Cooperative Constraint Functional Logic Programming

Mircea Marin, Tetsuo Ida
Institute of Information Sciences and Electronics
University of Tsukuba,
1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan
mmarin@score.is.tsukuba.ac.jp, ida@score.is.tsukuba.ac.jp

Taro Suzuki
Research Institute of Electric Communication,
Tohoku University, 980-8577 Sendai,Japan
taro@nue.riec.tohoku.ac.jp

Abstract

We describe the current status of the development of CFLP*, a system which aims at the integration of the best features of functional logic programming (FLP), cooperative constraint solving (CCS), and distributed constraint solving. FLP provides support for defining one's own abstractions (user-defined functions and predicates) over a constraint domain in an easy and comfortable way, whereas CCS is employed to solve systems of mixed constraints by iterating specialized constraint solving methods in accordance with a well defined strategy.*

CFLP *is a distributed implementation of a cooperative constraint functional logic programming scheme obtained from the integration of higher-order lazy narrowing for functional logic programming with cooperative constraint solving. The implementation takes advantage of the existence of several constraint solving resources located in a distributed environment, which communicate asynchronously via message passing.*

1 Introduction

Integration of declarative programming paradigms into a unified framework that captures the best features of each of them has attracted much interest during the last decade. Of particular interest is the design and implementation of a system based on a clean combination of functional logic programming (FLP) with cooperative constraint solving (CCS).

CFLP [4] is an experimental system which integrates higher-order functional logic programming with cooperative constraint solving. Its computational model combines higher-order lazy narrowing with the principle of solving systems of mixed constraints with a cooperation of constraint solvers described by a given cooperation strategy. The system is implemented in *Mathematica* and consists of an interpreter based on a higher-order lazy narrowing calculus, and a cooperative constraint solving system.

Our paper is structured as follows. In Section 2 we illustrate the solving capabilities of CFLP. The language of CFLP—syntax and semantics—is outlined in Section 3. Section 4 describes the general architecture of the system and of its main components. Finally, in Section 5 we draw some conclusions and directions of further research.

2 Examples

We describe with two examples the solving capabilities of CFLP.

2.1 Electric Circuit Modeling

The first example shows how electric circuit layouts can be computed with CFLP. This example illustrates the expressive power of the FLP style extended with higher-order constructs such as λ-abstractions and function variables, and constraint solving capabilities for differential equations and systems of polynomial equations.

We first define a function spec which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The CFLP rules of spec correspond to a recursive definition, where the base case describes the behavior of elementary circuits such as resistors, capacitors, and inductors, and the inductive case describes the behavior 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.

```
(* declare the CFLP type ElComp with
  associated data constructors
  res, ind, cap, serial, par *)
Constructor[ElComp = res[Float]
                   | ind[Float]
                   | cap[Float]
                   | serial[TyList[ElComp]]
                   | par[TyList[ElComp]]];
(* specify the CFLP program *)
Prog:= {
 spec[res[r],v:Float → Float,i] → True
     ⇐ λ[{t},v[t]] ≈ λ[{t},r * i[t]],
 spec[ind[l],v,i:Float → Float] → True
     ⇐ λ[{t},v[t]] ≈ λ[{t},l*i'[t]],
 spec[cap[c],v:Float → Float,i] → True
     ⇐ λ[{t},i[t]] ≈ λ[{t},c*v'[t]],
 spec[serial[{}],λ[{t},0],i] → True,
 spec[serial[[comp | comps]],v,i] → True ⇐
     {spec[comp,v1,i] ≈ True,
     spec[serial[comps],v2,i] ≈ True,
     λ[{t},v[t]] ≈ λ[{t},v1[t] + v2[t]]},
 spec[par[{}],v:Float → Float,λ[{t},0]] → True,
 spec[par[[comp | comps]],v,i] → True ⇐
     {spec[comp,v,i1] ≈ True,
     spec[par[comps],v,i2] ≈ True,
     λ[{t},i[t]] ≈ λ[{t},i1[t] + i2[t]]}}
```

The variables and operators can be type annotated, and a polymorphic type checker is integrated into the system to verify the type consistency of the CFLP goal and program. The universally quantified variables are underlined. Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The CFLP system recognizes the following list specifications:

- $\{t_1, \ldots, t_n\}$: the list consisting of components $t_1, \ldots, t_n$,
- $[\, h \mid tl \,]$: CFLP list with head h and tail tl in Prolog-style notation.

Consider the problem of finding the behavior in time of the current in a RLC circuit with R = 2, an L = 1 and $C = 1/2$ (see Fig. 1), under the restrictions 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 := {spec[serial[{res[2], ind[1], cap[1/2]}],
     λ[{t},50],i] ≈ True,
     i[0] ≈ 0}
```

The logical variable of the goal is i, and it is overlined. Note

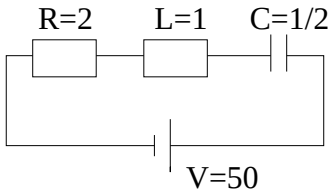


Fig. 1. RLC circuit

the usage of the expression λ[{t},50] for specifying that the voltage is constant ($50\ \Omega$) in time.

Now we can ask the CFLP system to solve the problem:

```
TSolve[G,Rules → Prog]
```

This call is similar to the Solve call of *Mathematica*, but TSolve has the specific option Rules which provides the functional logic program (i.e., conditional rewrite system) used upon solving the goal G.

The system computes the parametric solution $\{i \mapsto \lambda t.-k\, e^{-t} \sin(t)\}$ which is represented in *Mathematica* by {i → λ[{t}, − k e^−t Sin[t]}.

2.2 Program Calculation

This example illustrates the capabilities of a computing environment which can perform full higher-order pattern unification, and higher-order term rewriting.

Consider the problem of writing a program to check whether a list of numbers is *steep*, i.e, if every element of the list is greater than or equal to the average of the elements that follow it. A CFLP program that does this is:

```
Prog::={
  steep[{}] → True,
  steep[[a | x]] → And[a * len[x] ≥ sum[x], steep[x]],
  sum[{}] → 0, sum[[x | y]] → x + sum[y],
  len[{}] → 0, len[[x | y]] → 1 + len[y]}
```

Prog is modular and easy to understand, but very inefficient (quadratic complexity). It is possible—and desirable—to automatically compute an efficient (linear complexity) version steepOptimal of the function steep defined in Prog. To achieve this, we employ the fusion calculational rule

$$\frac{f[e] = e' \quad f[g[\underline{a}, \underline{ns}]] = h[\underline{a}, f[\underline{ns}]]}{f[\texttt{foldr}[g, e, \underline{ns}]] = \texttt{foldr}[h, e', \underline{ns}]} \qquad (1)$$

where foldr : $(\alpha \times \beta \to \beta) \times \beta \times \texttt{TyList}[\alpha] \to \beta$ is defined as usual:

$$\begin{aligned} \texttt{foldr}[\underline{g}, \underline{e}, \{\}] &= \underline{e}, \\ \texttt{foldr}[\underline{g}, \underline{e}, [\underline{n} \mid \underline{ns}]] &= g[\underline{n}, \texttt{foldr}[\underline{g}, \underline{e}, \underline{ns}]]. \end{aligned}$$

To see how this calculational rule can be employed, we first observe that the computation of steep[[n | ns]] with

Prog requires the computation of 3 additional quantities: steep$[ns]$, sum$[ns]$ and length$[ns]$. Thus, Prog actually specifies the computation of the function

$$\texttt{f} = \lambda[\{ns\}, \texttt{c3}[\texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]]$$

where c3 is a ternary constructor.

Note that $\texttt{f}[\underline{ns}] = \texttt{f}[\texttt{foldr}[g, e, \underline{ns}]]$, where $g = \texttt{Cons}$ and $e = \{\}$. From the fusion calculational rule (1) results that we can compute f$[ns]$ efficiently by computing foldr$[\texttt{h}, e', ns]$ instead, where h is the solution of the equation

$$\lambda[\{n, ns\}, \texttt{f}[g[n, ns]]] \approx \lambda[\{n, ns\}, \overline{\texttt{h}}[n, f[ns]]]. \quad (2)$$

and $e' = f[\{\}] = \texttt{c3}[\texttt{True}, 0, 0]$. Then:

$$\texttt{steepOpt} = \lambda[\{ns\}, \texttt{sel-c3-1}[\texttt{foldr}[\texttt{h}, e', ns]]] \quad (3)$$

where sel-c3-1 is the data selector of the first argument of a term constructed with c3 (see Subsect. 3.1 for details). To solve equation (2) with CFLP we perform the following calls:

```
(* Declare data constructor c3 with
 associated type constructor 'Triple' *)
Constructor[Triple=c3[Bool,Float,Float]];

(* add definition of f to Prog *)
AppendTo[Prog,f[ns] → c3[steep[ns], sum[ns], len[ns]]];

(* compute h; during the computation
  'Plus','Times','Power','GreaterEqual','AND'
  are regarded as mere constructors *)
TSolve[λ[{n, ns}, f[[n | ns]]] ≈
       λ[{n, ns}, h[n, c3[steep[ns], sum[ns], len[ns]]]],
  Constructor →{Plus,Power,
                Times,GreaterEqual,And},
  Rules → Prog]
```

CFLP yields the unique solution

$$\{\{h \mapsto \lambda[\{x\$13, x\$14\}, \texttt{c3}[\texttt{And}[x\$13\ \texttt{sel-c3-3}[x\$14] \geq \texttt{sel-c3-2}[x\$14]], \texttt{sel-c3-1}[x\$14], x\$13 + \texttt{sel-c3-1}[x\$14], 1 + \texttt{sel-c3-3}[x\$14]]]\}\}$$

3 The Language

CFLP is a distributed software system for solving systems of equations over various constraint domains for which specialized constraint solvers are available.

3.1 Syntax

The language of CFLP is built up from the following symbols:

- Built-in constant symbols: 2/3, -2912, Pi, etc.[1],
- External symbols: the elements of a predefined set $\mathcal{F}_e = \{\texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$,
- Built-in relational symbols:

  equational: $\approx$ (unoriented equality), $\triangleright$ (oriented equality), $\doteq$ (unoriented strict equality) and $\gg$ (oriented strict equality)

  logical: $\{\cdot, \cdots\}$ (sequential AND), $\|$ (sequential OR), and $\vee$ (parallel OR),

- Built-in type constructors:
  - base types: Int, Float, Compl, Bool,
  - TyList$[\alpha]$ (polymorphic list type) and $\{\}$ (empty list constant),
- Data constructors: the elements of a set $\mathcal{F}_c$ which contains the built-in data constructors Cons, $\{\}$ (for lists), and the data constructors defined with Constructor declarations,
- Defined function symbols: symbols of a set $\mathcal{F}_d$ which contains:
  - the data selectors introduced by the Constructor declarations,
  - the symbols defined by the rewrite rules of a given program
- Variables: symbols of a set $\mathcal{V}$ of *Mathematica* symbols with no predefined or given meaning.

The set $\mathcal{F} = \mathcal{F}_e \cup \mathcal{F}_c \cup \mathcal{F}_d$ is called the *signature* of CFLP. Next, we build the other syntactic objects of our language:

- Well-typed λ-term (or term): *Mathematica* representation of an expression built over the signature $\mathcal{F}$ and set of variables $\mathcal{V}$, but instead of Function we use λ. E.g., $\lambda[\{x, y\}, x * y]$ is the λ-term for the function which computes the product of its arguments. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$.
- Type-annotated symbol: an expression of the form $v\texttt{:}\tau$ where v is a variable or function symbol and τ is a type expression.
- Equation: an expression of the form $s \cong t$ where $\cong \in \{\approx, \triangleright, \doteq, \gg\}$ and s, t are terms of the same type.
- Goal: either an equation, or of the form $\{g_1, \ldots, g_n\}$ (sequential AND goal), $g_1 \| \ldots \| g_n$ (sequential OR goal), or $g_1 \vee \ldots \vee g_n$ (parallel OR-goal), where $g_1, \ldots, g_n$ are goals.

[1] These are the constants recognized by *Mathematica*

- Rewrite rule: expression of the form $f[l_1, \ldots, l_n] \rightarrow r \Leftarrow c$ where $f[l_1, \ldots, l_n]$, r are terms of the same base type, $f[l_1, \ldots, l_n]$ is a higher-order pattern, and c is a goal. f is the *defined* function, and c the *condition* of the rewrite rule.

- Program: a (possibly empty) set of rewrite rules.

A `Constructor` declaration has the form

`Constructor[`⟨*constr-spec*$_1$⟩, ..., ⟨*constr-spec*$_n$⟩`]`

where

⟨*constr-spec*⟩ ::= ⟨*type-spec*⟩"="⟨*data-spec*$_1$⟩"|" ... "|"⟨*data-spec*$_n$⟩

⟨*type-spec*⟩ ::= ⟨*type-name*⟩ | ⟨*type-name*⟩$[\alpha_1, \ldots, \alpha_n]$

⟨*data-spec*⟩ ::= ⟨*type-constr*⟩ | ⟨*type-constr*⟩$[\tau_1, \ldots, \tau_n]$

$\alpha_1, \ldots, \alpha_n$ distinct type variables

$\tau_1, \ldots, \tau_n$ type expressions

Example 1 We can declare binary trees with nodes of type α by:

```
Constructor[BTree[α]=
            BNil|
            T2[α,BTree[α],BTree[α]]
```

This declaration introduces the type constructor `BTree` together with its associated data constructors

`BNil:`$\forall\alpha.$`BTree[`α`]`

`T2` : $\forall\alpha.\alpha \times$ `BTree`$[\alpha] \times$ `BTree`$[\alpha] \rightarrow$ `BTree`$[\alpha]$.

In addition, the data-selectors `sel-T2-1`, `sel-T2-2` and `sel-T2-3` for the data constructor `T2` are defined implicitly. □

A `Constructor` call extends the set $\mathcal{F}_c$ of the language of `CFLP` with the newly declared data constructors, and the set $\mathcal{F}_d$ of defined symbols of the language of `CFLP` with the selectors of the non-constant constructors. For example, if c is a newly-declared n-ary data constructor, then the data selectors `sel-`c`-`$1, \ldots,$ `sel-`c`-`n are added to $\mathcal{F}_d$.

A program $\mathcal{R}$ adds the set of symbols

$$\{f \mid \exists (f[l_1, \ldots, l_n] \rightarrow r \Leftarrow c) \in \mathcal{R}\}$$

to the set $\mathcal{F}_d$ of defined symbols of the `CFLP` language. We denote by $\mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ the set of substitutions over a signature $\mathcal{F}$ and set of variables $\mathcal{V}$, by $\mathcal{D}(\theta)$ the domain of a substitution θ, and by ε the empty substitution.

3.2 Semantics

The semantics of the external symbols in $\mathcal{F}_e$ and of the data constructors in $\mathcal{F}_c$ is given by a predefined constraint domain $\mathcal{X}$ equipped with various solvers for solving systems of equational constraints.

A `CFLP` program $\mathcal{R}$ extends the language of the constraint domain $\mathcal{X}$ with symbols defined in a functional-logic programming style, and to give them a meaning. A program $\mathcal{R}$ induces a rewrite relation $\longrightarrow_{\mathcal{X},\mathcal{R}}$ on $\mathcal{X}$, where pattern matching is defined modulo the equality relation on $\mathcal{X}$. We define the semantics of the equational symbols of `CFLP` as follows:

- $\mathcal{X}, \mathcal{R} \models s \approx t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \;{}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some term u,

- $\mathcal{X}, \mathcal{R} \models s \rhd t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$,

- $\mathcal{X}, \mathcal{R} \models s \doteq t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \;{}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some constructor term u,

- $\mathcal{X}, \mathcal{R} \models s \gg t$ if t is a constructor term and $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$.

3.3 The Problem

`CFLP` is designed to solve problems of the following type:

Given a program $\mathcal{R}$ and an equation $s \cong t$ with $\cong \in \{\approx, \rhd, \doteq, \gg\}$,

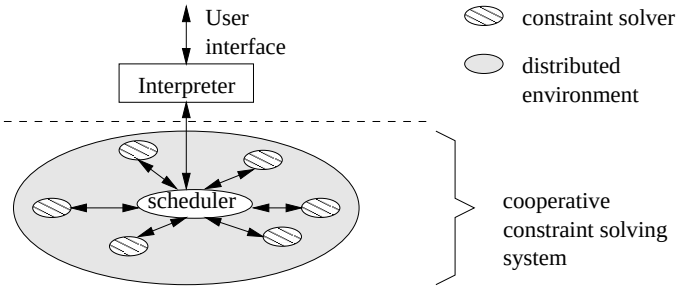
Compute $\theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ such that $\mathcal{X}, \mathcal{R} \models s\theta \cong t\theta$.

We call such a θ an $\mathcal{R}$*-solution* of $s \cong t$. This problem is extended to goals in the natural way. We denote by $\mathcal{U}_{\mathcal{R}}(G)$ the set of $\mathcal{R}$-solutions of a goal G. In general, we are interested to compute $\mathcal{R}$*-normalized* solutions of a goal G, i.e. substitutions $\theta \in \mathcal{U}_{\mathcal{R}}(G)$ such that $X\theta$ is $\mathcal{R}$-normalized for all $X \in \mathcal{D}(\theta)$. We denote the set of $\mathcal{R}$-normalized solutions of G by $\mathcal{U}^n_{\mathcal{R}}(G)$.

To solve such problems, we have designed and implemented a computation model which integrates two operational principles: lazy narrowing for conditional pattern rewrite systems, and concurrent constraint solving. Lazy narrowing solves equations containing symbols from $\mathcal{F}_c \cup \mathcal{F}_d$, whereas concurrent constraint solving is employed to solve systems of equations over $\mathcal{X}$.

4 System Structure

The combination of lazy narrowing with CCS is reflected in the structure of the system: it consists of a functional logic interpreter based on lazy narrowing, which is integrated with a distributed implementation of a cooperative constraint solving system (see below).



4.1 The Interpreter

The interpreter of CFLP is based on a calculus $\mathcal{C}$ which is an extension of a pure lazy narrowing calculus $\mathcal{K}$ for conditional pattern rewrite systems with inference rules to recognize and process equations with external symbols. $\mathcal{C}$ can be described by a set of inference rules which act on *states* of the form $\langle W \mid G \mid \mathit{Store}\rangle$ where W is a set of variables, G is a CFLP goal, and *Store* is a set of constraints $s \approx t$ collected so far. A state $\langle W \mid G \mid \mathit{Store}\rangle$ is interpreted as

$$[\![\langle W \mid G \mid \mathit{Store}\rangle]\!] = \{\gamma \mid \gamma \in \mathcal{U}_{\mathcal{R}}(G), \gamma\restriction_W \text{ is } \mathcal{R}\text{-normalized, and } \gamma \in \mathcal{U}_{\mathcal{R}}(e) \text{ for all } e \in \mathit{Store}\}.$$

The inference rules of the calculus $\mathcal{C}$ are presented as relations of the form

$$\langle W \mid G \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\theta \langle W' \mid G' \mid \mathit{Store}'\rangle$$

where θ is a substitution, called *computed* substitution. Such a relation is called *$\mathcal{C}$-step*. The interpreter computes *$\mathcal{C}$-derivations*, i.e. sequences

$$\langle W_0 \mid G_0 \mid \mathit{Store}_0\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta_1} \ldots \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta_N} \langle W_N \mid G_N \mid \mathit{Store}_N\rangle$$

of $\mathcal{C}$-steps, abbreviated

$$\langle W_0 \mid G \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}^*_{\theta_1 \ldots \theta_N} \langle W_N \mid G_N \mid \mathit{Store}_N\rangle.$$

The $\mathcal{C}$-derivations which are useful in computing a representation of $\mathcal{U}^n_{\mathcal{R}}(G)$ are the so-called $\mathcal{C}$-refutations. A *$\mathcal{C}$-refutation* is a finite $\mathcal{C}$-derivation of maximum length

$$\langle \mathcal{V}(G) \mid G \mid \{\}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}^*_\theta \langle W' \mid G' \mid \mathit{Store}\rangle \qquad (4)$$

where $\mathcal{V}(G)$ is the set of free variables in G. The set of answers computed by CFLP for a given goal G is

$$\mathit{Answ}_{\mathcal{R}}(G) = \{\ \langle \theta, G', \mathit{Store}\rangle \mid \exists\, \mathcal{C}\text{-refutation } \langle \mathcal{V}(G) \mid G \mid \{\}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}^*_\theta \langle W' \mid G' \mid \mathit{Store}\rangle\}$$

The calculus $\mathcal{C}$ is designed to satisfy the following two conditions:

soundness: For any $\langle \theta, G', \mathit{Store}\rangle \in \mathit{Answ}_{\mathcal{R}}(G)$ and any $\mathcal{R}$-solution γ of G' and *Store*, we have $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$

completeness: For any $\gamma \in \mathcal{U}^n_{\mathcal{R}}(G)$ there is a $\langle \theta, G', \mathit{Store}\rangle \in \mathit{Answ}_{\mathcal{R}}(G)$ such that $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$ for some $\mathcal{R}$-solution γ' of G' and *Store*.

A $\mathcal{C}$-refutation is constructed in two stages:

1. *Lazy Narrowing Stage:* starting with the state $\langle \mathcal{V}(G) \mid G \mid \{\}\rangle$, the goal G is narrowed until a goal made of equations which can not be narrowed anymore. The equations which can not be narrowed anymore are either:
   - *constraints*, i.e. equations over the constraint domain $\mathcal{X}$, or
   - certain *flex/flex equations*, i.e. equations between terms of the form
     $$\lambda[\{x_1, \ldots, x_p\}, X[s_1, \ldots, s_m]]$$
     with X a free variable.
     Remark: The design a calculus to solve all flex/flex equations is unrealistic, since full higher-order unification is highly intractable [1]. Our design of $\mathcal{C}$ is based on Huèt's idea of pre-unification [3].
2. *Constraint Solving Stage:* the constraints produced during the lazy narrowing stage are solved with a cooperative constraint solver.

4.1.1 The Lazy Narrowing Stage

The rules of $\mathcal{C}$ which realize this stage are:

$[\vee]$ *parallel OR*
$\langle W \mid G_1 \vee \ldots \vee G_n \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\varepsilon \langle W \mid G_k \mid \mathit{Store}\rangle$ where $k \in \{1, \ldots, n\}$

$[||]$ *sequential OR*
$\langle W \mid G_1 \| \ldots \| G_n \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\varepsilon \langle W \mid G_k \mid \mathit{Store}\rangle$ where $k \in \{1, \ldots, n\}$

The difference between these rules is that the nondeterminism due to selection of G_k is explored by breadth-first search for $[\vee]$ and by depth-first search for $[||]$.

$[\{\cdot\}]$ *sequential AND*
$\langle W | \{G_1, \ldots, G_n\} | \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\theta \langle W' | \{G'_1, \ldots, G'_n\} | \mathit{Store}'\rangle$ if $\langle W \mid G_k \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\theta \langle W' \mid G'_k \mid \mathit{Store}'\rangle$ and $G'_i = G_i\theta$ for all $i \neq k$.

[xf] *constraint accumulation*
$\langle W \mid s \cong t \mid \mathit{Store}\rangle \stackrel{\mathcal{C}}{\Longrightarrow}_\varepsilon \langle W \mid \{\} \mid \mathit{Store} \cup \{s \approx t\}\rangle$ if $s \approx t$ is a constraint

[xi] *imitation for external symbols*

$$\langle W \mid \frac{\lambda[\{\overline{x_p}\}, f[\overline{s_m}] \cong \lambda[\{\overline{x_p}\}, t] \mid \mathit{Store}\rangle}{\lambda[\{\overline{x_p}\}, s_m] \cong \lambda[\{\overline{x_p}\}, Y_m[\overline{x_p}]],} \xrightarrow{\mathcal{C}}_{\varepsilon}$$
$$\langle W \mid \lambda[\{\overline{x_p}\}, Y[\overline{x_p}]] \cong \lambda[\{\overline{x_p}\}, t] \mid \mathit{Store}\rangle$$

with $Y_1, \ldots, Y_m$ fresh variables and $\cong \in \{r, r^{-1} \mid r \in \{\approx, \doteq, \rhd, \gg\}\}$ if $f \in \mathcal{F}_e$.

[flp] *lazy narrowing step*

These steps are governed by a given lazy narrowing calculus $\mathcal{K}$

$\langle W \mid s \cong t \mid \mathit{Store}\rangle \xrightarrow{\mathcal{C}}_{\theta} \langle W \mid G \mid \mathit{Store'}\rangle$
if $s \cong t \xrightarrow{\mathcal{K}}_{\theta} G'$ is a $\mathcal{K}$-step and $\mathit{Store}' = \{s\theta \approx t\theta \mid (s \approx t) \in \mathit{Store}\}$.

The interpreter of `CFLP` can make use of different built-in lazy narrowing calculi $\mathcal{K}$, depending on the preference of the user. We have designed and implemented sound and complete lazy narrowing calculi for pattern rewrite systems (PRS for short), left-linear confluent fully-extended PRS, and left-linear constructor fully extended PRS (see [5]). Moreover, we have extended these calculi to handle strict equations and conditional PRSs. Some of these calculi are higher-order generalizations of the deterministic refinements of the lazy calculus LNC [7, 6].

4.1.2 The Constraint Solving Stage

The constraints accumulated in the constraint store during the lazy narrowing stage are submitted to be solved with a cooperative constraint solving system. Formally, this stage can be described by

[cs] *constraint solve*

$\langle W \mid G \mid \mathit{Store}\rangle \xrightarrow{\mathcal{C}}_{\theta} \langle W' \mid G\theta \mid \mathit{Store}'\rangle$
if $\langle \theta, \mathit{Store}'\rangle \in \mathcal{S}(\langle\{\}, \mathit{Store}\rangle)$ (see next subsection) and W' are the free variables in $\{X\theta \mid X \in W\}$.

4.2 The Cooperative Constraint Solving Component

We assume given a constraint domain $\mathcal{X}$ over a signature $\mathcal{F}_e$ of *external* operators, and a set of specialized constraint solvers $CS_1, \ldots, CS_n$. Each solver is a function

$$CS_k : \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \to \mathcal{P}_{\text{fin}}(\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}))$$

where $\mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \subset \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V}))$ is the set of constraints that can bw solved with CS_k, $\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V})$ is the set of idempotent substitutions over $\mathcal{F}_e$. The individual solvers are canonical simplifiers, i.e. if $CS_k(S) = \{\langle \theta_i, S_i\rangle \mid 1 \leq i \leq N\}$ then

- γ is a solution of S iff γ is of the form $\theta_p\gamma'_p$ for some $p \in \{1, \ldots, N\}$ and solution γ'_p of S_p.
- every S_i is a CS_k-canonical form, i.e.

$$CS_k(S_i) = \{\langle\{\}, S_i\rangle\}.$$

CS_k is extended to an operator CS^e_k on $\mathcal{P}_{\text{fin}}(\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V})))$ defined by

$$CS^e_k(\{\langle\gamma_p, S_p\rangle \mid 1 \leq p \leq N\} = \bigcup_{p=1}^{N}\{\langle\gamma_k\theta', S''\gamma_k \cup S'\rangle \mid S' = S \cap \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}), S'' = S - S', \langle\theta', S'\rangle \in CS_k(S_p)\}.$$

The operational principle of the cooperative constraint solving component is defined by a method $\mathcal{S}$ which describes how the computations of $CS_1, \ldots, CS_n$ are combined. We call $\mathcal{S}$ *cooperation strategy*, and define $\mathcal{S}(CS_1, \ldots, CS_n)$ as the fixed-point of $CS^e_n \circ \cdots \circ CS^e_1$. This strategy has been proposed and extensively investigated by Hong [2] in the framework of cooperative CLP. Obviously, when defined, the result of $\mathcal{S}$ is a CS_k-canonical form for all $1 \leq k \leq N$.

Example 2 Assume we want to solve

$$S = \{\lambda t.y'(t) \approx \lambda t.k^2 y(t),\ y(0) \approx 1,\ y(2) \approx 3, y(r) \approx 5\}$$

in variables y, k, r, by using a cooperation of 3 solvers: a solver CS_1 for differential equations, a solver CS_2 for systems of monomial equations, and a solver CS_3 for equations with invertible functions. Then
$CS^e_1(\langle\{\}, S\rangle) = S_1$ where $S_1 = \{\langle\{y \mapsto \lambda t.c\ e^{k^2 t}\}, \{c \approx 1,\ c\ e^{2\,k^2} \approx 3,\ c\ e^{r\,k^2} \approx 5\}\rangle\}$ with c a new variable; $CS^e_2(S_1) = S_2$ where $S_2 = \{\langle\{y \mapsto e^{k^2 t}, c \mapsto 1\}, \{e^{2\,k^2} \approx 3,\ e^{r\,k^2} \approx 5\}\rangle\}$; $CS^e_3(S_2) = S_3$ where $S_3 = \{\langle\{y \mapsto \lambda t.e^{k^2 t}, c \mapsto 1\}, \{2\,k^2 \approx \log(3), r\,k^2 \approx \log(5)\}\rangle\}$; $CS^e_1(S_3) = S_3$; $CS^e_2(S_3) = S_4$ where $S_4 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\log(3)/2}\}, \{-r\ \log(3)/2 \approx \log(5)\}\rangle, \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto \sqrt{\log(3)/2}\}, \{r\ \log(3)/2 \approx \log(5)\}\rangle\}$; $CS^e_1(S_4) = CS^e_2(S_4) = S_4$; and $CS^e_3(S_4) = S_5$ where $S_5 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\frac{\log(3)}{2}}, r \mapsto -\frac{2\log(5)}{\log(3)}\}, \{\}\rangle,\ \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2},\ c \mapsto 1,\ k \mapsto \sqrt{\frac{\log(3)}{2}}, r \mapsto \frac{2\ \log(5)}{\log(3)}\}, \{\}\rangle\}$. In this example, the solution S_5 of S is computed in 9 steps.

To compute $CS^e_k(S)$ we must call CS_k n_k times where n_k is the number of elements in S, and these calls can be realized in parallel. To take advantage of this fact, we have implemented a distributed constraint solving system consisting of

- several instances of solvers running on various machines, and
- a scheduler, which implements the strategy $\mathcal{S}$ by fairly allocating the constraint solvers available to eventually solve each element of S.

The general structure of the distributed constraint solving subsystem of CFLP is depicted in Fig. 2.

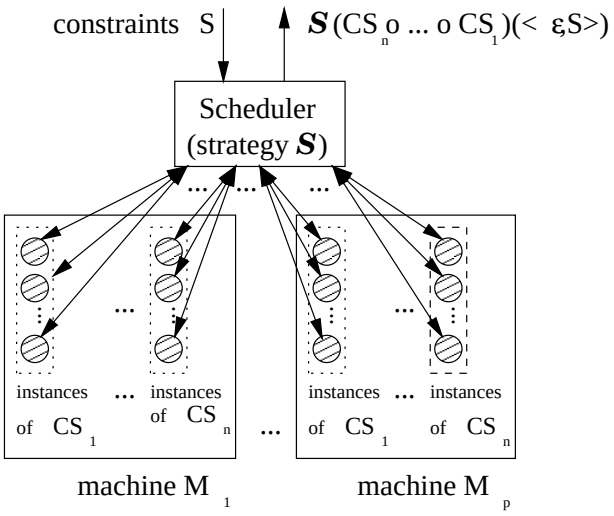


Fig. 2. CFLP**: the cooperative constraint solving component**

The current version of CFLP has integrated four types of constraint solvers:

1. CS_1 : solver which can solve algebraic equations with invertible functions, and yields solutions in terms of formal inverse functions.
2. CS_2 : solves systems of equations between multivariate polynomials over algebraic extensions of the domain of complex numbers,
3. CS_3 : solver for differential equations over algebraic extensions of $\mathbb{C}$,
4. CS_4 : solver for partial differential equations over algebraic extensions of $\mathbb{C}$.

All the components of the cooperative constraint solving system—scheduler and constraint solvers—are implemented in *Mathematica* [8] as *MathLink*-compatible processes which communicate asynchronously by message-passing over *MathLink* connections.

CFLP makes distinction between two types of solving resources:

1. local constraint solvers: these are solvers which run locally as sub-processes of a CFLP session,
2. remote constraint solvers: these solvers are started to run on various machines from outside the CFLP session, and can be shared between different CFLP sessions. The distribution of CFLP provides shell scripts to start and stop running remote constraint solvers.

The user can adjust the computation session by specifying

- the machines $M_1, \ldots, M_p$ on which to connect to the remote constraint solvers, and
- the number of local constraint solvers.

5 Conclusion

The system described in this paper is based on a computational model which integrates lazy narrowing for conditional PRS with CCS.

Currently, only a few theoretical results have been generalized to the conditional case. We will continue our research to design efficient and complete calculi for various classes of conditional PRS.

The main intention of our system CFLP was to prove the suitability of our evolvable distributed model for cooperative constraint solving. We didn't focus yet on the design of an efficient implementation.

References

[1] W. Gould. *A matching procedure for ω-order logic*. Scientific Report 4. Air Force Cambridge Research Laboratories, 1966.

[2] H. Hong. Non-linear Constraints Solving over Real Numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical Report 92-08, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[3] G. Huèt. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[4] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Schloss Hagenberg, April 2000.

[5] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[6] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

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

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