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— **CFLP is a distributed software system which aims at integrating constraint solving capabilities over various domains in a functional logic programming environment. In this paper we report on the current status of the development of the system. The general structure and the functionality of the system components are presented.**

Keywords— **Functional Logic Programming, Constraint Solving, Distributed Systems.**

I Introduction

Lazy narrowing (LN) [5] is a goal directed method in functional logic programming (FLP) for solving equational goals in a top-down or outside-in manner. Recent refinements of LN towards programming applications lead to successful implementations that can be used as starting point of developing an FLP language.

One of the recent trends in declarative programming is the possibility to extend FLP with constraint solving capabilities. The incorporation of the CLP scheme proposed by Jaffar and Lassez [3] 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. Since in our opinion the computational effort of most real, time consuming applications is in solving the constraints, we focused on developing an FLP language based on a distributed constraint solving model. Our investigation showed that it is possible to refine a lazy narrowing calculus (LNC) with inference rules for specifying constraint solving capabilities, and to develop a model for solving the constraints in parallel.

The system is called CFLP and is under development at RISC-Linz [4] in cooperation with the University of Tsukuba. It consists of an FLP interpreter based on the lazy narrowing calculi LCNC [2] and Higher-Order LNC [6], and a number of constraint solvers that may run on possibly different machines. The interpreter is implemented in the language of the computer algebra system Mathematica. The cooperation between interpreter and constraint solvers is coordinated by a component called scheduler which schedules the tasks generated by the interpreter among the solvers.

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

The paper is structured as follows: in Section II we illustrate by an example the system capabilities. Section III describes the general architecture of the CFLP system. Sections IV, V and VI describe the three main components of the system: interpreter, scheduler and constraint solvers. The last section presents conclusions and further remarks.

II Example

We illustrate by an example the possibility to solve with CFLP systems of equations that can not be solved with the FLP scheme. Consider the following system of equations:

$$\mathcal{S}:\ f[x]=g[y],\ h'[u]=y*u^y,\ h'[1]=4$$

in complex variables x, y, z, u and higher order variable h, where g is a constructor (i.e., an uninterpreted function symbol) and f is a function symbol defined by the rewrite rules $f[x]\to g[y] \Leftarrow x+y=3$, $f[x]\to g[y^2] \Leftarrow x-y=3$, $f[x+1]\to g[y(y-1)] \Leftarrow x+2y=7$. Notice that $\mathcal{S}$ contains expressions involving linear and polynomial terms over the complex numbers, derivatives, and a symbol defined by a term rewriting system. For solving these constraints, we first start the CFLP system in a Mathematica 3.0 session. This is done by loading the package `TSolve`:

```
In[1]:= <<TSolve.m
   connection to scheduler established.
   connection to constraint solvers established.
```

Some messages are displayed to inform the user if the system components were successfully started and interconnected. Also, two windows for tracing the evolution of the FLP interpreter and of the constraint scheduling component show up on the screen.

Next, we encode the problem according to our syntax. The Mathematica assignment:

`In[2]:=` $S := \{f[x]\approx g[y], h'[u]\approx y*u^y, h'[1]\approx 4\};$

defines S as the system of equations to be solved. For solving S, call:

```
In[3]:=TSolve[S, {x, y, h}, {u},
        Rule->{
          f[x] → g[y] ⇐ x + y ≈ 3,
          f[x] → g[y^2] ⇐ x^2 − y ≈ 3,
          f[x + 1] → g[y ∗ (y − 1)] ⇐ x + 2 ∗ y ≈ 7},
        DefinedSymbol->{f::Compl → Compl},
        Constructor->{g::Compl → Compl}]
```

The first argument represents the system of equations (the goal), the second is the list of system variables, and the third is the list of eliminable variables. The option `Rule` specifies the equational theory (the program), `DefinedSymbol` the type annotated list of defined function symbols, and `Constructor` the type annotated list of constructor symbols. The call returns the answer substitutions:

`Out[3]:=`

$$\begin{array}{l}\{\{x \to -1, y \to 4, h \to \lambda[\{u\}, c1 + 4\ u^5/5]\},\\ \{x \to 1, y \to 4, h \to \lambda[\{u\}, c2 + 4\ u^5/5]\},\\ \{x \to 5, y \to 4, h \to \lambda[\{u\}, c3 + 4\ u^5/5]\},\\ \{x \to 7 - \sqrt{17}, y \to 4, h \to \lambda[\{u\}, c4 + 4\ u^5/5]\},\\ \{x \to 7 + \sqrt{17}, y \to 4, h \to \lambda[\{u\}, c5 + 4\ u^5/5]\}\}\end{array}$$

The computing steps performed by the functional logic interpreter of CFLP and the constraint solvers can be viewed in the trace windows that were opened when CFLP was loaded. For this simple example, solving the system S of equations reduces to solving three disjoint sets of constraints consisting of linear, polynomial, and differential equations over the complex numbers. Solving them is done in a distributed way as explained in Section V.

III The Structure of CFLP

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. More precisely, the system is designed to solve problems of the following type:

Given An equational goal G defined by the grammar

$$G ::= t_1 \approx t_2 \mid G_1 \vee \ldots \vee G_n \mid G_1 \wedge \ldots \wedge G_n \mid \{G_1, \ldots, G_n\}$$

where t_1, t_2 are simply typed terms over a signature $\mathcal{F}$ and a set V of variables, and an equational theory R presented as a set of conditional rewrite rules $t_1 \to t_2 \Leftarrow (G_1, \ldots, G_m)$

Find the substitutions over V such that the formula $G\theta$ holds in R.

The construct $G_1 \vee \ldots \vee G_n$ represents a disjunction of formulas (an OR-formula). The last two constructs: $G_1 \wedge \ldots \wedge G_n$ (parallel AND-formula) and $\{G_1, \ldots, G_n\}$ (sequential AND-formula) denote the logical conjunction of the formulas $G_1, \ldots, G_n$. In the sequel we will identify a goal G with the sequential AND-formula $\{G\}$.

In addition to the usual constructors and defined symbols of a pure functional logic language, $\mathcal{F}$ contains also external operators. These are the operators used for expressing constraints over various domains (see next subsection). The system is composed of three main components (see Figure 1):

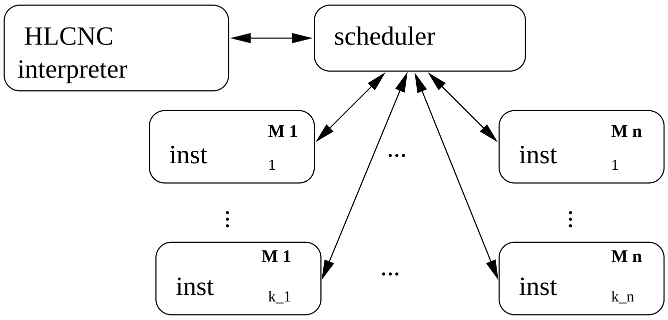

Fig. 1. The architecture of CFLP

- A functional logic *interpreter* running on one machine. It is based on the calculus HLCNC [4], an extension of the calculi LCNC and Higher-Order LNC.
- Various specialized *constraint solvers* running concurrently on different machines. Each constraint solver is an *instance* of a particular constraint solving method.
- A *scheduler* which coordinates the solving process of the constraints appearing in the equational goal.

Every constraint solving method $\mathbf{M}_i$ has associated a number of instances (denoted in Figure 1 by $\mathbf{inst}_j^{\mathrm{M}_i}$) which are processes that may run in parallel (also on different machines of a network). Because constraint solvers are an expensive resource, they can be shared: different schedulers corresponding to different CFLP sessions can access simultaneously the same constraint solvers.

IV The Interpreter

The mechanism of inference of the CFLP interpreter is based on HLCNC, a deterministic refinement of higher order lazy narrowing with conditional equations. HLCNC essentially consists of the rules for higher-order unification plus the lazy narrowing rules. We extended this calculus 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, (b) the possibility to specify explicit OR-parallelism and AND-parallelism at the level of CFLP goals.

IV.A Constraints

Constraints are equations that can not be solved in a pure functional logic framework. The interpreter recognizes them by analyzing the outermost operator of the equational terms, preprocesses them by decomposing them into equations that can be solved outside the equational theory R, and collects them into a so-called *constraint store*. The accumulated constraints are sent at certain times to the CFLP scheduler which is in charge of solving them. This behavior is captured in the HLCNC calculus by extending the configurations on which inference rules act to pairs of the form $\langle \mathcal{S}, \mathcal{C} \rangle$, where $\mathcal{S}$ represents the equations for the interpreter and $\mathcal{C}$ are the constraints collected so far. Constraint processing is captured by the following inference rules:

$$\langle \{f[t_1,\ldots,t_n] \approx t, F_2,\ldots,F_n\},\mathcal{C}\rangle \rightsquigarrow \langle\{t_1 \approx x_1,\ldots,t_n \approx x_n, y \approx t, \lceil f[x_1,\ldots,x_n] \approx y\rceil\},\mathcal{C}\rangle$$
$$\langle \{t \approx f[t_1,\ldots,t_n], F_2,\ldots,F_n\},\mathcal{C}\rangle \rightsquigarrow \langle\{t \approx y, x_1 \approx t_1,\ldots,x_n \approx t_n, \lceil f[x_1,\ldots,x_n] \approx y\rceil\},\mathcal{C}\rangle$$
$$\langle \{\lceil F_1\rceil, F_2,\ldots,F_n\},\{eq_1,\ldots,eq_m\}\rangle \rightsquigarrow \langle\{F_2,\ldots,F_n\},\{eq_1,\ldots,eq_m,F_1\}\rangle$$

where f is an external operator and $x_1,\ldots,x_n,y$ are new variables. Note that the first two rules are much in the spirit of the imitation rule of the lazy narrowing calculus: they decompose an equation in simpler ones by introducing new variables. Also, the constraint $f[x_1,\ldots,x_n] \approx y$ which does not depend on R is generated and marked for being collected later in the constraint store.

Usually, the constraints are delivered to the CFLP scheduler when no more equations are left for the interpreter. Formally, this means that $\langle\mathcal{S},\{\}\rangle \rightsquigarrow^*_\theta \langle\{\},\mathcal{C}\rangle \rightsquigarrow^*_\sigma \langle\{\},\{\}\rangle$ where the first subderivation $\langle\mathcal{S},\{\}\rangle \rightsquigarrow^*_\theta \langle\{\},\mathcal{C}\rangle$ represents the sequence of HLCNC inference steps, and the last one $\langle\{\},\mathcal{C}\rangle \rightsquigarrow^*_\sigma \langle\{\},\{\}\rangle$ specifies the scheduling process which finds the answer substitution σ to $\mathcal{C}$.

IV.B Sources of Parallelism

An OR-formula is an *explicit* source of parallelism: it generates disjoint subsets of constraints by reducing all the alternative subsystems of equations. Apart from the OR-rule, there are also *implicit* sources of parallelism. One of them is the nondeterministic lazy narrowing inference rules. In the example given in Section II, for solving the equation $f[x] \approx g[y]$ we have to consider all the three conditional rewrite rules that define f. This will give rise to three disjoint sets of equations. For example, by narrowing with respect to the last rule associated to f, we get $\{x \approx x_1{+}1, g[y_1{*}(y_1{-}1)] \approx g[y], x_1{+}2y_1 \approx 7, h'[u] \approx y{*}u^y, h'[1] \approx 4\}$, where x_1, y_1 are new variables. This system which will finally reduce to a set of constraints, and the same thing will happen when considering the other two rules.

Another source of explicit parallelism are parallel AND-formulas. For a formula $G = G_1 \wedge \ldots \wedge G_n$, a so-called *AND-context* is generated: it is the tuple $\langle\langle G_1,\{\}\rangle, \ldots, \langle G_n,\{\}\rangle\rangle_{\text{AND}}$. This construct expresses the decomposition of G into n distinct goals together with their own sets of collected constraints, which are processed in parallel. The inference rules for processing parallel AND-formulas are:

$$\langle\{\langle\langle G_1,\mathcal{C}_1\rangle,\ldots,\langle G_n,\mathcal{C}_n\rangle\rangle_{\text{AND}}, G_{n+1},\ldots,G_{n+m}\},\mathcal{C}\rangle \rightsquigarrow_\theta \langle\{\langle\langle G'_1,\mathcal{C}'_1\rangle\bar\theta_1,\ldots,\langle G'_n,\mathcal{C}'_n\rangle\bar\theta_n\rangle_{\text{AND}}, G_{n+1}\theta,\ldots,G_{n+m}\theta\},\mathcal{C}\theta\rangle$$

where $\theta = \theta_1\ldots\theta_n$ and $\bar\theta_i = \theta_1\ldots\theta_{i-1}\theta_{i+1}\ldots\theta_n$, if $\langle\{G_i,\mathcal{C}_i\rangle \rightsquigarrow_{\theta_i} \langle G'_i,\mathcal{C}'_i\rangle$ or $\theta_i = \varepsilon$ and $\langle G_i,\mathcal{C}_i\rangle = \langle G'_i,\mathcal{C}'_i\rangle$, and there are not two substitutions θ_i,θ_j with $1 \le i \ne j \le n$ for the same variable.

$$\langle\{\langle\langle\{\},\mathcal{C}_1\rangle,\ldots,\langle\{\},\mathcal{C}_n\rangle\rangle_{\text{AND}}, G_{n+1},\ldots,G_{n+m}\},\mathcal{C}\rangle \rightsquigarrow \langle\{G_{n+1},\ldots,G_{n+m}\},\mathcal{C}\cup\mathcal{C}_1\cup\ldots\cup\mathcal{C}_n\rangle$$

V The Constraint Scheduler

The constraint scheduler (see Figure 2) is in charge of solving the sets of constraints received from the interpreter.

Algorithm: Schedule
Globals **M** : the ordered set of all method identifiers
Locals

CS	a set of tuples $\langle i, Cs_i, M_i\rangle$, where Cs_i is a set of constraints and M_i is the ordered set of the identifiers of the methods which have not yet been applied on Cs_i. i is an integer used for distinguishing the tuples.
state$[i]$	the state of solving the constraint set Cs_i (*idle* if no solver acts on them, *wait* if a solver acts on them, *ready* if a solver has data available after processing Cs_i)
links$[M]$	the set of *idle* links to M-solvers.

Begin
  Listen ← `False`;
  Loop
    If *Listen* and LinkReadyQ[*Link*] then
      Q ← `LinkRead`[*Link*]
      Cases Q of
        CSolve$[C,V]$:
          IsOver ← `False`; i ← NewIndex[];
          CS ← `Append`[**CS**, $\langle i, Cs, \mathbf{M}\rangle$];
          state$[i]$ ← *idle*
        GetAnswers:
          Listen ← `False`
      End Cases
    If not(*IsOver*) then for all $\langle i, Cs_i, M_i\rangle \in$ **CS** do
      Cases **state**$[i]$ of
        idle:
          If $M_i \neq \{\}$ then
            M^i ← First$[M_i]$; $Cs_i^{M^i}$ ← `Filter`$_{M^i}[Cs_i]$;
            If $Cs_i^{M^i} = \{\}$ then $M_i \leftarrow M_i - M^i$
            else *IsOver* ← `False`;
              If **links**$[M^i] \neq \{\}$ then
                state$[i]$ ← *wait*; l ← `First`[**links**$[M^i]$]
                LinkWrite[l, **Solve**$[Cs_i]$];
        wait:
          QueryLinks[**links**$[M^i]$]; *IsOver* ← `False`;
        ready:
          If $Cs_i^{M^i}$ is inconsistent then $M_i \leftarrow \{\}$
          else if $Cs_i^{M^i}$ could not be solved then
            $M_i \leftarrow M_i - M^i$;
            If $M_i \neq \{\}$ then *IsOver* ← `False`
          else
            Let $Cs_i^1,\ldots,Cs_i^{k_i}$ be the answers;
            CS ← **CS** − Cs_i; *IsOver* ← `False`;
            For $1 \le j \le k_i$ do
              j' ← `NewIndex`[]; **state**$[j]$ ← *idle*
              CS ← `Append`[**CS**, $\{j', Cs_i^j, \mathbf{M} - M^i\}$]
        done:
          If *IsOver* then
            LinkWrite[*Link*, **ConstraintSets**[**CS**]];
            IsOver ← `False`;
            CS ← {}; *Listen* ← *True*
      End Cases
End

Fig. 2. The constraint scheduling algorithm

The communication between interpreter and scheduler is done via a MathLink connection *Link* and can be described as follows.

The sets Cs_i of constraints accumulated by the interpreter are sent to the scheduler through a message $\mathbf{CSolve}[Cs_i, V_i]$, where V_i are the constraint variables of Cs_i. After sending all the constraint sets that can be generated from the initial goal, the interpreter sends a **GetAnswers** message to inform the scheduler that the interpreter has finished generating sets of constraints and starts waiting for solutions to the sets of constraints generated so far. At the other end, the scheduler reads the messages and starts to coordinate the constraint solvers to solve the constraint sets received so far. After the answers are computed and **GetAnswers** is received, it will send back the list of answers. After all the answers have been received, the interpreter will continue its execution.

The scheduling algorithm is described in Figure 2. Constraints are solved in cooperation with a number of constraint solvers that implement a predefined set of constraint solving methods. Every solver is identified by the constraint solving method it implements and by a link object to it. An *M-solver* is a solver implementing the method M and can be regarded as an operator $\mathbf{op}^M$ that acts on the constraint sets contained in **CS** until they become invariant. In other words, for a given set Cs of constraints, the scheduler computes the fixed-point set $Cs' = \mathbf{FixPoint}_{\mathbf{op}^\mathbf{M}}(Cs)$ where $\mathbf{op}^\mathbf{M} = \mathbf{op}^{M_n} \circ \cdots \circ \mathbf{op}^{M_1}$ if $\mathbf{M} = \{M_1, \ldots, M_n\}$.

In the pseudo-code we denoted by $\texttt{Filter}_M$ a filter that selects the constraints which can be handled by an M-solver.

VI The Constraint Solvers

The constraint solvers are implementations of the constraint solving methods specified to the scheduler through the ordered set $\mathbf{M}$. The current implementation provides four methods for solving constraints over the domain of complex numbers: (1) `Linear`, for linear equations (the Simplex algorithm), (2) `Polynomial`, for polynomial equations (the Gröbner basis algorithm [1]), (3) `Derivative`, for ordinary differential equations, and (4) `PartialDerivative`, for partial differential equations. These methods are tried in the denoted order.

All solvers are currently implemented by separate Mathematica processes executing in parallel and communicating with the constraint scheduler via the MathLink interface.

VII Conclusions and Future Work

We have presented a distributed constraint solving system for functional logic programming. It is intended to be used by researchers in logic programming languages and in functional logic programming languages as well as by researchers in constraint solving who will utilize its expressive and its computational power.

The current implementation has integrated solvers for linear, polynomial, differential and partial differential equations over the domain of complex numbers, and all of them were implemented in Mathematica. We will extend the functionality of the system in the following ways:

- For handling more complex examples, we will integrate external solvers implemented in a compiled language (C/C++) over more powerful constraint domains.
- While the constraint solving engines may run on different processors or different machines, not much emphasis has been yet placed on the efficiency of the system with respect to its concurrent execution. We will investigate the dynamic behavior of the system in more detail in order to improve its runtime efficiency and yield good absolute speedup.
- Applicative packages written on top of the system will be provided for particular applicative domains. Currently a sample package modeling the behavior of electrical circuits is available.

Of particular interest is also the extension of the calculus to higher order logic. Such extensions are difficult because of the huge search space for solutions that has to be considered. We started to investigate efficient implementations of a restricted form of higher order narrowing and hope that further investigation will boost the constraint solving capability of our system.

References

[1] B. Buchberger, "*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] M. Hamada, A. Middeldorp, "*Strong Completeness of a Lazy Conditional Narrowing Calculus*", in *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, World Scientific, Singapore, 1997.

[3] J. Jaffar, J.-L. Lassez, *Constraint Logic Programming*, Technical Report, Department of Computer Science, Monash University, Clayton, 1987.

[4] M. Marin, W. Schreiner, *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/

[5] A. Middeldorp, S. Okui, T. Ida, "Lazy Narrowing: Strong Completeness and Eager Variable Elimination, Theoretical Computer Science, 167(1,2):95-130, 1996.

[6] T. Suzuki, K. Nakagawa, T. Ida, "*Higher Order Lazy Narrowing Calculus: a Computation Model for a Higher-Order Functional Logic Language*", in *Proceedings of Sixth International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science, 1997.