CFLP: a Distributed Constraint Solving System for Functional Logic Programming

Mircea Marin
Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
`Mircea.Marin@risc.uni-linz.ac.at`
`Wolfgang.Schreiner@risc.uni-linz.ac.at`

Abstract. In this paper we present the distributed software system CFLP for solving constraints over the complex numbers in a functional logic programming environment. We describe the general architecture of the system and the functionality of its main components.

1 Introduction

Functional and logic programming are the most important declarative programming paradigms and interest in combining them has grown over the last decade (see [7] for a survey). A sound and complete method to solve equations with respect to a terminating and confluent term rewriting systems is *narrowing* [9]. There are many variants of narrowing that can be successfully used as a implementation basis of a functional logic programming (FLP) language. Among them, the lazy narrowing calculi (LNC) developed by **S**ymbolic **C**omputation **RE**search group (SCORE) at the University of Tsukuba were successfully implemented in the *Mathematica* system [5]. In particular, the calculi LCNC [6] and Higher-Order LNC [14] attract interest because of their generalization to functional logic programs given conditional term rewriting systems and a variant of higher order rewriting. An important feature of these lazy narrowing calculi is the possibility to extend them with special rules for handling constraints.

Extending logic programming (LP) with constraint solving capabilities is one of the recent trends in declarative programming. The CLP scheme identified by Jaffar and Lassez in 1987 [10] has been successfully incorporated to LP for integrating constraint solving capabilities over domains like the reals (CLP($\mathcal{R}$) [11] or RISC-CLP(Real) [8]), booleans and finite domains (CHIP [3]), rational and infinite trees (Prolog III [2]). Notably, almost all these languages have a sequential implementation.

We will present here the results of our investigation in extending an LNC calculus inspired by HLNC and CFLP with distributed constraint solving capabilities. Our investigation showed that it is possible to extend an FLP language based on a lazy narrowing calculus with constraint solving capabilities by developing a model for solving the constraints in parallel.

The system is called CFLP and is under development at RISC-Linz [12]. It is implemented as a set of Mathematica packages based on the earlier implementation of the calculus LCNC developed by T. Ida and his SCORE group coworkers at University of Tsukuba, Japan.

The system components—interpreter, constraint scheduler and constraint solvers—are separate processes that may run asynchronously on different machines and communicate by message passing. For the moment, the CFLP components are implemented as *Mathematica* packages running in different Mathematica sessions and communicating via *MathLink* connections. MathLink is a library of functions that implement a protocol for sending and receiving Mathematica expressions over the network(see [4]).

Currently our implementation focuses on solving constraints over the domain of complex numbers involving systems of linear equations, systems of polynomial equations, systems of differential equations and partial differential equations.

The paper is structured as follows: In Section 2 we present some example programs to illustrate how CFLP finds solutions. Section 3 describes the general architecture of the CFLP system. Sections 4, 5 and 6 describe the three main components of the system: interpreter, constraint scheduler and constraint solvers. The last section presents conclusions and further remarks.

2 Examples

We illustrate on two 3examples the functionality of our system.

2.1 Example 1

Consider the system of constraints $S : f[x] \approx g[z]$ in variables x and z, where g is a constructor operator and f is a defined function symbol defined by the conditional rewrite rule

$$f[x] \longrightarrow g[y] \Leftarrow (x + y \approx 3) \wedge (x^2 - y \approx 3)$$

in variables x and y. Here, '$\approx$' stands for the equality relation defined by the reflexive, symmetric and transitive closure of the term reduction relation $\Rightarrow_{\mathcal{R}}$ where

$$\mathcal{R} = \{[f[x] \to g[y] \Leftarrow (x + y \approx 3) \wedge (x^2 - y \approx 3)\}$$

is the given set of conditional rewrite rules.

For solving this system in CFLP, we perform the call

TSolve$[f[x] \approx g[z], \{x, y, z\}$, Rule -> $\{[f[x] \longrightarrow g[y] \Leftarrow (x + y \approx 3) \wedge (x^2 - y \approx 3) \quad \}$
DefinedFunctionSymbol -> $\{f$::Complex $\to$ Complex$\}$,
Constructor->$\{g\}]$

It takes as first argument the equality or list of equalities which we want to get solved, and as second argument the variables for which we want to find answer substitutions. TSolve calls accept the following options:

– Rule, for specifying the equational theory $\mathcal{R}$ with respect to which we try to find solutions. In our setting, $\mathcal{R}$ is always a list of (conditional) rewtite rules. By default (i.e., if not specified in the call), $\mathcal{R}$ is the empty list.
– DefinedFunctionSymbol, for specifying the list of operators for which we provide rewrite rules (so called defined function symbols). These symbols may be type annotated. In our example, we have only one function symbol, f, which is a function over the complex numbers.
– Constructor, for specifying the list of constructors of our equational theory.

The Mathematica palette `CFLPsyntax.nb` is provided for typing in expressions in CFLP syntax.

CFLP finds the answer substitutions

$$\{\{x \mapsto 2, z \mapsto 1\}, \{x \mapsto -3, z \mapsto 6\}\}$$

by performing the following refutations. First, the following derivations are produced by the CFLP interpreter:

$\{f[x] \approx g[z]\} \rightsquigarrow \{x \approx y0, g[y0] \approx g[z], x0 + y0 \approx 3, \wedge, x0^2 - y0 \approx 3\} \rightsquigarrow_{x \mapsto y_0} \{g[y0] \approx g[z], x0 + y0 \approx 3, x0^2 - y0 \approx 3\} \rightsquigarrow \{y0 \approx z, x0 + y0 \approx 3, x0^2 - y0 \approx 3\} \rightsquigarrow_{y0 \mapsto z} \{x0 + z \approx 3, x0^2 - z \approx 3\} \rightsquigarrow \{\lceil x0 + z \approx 3 \rceil, x0^2 - z \approx 3\} \rightsquigarrow \{\lceil x0 + z \approx 3 \rceil, \lceil x0^2 - z \approx 3 \rceil\}$. Here, $x0$ and $y0$ are new variables introduced by the application of the outermost narrowing inference rule of the underlying calculus of the interpreter. In the last two inference steps the constraints are detected: they are marked, here with the brackets $\lceil \cdot \rceil$, for being treated later by the constraint scheduling component (CS) of the CFLP system. CS is triggered when all the equations in the goal get marked.

Since the goal $\{\lceil x0 + z \approx 3 \rceil, \lceil x0^2 - z \approx 3 \rceil\}$ has marked all its equations, the constraint scheduler will be invoked. The interpreter sends to CS the substitution $\theta = \{x \mapsto y_0, y0 \mapsto z\}$ computed so far together with the system $\{x0 + z \approx 3, x0^2 - z \approx 3\}$ of constraints to be solved. As a result, the scheduler will find the answer substitutions $\{\{x0 \mapsto 2, z \mapsto 1\}, \{x0 \mapsto -3, z \mapsto 6\}\}$ which are composed with θ to get the answer substitution returned by the call. The next example illustrates the solving mechanism of the constraint scheduler.

2.2 Example 2

Consider the system of constraints:

$$S : \begin{cases} y'[x] = k * x \\ y[2] = 3 \\ 2 * y[1] = 3 \\ y[t] = 5.5 \end{cases}$$

For solving this system in CFLP, we first encode it according to the CFLP syntax. The Mathematica assignment

$S := \{y'[x] \approx k * x, y[2] \approx 3, 2 * y[1] \approx 3, y[t] \approx 5.5\};$

defines S as the system of equations to be solved. The CFLP command for solving S is

$\mathbf{TSolve}[S, \{y, k, t\}]$

where the second argument denotes the variables of S. CFLP finds the answer substitution $\{y \mapsto \lambda[\{u\}, \frac{u^2}{2} + 1], k \mapsto 1, t \mapsto 3\}$. The derivation produced by the interpreter for computing the answer is

$$\begin{aligned}
&S \leadsto \{\lceil y'[x] \approx k * x\rceil, y[2] \approx 3, 2 * y[1] \approx 3, y[t] \approx 5.5\} \leadsto \\
&\{\lceil y'[x] \approx k * x\rceil, \lceil y[2] \approx 3\rceil, 2 * y[1] \approx 3, y[t] \approx 5.5\} \leadsto \\
&\{\lceil y'[x] \approx k * x\rceil, \lceil y[2] \approx 3\rceil, \lceil 2 * y[1] \approx 3\rceil, y[t] \approx 5.5\} \leadsto \\
&\{\lceil y'[x] \approx k * x\rceil, \lceil y[2] \approx 3\rceil, \lceil 2 * y[1] \approx 3\rceil, \lceil y[t] \approx 5.5\rceil\}.
\end{aligned}$$

We use $\lceil\cdot\rceil$ to mark the equations (so called constraints) that are sent to specialized constraint solvers. After all the remaining equations are marked as constraints, we try to solve them by applying a sequence of constraint solving methods over the complex numbers. We get the following constraint solving refutation:

$$\begin{aligned}
&\{\underline{y'[x] \approx k * x}, y[2] \approx 3, 2 * y[1] \approx 3, y[t] \approx 5.5\} \\
&\quad \leadsto^{\mathtt{Derivative}}_{y \mapsto \lambda[\{u\}, k\frac{u^2}{2} + d]} \{\underline{2 * k + d \approx 3, k + 2 * d \approx 3}, k * \tfrac{t^2}{2} + d \approx 5.5\} \\
&\quad \leadsto^{\mathtt{Linear}}_{k \mapsto 1, d \mapsto 1} \{\tfrac{t^2}{2} + 1 \approx 5.5\} \leadsto^{\mathtt{Polynomial}}_{t \mapsto 3} \{\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \leadsto^{\mathtt{Polynomial}}_{t \mapsto -3} \{\}
\end{aligned}$$

In this case we successively applied methods `Linear`, `Derivative` and `Polynomial`. The curly arrow $\leadsto$ denotes an inference step and has as superscript the method considered and as subscript the substitution computed at that step. The underlined equations represent the constraints to which the method is applied.

By composing the individual step substitutions we get the computed substitution $\{y \mapsto \lambda[\{x\}, \frac{x^2}{2} + 1], k \mapsto 1, t \mapsto 3, z \mapsto x\}$. By restricting it to the set of variables given in the **TSolve** call, we get the answer substitution returned by the call.

3 The CFLP System

CFLP is a distributed software system built on top of the *Mathematica* computer algebra system to increase the solving capabilities of a normal Mathematica session. It is composed of three main components:

- A functional logic *interpreter* running on one machine. The interpreter is based on the calculus HLCNC, an extension of the calculi LCNC and Higher-Order LNC [5].
- Various specialized *constraint solvers* running concurrently on different machines. Each constraint solver is an *instance* of a particular constraint solving method.
- A *constraint solving scheduler* which is responsible with coordinating the solving process of the constraints appearing in the user program.

The CFLP components are implemented as Mathematica packages that are loaded in Mathematica sessions which communicate via *MathLink* connections (see Figure 1).

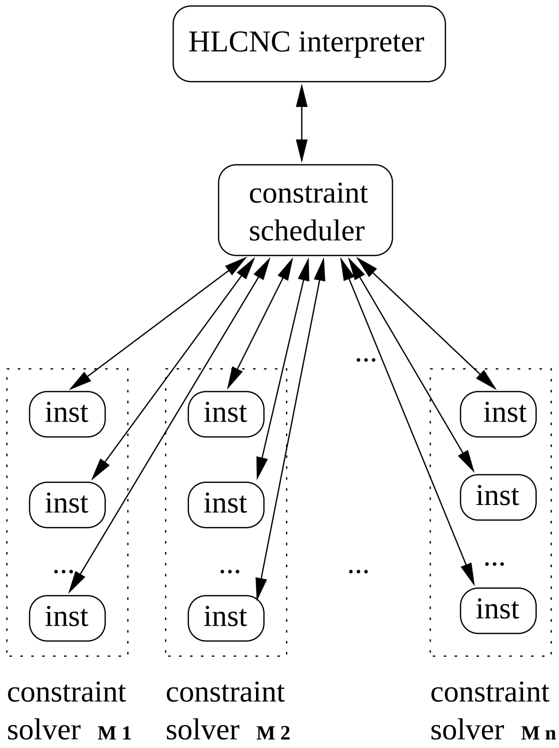

Fig. 1. CFLP: general architecture

Every constraint solving method $\mathbf{M}_i$ has associated a number of active instances (denoted in Figure 1 by **inst**) which may run in parallel (also on different machines of a network).

The communication between interpreter and scheduler is explained in Section 4 and 5. The communication between scheduler and constraint solvers is explained in Section 5 and 6.

4 The Functional Logic Interpreter

First, we will briefly describe the language of the CFLP interpreter. It is based on the calculus LCNC proposed recently by Hamada and Middledorp [6], which we have extended in two main directions:

- The set of *function symbols* $\mathcal{F}$ now partitioned into three disjoint sets: $\mathcal{F}_c$ (constructors), $\mathcal{F}_d$ (defined function symbols), and $\mathcal{F}_e$ (external symbols).

The meaning of the first two sets is the same as for LCNC, whereas external function symbols identify the so called *constraints*, or equations that can not be handled in a pure logical programming framework.

– A higher-order lazy narrowing was introduced by extending LCNC with the special inference rules proposed by Suzuki, Nakagawa and Ida in [14].

Due to lack of space we will omit the details of the second extension and concentrate on the first one.

4.1 The Underlying Calculus

A *CFLP program* is a set $\mathcal{R}$ of conditional equations (or conditional rewrite rules) of the form $l \to r \Leftarrow (l_1 \approx r_1 \wedge \ldots \wedge l_n \approx r_n)$ where l, r, l_i, r_i $(1 \leq i \leq n)$ are CFLP terms. The sign "$\approx$" in the condition part denotes equality interpreted as joinability. That is, for a given CFLP program $\mathcal{R}$ we have $s \to_{\mathcal{R}} t$ if there exist $l \to r \Leftarrow (l_1 \approx r_1 \wedge \ldots \wedge l_n \approx r_n) \in \mathcal{R}$, a substitution σ and a context $C[\,]$ such that $s \equiv C[l\sigma]$, $r \equiv C[r\sigma]$ and $l_i\sigma \downarrow_{\mathcal{R}} r_i\sigma$ for $1 \leq i \leq n$.

A *program goal* is a set $\mathcal{S}$ of the form $\{g_1, \ldots, g_n\}$, where g_i is either an equation $u_{i,1} \approx v_{i,1}$, a parallel OR-equation $u_{i,1} \approx v_{i,1} \vee \ldots \vee u_{i,n_i} \approx v_{i,n_i}$, or a parallel AND-equation $u'_{i,1} \approx v'_{i,1} \wedge \ldots \wedge u'_{i,n_i} \approx v'_{i,n_i}$. A *solution* w.r.t. $\mathcal{R}$ for $\mathcal{S}$ is a substitution σ such that $(\exists j. u_{i,j}\sigma =_{\mathcal{R}} v_{i,j}\sigma) \wedge (\forall k. u'_{i,k}\sigma =_{\mathcal{R}} v'_{i,k})$ for all $1 \leq i \leq n$ where $=_{\mathcal{R}}$ is the reflexive, symmetric, transitive closure of $\to_{\mathcal{R}}$.

LCNC is a suitable calculus for computing the solutions of program goals. It consists of five inference rules (see Appendix A) that simulate the conditional narrowing calculus without losing completeness when the user program has some special properties, like for example being decreasing and confluent. Roughly speaking, these rules successively decompose the user goal until we reach to equalities having a variable as left or right hand side. At this point, variable binding is performed. If no contradictions occur (e.g., by performing variable occur check), we end up with a computed answer substitution.

In the sequel we will denote program goals by $\mathcal{S}, \mathcal{S}', \mathcal{S}''$, equalities by $e_1, e_2, \ldots$, and sequences of equalities by G, G', G''.

4.2 Inference Rules for Constraint Processing.

Once we consider external operators, we have equations that can not be decomposed the same way as before. These equations express constraints that may be solved by some external solvers. For handling them, two new inference rules were added to the calculus.

The inference rules act on configurations which are pairs of sets of equations: the first component represents the set of equations that remained to be processed by the HLCNC interpreter, and the second contains the constraints collected so far (the constraint pool).

[xi] *imitation rule for external functions*

$$\frac{\langle \{\phi[s_1, \ldots, s_m] \approx \psi[t_1, \ldots, t_n], G'\}, \mathcal{S}' \rangle}{\begin{array}{l}\langle \{x_1 \approx s_1, \ldots, x_m \approx s_m, y_1 \approx t_1, \ldots, y_n \approx t_n, \\ \lceil \phi[x_1, \ldots, x_m] \approx \psi[y_1, \ldots, y_n] \rceil, G'\}, \mathcal{S}' \rangle\end{array}}$$

$$\frac{\langle\{\phi[s_1,\ldots,s_m] \approx t, G'\}, \mathcal{S}'\rangle}{\langle\{x_1 \approx s_1,\ldots,x_m \approx s_n, x_{n+1} \approx t, \lceil \phi[x_1,\ldots,x_m] \approx x_{m+1}\rceil, G'\}, \mathcal{S}'\rangle}$$

where ϕ, ψ are *external terms* and $x_1,\ldots,x_m,x_{m+1},y_1,\ldots,y_n$ are fresh variables. Currently, an external term is either an external operator like '+', '∗', 'Sin', ..., or a term Derivative$[i_1,\ldots,i_n][f][x_1,\ldots,x_n]$ denoting the (partial) differential $\frac{\delta^{i_1+\cdots+i_n} f}{\delta x_1^{i_1}\ldots\delta x_n^{i_n}}$.

[cs] *constraint collection*

$$\frac{\langle\{\lceil u \approx v\rceil, G'\}, \{G''\}\rangle}{\langle\{G'\}, \{G'', u \approx v\}\rangle}$$

Constraints are collected only after they have been marked by an application of rule [xi]. All our inference rules act on the *leftmost* equality of the first pair component. This decision is justified by the strong completeness results known for this selection strategy.

A convenient notation for denoting HLCNC inference steps is $\langle\mathcal{S}_0, \mathcal{S}'_0\rangle \rightsquigarrow_\theta \langle\mathcal{S}_1, \mathcal{S}'_1\rangle$ where θ is the substitution involved in applying the respective inference rule. An HLCNC *refutation* is a sequence of inference steps:

$$\langle\mathcal{S}, \{\}\rangle \rightsquigarrow_{\theta_1} \cdots \rightsquigarrow_{\theta_n} \langle\{\}, \mathcal{S}'\rangle$$

where $\mathcal{S}$ is the program goal. The substitution $\theta_1 \cdots \theta_n$ represents a partial solution of $\mathcal{S}$. What remains to be solved is the system $\mathcal{S}'$ of constraints. For this purpose, the inference rule [xs] is provided:

[xs] *constraint solve*

$$\frac{\langle\{\}, \mathcal{S}'\rangle}{\langle\mathcal{S}'', \{\}\rangle\sigma}$$

The meaning of [xs] is dependent on the constraint solving capability of CFLP: σ represents an answer substitution computed for some of the variables of $\mathcal{S}'$, and $\mathcal{S}''$ is the system of constraints that could not be solved with the constraint solving methods of CFLP.

By composing this inference step with the HLCNC refutation we get the CFLP derivation $\langle\mathcal{S}, \{\}\rangle \rightsquigarrow^*_{\theta\sigma} \langle\mathcal{S}'', \{\}\rangle$. In this case, is $\mathcal{S}''$ is $\{\}$, the CFLP derivation is a *CFLP refutation* and we say that $\theta\sigma$ represents a *CFLP answer substitution* for $\mathcal{S}$.

4.3 Sources of Parallelism

Among the sources of parallelism of this calculus we enumerate the outermost narrowing inference rule (see Appendix A) and the syntax constructs for AND- and OR-parallelism specified in program goals. We extended the syntax of program goals with OR-equations and AND-equations.

The inference rule corresponding to an OR-equation is

[OR] *OR rule*

$$\frac{\langle\{e_1 \vee \ldots \vee e_i \vee \ldots \vee e_n, G\}, \mathcal{S}'\rangle}{\langle\{e_i, G\}, \mathcal{S}'\rangle}$$

We didn't investigate yet how to parallelize the resolution of parallel AND-equations. Currently, we sequentialize them by applying the following rule:

[AND] *AND rule*

$$\frac{\langle \{e_1 \wedge \ldots \wedge e_n, G\}, \mathcal{S}' \rangle}{\langle \{e_1, \ldots, e_n, G\}, \mathcal{S}' \rangle}$$

Obviously, if we want to preserve the completeness of the calculus, we have to consider all the alternatives of applying the OR-rule and outermost narrowing rule. As a consequence, we will have the following scenario:

1. Whenever an HLCNC derivation $\langle \mathcal{S}, \{\} \rangle \rightsquigarrow^*_{\theta_1} \langle \{\}, \mathcal{S}'_1 \rangle, \ldots, \langle \mathcal{S}, \{\} \rangle \rightsquigarrow^*_{\theta_1} \langle \{\}, \mathcal{S}'_n \rangle$ is generated, the partial solution θ_i together with the system $\mathcal{S}'_i$ of collected constraints are sent to the scheduler.
2. After all HLCNC refutations are exhausted, the interpreter asks the scheduler for the CFLP answer substitutions corresponding to the pairs $\langle \theta_i, \mathcal{S}'_i \rangle$.

5 The Scheduler

The constraint scheduler coordinates the resolution of the sets of constraints received from the interpreter. We already mentioned that the scheduler receives from the interpreter pairs of the form $\langle \theta_i, \mathcal{S}'_i \rangle$ corresponding to the HLCNC refutations $\langle \mathcal{S}, \{\} \rangle \rightsquigarrow^*_{\theta_i} \langle \{\}, \mathcal{S}' \rangle$. The scheduler builds a *solution tree* such that leaf nodes correspond either to CFLP solutions of $\mathcal{S}$ or to failure nodes. The solution tree has the structure depicted in Figure 2:

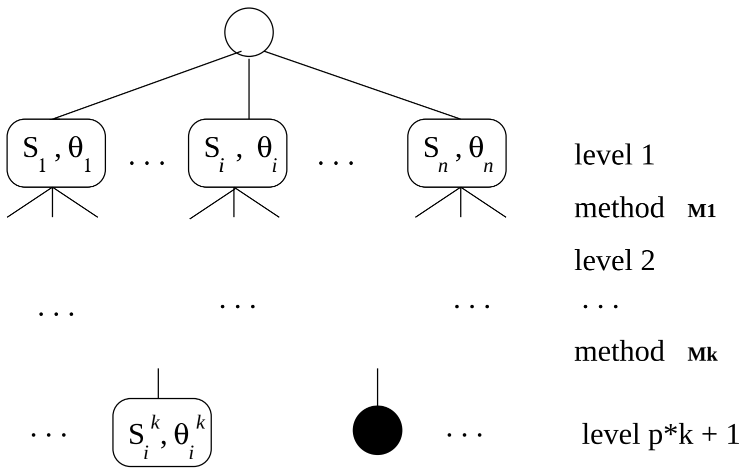

Fig. 2. Constraint solution tree

The constraint scheduler generates the solution tree with respect to a given set $\mathcal{M} = \{\mathbf{M}_1, \ldots, \mathbf{M}_k\}$ of constraint solving methods. The constraint solving capability of the CFLP system depends on the size k of $\mathcal{M}$ and on the strength of the method components to find answer substitutions.

All nodes on tree level $i+1$ are generated by the application of a constraint method $\mathbf{M}_i$ on the constraints stored in nodes at level i. A method $\mathbf{M}_i$ applied to a node with content $\langle\theta', \mathcal{S}'\rangle$ performs the following operations:

1. Select the subset $\mathcal{S}'_i \subseteq \mathcal{S}'$ of constraints on which $\mathbf{M}_i$ can act.
2. Apply $\mathbf{M}_i$ on $\mathcal{S}'_i$. We distinguish three cases.
   (a) $\mathcal{S}'_i$ is inconsistent w.r.t. $\mathbf{M}_i$; then $\mathcal{S}'$ is inconsistent and we generate a failure node as son of N,
   (b) $\mathbf{M}_i$ is unable to solve $\mathcal{S}'_i$; then we generate N as son of N at level $i+1$,
   (c) $\mathcal{S}'{}_i$ has the answer substitutions $\sigma_1, \ldots, \sigma_m$; then we generate sons $N_1, \ldots,$ N_m of N at level $i+1$ such that N_i contains $\langle\theta'\sigma_i, (\mathcal{S}' - \mathcal{S}'_i)\sigma_i\rangle$.

In Figure 2, constraint nodes are represented by black circles.

Methods are applied repeatedly in the order $\mathbf{M}_1, \ldots, \mathbf{M}_k, \ldots, \mathbf{M}_1, \ldots, \mathbf{M}_k$. A non-failure leaf node is a node at a level $p \cdot k + 1$ such that it coincides with its ancestors at levels $(p-1) \cdot k + 2, \ldots, p \cdot k$, that is, no method $\mathbf{M}_i \in \mathcal{M}$ is able to solve it.

The branches of the solution tree are generated concurrently by actions of constraint solvers on the tree nodes. Constraint solvers are instances of constraint solving methods (see Figure 1). Whenever a method is applied on a node, the scheduler looks for an idle constraint solver that implements that method. If such a solver exists, it is fired and that node and the scheduler starts to wait for the solutions to that node. Otherwise, the scheduler delays the treatment of that node until a corresponding constraint solver becomes idle. When a non-failure leaf node is found, the scheduler does one of the following:

- if the interpreter already waits for solutions, it sends the content of the leaf node to the interpreter.
- otherwise, it stores it until the interpreter is ready to receive solutions from the scheduler.

The constraint scheduler stops execution after the solution tree is completely generated and all its non-failure leaf nodes have their content sent to the interpreter.

6 The Constraint Solvers

The CFLP constraint solvers implement the methods of the list $\mathcal{M}$ of methods specified by the scheduler. They are currently implemented in Mathematica as packages that run in separate sessions and communicate with the constraint scheduler via MathLink connections.The current implementation provides four methods for solving constraints over the domain of complex numbers:

1. `Linear`, for solving linear systems of equations (the Simplex algorithm),
2. `Polynomial`, for solving polynomial systems of equations (the Gröbner basis algorithm [1]),
3. `Derivative`, for solving linear differential equations,
4. `PartialDerivative`, for solving systems of partial differential equations.

These methods are tried in the denoted order.

In order to increase the power and efficiency of the special constraint solvers, we plan to replace the current Mathematica implementations with solvers implemented in a compiled language, like for example C or C++. The communication protocol between the CFLP components will remain MathLink, since one of its uses is to allow integrating external functions written in other languages to be called from within the Mathematica environment.

7 Conclusions

We presented here a distributed constraint solving system for functional logic programming. The constraints considered by now are linear, polynomial, differential or partial differential equations over the complex numbers. We intend to enlarge the set of allowable language constraints in two ways:

- Consider also other kinds of constraints over the complex numbers,
- Consider also constraints over other domains, like reals, integers, or booleans.

Both directions require to extend the set of external solvers of CFLP. For supporting the second extension, we have already introduced a polymorphic type checker and started to experiment its use in classifying the constraints with respect to the domains on which they are defined.

References

1. Buchberger, B.: Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In N.K.Bose, eds., Multidimensional Systems Theory, Chapter 6, 184-232, Dordrecht: D.Riedel Publishing Company, 1985.
2. Colmerauer, A.: An Introduction to Prolog III. Comm.of the ACM, 33, No.7, 69-90, 1990.
3. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., Bertyer, F.: The Constraint Logic Programming Language CHIP. Proc. of the Int. Conf. on Fifth Generation Computer Systems, 1988.
4. Todd Gayley: A MathLink Tutorial
Wolfram Research, Technical Report, 1997.
5. Hamada, M., Ida, T.: Implementation of Lazy Narrowing Calculi in Mathematica. Technical Report 97-02, RISC, Johannes Kepler University, Austria, 1997.
6. Hamada, M., Middeldorp, A.: Strong Completeness of a Lazy Conditional Narrowing Calculus. Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, World Scientific, Singapore, 1997.

7. Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. J. Logic Programming 1994: 19,20:583-628
8. Hong, H.: RISC-CLP(Real): Constraint Logic Programming over Real Numbers. In F.Benhamou and A.Colmerauer, eds., Constraint Logic Programming: Selected Research, MIT Press, 1993.
9. Hullot, J.-M.: Canonical Forms and Unification. Proceedings of the 5th Conference on Automated Deduction, LNCS 87, 314-334, 1980.
10. Jaffar, J., Lassez,J.-L.: Constraint Logic Programming. Technical Report, Department of Computer Science, Monash University, Clayton, 1987.
11. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP($\mathcal{R}$) Language and System. IBM Research Division, T.J.Watson Research Centre, Yorktown Heights, 1990.
12. Marin, M., Schreiner, W.: The CFLP calculus: A Mathematica Implementation. Technical Report, RISC, Johannes Kepler University, Austria, 1998. Also available in postscript format from
http://www.risc.uni-linz.ac.at/projects/basic/distcon/deliverables/
13. Middeldorp, A., Okui, S., Ida, T.: Lazy Narrowing: Strong Completeness and Eager Variable Elimination. Theoretical Computer Science, 167(1,2):95-130, 1996.
14. Suzuki, T., Nakagawa, K., Ida, T.: Higher Order Lazy Narrowing Calculus: a Computation Model for a Higher-Order Functional Logic Language. Technical Report, SCORE Research Group, Tsukuba University, Japan, 1997.

A The Lazy Conditional Calculus: Inference Rules

[o] *outermost narrowing*

$$\frac{\{G', f[s_1, \ldots, s_n] \simeq t, G''\}}{\{G', s_1 \approx t_1, \ldots, s_n \approx t_n, r \approx t, l_1 \approx r_1, \ldots, l_n \approx r_n, G''\}}$$

if there exist a fresh variant $f[t_1, \ldots, t_n] \to r \Leftarrow (l_1 \approx r_1 \wedge \cdots \wedge l_n \approx r_n)$ of a rule in $\mathcal{R}$,

[i] *imitation*

$$\frac{\{G', f[s_1, \ldots, s_n] \simeq x, G''\}}{\{G', s_1 \approx x_1, \ldots, s_n \approx x_n, G''\}\theta}$$

if $\theta = \{x \mapsto f[x_1, \ldots, x_n]\}$ with $x_1, \ldots, x_n$ fresh variables,

[d] *decomposition*

$$\frac{\{G', f[s_1, \ldots, s_n] \approx f[t_1, \ldots, t_n], G''\}}{\{G', s_1 \approx t_1, \ldots, s_n \approx t_n, G''\}}$$

[v] *variable elimination*

$$\frac{\{G', s \simeq x, G''\}}{\{G', G''\}\theta}$$

if $x \notin \mathit{Vars}[s]$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{\{G', t \approx t, G''\}}{\{G', G''\}}$$

Notes.
G', G'' denote arbitrary sequences of goal equalities. $\simeq$ is the symmetric closure of $\approx$.