A Distributed Constraint Solving System for Functional Logic Programming

Mircea Marin[†] Tetsuo Ida[‡]

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

Abstract

The need for combining and making various constraint solvers cooperate is widely recognized. Such an integrated system would allow solving problems that can not be solved by a single solver. **CFLP** (Constrained Functional Logic Programming language) is a distributed software system consisting of a functional logic interpreter running on one machine and a number of constraint solving engines running on other machines. The interpreter is based on a deterministic version of a lazy narrowing calculus which was extended in two main directions: (a) the possibility to specify explicit OR-parallelism, and (b) the possibility to specify constraints over various domains. The OR parallel features of the interpreter allow the decomposition of the solution space into different subspaces denoted by various sets of constraints; the individual sets are solved by different constraint solving engines in parallel and joined together to form the total solution set. This allows to investigate problems with large solution spaces using the computational power available in large computer networks.

1 Introduction

The need for combining and making various constraint solvers cooperate is widely recognized. Such an integrated system would allow solving problems that can not be solved by a single solver and further open the possibility of global problem solving over the web.

We designed the lazy narrowing calculi [5] for solving equations over the domain of terms in a functional logic setting. 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. Our investigation showed that it is possible to extend the

lazy narrowing calculus with inference rules for specifying constraints and to develop a model for solving the constraints in a parallel and distributed environment.

We designed and implemented a system called CFLP [4] (**C**onstrained **F**unctional **L**ogic **P**rogramming) consisting of a functional logic interpreter and a number of constraint solvers that may be deployed over different machines. 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 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 a few examples.

2.1 Polynomial Approximation

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

$$map[F, \{\}] \to \{\},$$
$$map[F, [y \mid z]] \to [F[y] \mid t] \Leftarrow t \approx map[F, z]$$

In variables F, x, y, z. Note that in this example F is a higher-order variable. We write $f[t1, \ldots, tn]$ for the term obtained by applying f to arguments $t1, \ldots, tn$. 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. $[h \mid t]$ denotes a list with head h and tail t, $\{\}$ is the empty list, and an expression of the form $\{a1, a2, \ldots, an\}$ is syntactic sugar for the list $[a1 \mid [a2\ \ldots\ [an \mid \{\}]\ldots]]$. The rewrite rule which defines the polynomial f is

$$f[x] \to a\ x^3 + b\ x^2 + c\ x + d$$

in variable x. In the theory defined by these rules our problem reduces to solving the equation:

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

in variables m, n, p, q, r 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. A polymorphic type checker is provided 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:

`TSolve`[
$map[\lambda[\{z\}, \bar{m}\ z^4 + \bar{n}\ z^3 + \bar{p}\ z^2 + \bar{q}\ z + \bar{r}], \{1, 2, 3, 4\}]$
$\approx$
$map[f, \{1, 2, 3, 4\}]$,
`DefinedSymbol->` {
map:`Float` $\times$ `Float` $\times$ `TyList[Float]`
$\to$ `TyList[Float]`,
f : `Float` $\to$ `Float`},
`Rules->`{ $f[\underline{x}] \to a\ x^3 + b\ x^2 + c\ x + d$,
$map[\underline{F}, \{\}] \to \{\}$,
`map[`$\underline{F}$`,[`$\underline{y}$ $\mid$ $\underline{z}$`]]` $\to [F[y] \mid \underline{t}] \Leftarrow t \approx map[F, z]$},
`Constructor->`{
a : `Float`, b : `Float`,
c : `Float`, d : `Float`}]

Logical variables are declared in the goal by annotating them with an overbar, and rule variables are underlined.

The `TSolve` options used for this call are:

- `Rules`: the set of conditional rewrite rules,
- `Constructor`: the constructor symbols,
- `DefinedSymbol`: the defined symbols.

The answer computed by `TSolve` is

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

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 A Problem Involving Solver Cooperation

Consider the following program:

$$f[x] \to g[y] \Leftarrow (x + y \approx 3 \ \vee \ x^2 - y \approx 9)$$

in complex variables x, y and the goal:

$$f[x] \approx g[y], g[y^2] \approx g[z^2 - 1],\\ \lambda[\{u\}, H'[u]] \approx \lambda[\{u\}, z\ u^y], H'[1] \approx 4$$

in variables x, y, z, H. In this example the operator $\vee$ denotes logical disjunction, and it can be used in goals and conditional parts of rewrite rules to express alternative solutions.

Solving this goal requires constraint solvers for linear, polynomial and differential equations over the domain of complex numbers. Upon the query:

`TSolve`$[\{f[\bar{x}] \approx g[\bar{y}], g[y^2] \approx g[z^2-1], \lambda[\{u\},$
$H'[u]] \approx \lambda[\{u\}, \bar{z}\ u^y], H'[1] \approx 4\}, \{x, H\}$
`DefinedSymbol->` $\{f$`:Compl` $\to$ `Compl`$\}$,
`Rules->`$\{f[\underline{x}]g[\underline{y}] \Leftarrow (x + y \approx 3 \ \vee \ x^2 - y \approx 9)\}$,
`Constructor->`$\{g$`:Compl` $\to$ `Compl`$\}]$

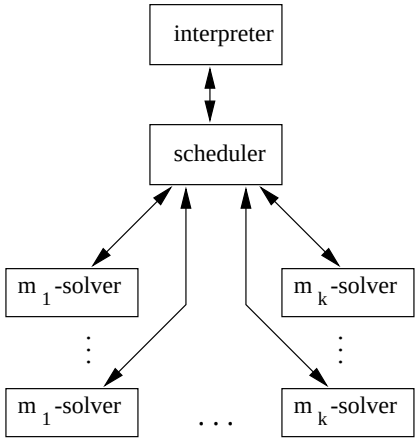

Figure 1: The architecture of CFLP

all solutions are computed:

$$\{\{x \mapsto 3 + \sqrt{15}, H \to \lambda[\{u\}, c1 + \tfrac{4\ u^{1-\sqrt{15}}}{1-\sqrt{15}}]\},\\ \{x \mapsto 3 - \sqrt{15}, H \to \lambda[\{u\}, c2 + \tfrac{4\ u^{1+\sqrt{15}}}{1+\sqrt{15}}]\},\\ \{x \mapsto -\sqrt{9-\sqrt{15}}, H \to \lambda[\{u\}, c4 + \tfrac{4\ u^{1-\sqrt{15}}}{1-\sqrt{15}}]\},\\ \{x \mapsto -\sqrt{9+\sqrt{15}}, H \to \lambda[\{u\}, c4 + \tfrac{4\ u^{1+\sqrt{15}}}{1+\sqrt{15}}]\},\\ \{x \mapsto \sqrt{9-\sqrt{15}}, H \to \lambda[\{u\}, c4 + \tfrac{4\ u^{1-\sqrt{15}}}{1-\sqrt{15}}]\},\\ \{x \mapsto \sqrt{9-\sqrt{15}}, H \to \lambda[\{u\}, c4 + \tfrac{4\ u^{1+\sqrt{15}}}{1+\sqrt{15}}]\}\}$$

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,
- various specialized constraint solvers.

The system architecture is depicted in Fig. 1.

3.1 The Interpreter

The CFLP interpreter is based on a deterministic extension of the calculi Higher-order LNC and LCNC [6, 1]. The 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 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.

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 the component called constraint scheduler.

Note that the non-deterministic selection of an inference rule for a defined symbol and explicit OR-formulas cause the initial goal to be reduced to disjoint sets of constraints that have to be solved in parallel. For example, in the second example the reduction of the initial goal involves the decomposition of the equation $f[\bar{x}] \approx g[\bar{y}]$ into simpler equations. The inference step performed by our calculus is:

$$f[x] \approx g[y] \Rightarrow$$
$$x \approx x0, (x0 + y0 \approx 3) \vee (x0^2 - y0 \approx 9), g[y0] \approx g[y]$$

where $x0, y0$ are new variables. In this step we used the fresh variant $f[x0] \rightarrow y0 \Leftarrow (x0 + y0 \approx 3 \vee x0^2 - y0 \approx 9)$ of the rewrite rule which defines f. Upon this step an OR-subgoal is introduced and as a result the goal is finally decomposed into two disjoint sets of constraints. These sets of constraints are sent to be solved to the constraint scheduler.

3.2 The Constraint Scheduler

The constraint scheduler coordinates the process of solving the systems of constraints received from the interpreter. In order to solve these sets of constraints, the constraint scheduler maintains a dynamic data structure called *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. Here ε is the empty substitution. The scheduler expands this tree by applying constraint solving methods in parallel to all its leaf nodes. The application of a constraint solving method m to a node $\langle \theta, cs \rangle$ involves the call of a constraint solver which implements the method m. Upon this call the system cs of constraints may be found inconsistent or may be decomposed into a finite sequence of pairs $\langle \theta 1, cs1 \rangle, \ldots, \langle \theta p, csp \rangle$, with the property that θ is a solution of cs iff there exists a solution σi of csi $(1 \leq i \leq p)$ such that $\theta = \sigma i \circ \theta i$. The expansion of a node $\langle \theta, cs \rangle$ stops either when cs is detected inconsistent or if it can not be reduced further by any constraint solver.

The implementation of the scheduling algorithm is inspired from the work of Hong [3]. The scheduler can be regarded as a component parameterized with respect to a list $\mathbf{M} = \{m1, \ldots, mk\}$ of constraint solving methods. Every method has associated one or more running constraint solvers.

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: (a) `Linear`, for linear equations (the Simplex algorithm), (b) `Polynomial`, for polynomial equations (the Gröbner basis algorithm), (c) `Derivative`, for ordinary differential equations, and (d) `PartialDerivative`, for partial differential equations. These methods are tried in the order presented.

All constraint solvers are implemented by separate processes executing in parallel and communicating with the interpreter via the constraint scheduler. There are two types of CFLP constraint solvers:
(A) Local solvers. These solvers run as subsidiary Mathematica kernel processes of the CFLP constraint scheduler.
(B) Shared solvers. These solvers are started from outside a CFLP session and can be shared by different CFLP constraint schedulers running on different machines. This means that we may have the situation depicted in Figure 2.

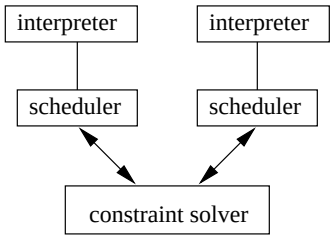

Figure 2: Shared constraint solver

The user can adjust the constraint solving component of the system by specifying the number of local constraint solvers which are started at system initialization and the remote machines on which to look up for shared constraint solvers. The communication mechanism between the scheduler and constraint solvers is implemented completely in MathLink [7]. Therefore the CFLP system is machine independent and can be used in heterogeneous networks.

4 Conclusions

CFLP is a software system consisting of a functional logic interpreter and a distributed constraint solving system. All the system components: interpreter, scheduler, constraint solvers, are implemented in Mathematica 3.0 [7] as separated processes that may run in a distributed environment and communicate via MathLink connections. In the current implementation we have integrated solvers for linear, polynomial, differential and partial differential equations over the domain of complex numbers.

We intend to further develop the system by integrating more constraint solvers and deploying them over the web, each concurrently accessing their constraints.

References

[1] M. Hamada, T. Ida: Deterministic and Nondeterministic Lazy Conditional Narrowing and their Implementations. Transactions of Information Processing Society of Japan. Vol.39, No.3, Mar.1998.

[2] M. Hamada, A. Middeldorp, T. Suzuki: Completeness Results for a Lazy Conditional Narrowing Calculus. Proceedings of DMTCS'99 and CATS'99. Auckland, New Zealand, 18-21 January 1999.

[3] H. Hong: RISC-CLP(CF): Constraint Logic Programming over Complex Functions. Technical Report. Research Institute for Symbolic Computation. Linz 1994.

[4] M. Marin, W. Schreiner: CFLP: a Distributed Constraint Solving System for Functional Logic Programming, P. Kacsuk and G. Kotsis (eds)., DAPSYS'98 Workshop on Distributed and Parallel Systems, September 28-30, 1998, Budapest, Hungary, pp.133-136. Technical Report TR-98102, Department of Applied Computer Science, University of Vienna, Austria.

[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. Proceedings of Sixth International Conference on Algebraic and Logic Programming. LNCS, 1997.

[7] S. Wolfram: The Mathematica Book. Third Edition, Mathematica Version 3, Wolfram Media, Cambridge University Press.