CFLP: a *Mathematica* Implementation of a Distributed Constraint Solving System

Mircea Marin[1] **Tetsuo Ida**[1] **Wolfgang Schreiner**[2]

[1] *Institute of Information Sciences and Electronics*
University of Tsukuba
Tsukuba 305−8573,Japan
{mmarin,ida}@score.is.tsukuba.ac.jp
[2] *Research Institute for Symbolic Computation (RISC−Linz)*
Johannes Kepler University
A−4040Linz, Austria
schreine@risc.uni-linz.ac.at

The need for combining and making cooperate various constraint solvers is widely recognized. Such an integrated system would allow solving problems that can not be solved by a single solver.
CFLP (**C**onstraint **F**unctional **L**ogic **P**rogramming System) is a distributed software system consisting of a functional logic programming interpreter running on one machine and a number of constraint solving engines running on other machines. The interpreter is based on a deterministic version of a lazy narrowing calculus which was extended in two main directions: (a) the possibility to specify explicit OR−parallelism,and (b) the possibility to specify constraints over various domains. The OR−parallelfeatures of the interpreter allow the decomposition of the solution space into different subspaces denoted by various sets of constraints; the individual sets are solved by different constraint solving engines in parallel and joined together to form the total solution set. This allows to investigate problems with large solution spaces using the computational power available in large computer networks.
The **CFLP** system is written entirely in *Mathematica* and uses the *MathLink* protocol for interprocess communication. The current implementation can solve problems expressible in functional logic and involving constraints such as systems of linear, polynomial, differential equations, and equations with invertible functions.

Introduction

Recent refinements of lazy narrowing calculi for solving equations over the domain of terms lead to successful implementations that can be used as starting point of developing a system based on equational reasoning. However, there are many applications such as theorem proving and computational geometry, where the expressive power of pure functional logic programming is not sufficient. These problems usually involve solving systems of constraint over various, like polynomial equations, differential equations, linear equations and inequations. The CLP scheme proposed by Jaffar and Lassez [JL:87] showed the possibility to integrate constraint solving capabilities over domains like reals, booleans, finite domains, rational and infinite trees in logic programming languages. Notably, almost all of these languages have a sequential implementation. Our investigation showed that it is possible to refine a lazy narrowing calculus with inference rules for handling constraints, and to develop a model for solving the constraints in a distributed environment.
The system is called **CFLP** (**C**onstraint **F**unctional **L**ogic **P**rogramming System) [MIS:99a] and consists of a functional logic interpreter based on a higher−order lazy narrowing calculus and a number of constraint solvers that may run on possibly different machines. The interpreter is implemented in *Mathematica*. The cooperation between interpreter and constraint solvers is coordinated by a constraint scheduler which schedules the individual constraint solvers to handle the solving tasks generated by the interpreter.
The paper is structured as follows. First, we illustrate by examples the system capabilities. Next, we describe the system architecture. The following subsections describe the three main components of the system: interpreter, scheduler, and constraint solvers. Finally, we draw some conclusions and directions of further research.

Examples

CFLP is delivered as a collection of *Mathematica* packages which add solving power to a normal *Mathematica* session. If **CFLP** is properly installed, a *Mathematica* user can enable it by loading the package TSolve.m

```
<< TSolve.m
```

In this section we illustrate the functionality of **CFLP** with a few examples.

■ A Polynomial Approximation Problem

Consider the problem of finding the relationship between the coefficients of a uni−variatepolynomial $f \in C[x]$ of degree 3 and a uni−variatepolynomial $g \in C[x]$ of degree 4 related by the constraint that they have the same values for $x = 1,2,3,4$.
In **CFLP**, the polynomial f of degree 3 can be specified by the λ−term $\lambda[\{x\}, a\,x^3+b\,x^2+c\,x+d]$ where *a, b, c, d* are constants, and the polynomial g of degree 4 can be specified by the λ−term $\lambda[\{x\}, m\,x^4+n\,x^3+p\,x^2+q\,x+r]$ where *m, n, p, q, r* are variables whose values have to be computed from the conditions

```
f[1] == g[1], f[2] == g[2], f[3] == g[3], f[4] == g[4]
```

which can be specified in the compact form

```
map[f, {1, 2, 3, 4}] == map[g, {1, 2, 3, 4}]                    (1)
```

where *map*[*f, list*] applies *f* to each element of *list*. In **CFLP**, *map* can be specified via the following functional logic program:

```
Prog = {
   map[F, {}] → {},
   map[F, [H | Tl]] → [F[H] | map[F, Tl]]};
```

The underlined symbols denote the free variables in the rewrite rules, and [*H* | *Tl*] denotes a list with head *H* and tail *Tl*.
To solve equation (1), we call

```
TSolve[map[λ[{x}, a x^3 + b x^2 + c x + d], {1, 2, 3, 4}] ≐
  map[λ[{x}, m x^4 + n x^3 + p x^2 + q x + r], {1, 2, 3, 4}],
 {a, b, c, d}, {},
 DefinedSymbol →
  {map : (Float → Float) × TyList[Float] → TyList[Float],
   f : Float → Float},
 Rules → {map[F, {}] → {}, map[F, [H | Tl]] → [F[H] | map[F, Tl]]},
 Constructor → {m : Float, n : Float, p : Float, q : Float, r : Float}]
```

```
Type checking program ...

Type checking goal ...

Enabling distributed constraint solving subsystem ...
```

$$\left\{d \to -24\,m + r,\ c \to 15\,m + 7\,n - \frac{7}{6}\,(60\,m + 6\,n) + 3\,p - \frac{3}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p) + q,\ b \to \frac{1}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p),\ a \to \frac{1}{6}\,(60\,m + 6\,n)\right\}$$

$$\left\{\left\{d \to -24\,m + r,\ c \to 15\,m + 7\,n - \frac{7}{6}\,(60\,m + 6\,n) + 3\,p - \frac{3}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p) + q,\ b \to \frac{1}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p),\ a \to \frac{1}{6}\,(60\,m + 6\,n)\right\}\right\}$$

The equational symbol ≐ denotes *strict* equality, i.e., the fact that the *values* of the two sides of the equation coincide. We identify the value of an expression with the result of rewriting it to an expression without defined symbols.
The syntax call of **TSolve** is similar to the syntax call of **Solve**. The first 3 arguments specify the goal to be solved, the main variables of the goal, and the eliminable variables of the goal respectively. In addition, the user of **CFLP** can control the behavior of the computation through various **TSolve** options. In this example, the options are:

DefinedSymbol: the list of type-annotated symbols defined by the user,
Rules: the list of definitions for the user–definedsymbols (the functional logic program),
Constructor: the list of type-annotated data constructors.

The constant coefficients of *f* are given as data constructors, and the unknown coefficients of *g* are given as main variables.
CFLP returns the unique solution:

$$\left\{d \to -24\,m + r,\ c \to 15\,m + 7\,n - \frac{7}{6}\,(60\,m + 6\,n) + 3\,p - \frac{3}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p) + q,\ b \to \frac{1}{2}\,(50\,m + 12\,n - 2\,(60\,m + 6\,n) + 2\,p),\ a \to \frac{1}{6}\,(60\,m + 6\,n)\right\}$$

Note the use of higher–ordervariables and λ–abstractionsin the specification of the goal and functional logic program. CFLP is able to handle equations involving operators defined outside the functional logic program. Furthermore, the computed answer is parametric, since *r* is a variable.

■ Electrical Circuits

This example illustrates the expressive power of the functional logic programming style extended with higher–order constructs (function variables and λ–abstractions)and capabilities to solve differential equations, linear equations, and systems of polynomial equations. We consider the problem of modelling the behavior in time of electrical circuits built from serial and parallel connections of elementary components such as resistors, inductors and capacitors.
First we introduce a new type constructor **ElComp** with associated data types **res**, **ind**, **cap**, **serial** and **parallel** to specify the characteristics of the electrical circuits. E.g., a resistor with resistance R=2 is specified as **res**[2], a serial connection of two resistors R1 and R2 is specified as **serial**[{**res**[R1],**res**[R2]}], and a parallel connection of a capacitor C and an inductor L is specified as **parallel**[{**cap**[C],**ind**[L]}]. The keyword **TyList** is the **CFLP** type constructor for lists.

```
TypeConstructor[ElComp = res[Float] | ind[Float] | cap[Float] |
   serial[TyList[ElComp]] | parallel[TyList[ElComp]]]
```

```
{ElComp}
```

Next, we define a function *spec* which describes the behavior in time *t* of an electrical component as a function of current *i*[*t*] and voltage *v*[*t*] in the circuit. The **CFLP** definition of *spec* is recursive, where the base case describes the behavior in time of the elementary circuits (resistor, inductor, capacitor) and the inductive case describes the behavior in time of serial and parallel connections of electrical components.

The **CFLP** program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
Prog =
  {spec[res[r], v : Float → Float, i : Float → Float] →
    True ⇐ {λ[{t}, v[t]] ≈ λ[{t}, r i[t]]},
   spec[ind[l], v : Float → Float, i : Float → Float] →
    True ⇐ λ[{t}, v[t]] ≈ λ[{t}, l i'[t]],
   spec[cap[c], v : Float → Float, i : Float → Float] →
    True ⇐ λ[{t}, i[t]] ≈ λ[{t}, c v'[t]],
   spec[serial[{}], λ[{t}, 0], i : Float → Float] → True,
   spec[serial[[comp | T]],
     v : Float → Float, i : Float → Float] → True ⇐
    {spec[comp, v1, i] ≈ True,
     spec[serial[T], v2, i] ≈ True,
     λ[{t}, v[t]] == λ[{t}, v1[t] + v2[t]]},
   spec[parallel[{}], v, λ[{t}, 0]] → True,
   spec[parallel[[comp | T]], v, i] → True ⇐
    {spec[comp, v, i1] ≈ True,
     spec[parallel[T], v, i2] ≈ True,
     λ[{t}, i[t]] == λ[{t}, i1[t] + i2[t]]}};
```

Consider the problem of finding the behavior in time of the current in an RLC circuit, under the restriction that the voltage is constant in time and the current was initially set to 0.

In this case, the goal which we want to solve is

```
G = {i[0] == 0, spec[serial[{res[R0], ind[L0], cap[C0]}],
     λ[{t : Float}, V], λ[{t : Float}, i[t]]] ≐ True};
```

in variable *i*, and R0, L0, C0 and V are data constructors which denote arbitrary but fixed characteristic values for the resistor, inductor, capacitor and voltage of the circuit.
Now we can ask CFLP to solve the problem:

```
TSolve[G, {i : Float → Float}, {},
 Rules → Prog,
 Constructor → {R0 : Float, L0 : Float, C0 : Float, V : Float},
 DefinedSymbol →
  {spec : ElComp × (Float → Float) × (Float → Float) → Bool}]
```

```
Type checking program ...
```

```
Type checking goal ...
```

$$\left\{i \to \lambda\left[\{t\},\ e^{\frac{\left(-\sqrt{C0}\ R0-\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1 - e^{\frac{\left(-\sqrt{C0}\ R0+\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1\right]\right\}$$

$$\left\{\left\{i \to \lambda\left[\{t\},\ e^{\frac{\left(-\sqrt{C0}\ R0-\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1 - e^{\frac{\left(-\sqrt{C0}\ R0+\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1\right]\right\}\right\}$$

The system computes the parametric solution

$$\left\{i \to \lambda t.\, e^{\frac{\left(-\sqrt{C0}\ R0-\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1 - e^{\frac{\left(-\sqrt{C0}\ R0+\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1\right\}$$

which is represented in *Mathematica* by

$$\left\{i \to \lambda\left[\{t\},\ e^{\frac{\left(-\sqrt{C0}\ R0-\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1 - e^{\frac{\left(-\sqrt{C0}\ R0+\sqrt{-4\ L0+C0\ R0^2}\right)\ t}{2\sqrt{C0}\ L0}}\ C1_1\right]\right\}.$$

Note the usage of type annotations in the goal, list of variables, and options of the **TSolve** call; **CFLP** has integrated a polymorphic type checker to verify the type consistency of the program and goal provided by the user.

■ A Problem Involving Solver Cooperation

Consider the following program

```
<< TSolve.m
```

```
Prog = {f[X : Compl] → g[Y : Compl] ⇐ (X + Y == 3 ⋁ X^2 - Y == 9)}
```

```
{f[X : Compl] → g[Y : Compl] ⇐ X + Y == 3 || X^2 - Y == 9}
```

in complex variables X, Y, and the goal

```
G = {f[X] ≈ g[Y], g[Y^2] ≈ g[Z^2 - 1],
  λ[{x : Compl}, H'[x]] == λ[{x : Compl}, Z x^Y], H'[1] == 4}
```

```
{f[X] ≈ g[Y], g[Y^2] ≈ g[-1 + Z^2],
 λ[{x : Compl}, H'[x]] == λ[{x : Compl}, x^Y Z], H'[1] == 4}
```

with indeterminates X,H,Y,Z. In this example, the operator ⋁ denotes logical disjunction, and it can be used in goals and conditional parts of rewrite rules to express alternative solutions.

Solving this goal requires a cooperation among solvers for linear, polynomial and differential equations over the domain of complex numbers. To solve G, we call:

```
TSolve[G, {X, H, Y, Z}, {},
 Rules → Prog,
 DefinedSymbol → {f : Compl → Compl},
 Constructor → {g : Compl → Compl}]
```

```
Type checking program ...

Type checking goal ...
```

$$\left\{Y \to \sqrt{15},\ X \to 3 - \sqrt{15},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_7\right],\ Z \to 4\right\}$$

$$\left\{Y \to -\sqrt{15},\ X \to 3 + \sqrt{15},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_8\right],\ Z \to 4\right\}$$

$$\left\{Y \to -\sqrt{15},\ X \to -\sqrt{9-\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_9\right],\ Z \to 4\right\}$$

$$\left\{Y \to -\sqrt{15},\ X \to \sqrt{9-\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_{10}\right],\ Z \to 4\right\}$$

$$\left\{Y \to \sqrt{15},\ X \to -\sqrt{9+\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_{11}\right],\ Z \to 4\right\}$$

$$\left\{Y \to \sqrt{15},\ X \to \sqrt{9+\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_{12}\right],\ Z \to 4\right\}$$

$$\left\{\left\{Y \to \sqrt{15},\ X \to 3 - \sqrt{15},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_7\right],\ Z \to 4\right\},\right.$$
$$\left\{Y \to -\sqrt{15},\ X \to 3 + \sqrt{15},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_8\right],\ Z \to 4\right\},$$
$$\left\{Y \to -\sqrt{15},\ X \to -\sqrt{9-\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_9\right],\ Z \to 4\right\},$$
$$\left\{Y \to -\sqrt{15},\ X \to \sqrt{9-\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}} + C1_{10}\right],\ Z \to 4\right\},$$
$$\left\{Y \to \sqrt{15},\ X \to -\sqrt{9+\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_{11}\right],\ Z \to 4\right\},$$
$$\left.\left\{Y \to \sqrt{15},\ X \to \sqrt{9+\sqrt{15}},\ H \to \lambda\left[\{x \bullet 1\},\ \frac{4\, x \bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}} + C1_{12}\right],\ Z \to 4\right\}\right\}$$

In this case, the solutions computed by **CFLP** are parametric, and the system has generated the auxiliary variables $C1_7, C1_8, C1_9, C1_{10}, C1_{11}, C1_{12}$ to express them. It is not hard to see that these are all the solutions of G.

The Structure of the System

CFLP is a distributed software system for solving systems of equational goals in theories that can be represented as sets of conditional rewrite rules over a term algebra whose signature is extended with external operators. The external operators are used for expressing constraints over various domains.
The system consists of 3 components:

- an interpreter based on a higher−orderlazy narrowing calculus ***C***
- a cooperative constraint solver consisting of

- a scheduler which implements a strategy ***S*** for solver cooperation,
- various specialized constraint solvers.

The general architecture of the system is depicted below.

Figure 1

■ The Interpreter

The **CFLP** interpreter is designed to solve systems of equations between simply−typed λ−terms in theories axiomatized by finite sets of conditional rewrite rules of a certain kind, called *pattern rewrite rules*.
Roughly speaking, simply−typed λ−terms are analogous to the **Function** construct of *Mathematica*, modulo the following differences:

1. the keyword **Function** is replaced with the keyword λ,
2. each such construct (which we will henceforth call a λ-*term*) should be well−typed, i.e. a type can be inferred from the types of its base components (variables, constants, and function symbols).

The current version of CFLP recognizes the following base types:

Type	**literals**
`Int`	`Integer literals of` *`Mathematica`*
`Float`	`Real literals of` *`Mathematica`*
`Compl`	`Complex literals of` *`Mathematica`*
`String`	`String literals of` *`Mathematica`*
`Bool`	`True, False`

and the following constructed types:

$\tau_1 \times \ldots \times \tau_n \to \tau$	type of functions with n arguments of types τ_1, ..., τ_n, and result of type τ
TyList[τ]	type of lists with elements of type τ

For lists, we have adopted both the *Mathematica* notation $\{a_1,\ldots,a_n\}$ to represent a list with elements $a_1,\ldots,a_n$, and the *Prolog*−likenotation [H | T] to represent a list with head H and tail T.

The **CFLP** interpreter is based on a higher−order lazy narrowing calculus *C* for pattern rewrite systems [MIS:99b] extended to handle also conditional pattern rewrite systems [Pre:98]. The underlying calculus essentially consists of the unification rules of higher−order patterns [Nip:93] and the lazy narrowing rules. Since the search space of such a calculus is extremely large, we have implemented a couple of refinements with smaller search space for solutions, which are sound and complete for classes of functional logic programs which are of practical interest.
We have extended these calculi in two main directions:

(a) the possibility to specify *constraints*, i.e., equations that can not be solved by narrowing, but for which specialized solvers are available, and

(b) the possibility to specify explicit OR−parallelism.

The interpreter successively decomposes the goal towards an answer substitution by applying the inference steps of the underlying lazy narrowing calculus. The only equations which can not be solved in this way are the constraints. The constraints generated upon derivations are collected and sent to be solved by specialized constraint solvers via the component called *scheduler* (see Fig. 1).

Note that the nondeterministic selection of a rewrite rule for a defined symbol and the explicit OR−formulas cause the initial goal to be reduced to disjoint sets of constraints that can be solved in parallel. E.g., in the last illustrative example, the reduction of the initial goal

$$f[X] \approx g[Y],\ g[Y^2] \approx g[Z^2 - 1],$$
$$\lambda[\{x : \text{Compl}\},\ H'[x]] \approx \lambda[\{x : \text{Compl}\},\ Z\,x^Y],\ H'[1] == 4$$

involves the decomposition of the equation f[X]≈g[Y] into simpler equations. The transformation step performed by our lazy narrowing calculus is:

$$f[X] \approx g[Y] \Rightarrow X \approx X0,\ g[Y0] \approx g[Y],\ (X0 + Y0 == 3 \bigvee X0^2 - Y0 == 9) \qquad (2)$$

where X0,Y0 are new variables. In this step we have used the fresh variant

$$f[X0] \to g[Y0] \Leftarrow (X0 + Y0 == 3 \bigvee X0^2 - Y0 == 9)$$

of the rewrite rule which defines f.

During step (2), an OR−subgoal is produced and as a result the initial goal is finally decomposed in two disjoint sets of constraints. These sets of constraints are sent to be solved to the scheduler.

■ The Scheduler

The scheduler coordinates the process of solving the systems of constraints received from the interpreter. In order to solve these sets of constraints, the constraint scheduler maintains a dynamic data structure called a *constraint tree*. The nodes of the constraint tree are pairs of the form $\langle\theta,cs\rangle$ where θ is a substitution and *cs* is a set of constraints.
Whenever a set of constraints *cs* is received from the interpreter, the scheduler adds a new son $\langle\epsilon,cs\rangle$ to the root of the constraint tree. Here ϵ stands for the empty substitution. The scheduler expands this tree by applying constraint solving methods in parallel to its leaf nodes.
A leaf node $\langle\theta,cs\rangle$ is expanded w.r.t. a method *m* as follows:

1) cs is decomposed into a set cs_1 of constraints to which method m can be applied, and a set cs_2 of other constraints.

2) cs_1 is sent to be solved to a constraint solver which implements method m. We call such a solver an m−solver.

3) If the m-solver detects cs_1 inconsistent then the node $\langle\theta,cs\rangle$ is marked as inconsistent. Otherwise the m−solver returns $\langle\epsilon,cs\rangle$ if it can not reduce cs_1, or it computes a finite sequence of pairs $\langle\theta_1, \text{cs}_1^{'}\rangle,\ldots,\langle\theta_p, \text{cs}_p^{'}\rangle$ with the property that θ is a solution of cs iff there exists a solution σ_i of some $\text{cs}_i^{'}$ $(1\leq i\leq p)$ such that $\theta = \theta_i\sigma_i$.

4) If the sequence $\langle\theta_1, \text{cs}_1^{'}\rangle,\ldots,\langle\theta_p, \text{cs}_p^{'}\rangle$ is computed by the m−solver, then the nodes $\langle\theta\ \theta_i,\ cs_2\theta_i \bigcup \text{cs}_i^{'}\rangle$ $(1\leq i\leq p)$ are added to the constraint tree as sons of the node $\langle\theta,cs\rangle$.

A node $\langle\theta,cs\rangle$ is *final* if it can not be reduced by any m−solverwhich is available.

The implementation of the scheduling algorithm is inspired by the work of Hong [Hong:94]. The scheduler can be regarded as a component parameterized with a list $\mathbf{M}=\{m_1,\ldots,m_k\}$ of constraint solving methods. The scheduler implements a cooperation strategy S which repeatedly applies the sequence $m_1,\ldots,m_k$ of methods to the leaves of the constraint tree until they become either final or inconsistent. As soon as a final node is generated, it is made accessible to the interpreter.

■ The Constraint Solvers

The constraint solvers are implementations of the constraint solving methods specified to the scheduler through the list **M**. The current implementation of **CFLP** provides constraint solvers which implement the following methods for solving constraints over the domains of real and complex numbers:

1) **Linear**: a method for solving systems of linear equations and systems of equations with invertible functions,
2) **Polynomial**: a method for solving systems of polynomial equations,
3) **Derivative**: a method for solving derivative equations,
4) **PartialDerivative**: a method for solving partial derivative equations.

These methods are tried in the order presented.

All solvers are implemented by separate *Mathematica* processes executing in parallel and communicating with the constraint scheduler via *MathLink* connections. There are two types of constraint solvers in **CFLP**:

1) Local constraint solvers: these solvers run as subsidiary *Mathematica* processes of the **CFLP** constraint scheduler.

2) Shared constraint solvers: a shared constraint solver is started from outside a **CFLP** session and can be connected later to more **CFLP** schedulers which may run on possibly different machine.

The user can adjust the distributed constraint solving component of the system by specifying the number of local constraint solvers which are started at system initialization, and the remote machines on which to look for shared constraint solvers.
The communication mechanism between scheduler and constraint solvers is implemented completely in *MathLink* [Wol:96]. Therefore, **CFLP** is a platform−independentsoftware system and can be used in heterogeneous networks.

Conclusion and Future Work

CFLP is a software system consisting of a functional logic interpreter and a distributed constraint solving system, which provides support for solving systems of equations over various constraint domains. The functional logic component allows to define one's own abstractions in an easy and comfortable way. In the current implementation we have integrated solvers for linear equations and equations with invertible functions, polynomial equations, differential equations, and partial differential equations. The constraint solvers are all implemented on top of the constraint solving capabilities of *Mathematica*.

We intend to further develop the system by integrating more constraint solving capabilities. The constraint solvers may act either on disjoint sets of constraints or on overlapping subsystems. Currently, the constraint solvers are not allowed to act simultaneously on a leaf node of the constraint tree. An optimization would be to act simultaneously with more solvers on the same node in situations when the subsystems of equations are non-overlapping.

Because of the higher−order extensions of the functional logic programming framework (function variables and λ−abstractions), the search space of lazy narrowing is very large. We have identified and implemented various refinements of a lazy narrowing calculus which perform a more deterministic search for solutions for classes of functional logic programs of practical interest. We intend to continue our research and identify better refinements.

The system is intended to be used by researchers in functional and logic programming languages, and by researchers in constraint solving who are willing to make use of its expressive and computational power.

■ References

[Hong:94] H. Hong. RISC−CLP(CF):Constraint Logic Programming over Complex Functions. In F. Pfennig (ed.): *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, pp. 99−113,Springer−VerlagBerlin Heidelberg, 1994.

[JL:87] J. Jaffar and J.−L. Lassez. *Constraint Logic Programming*. Technical Report. Department of Computer Science, Monash University. Clayton 1987.

[MIS:99a] M. Marin, T. Ida and W. Schreiner. A Distributed System for Solving Equational Constraints based on Lazy Narrowing Calculi. In JSSST *Workshop on Programming and Programming Languages (PPL'99)*, Togawa, Japan, March 17−19,1999.

[MIS:99b] M. Marin, T. Ida and T. Suzuki. On Reducing the Search Space of Higher−OrderLazy Narrowing. In A. Middeldorp, T. Sato (eds.): *FLOPS'99, LNCS 1722*, pp.319−334, November 1999. Springer−Verlag Berlin Heidelberg, 1999.

[Nip:93] T. Nipkow. Functional Unification of Higher−orderPatterns. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, pp.64−74,1993.

[Pre:98] C. Prehofer. *Solving Higher−orderequations. From Logic to Programming*. Birkhäuser Boston, 1998.

[Wol:98] S. Wolfram. The *Mathematica* Book. Third Edition. Wolfram Media and Cambridge University Press, 1996.