A Distributed System for Solving Equational Constraints Based on Lazy Narrowing Calculi

Mircea Marin[1], Tetsuo Ida[2], and Wolfgang Schreiner[1]

[1] 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
[2] Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba 305-8573, Japan
ida@score.is.tsukuba.ac.jp

Abstract. In this paper we describe the architecture and implementation of a system that aims at extending in a consistent way a functional logic programming language with solving techniques of various constraint solving systems. The system is called CFLP (Constrained Functional Logic Programming language), and consists of a lazy functional logic interpreter extended in two directions: the possibility to specify constraints, and the possibility to specify AND- and OR-parallelism. For solving the constraints, a distributed constraint solving system was implemented.

1 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 a pure functional logic language is not sufficient. These problems usually involve solving systems of constraints over various domains, like polynomial equations, differential equations, linear equations and inequations. The CLP scheme proposed by Jaffar and Lassez [4] 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 specifying constraints and to develop a model for solving the constraints in a distributed environment.

The system is called CFLP (**C**onstrained **F**unctional **L**ogic **P**rogramming) and consists of a functional logic interpreter based on the lazy narrowing calculi LCNC [2] and Higher-Order LNC [14] and a number of constraint solvers that may run on possibly different machines. The interpreter is implemented in the symbolic computation system Mathematica [15]. The cooperation between interpreter and constraint solvers is coordinated by a constraint scheduler, which schedules the tasks generated by the interpreter among the solvers.

The paper is structured as follows: in Section 2.1 we illustrate by two examples the system capabilities. Section 3 describes the system architecture. Subsections 3.1, 3.2 and 3.3 describe the three main components of the system: interpreter, scheduler and constraint solvers.

2 Examples

In this section we illustrate the functionality of CFLP with two examples.

2.1 A Polynomial Approximation Problem

Consider the problem of finding the relationship between the coefficients of a uni-variate polynomial $\mathtt{f} \in \mathbb{C}[\mathtt{x}]$ of degree 3 and a uni-variate polynomial $\mathtt{g} \in \mathbb{C}[\mathtt{x}]$ of degree 4 related by the constraint that they have the same values for $\mathtt{x} = 1, 2, 3, 4$. In CFLP, this problem can be modeled as an equality between the lists $\{\mathtt{f}[1], \mathtt{f}[2], \mathtt{f}[3], \mathtt{f}[4]\}$ and $\{\mathtt{g}[1], \mathtt{g}[2], \mathtt{g}[3], \mathtt{g}[4]\}$. In order to express lists of function evaluations, we make use of the higher-order function `map` which is defined by the conditional rewrite rule:

$$\begin{array}{l}\mathtt{map}[\mathtt{F}, \{\}] \to \{\},\\ \mathtt{map}[\mathtt{F}, [\mathtt{y} \mid \mathtt{z}]] \to [\mathtt{F}[\mathtt{y}] \mid \mathtt{t}] \Leftarrow \mathtt{t} \approx \mathtt{map}[\mathtt{F}, \mathtt{z}].\end{array}$$

in variables `F`, `y`, `z`, `t`. Here, `F` is a higher-order variable. We write $f[t_1, \ldots, t_n]$ for the term obtained by applying f to arguments $t_1, \ldots, t_n$. The construct *lhs* $\to$ *rhs* $\Leftarrow$ *cond* is the CFLP notation for a conditional rewrite rule, and *lhs* $\to$ *rhs* an unconditional rewrite rule. The condition part *cond* of a rewrite rule is in this example an equation, but in general it can be any CFLP goal (see Subsection 3.1). $[h \mid t]$ denotes a list with head h and tail t, $\{\}$ is the empty list, and an expression of the form $\{a_1, a_2, \ldots, a_n\}$ is syntactic sugar for the list $[a_1 \mid a_2 \ \ldots \ [a_n \mid \{\}] \ldots]$. Note that the second definition of `map` can be replaced with the unconditional rewrite rule

$$\mathtt{map}[\mathtt{F}, [\mathtt{y} \mid \mathtt{z}]] \to [\mathtt{F}[\mathtt{y}] \mid \mathtt{map}[\mathtt{F}, \mathtt{z}]]$$

but for explanatory purpose we use this definition.

The rewrite rules describing the polynomial `f` is

$$\mathtt{f[x]} \to \mathtt{a\ x}^3 + \mathtt{b\ x}^2 + \mathtt{c\ x} + \mathtt{d}.$$

In the equational theory defined by these conditional rewrite rules, our problem reduces to solving the equation:

$$\mathtt{map}[\lambda[\{\mathtt{z}\}, \mathtt{m\ z}^4 + \mathtt{n\ z}^3 + \mathtt{p\ z}^2 + \mathtt{q\ z} + \mathtt{r}], \{1, 2, 3, 4\}] \approx \mathtt{map}[\mathtt{f}, \{1, 2, 3, 4\}]$$

in variables m, n, p, q, r, z and constants a,b,c,d. Here, λ is the symbol for lambda-abstraction.

For solving CFLP queries, our system provides the function TSolve. The first argument of TSolve is the equational goal to be solved, the second one is the list of variables to be computed, and the third (optional) argument is the list of other variables appearing in the equational goal. Variables may be type annotated. CFLP includes a polymorphic type checker for verifying the type-correctness of the goal and conditional rules. The rest of the information necessary for solving the goal is given by passing specific options of TSolve. For our first example, the call is as follows:

```
TSolve[
  map[λ[{z},m z^4 + n z^3 + p z^2 + q z + r],{1,2,3,4}] ≈ map[f,{1,2,3,4}],
  {m,n,p,q,r},{z},
  DefinedSymbol->{
    map:Float×Float×TyList[Float]→ TyList[Float],
    f:Float → Float},
  RuleVars → {F,y,z,t},
  Rule->{
    f[x] → a x^3 + b x^2 + c x + d,
    map[F,{}] → {},
    map[F,[y | z]] → [F[y] | t] ⇐ t ≈ map[F,z]},
  Constructor->{a:Float,b:Float,c:Float,d:Float}]
```

The TSolve options used for this call are

- Rule: the set of conditional rewrite rules,
- RuleVars: the variables which are used in the set of conditional rewrite rules,
- Constructor: the constructor symbols,
- DefinedSymbol: the defined symbols.

The answer computed by TSolve is

$$\left\{\left\{\mathtt{m} \to \frac{1}{24}\,(-\mathtt{d}+\mathtt{r}), \mathtt{n} \to \mathtt{a} + \frac{5(\mathtt{d}-\mathtt{r})}{12}, \mathtt{p} \to \mathtt{b} - \frac{35(\mathtt{d}-\mathtt{r})}{24}, \mathtt{q} \to \mathtt{c} + \frac{25(\mathtt{d}-\mathtt{r})}{12}\right\}\right\}$$

Note the use of higher-order variables and lambda-abstractions in the formulation of the query and rewrite rules. The system is able to handle equations involving operators defined outside the functional logic program. Furthermore, the computed answer is a parametric solution, since r is a variable.

2.2 The "Piano Mover's" Problem

Next, we consider a restricted version of the well-known "piano mover's" problem stated in [9]. The problem is to move a rectangle of length L and width w through a right-angled corridor with horizontal width W and vertical height H. We have chosen the movement depicted in Fig. 1. Obviously, if there exists a movement of the piano through the corridor, then the depicted movement is also possible.

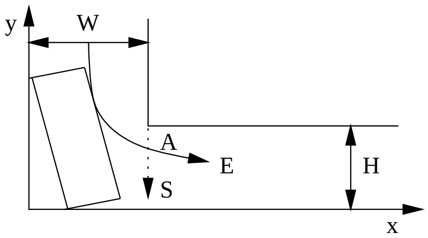
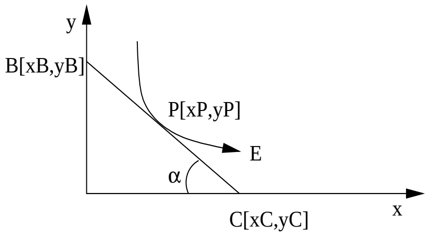


Fig. 1. "piano mover's" problem

The system is required to determine under which conditions this movement is valid.

The problem reduces to the computation of the intersection point of two curves: E, the upper envelope of the moving rectangle and S, the vertical line corresponding to the vertical wall of the right-angled corridor. If $A[\mathtt{xA}, \mathtt{yA}]$ is the intersection point of the two curves, the piano can be moved through the corridor only if $\mathtt{yA} \geq l$. To compute the coordinates of A, we will first compute a parametric representation of E. We parameterize the coordinates xP and yP of a point $P \in E$ in terms of t, the cosine of the angle α between the tangent to E in P and the x-axis. Let $B[\mathtt{xB}, \mathtt{yB}]$ and $C[\mathtt{xC}, \mathtt{yC}]$ be the intersection points of the tangent to E in $P[\mathtt{xP}, \mathtt{yP}]$ and the axes (see Figure below).



The coordinates xP and yP can now be computed in terms of t:

$$
\begin{aligned}
\mathtt{TSolve}[\ &\{\mathtt{xC}[t] \approx L\ t + \frac{w}{\sqrt{1-t^2}}, \mathtt{yB}[t] \approx L\ \sqrt{1-t^2} + \frac{w}{t},\\
&\mathtt{xP} \approx \lambda[\{t\}, \mathtt{xC}[t] - \mathtt{u}[t]\ \mathtt{xC}[t]], \mathtt{yP} \approx \lambda[\{t\}, \mathtt{u}[t]\ \mathtt{yB}[t]],\\
&\mathtt{yP}'[t]\ \mathtt{xC}[t] + \mathtt{xP}'[t]\ \mathtt{yB}[t] \approx 0\},\\
&\{\mathtt{xP} : \mathtt{Float} \to \mathtt{Float}, \mathtt{yP} : \mathtt{Float} \to \mathtt{Float}\},\\
&\{t : \mathtt{Float}, \mathtt{u} : \mathtt{Float} \to \mathtt{Float}\},\\
&\mathtt{Constructor\text{-}>}\{w : \mathtt{Float}, L : \mathtt{Float}\}]
\end{aligned}
$$

The specification of this problem involves the use of lambda-abstractions, higher-order-variables, and equational constraints like differential and polynomial equations over the domain of reals. The first two equations express the relations between the coordinates xB and yC and t. The third equation expresses the collinearity of P, B, C, and the fourth equation constrains AB to be parallel with the tangent to E in P.

Upon this query CFLP finds the variable bindings:

$$\mathtt{xP} \to \lambda\left[\{t\}, L\ t + \frac{w}{\sqrt{1-t^2}} - t\left(L\ t + \frac{w}{\sqrt{1-t^2}}\right)\frac{\left(w^2 t + w\ L\ \sqrt{1-t^2} + L^2 t(1-t^2)^2\right)}{w^2 + 2\ w\ L\ t\ \sqrt{1-t^2} - L^2 t^2(-1+t^2)}\right],$$
$$\mathtt{yP} \to \lambda\left[\{t\}, t\ \left(\frac{w}{t} + L\ \sqrt{1-t^2}\right)\frac{\left(w^2 t + l\ L\ \sqrt{1-t^2} + L^2 t(1-t^2)^2\right)}{w^2 + 2\ w\ L\ t\ \sqrt{1-t^2} - L^2 t^2(-1+t^2)}\right]$$

3 The System Structure

CFLP is a distributed software system for solving 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. External operators are used for expressing constraints over various domains.

The system consists of three components: an interpreter, a scheduler, and various specialized constraint solvers. The interpreter successively decomposes the goal towards an answer substitution by applying the inference steps of the underlying functional logic calculus. The only equations which can not be solved in this way are those which involve external operators. Such equations are factored into a sequence of simpler equations and a constraint, i.e., an equation which contains only external operators. The constraints generated upon derivations are sent to specialized constraint solvers via a component called constraint scheduler. The general architecture of the system is depicted in Fig. 2.

3.1 The Interpreter

The CFLP interpreter is based on a deterministic extension of the calculi Higher-order LNC and LCNC ([14, 2]). This calculus essentially consists of the rules for higher-order unification plus the lazy narrowing rules, and it was proven to be sound and complete for various classes of equational theories of practical interest. 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, and (b) the possibility to specify explicit OR- and AND-parallelism.

In the rest of this section we briefly describe the language and underlying calculus of the interpreter.

Goals. A CFLP *goal* is an expression of the syntax domain ($g \in$)`Goal` defined by:

$$\begin{aligned} g ::= {} & t_1 \approx t_2 \mid \{g_1, \ldots, g_1\} \mid g_1 \parallel \ldots \parallel g_n \mid g_1 \vee \ldots \vee g_n \\ & \circledast \mid \mathtt{fork} \mid \mathtt{False} \end{aligned}$$

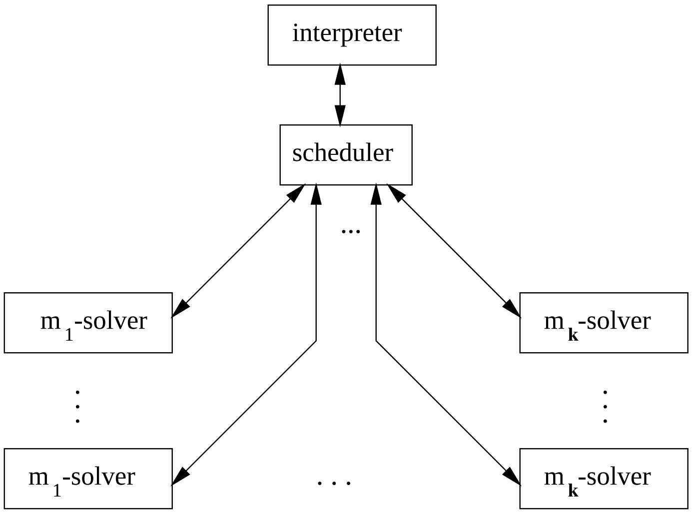


Fig. 2. The architecture of CFLP

where $t_1 \approx t_2$ is an equation. In CFLP, terms are built over a signature $\mathcal{F}$ which is the disjoint union of three sets: the set $\mathcal{F}_c$ of *constructors*, the set $\mathcal{F}_d$ of *defined symbols* and the set $\mathcal{F}_e$ of *external symbols*. The notions of constructors and defined symbols are those from functional logic programming. External symbols are the symbols which are used to write *constraints*. Formally, a constraint is an equation $t_1 \approx t_2$ where t_1, t_2 are terms built from variables and external symbols.

The interpreter is able to handle logical conjunctions ($\{g_1, \ldots, g_n\}$) and disjunctions ($g_1 \parallel \ldots \parallel g_n$, $g_1 \vee \ldots \vee g_n$) of goals. We use different notations for logical disjunction because of the different operational interpretations we are about to attribute to them. A goal of the form $g_1 \vee \ldots \vee g_n$ (sequential OR-goal) is solved sequentially, from left to right, whereas a goal of the form $g_1 \parallel \ldots \parallel g_n$ (parallel OR-goal) is solved by solving $g_1, \ldots, g_n$ concurrently. `False` denotes the logical falsity. The goals $\circledast$ and `fork` can be read as logical truth, but they are operators which can be used to control the solving process. The meaning of these operators is given in the subsection describing the transition rules of the CFLP interpreter.

In order to describe the underlying calculus of the interpreter, we introduce the notions of *states* and *transition rules.*

States. A state is an element of the syntax domain ($st \in$)`State` defined by

$$\begin{aligned} st ::= \; & \langle \theta, g, cs \rangle \mid st_1 \wedge \ldots \wedge st_n \\ & \mid st_1 \vee \ldots \vee st_n \mid st_1 \| \ldots \| st_n \\ & \mid \mathtt{wait}[cs] \mid \mathtt{fail} \end{aligned}$$

where θ is a substitution, g is a goal and cs is a set of constraints.

We interpret a state of the form $\langle\theta, g, cs\rangle$ as the space of solutions of the logical formula which is the conjunction of θ, g and cs. With this interpretation, the state $st_1 \wedge \ldots \wedge st_n$ represents the intersection of the spaces of solutions of $st_1, \ldots, st_n$. We call such a state a *sequential AND-state*. A *sequential OR-state* is a state of the form $st_1 \vee \ldots \vee st_n$. A *parallel OR-state* is a state of the form $st_1 \| \ldots \| st_n$. OR-states represent the union of the spaces of solutions of $st_1, \ldots, st_n$. A state of the form $\texttt{wait}[cs]$ is a *wait state*, and it represents the space of solutions of the system of equations contained in cs. `fail` is the failure state, i.e., the state with no solutions.

We call a state $st = \langle\theta, g, cs\rangle$ *elementary state* and say that θ is the substitution of st, g is the *goal* of st, and cs the *constraint store* of st.

A *final state* is a state of one of the following forms:

- `fail`,
- $\langle\theta, \{\}, cs\rangle$ where cs is a (possibly empty) set of constraints that can not be reduced by the constraint scheduler; we denote such a final state by adding the subscript f, i.e. we write $\langle\theta, \{\}, cs\rangle_f$,
- $\langle\theta_1, \{\}, cs_1\rangle_f \| \ldots \| \langle\theta_n, \{\}, cs_n\rangle_f$.

Transition Rules. In order to solve a goal g, the interpreter first generates the initial configuration $\langle\epsilon, g, \{\}\rangle$ and successively transforms it towards a final state. If the interpreter reaches the final state `fail`, then g is inconsistent. If the interpreter reaches a final state of the form $\langle\theta, \{\}, cs\rangle_f$, we know that the solutions of g are of the form $\sigma \circ \theta$, where σ is a solution of the system of constraints cs. If the interpreter reaches a final state of the form $\langle\theta_1, \{\}, cs_1\rangle_f \| \ldots \| \langle\theta_n, \{\}, cs_n\rangle_f$, we know that θ is a solution of g iff $\theta = \sigma_i \circ \theta_i$, for some solution σ_i of the system of constraints cs_i.

We distinguish two types of transition rules: transition rules for elementary states and transition rules for non-elementary states. A transition rule for an elementary state $st = \langle\theta, g, cs\rangle$ aims to simplify either the constraint store or the goal of st. Such a simplification results in performing at least one of the following operations:

- st is decomposed into simpler states,
- cs is found inconsistent or is decomposed into a finite disjunction $\mathcal{D}(cs) = \{\langle\sigma_i, cs_i\rangle \mid 1 \leq i \leq n\}$, where σ_i are substitutions and cs_i are sets of constraints which can not be reduced anymore. In order to get $\mathcal{D}(cs)$, the interpreter first tells the scheduler to compute it and next asks the scheduler to return it. Since the computation of $\mathcal{D}(cs)$ is time-consuming, we have designed two types of transition rules for processing constraints:
  - `tell` inference rules, which tell the scheduler to start computing $\mathcal{D}(cs)$,
  - `ask` inference rules, which ask the scheduler to retrieve the fragment of $\mathcal{D}(cs)$ computed so far.
- g is found inconsistent or is reduced to simpler goals. A particular case is when g is an equation. In this case, the reduction process is described by a relation

$$t_1 \approx t_2 \Rightarrow^{\texttt{FLP}} \langle\sigma, g, cs\rangle$$

which describes the decomposition process of an equation $t_1 \approx t_2$ into a substitution σ, goal g and set of constraints cs. The relation $\Rightarrow^{\mathtt{FLP}}$ consists of the inference rules of processing equations taken from the calculi LCNC and Higher-order LNC, plus specialized rules for handling constraints.

We describe here the transition rules for elementary states. Since the specification of the transition rules for non-elementary states is quite straightforward, we omit them here.

[$\vee$] *sequential-OR rule for goals*

$$\frac{\langle \theta, g_1 \vee \ldots \vee g_n, cs\rangle}{\langle \theta, g_1, cs\rangle \vee \ldots \vee \langle \theta, g_n, cs\rangle}$$

[$||$] *parallel-OR rule for goals*

$$\frac{\langle \theta, g_1 || \ldots || g_n, cs\rangle}{\langle \theta, g_1, cs\rangle || \ldots || \langle \theta, g_n, cs\rangle}$$

[`fork`] *parallel AND-rule for goals*

$$\frac{\langle \theta, \{\mathtt{fork}, g_1, \ldots, g_n\}, cs\rangle}{\langle \varepsilon, g'_1, \{\}\rangle \wedge \ldots \wedge \langle \varepsilon, g'_m, \{\}\rangle \wedge \langle \theta, \{\}, cs\rangle}$$

where $g'_1, \ldots, g'_m$ is a maximal partition into subsequences of the sequence $g_1, \ldots, g_n$, such that any two subsequences g'_i, g'_j with $i \neq j$ have no common variables.
This transition rule gives the operational semantics of the `fork` operator. The current state is decomposed into a suitable combination of states which can be processed concurrently and combined afterwards into a new state.

[`tell`] *tell rules*

$$\frac{\langle \theta, \{\circledast, g_1, \ldots, g_n\}, cs\rangle}{\langle \theta, \{\mathtt{wait}[cs], \{g_1, \ldots, g_n\}\}, \{\}\rangle} \qquad \frac{\langle \theta, \{\}, cs\rangle}{\langle \theta, \{\mathtt{wait}[cs], \{\}\}, \{\}\rangle}$$

These rules model the mechanism of telling the scheduler to start computing the decomposition $\mathcal{D}(cs)$. When the interpreter performs a `tell` transition step it sends the set cs of constraints to the scheduler.

[`ask`] *ask rules*

$$\frac{\langle \theta, \{\mathtt{wait}[cs], g\}, \{\}\rangle}{\langle \sigma_1 \circ \theta, \sigma_1(g), cs_1\rangle_f || \ldots || \langle \sigma_n \circ \theta, \sigma_n(g), cs_n\rangle_f || \langle \theta, \{\mathtt{wait}[cs], g\}, cs\rangle}$$

if, since the last application of an `ask` rule to $\mathtt{wait}[cs]$, the scheduler has computed the fragment $\{\langle \sigma_1, cs_1\rangle, \ldots, \langle \sigma_n, cs_n\rangle\}$ of $\mathcal{D}(cs)$ and the computation of $\mathcal{D}(cs)$ is not finished.

$$\frac{\langle \theta, \{\mathtt{wait}[cs], g\}, \{\}\rangle}{\langle \sigma_1 \circ \theta, \sigma_1(g), cs_1\rangle_f || \ldots || \langle \sigma_n \circ \theta, \sigma_n(g), cs_n\rangle_f}$$

if, since the last application of an `ask` rule to $\mathtt{wait}[cs]$, the scheduler has computed the fragment $\{\langle\sigma_1, cs_1\rangle, \ldots, \langle\sigma_n, cs_n\rangle\}$ of $\mathcal{D}(cs)$ and the computation of $\mathcal{D}(cs)$ is finished.

$$\frac{\langle\theta, \{\mathtt{wait}[cs], g\}, \{\}\rangle}{\mathtt{fail}}$$

if the scheduler has detected cs inconsistent.

[eq] *equation refinement rules*

$$\frac{\langle\theta, \{eq, g_2, \ldots, g_n\}, cs\rangle}{\langle\sigma \circ \theta, \sigma(\{g_1, g_2, \ldots, g_n\}), \sigma(cs) \cup cs'\rangle}$$

if $eq \in \mathtt{Equation}$ and $eq \Rightarrow^{\mathtt{FLP}} \langle\sigma, g_1, cs'\rangle$.

[eq] *simplification rules*

$$\frac{\langle\theta, eq, cs\rangle}{\langle\theta, \{eq\}, cs\rangle} \quad \text{if } eq \in \mathtt{Equation}$$

$$\frac{\langle\theta, \{\mathtt{False}, g_2, \ldots, g_n\}, cs\rangle}{\mathtt{fail}} \qquad \frac{\langle\theta, \{\{g_1, \ldots g_m\}, g_{m+1}, \ldots, g_{m+n}\}, cs\rangle}{\langle\theta, \{g_1, \ldots, g_{m+n}\}, cs\rangle}.$$

3.2 The Constraint Scheduler

The constraint scheduler coordinates the process of solving the systems of constraints received from the interpreter upon `tell` transition rules. In order to solve the sets of constraints received from the interpeter, the constraint scheduler maintains a dynamic data structure called *constraint tree*. The nodes of the constraint tree are tuples of the form $\langle\sigma, cs\rangle$, where σ is a substitution and cs is a system of constraints.

Whenever a set of constraints is received from the interpreter, a new son $\langle\varepsilon, cs\rangle$ of the root of the constraint tree is created. The scheduler expands this tree by applying constraint solving methods in parallel to all its leaf nodes. A leaf node $\langle\theta, cs\rangle$ is expanded w.r.t. a method $m \in \mathbf{M}$ as follows:

1. cs is decomposed into a set cs_1 of constraints to which m can be applied, and the set cs_2 of other constraints,
2. cs_1 is sent to be solved to a constraint solver which implements the 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 $\langle\varepsilon, cs_1\rangle$ if it can not reduce cs_1, or it computes a finite sequence of pairs $\langle\theta_1, cs_1'\rangle, \ldots, \langle\theta_p, cs_p'\rangle$, with the property that θ is a solution of cs iff there exists a solution σ_i of cs_i $(1 \le i \le p)$ such that $\theta = \sigma_i \circ \theta_i$.
4. If the sequence $\langle\theta_1, cs_1'\rangle, \ldots, \langle\theta_p, cs_p'\rangle$ is computed by the m-solver then the nodes $\langle\theta_i \circ \sigma, \theta_i(cs_2) \cup cs_i'\rangle$ $(1 \le i \le p)$ are added to the constraint tree as sons of $\langle\theta, cs\rangle$.

A node $\langle \sigma, cs \rangle$ is *final* if cs can not be reduced by any m-solver, where $m \in \mathbf{M}$.

The implementation of the scheduling algorithm is inspired from the work of [3]. The scheduler can be regarded as a component parameterized with respect to a list $\mathbf{M} = \{m_1, \ldots, m_k\}$ of constraint solving methods. The scheduler repeatedly applies the sequence $m_1, \ldots, m_k$ of methods to the leaves of the constraint trees until they become final nodes or become inconsistent.

The decomposition $\mathcal{D}(cs)$ is read from the final nodes of the subtree with root node $\langle \varepsilon, cs \rangle$. Whenever the interpreter applies an `ask` transition rule to a state `wait`$[cs]$, the scheduler returns the fragment of $\mathcal{D}(cs)$ computed since the last application of that transition rule.

3.3 The Constraint Solvers

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

– Local constraint solvers. These solvers run as subsidiary Mathematica kernel processes of the CFLP constraint scheduler.
– Shared constraint solvers. A shared constraint solver is started from outside a CFLP session and can be connected later to more CFLP constraint schedulers, which may run on possibly different machines. This means that we may have the situation depicted in Figure 3, where the constraint schedulers $\mathbf{scheduler}_1, \ldots, \mathbf{scheduler}_m$ may run on different machines.

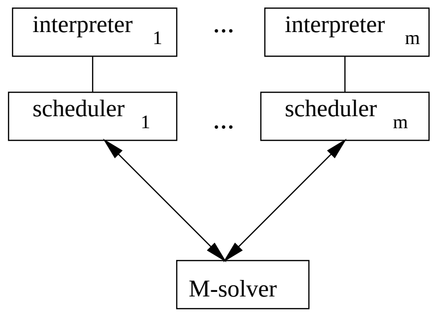


Fig. 3.

4 Conclusions

CFLP is a software system consisting of a functional logic interpreter and a distributed constraint solving system. In the current implementation we have integrated solvers for linear, polynomial, differential and partial differential equations over the domain of complex numbers. The constraint solvers are all implemented in Mathematica.

We intend to further develop the system by integrating more constraint solvers. These solvers may act on disjoint subsystems of constraints or on overlapping subsystems. Currently, only one constraint solver acts on a leaf node of the constraint tree. An optimization would be to simultaneously act with more constraint solvers on the same node in situations when the subsystems of constraints are non-overlapping.

The functional logic extension of the underlying calculus to higher order logic is still incomplete. Still, the current implementation can successfully handle restricted versions of higher order goals and conditional term rewriting systems. We will continue research in this field to identify more suitable extensions.

The system 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 are willing to use its expressive and computational power.

References

1. M. Hamada, T. Ida. *Deterministic and Non-deterministic Lazy Conditional Narrowing and their Implementations.* In Transactions of Information Processing Society of Japan, Vol. 39, No. 3, pp. 656–663, March 1998.
2. M. Hamada, A. Middeldorp: Strong Completeness of a Lazy Conditional Narrowing Calculus. *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming.* World Scientific, Singapore, 1997.
3. H. Hong. CLP(CF): Constraint Logic Programming over Complex Functions. Technical Report 94-09, RISC-Linz, Castle of Hagenberg, Austria, 1992.
4. J. Jaffar, J.-L. Lassez. *Constraint Logic Programming.* Technical Report. Department of Computer Science, Monash University, Clayton.
5. A. Middeldorp, E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing,* 5:213-253, 1994.
6. A. Middeldorp, S. Okui. A Deterministic Lazy Narrowing Calculus. In *Journal of Symbolic Computation* 25(6), pp. 733-757, 1998.
7. A. Middeldorp, S. Okui, T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science* 167(1,2), pp. 95-130, 1996.
8. E. Monfroy. *Solver Collaboration for Constraint Logic Programming.* PhD Thesis. Centre de Recherche en Informatique de Nancy INRIA-Lorraine 1996.
9. E. Monfroy, M. Rusinowitch, R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of ACM Symposium on Applied Computing (SAC'96),* Philadelphia, PA, USA, pages 63-72. ACM Press, February 1996.
10. T. Nipkow. *Functional unification of higher-order patterns.* In Proceedings of 8th IEEE Symposium on Logic in Computer Science, pp. 64-74, 1993.

11. Tobias Nipkow, Christian Prehofer. *Higher-Order Rewriting and Equational Reasoning.* In: W. Bibel, P. Schmitt (eds.). Automated Deduction - A Basis for Applications. Volume I. Kluwer, 1998, 399-430.
12. V. van Oostrom. Higher-order Families. In *International Conference on Rewriting Techniques and Applications '96, Lecture Notes in Computer Science, 1996.*
13. C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Birkhäuser Boston, 1998.
14. T. Suzuki, K. Nakagawa, and T. Ida. Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-order Functional Logic Language. In *Proceedings of Sixth International Joint Conference, ALP '97 - HOA '97*, volume 1298 of *LNCS*, pages 99–113, Southampton, 1997.
15. S. Wolfram. *The Mathematica Book.* Third Edition. Wolfram Media and Cambridge University Press, 1996.