$ by solving α_1

$\{\perp\}$ by solving $\alpha_2\alpha_3^1$

$\{\perp\}$ by solving $\alpha_2\alpha_4^1\alpha_3^2$

$\{m = 207 + 413/641\}$ by solving $\alpha_2\alpha_4^1\alpha_4^2\alpha_3^3$

We will find that all the remaining solving plans are of the form $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3\ldots$ and that they are unsatisfiable since the constraint corresponding to $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3$ is unsatisfiable. Therefore, the computation ends with the solution `m = 207+413/641` as the solved form of the goal constraint. In this example, $\alpha_2\alpha_4^1\alpha_4^2\alpha_3^3$ is the minimal constraint which causes the unsatisfiability of any further computation. Such a string is regarded as an **unsatisfiable constraint pattern**. As we will see in the next section, unsatisfiable constraint patterns provide important control information for pruning useless computational paths.

4 Machine Organization

In section 3, we illustrated the basic principle of the scheme in a "producer-filter" manner. The grammar behaves as a "producer", generating all solving plans denoting primitive constraints computed by the resolution steps and the constraint solver behaves as a filter, solving all satisfiable constraints and discarding all unsatisfiable ones. To construct a practical computational system, we must refine this "open loop" system to a "closed loop" system by designing a proper cooperation between constraint generation and constraint solving, and by exploiting fully the control information from the solver, to control the whole computation. A control strategy of the system will decide:

1. How to generate solving plans.
2. How to solve the corresponding constraints.
3. How the information about constraint solving can be used to prune useless computation.

To design the control strategy, we first define the **computational state** of the scheme.

Definition 4.1 *Given a grammar $M = (\Sigma, T, P, I)$ and $I \longleftarrow \omega \in P$, the computational state is defined inductively as*

$$I(0) = \omega$$

$$I(i+1) = \omega^{i+1}$$

where ω^{i+1} is derived by rewriting some non-terminal in I(i) In general $I(i)$ has the form

$$\sum_{m=1}^{n}(\prod_{j=1}^{r}\alpha_{mj})(\prod_{k=1}^{p}S_{mk})$$

Computational states provide a proper way to specify the execution behaviour of the scheme. The control strategy can be regarded as the transformation of computational states. The first issue is considered by deciding how to rewrite some non-terminal S_{mk} in ω^i in order to proceed to the next state $I(i+1)$. This determines the search strategy of the computation. If we rewrite only one particular non-terminal and insist on extracting only one solving plan at each level, it corresponds to depth-first search . If, by contrast, we rewrite all non-terminals and delay the checking for possible unsatisfiable constraint patterns, then we have breadth-first search. As to the second issue, the constraint solver should always be used to check the satisfiability at each stage for all newly generated solving plans. Particularly, the constraint solver should also check the satisfiability of a partially generated solving plan $(\prod_{j=1}^{r}\alpha_{mj})$ for a term $(\prod_{j=1}^{r}\alpha_{mj})(\prod_{k=1}^{p}S_{mk})$ of state $I(i)$ to discover unsatisfiable constraint patterns. This means that we don't prefer a pure depth-first search strategy. The third decision involves control over the search. It is closely related to the second decision. Since the constraint solver is assumed to have the incremental property, discussed in the section 1, the partially generated solving plans should be checked for unsatisfiable constraint patterns each time non-terminals are rewritten. This means, on the otherhand, we certainly don't explore the explosive breadth-first search strategy. Now, the inductive definition of the computational state can be refined to:

$$I(0) = \omega$$

and if

$$I(i) = \sum_{m=1}^{n}(\prod_{j=1}^{r}\alpha_{mj})(\prod_{k=1}^{p}S_{mk})$$

then $I(i+1) = \omega^{i+1}$ where ω^{i+1} is generated by the following steps:

1. Deleting all solving plans in $I(i)$, since all the corresponding constraints are being solved by the constraint solver.
2. Checking the partially generated solving plan $(\prod_{j=1}^{r}\alpha_{mj})$ for each $(\prod_{j=1}^{r}\alpha_{mj})(\prod_{k=1}^{p}S_{mk})$ of state $I(i)$ and then deleting all the terms $(\prod_{j=1}^{r}\alpha_{mj})(\prod_{k=1}^{p}S_{mk})$ whose partially generated solving plan $(\prod_{j=1}^{r}\alpha_{mj})$ is an unsatisfiable constraint pattern.
3. Rewriting non-terminals in $I(i)$ in terms of the chosen search strategy.

The whole computation terminates at the nth-stage iff $I(n)$ is empty. It is a distinguished feature of our scheme that the search strategy is open for the designer instead of traditionally fixed for an abstract machine. Many optimizations can be achieved by taking advantage of this feature. This "closed loop" system configuration is illustrated in Fig 1.

If we use the **level by level breadth-first search** mechanism by rewriting all non-terminals in w_i simultaneously in one step, we can illustrate the execution sequence of the above `InstalmentsCapital` program as follows:

$$1) I(0) = \alpha_1 + \alpha_2 \times S_{ic}(2)$$

1.1) Submit the solving plan α_1 to the solver to compute the solved form and α_2 to the solver to check its satisfiability. Since α_1 is unsatisfable, it is computed to $\perp$.
1.2) Delete α_1 in $I(0)$. Since α_2 is satisfiable, rewrite $S_{ic}(2)$ once. Then,

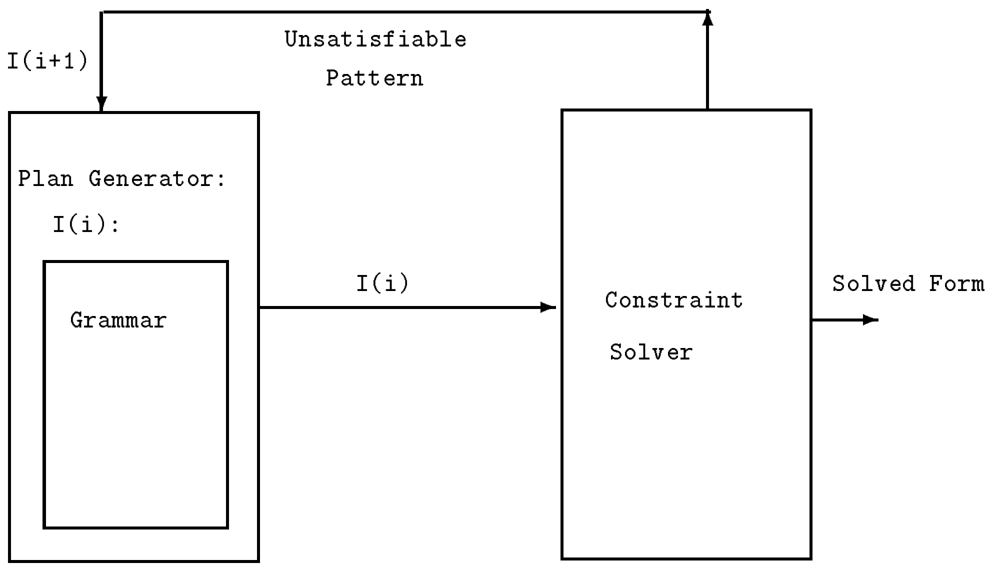

Figure 1: Machine Organization

$2) I(1) = \alpha_2\alpha_3^1 + \alpha_2\alpha_4^1 S_{ic}(2)$

2.1) Submit the solving plan $\alpha_2\alpha_3^1$ to the solver to solve the corresponding constraint and then check $\alpha_2\alpha_4^1$ for satisfiability. $\alpha_2\alpha_3^1$ is still unsatisfiable. Therefore, it is reduced to $\perp$.
2.2) Delete $\alpha_2\alpha_3^1$ in $I(1)$. Since $\alpha_2\alpha_4^1$ is satisfiable rewrite $S_{ic}(2)$ in $I(1)$ once. Then,

$3) I(2) = \alpha_2\alpha_4^1\alpha_3^2 + \alpha_2\alpha_4^1\alpha_4^2 S_{ic}(2)$

3.1) Submit the solving plan $\alpha_2\alpha_4^1\alpha_3^2$ to the solver to solve the corresponding constraint and then check $\alpha_2\alpha_4^1\alpha_4^2$ for satisfiability. $\alpha_2\alpha_4^1\alpha_3^2$ is still unsatisfiable. We get $\perp$ again.
3.2) Delete $\alpha_2\alpha_4^1\alpha_3^2$ in $I(2)$. Since $\alpha_2\alpha_4^1\alpha_4^2$ is satisfiable, we rewrite $S_{ic}(2)$ in $I(2)$ once. Then,

$4) I(3) = \alpha_2\alpha_4^1\alpha_4^2\alpha_3^3 + \alpha_2\alpha_4^1\alpha_4^2\alpha_4^3 S_{ic}(2)$

4.1) Submit the solving plan $\alpha_2\alpha_4^1\alpha_4^2\alpha_3^3$ to the solver to solve the corresponding constraint and then check $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3$ for satisfiability. $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3$ is satisfiable. Its solved form `m = 207+413/641` is then computed.
4.2) Whereas, $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3$ is unsatisfiable. Thus we delete the term $\alpha_2\alpha_4^1\alpha_4^2\alpha_3^3$ as well as $\alpha_2\alpha_4^1\alpha_4^2\alpha_4^3 S_{ic}(2)$ in $I(3)$. Then,

$5) I(4) = \varepsilon$

The computation terminates.

5 Conclusion and Related Work

We have presented in this paper a computational scheme for CLP programs. Due to the similar computational behaviour of all declarative constraint programming paradigms the scheme is suitable for modelling implementations of other declarative constraint programming systems such as constraint functional logic programming systems [8]. The novelty of the system comes from its concise separation of deduction (resolution) and constraint solving. Deduction is performed at compile time by partially evaluating the source program. Constraint solving then becomes the main computational task at run time. The control strategy over a computation is flexible and performed by exploiting fully the dynamic control information provided by constraint solving. These ideas originated from C.L.Chang and

J.R.Slagle's work on the connection graph method of theorem proving [1]. In their system, a grammar-based inference mechanism is used as a resolution proving procedure. Jiwei Wang has applied this mechanism to the implementation of Horn Clause Logic [20] together with a graph-oriented unification procedure. From an extensive investigation of these mechanisms we designed a general scheme that handles the major issues in implementing CLP language — organizing a flexible control strategy without excessive run time overheads. We are using the proposed scheme to implement a constraint logic programming system.

6 Acknowledgments

We would like to thank Dr. Joxan Jaffer for helpful discussions and Mr. Hendrich Lock for many discussions and comments. Yi-ke Guo would like to thank Mr. Jiwei Wang for many invaluable discussions and particularly, for educating him in the computational models of Horn Clause Logic. His Ph.D research on that topic directly inspired our work on the computational model for constraint logic programming systems. This research is partly supported by E.E.C. Phoenix Basic Research Action.

References

[1] Chang,C.L,Slagle,J.R. *Using Rewriting Rules for Connection Graphs to Prove Theorems* Artificial Intelligence Dec,1979.

[2] Colmerauer,A. *Prolog and Infinite Trees* in Logic Programming, ed. K.L.Clark and S.A.Tarnlund, Academic Press, New York, 1982.

[3] Colmerauer,A. *Opening the Prolog III Universe* BYTE, July, 1987.

[4] Darlington, J. Field, A.J. and Pull, H. *The unification of Functional and Logic languages* in Logic Programming: Functions, Relations and Equations, ed. Doug Degroot and G. Lindstrom, Prentice-Hall, 1986.

[5] Dershowitz, N. and Plaisted, D.A. *Equational Programming* in Machine Intelligence, D.Michie, J.E. Hayes and J. Richards, eds.,1986.

[6] Dershowitz, N. and Okada, M., *Conditional Equational Programming and the Theory of Conditional Term Rewriting* in Proc. of. FGCS 88, ed. by ICOT. 1988.

[7] Darlington, J, Guo, Y.K. *Narrowing and unification in Functional Programming* Proc.of RTA-89, LNCS 355, Apr.1989.

[8] Darlington,J, Guo Y.K.and Pull, H. *Introducing Constraint Functional Logic Programming* Technical Report , Dept. of Computing, Imperial College, London Feb.1991

[9] Guo,Y.K. *Constrained Resolution* Technical Report , Dept. of Computing, Imperial College, London Nov.1990

[10] Darlington,J, Guo, Y.K. *Constrained Equational Deduction* 2nd International Workshop on Conditional and Typed Term Rewriting System (CTRS90) June, 1990

[11] Darlington,J, Guo, Y.K. and Lock, H *A Classification for Integrating Functional and Logic Programming* Phoenix Project Report, Nov, 1989

[12] Darlington,J, Guo,Y.K. and Lock, H *Developing Phoenix Design Space* Esprit Phoenix Project Report, Apr, 1990

[13] Fay, M. J., *First-order Unification in an Equational Theory* in 4th Workshop on Automated Deduction, 1979.

[14] Jaffar, J., Lassez, J. and Maher, M. *Constraint Logic Programming* in Proc. of. 14th ACM symp. POPL. 1987.

[15] Lassez, J, M. Maher, and K. Marriot. *Unification revisited.* in J. Minker, editor, *Foundations of Deductive Databases and Logic Programming* . Morgan-Kaufman, 1988

[16] Martelli A, Montanari U. *An Efficient Unification Algorithm* ACM TPLS 1982, Vol 4 No.2.

[17] Saraswat V.A. (1987) *The Concurrent Logic Programming Language CP : Definition and Operational Semantics* Proc of SIGACT-SIGPLAN Symposium on Principles of Programming Languages p49-63 ACM New York.

[18] G.Smolka *Logic Programming over Polymorphically Order-Sorted Types* Ph.D Thesis Universitat Kaiserslautern 1989.

[19] Steele, G.L. *The Definition and Implementation of a Computer Programming Language Based on Constraints* Ph.D Thesis, M.I.T. AI- TR 595, 1980.

[20] Jiwei, Wang *Towards a New Computational Model for Logic Languages* CSM-128 Dept. of. Computer Science. University of Essex, March.1989