CFLP: a Mathematica Implementation of a Distributed Constraint Solving System

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

April 15, 1999

Abstract

The need for combining and making cooperate various constraint solvers 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.

The system is written entirely in Mathematica and uses the MathLink protocol for inter-process communication. The current implementation can solve problems expressible in functional logic and involving constraints such as linear, polynomial and differential equations.

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 [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. 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 [1] and Higher-Order LNC [7] and a number of constraint solvers that may run on possibly different machines. The interpreter is implemented in the symbolic computation system Mathematica [8]. 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 A Polynomial Approximation Problem

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 rule:

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

In variables F, x, y, z. Note that in this example F is a higher-order variable. We write $f[t_1, \ldots, t_n]$ for the term obtained by applying f to arguments $t_1, \ldots, t_n$.

The construct $lhs \rightarrow rhs \Leftarrow cond$ is the CFLP notation for a conditional rewrite rule, and $lhs \rightarrow rhs$ 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 $\{a_1, a_2, \ldots, a_n\}$ 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

$$map[F, [y \mid z]] \rightarrow [F[y] \mid map[F, z]]$$

in variables F, y, z but for explanatory purpose we consider the previous definition.

The rewrite rules describing the polynomial f is

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

in variable x. In the equational theory defined by these conditional rewrite 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. 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`
$[map[\lambda[\{z\}, \bar{m}\ z^4 + \bar{n}\ z^3 + \bar{p}\ z^2 + \bar{q}\ z + \bar{r}], \{1,2,3,4\}] \approx map[f, \{1,2,3,4\}],$
`DefinedSymbol->` {
map:`Float` $\times$ `Float` $\times$ `TyList[Float]` $\rightarrow$ `TyList[Float]`,
f : `Float` $\rightarrow$ `Float`},
`Rules->`{
$f[\underline{x}] \rightarrow a\ x^3 + b\ x^2 + c\ x + d$,
$map[\underline{F}, \{\}] \rightarrow \{\}$,
`map[`$\underline{\texttt{F}}$`, [`$\underline{\texttt{y}}$ ` | `$\underline{\texttt{z}}$`]]` $\rightarrow [F[y] | \underline{t}] \Leftarrow t \approx map[F, z]$},
`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. In order to type in expressions in the CFLP language, a suitable palette is provided for the user convenience.

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

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

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 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:

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

the following solutions are computed:

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

It is not hard to see that these are all the solutions to the query.

2.3 Electrical Circuits

This example illustrates how the behavior of electrical circuits can be expressed with our system. We consider circuits built from serial and parallel connections of elementary components such as resistors and capacitors. Electrical components act on electrical signals specified as pairs of the form $\{V, I\}$, where V is the voltage and I the intensity of the signal.

The rules given below define an underlying theory of electrical circuits consisting of serial connections of resistors/capacitors. Here, *resistor*$[R, \{V1, I1\}, \{V2, I2\}]$ defines the relation between input $\{V1, I1\}$ and output $\{V2, I2\}$ for a resistor with characteristic R. In a similar way is specified a capacitor. A serial connection is specified by a predicate *serial* $[comps, S1, S2]$ where S1 is the input signal, $S2$ the output signal, and *comps* is a list of electrical components specified in a functional way. E.g., the predicate

$$\begin{array}{ll} serial\,[& \{\lambda[\{SIn, SOut\}, resistor[R1, SIn, SOut]], \\ & \lambda[\{SIn, SOut\}, resistor[R2, SIn, SOut]]\}, S1, S2] \end{array}$$

describes a serial connection of two resistors (with characteristics) $R1$ and $R2$, input signal $S1$ and output signal $S2$.

$\texttt{ElTheory}= \{$
$\quad resistor[\underline{R}, \{\underline{V1}, \underline{I1}\}, \{\underline{V2}, \underline{I2}\}] \rightarrow \texttt{True} \Leftarrow \{V1 - V1 \approx I1\ R, I1 \approx I2\},$
$\quad capacitor[\underline{C0}, \{\underline{V1}, \underline{I1}\}, \{\underline{V2}, \underline{I2}\}] \rightarrow \texttt{True} \Leftarrow \{V1 - V2 \approx I1\ C0, I1 \approx I2\},$
$\quad serial\ [\{\}, \underline{S}, S] \rightarrow \texttt{True},$
$\quad serial\ [[\underline{Comp} \mid \underline{CompList}], \underline{SIn}, \underline{SOut}] \rightarrow \texttt{True} \Leftarrow$
$\qquad \{Comp[SIn, SBetw], serial[SBetw, SOut]\}\}$

In a similar way one can define parallel connections. When specifying electrical connections we find convenient to make use of the following abbreviations:

RComp[R]:=Function[{SIn,SOut},*resistor*[R,SIn,SOut]];
CComp[C1]:=Function[{SIn,SOut},*capacitor*[C1],SIn,SOut];

Consider now the following problem: Find the characteristic of a capacitor which can replace a serial connection of 3 capacitors. The CFLP query for solving this goal is:

$\texttt{TSolve}[$
$\quad serial\ [\{CComp[C1], CComp[C2], CComp[C3]]\}, \{V1, I1\}, \{\overline{V2}, \overline{I2}\}] \approx$
$\quad capacitor[\overline{C0}, \{V1, I1\}, \{V2, I2\}],$
$\quad \{\text{C0}\},$
$\quad \texttt{DefinedSymbol->}\{$
$\qquad capacitor\texttt{:Float} \times \texttt{TyList[Float]} \times \texttt{TyList[Float]} \rightarrow \texttt{Bool},$
$\qquad serial\ \},$
$\quad \texttt{Constructor->}\{\text{C1,C2,C3,I1,V1}\},$
$\quad \texttt{Rules->ElTheory}]$

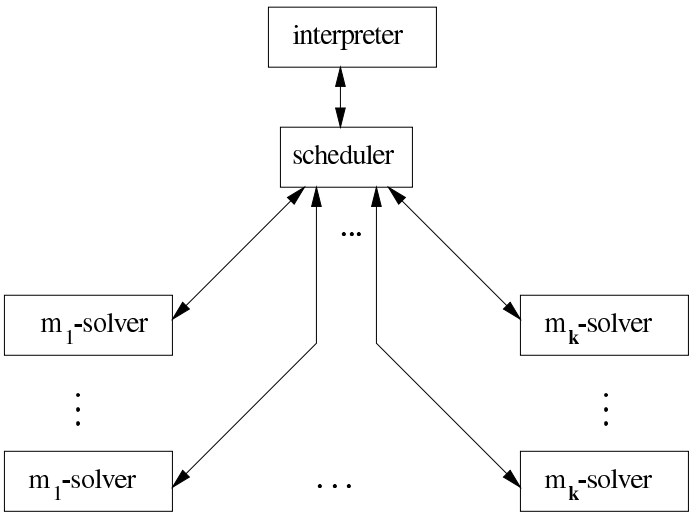

Figure 1: The architecture of CFLP

The call resumes with the computed answer:

$$\{\{C0 \rightarrow \frac{C1\ C2\ C3}{C2\ C3 + C1\ (C2 + C3)}\}\}$$

Note that this example we defined the components $I1, V1$ of the input signal as constructors, and those of the output signal as logical variables. Since we were interested only in finding the characteristic value $C0$, we provide `TSolve` with a second argument which gives the list of variables of interest. If the argument $\{C0\}$ is omitted then the computed answer will be a binding substitution for $C0, V2, I2$.

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 general architecture of the system 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 ([7, 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 ([1],[5]). 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.

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 illustrative example, the reduction of the initial goal

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

involves the decomposition of the equation $f[\bar{x}] \approx g[\bar{y}]$ into simpler equations. The transformation step performed by our lazy narrowing 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 rule of 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. 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 $(1 \leq i \leq p)$ 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 $\langle \theta_i \circ \sigma, \theta_i(cs_2) \cup cs'_i \rangle$ $(1 \leq i \leq p)$ 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 Hong [2]. 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 $\langle \varepsilon, cs \rangle$. As soon as a final node is generated, its content is made accessible to the interpreter.

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,
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 2, where the constraint schedulers $\mathbf{scheduler}_1, \ldots, \mathbf{scheduler}_m$ may run on different machines.

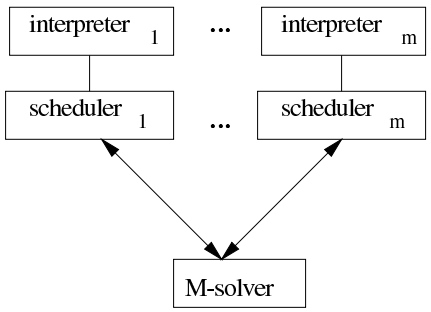

Figure 2: Shared constraint solver

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 for shared constraint solvers. The communication mechanism between the scheduler and constraint solvers is implemented completely in MathLink ([8]). 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. 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] Hamada, M., Middeldorp, A., Suzuki, T.: Completeness Results for a Lazy Conditional Narrowing Calculus. DMTCS/CATS'99. Proceedings of the 2nd Discrete Mathematics and Theoretical Computer Science Conference and the 5th Australasian Theory Symposium, Auckland, Springer-Verlag Singapore, pp. 217-231, 1999.

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

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

[4] Marin, M., Schreiner, W.: CFLP: A Distributed Constraint Solving System for Functional Logic Programming. Technical Report 98-23. Research Institute for Symbolic Computation. Linz 1998.

[5] Middeldorp, A., Okui: A Deterministic Lazy Narrowing Calculus. Journal of Symbolic Computation 25(6), pp. 733-757, 1998.

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

[7] Suzuki, T., Nakagawa, K., Ida, T.: 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.

[8] Wolfram, S.: The Mathematica Book. 3rd Edition. Wolfram Media and Cambridge University Press , 1996.