RISC-CLP(CF) Constraint Logic Programming over Complex Functions *

Hoon Hong

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria
e-mail: hhong@risc.uni-linz.ac.at

Abstract. A constraint logic programming system for the domain of complex functions is described. The intended users of the language are scientist and engineers who often reason/compute with constraints over complex functions, such as functional equalities, differential equations, etc. Constraints are solved by iterating several solving methods such as Laplace transformation, non-linear equation solving, etc. A prototype has been built and is illustrated in the paper.

1 Introduction

The constraint logic programming (CLP) framework [?, ?] provides a natural and general way to combine deduction and constraint solving, for the goal of reasoning with or solving non-trivial problems in a declarative way.

During last years, several important CLP systems have been developed such as (only to mention a few) Prolog III [?], CLP($\mathcal{R}$) [?], CHIP [?], CAL [?], CLP(BNR) [?, ?], RISC-CLP(Real) [?, ?] (See also [?] for an extension). The computation domain provided by these systems are trees, finite sets (domains), booleans, real numbers, and various combinations.

In this paper, we describe a new CLP system for the domain of complex functions. A complex function is a function of the type $\mathbf{C} \to \mathbf{C}$ where $\mathbf{C}$ is the set of all complex numbers. Thus a constant in the language denotes a complex function, and in the same way a variable ranges over the set of complex functions.

This particular domain was chosen because the system is intended to be used by scientist and engineer such as physicists, chemists, economists, electrical engineers, mechanical engineers etc. Scientist and engineers model their systems using "laws" which are very often differential equations or functional equalities, which are particular kinds of constraints over functions. As a simple example, the well known Newton's second law of the classical mechanics

$$f = m\frac{d^2x}{dt^2}$$

* This research was done in the framework of ACCLAIM, a research project of the basic research action in ESPRIT sponsored by the European Community.

is a constraint on the force f, the mass m, and the displacement x. Note that the variable x is *not* a number, but a *function* of time. If it were a number, the differentiation would be meaningless. There are numerous such examples, as can be found in any standard books on mathematical physics or advanced engineering mathematics.

Constraints are solved by the cooperation of several specialized sub-solvers such as non-linear equation solver, differential equation solver, etc. This design was crucial for the success of the RISC-CLP(CF) system because many interesting and important real life problems could not be solved by a single method.

The structure of the subsequent sections is as follows: In Section ??, we describe the syntax and the semantics of the language RISC-CLP(CF). In Section ??, we describe the inference engine which interprets the logical symbols in the language. In Section ??, we describe the constraint solver which handles the non-logical symbols in the language. In Section ??, we give several examples from scientific and engineering applications.

2 Language

In this section, we describe the language of RISC-CLP(CF), that is, the syntax and the semantics. The presentation will be intentionally informal and intuitive because we believe that it is better for human communication. Once one gets a grasp of the underlying idea, its formalization becomes easier.

Let us begin by recalling that any constraint logic programming language consists of two parts: logical part and constraint part. The logical part is already given by the general framework of CLP. Thus, a program consists of clauses, a clause consists of a head and a body, a head is an atomic formula, and a body consists of constraints and/or atomic formulas. (For a precise description, see the top part of Figure ??.)

One will notice that we introduced the construct called "guarded clause" which is indicated by the connective '`:?`'. We postpone the explanation of this until the next section since it is a construct for specifying "control", not logic.

Now we turn our attention to the constraint part of the language. For the syntax, we would like to provide sufficiently many symbols for expressing (possibly differential) constraints over complex numbers, rational functions, radicals, Dirac delta function, Heaviside step function, and all the elementary transcendental functions such as exponential, logarithm, trigonometric, hyperbolic, and their inverses. For instance, we will provide symbols such as

```
2  K  y  t  sin  exp  +  '  =
```

For the complete set of symbols and the precise syntax, see the middle part of Figure ??.

In order to discuss the semantics of these symbols, let us consider the following simple constraint:

```
y' = 2 * K * sin(t)
```

```
<program>     : <clause>...<clause>
<clause>      : <unguarded>
              | <guarded>
<unguarded>   : <atom>  :-  <literal>,...,<literal>
<guarded>     : <atom>  :?  <literal>,...,<literal>
<query>       : <literal>,...,<literal>.
<literal>     : <atom>
              | <constraint>
<atom>        : <id> [ <term>,...,<term> ]

<constraint>  : <term> = <term>
<term>        : <number>
              | <id>
              | exp | ln   | delta  | step
              | sin | sinh | arcsin | arcsinh
              | cos | cosh | arccos | arccosh
              | tan | tanh | arctan | arctanh
              | csc | csch | arccsc | arccsch
              | sec | sech | arcsec | arcsech
              | cot | coth | arccot | arccoth
              | <term>'
              | <term> + <term>
              | <term> - <term>
              | <term> * <term>
              | <term> / <term>
              | <term> ^ <term>
              | <term> ( <term> )
              | ( <term> )

<id>          : a string of letters, numbers, or '_', starting with a letter.
<number>      : an integer.
              | Pi | E | I      for π, e, i respectively.
```

Fig. 1. Syntax of RISC-CLP(CF)

where `t` is the so called "independent variable", `y` is the "dependent variable", and `K` is a "parameter" which does not depend on `t`.

In order to talk about the meaning of this expression, we first need to fix the intended domain. One might consider having two domains: the set of complex numbers (for the fixed number `2` and the variable number `K`) and the set of complex functions (for the fixed function `sin` and the variable function `y`). But this introduces the unnecessary trouble of type checking and coercion. Thus we will use only one domain: the set of all complex functions. This choice of the domain naturally leads to the following semantic decisions:

`y`	a variable symbol ranging over the set of all complex functions.
`'`	a function symbol denoting the differentiation operation on complex functions.
`=`	a predicate symbol denoting the functional equality.
`2`	a constant symbol denoting the complex function whose value is always the complex number 2.
`*`	a function symbol denoting the multiplication operation of two complex functions.
`K`	a variable symbol ranging over the set of all complex functions, with the additional property that its derivative is zero.
`sin`	a constant symbol denoting the sine function.

Next, we pay attention to the symbol: `t`. What is this? It is called an "independent variable", but this notion is very difficult to reconcile with the basic notions of the first order language. Fortunately there is a simple solution. We only need to interpret it as follows:

`t`	a constant symbol denoting the identity complex function $\lambda t.t$.

In fact, this interpretation is consistent with the following often used expression:

```
t' = 1
```

Here `t` is clearly meant to be the identity function.

Finally we are at the last remaining symbol: `()`. This symbol is usually interpreted as "evaluation" of a function on a number. But this collides with our choice of the domain. Again, there is a simple solution. We only need to relax the notion of evaluation to that of composition:

`()`	a function symbol denoting the composition operation of two complex functions.

Thus, `sin(t)` is interpreted as the composition of two complex functions sin and $\lambda t.t$. This has a side-benefit that `y` and `y(t)` denote the same complex function, justifying the usual practice of interchangeably using these two "at random".

All the remaining symbols such as "`cos tan arccos +`" can be interpreted in the exactly same way as above, and we will not describe them (tedious). On top of these, one can easily build all other semantic notions such as *interpretation of terms, solvability, model*, etc. We will not describe how they are defined since they are done in the usual standard way. See [?] for a succinct and elegant description of how this can be done.

Finally, note that the choice of the particular symbol `t` as the identity function is arbitrary. The language allows the user to choose any symbol for the identity function. In fact, the user can also specify which variables are parameters (function variables whose derivatives are zero) and which are function variables.

$t \leftarrow$ **Engine** $(\bar{C}, C, A, s)$

Inputs	$\bar{C}$	: a set of sets of constraints.		
	C	: a set of constraints.		
	A	: a set of atomic formulas.		
	s	: '`guarded`'	if the current clause is guarded.	
		'`unguarded`'	else	
Outputs	t	: '`stop_search`'	if the user asked to stop,	
		'`cont_search`'	otherwise.	
Globals	P	: a program.		

(1) [Check.]
If $A = \emptyset$ or $s =$ `guarded`,
(a) $\bar{C}^* \leftarrow$ **Solver** $(\bar{C}, C)$. *(see Figure ??.)*
(b) Set $\bar{C} \leftarrow \bar{C}^*$ and $C \leftarrow \emptyset$.
(c) If $\bar{C}$ is empty, return `cont_search`.

(2) [Answer.]
If $A = \emptyset$,
(a) Output $\bar{C}$ to the user.
(b) Ask the user whether to continue searching.
(c) If yes, return `cont_search`, else return `stop_search`.

(3) [Expand.]
Remove the first atomic formula A_1 from A.
For each matching clause R do
(a) Expand the atomic formula A_1 by using the clause R, obtaining a set C' of new constraints, and a set A' of new atomic formulas.
(b) If R is guarded, then $s' \leftarrow$ `guarded`, else $s' \leftarrow$ `unguarded`.
(c) If **Engine** $(\bar{C},\ C' \cup C,\ A' \cup A,\ s')$ is `stop_search`, then return `stop_search`.
Return `cont_search`. □

Fig. 2. Algorithm: **Engine**

For the examples shown in this paper, we used the following convention: the identity is t, the parameters are the identifiers starting with an uppercase letter, and the function variables are the ones starting with a lowercase letter.

3 Inference Engine

The inference engine interprets the logical symbols: the implication '`:-`', the conjunction '`,`', and the predicate symbols. Its operation is as usual. It maintains a list of constraints and atomic formulas and repeatedly expands atomic formulas by matching clauses in the program. When all atomic formulas disappear, the remaining constraints are reported to be an answer. It also follows the usual "depth first" strategy, namely first expanding the left-most atomic formula by the top-most matching clause.

However, there is one difference. By default, the engine does *not* call the solver at any internal node of the search tree. It calls the solver only when it is at a leaf node (that is when there is no atomic formula to expand). The decision is based on the observation that for numerous examples from science and engineering, the calls of the solver at internal nodes almost always succeed, and thus it seldom causes search space pruning. It only causes overhead.

In fact, this can be explained. When a scientist writes a program, he/she uses constrains in order to describe the relationships between physical quantities. The relationships are usually either physical laws or definitions of derived quantities. Thus they are meant to be consistent! Scientists are not interested in inconsistent systems. If inconsistency happens, it is a sign that something is wrong with their modeling process.

However, there might be times when the user does want to do some search space pruning. In such case, the user can override override this default behavior by annotating some clause to be "guarded". (For a concrete syntax of guarded clause, see the grammar in Figure ??.)

Figure ?? gives an algorithmic description of the inference engine incorporating the ideas explained above. The input $\bar{C}$ (a set of sets of constraints) represents a disjunction of conjunctions of constraints which have been computed by the previous call of the constraint solver. The reason for having *disjunction* of conjunctions is that sometimes constraints have "multiple" solutions. The input C (a set of constraints) represents a conjunction of constraints that have not yet been sent to the constraint solver. The flag s tells the type (guarded/unguarded) of the clause that has been just used to expand an atomic formula in the goal.

It is a recursive algorithm, and should be initially called with $\bar{C}, C, A, s$ where $\bar{C} = \{\{\}\}$, C is the set of constraints in a query, A is the set of atomic formulas in the query, and s is `unguarded`.

4 Constraint Solver

To motivate the discussion of the solver used in the RISC-CLP(CF) system, let us consider the following simple constraint solving problem:

```
y' = K * y,  y(0) = 1, y(2) = 3, y(T) = 5.
```

Let us observe how a human solver would tackle this problem.

1. Solve '`y' = K * y`' obtaining '`y = C * exp(K * t)`'.
2. Substitute this into '`y(0) = 1`', obtaining '`C * exp (K * 0) = 1`'.
3. Solve this, obtaining '`C  = 1`'.

$$\bar{C}^* \leftarrow \textbf{Solver}\,(\bar{C}, C)$$

Inputs $\bar{C}$: a set of sets of constraints,
(representing a disjunction of conjunctions of constraints).
C : a set of constraints,
(representing a conjunction of constrains).

Outputs $\bar{C}^*$: a set of sets of constraints,
(representing a disjunction of conjunctions of constraints),
such that $\bar{C}^*$ is equivalent to but "simpler" than $\bar{C} \wedge C$.

Globals M^*: the ordered set of all method identifiers.

Locals W : a set $\{\ldots, < C_i', M_i >, \ldots\}$ where
C_i' is a set of constraints and
M_i is the ordered set of the identifiers of the methods
which have *not* been applied on C_i'.

(1) [Normalize.]
Carry out basic normalization on C, such as cancellation of equal terms, symbolic differentiation, etc.

(2) [Setup.]
Let $\bar{C} = \{\bar{C}_1, \ldots, \bar{C}_n\}$.
Set $W \leftarrow \{W_1, \ldots, W_n\}$ where $W_i =< \bar{C}_i \cup C, M^* >$.
Set $\bar{C}^* \leftarrow \emptyset$.

(3) [Simplify.]
While W is not empty do
    Remove the first $< C', M >$ from W.
    If M is empty,
        Insert C' to $\bar{C}^*$.
    Else
        Let **method** be the first identifier in M.
        $\bar{C}' \leftarrow \textbf{method}\,(C')$.   *(see Figure ??.)*
        If $\bar{C}' = \{C'\}$,
            Insert $< C', M - \{\textbf{method}\} >$ to W,
        Else,
            For each set $\bar{C}_i'$ in $\bar{C}'$,
                Insert $< \bar{C}_i',\ M^* - \{\textbf{method}\} >$ to W. □

Fig. 3. Algorithm: **Solver**

Various Constraint Solving Methods

Inputs C : a set of constraints,
Outputs $\bar{C}$: a set of sets of constraints equivalent to C.

`Numerical_Substitution`

(a) While there is a constraint such as "`T = term`" where `T` is a parameter which does not appear in `term`, substitute `term` into everywhere `T` appears. Do the same for `y(0)`, `y(K)`, etc.

`Functional_Substitution`

(a) While there is a constraint such as "`y = term`" where `y` is a functional variable which does not appear in `term`, substitute `term` into everywhere `y` appears.

`Numerical_Equations`

(a) Let $C' \subseteq C$ be the set of the constraints involving only numeric unknowns such as parameters or $y(0)$.
(b) Solve C' numerically for the numerical unknowns.

`Functional_Equations`

(a) Let $C' \subseteq C$ be the set of the constraints involving function variables but not involving differentiation operator.
(b) Solve C' symbolically for the function variables.

`Numerical_Equalities`

(a) Let $C' \subseteq C$ be the set of the constraints involving no variables.
(b) Evaluate both sides of each constraints and compare for equality.
(d) If both sides of each constraint are the same, then return $\{\{\}\}$, else return $\{\}$.

`Functional_Equalities`

(a) Let $C' \subseteq C$ be the set of the constraints involving the identity function symbol but not any function variables.
(b) Carry out the Laplace transform on C'.
(c) Infer a system of equations on parameters from the Laplace transformed constraints.
(d) Solve the system of equations for the parameters.

`Differential_Equations`

(a) Let $C' \subseteq C$ be the set of constraints involving differentiation operator.
(b) Solve C' by using various techniques such as Laplace transformation, integrating factors, etc.

Fig. 4. Algorithms: **Methods**

4. Substitute this into '`y = C * exp(K* t)`' obtaining '`y = exp(K * t)`'.
5. Substitute this into '`y(2) = 3`', obtaining '`exp(K * 2) = 3`'.
6. Solve this, obtaining '`K = 1/2 * ln(3)`'.
7. Substitute this into '` y = exp(K*t)`', obtaining '`y = exp(1/2*ln(3)*t)`'.
8. Substitute this into '`y(T) = 5`', obtaining '`5 = exp(1/2 * ln(3) * T)`'.
9. Solve this, obtaining '`T = 2 * ln(5) / ln(3)`'.
10. We are done. The solution is:

```
T = 2 * ln(5)/ln(3)         = 2.929947040
K = 1/2 * ln(3)             = 0.549306145
y = exp(1/2 * ln(3) * t)    = exp(0.5493061445 * t)
```

We notice that the human solver applies several different solving methods repeatedly until the constraints cannot be changed anymore. In fact, it is the only way to solve deeply nested constrains such as `y(y(T)) = y(T)`.

Thus, we will use the same scheme in designing our solver. However, here one should be careful to ensure that

- the iteration eventually terminates,
- all methods are applied in a fair frequency, and
- no method is applied more than once consecutively.

It is a scheduling problem. This can be done by maintaining a list of the methods that still need to be applied. Initially this list contains all the methods. When a method is applied, it is removed from the list. If the resulting constraints are different from the input constraints, then we include all other methods into the list. We repeat this process until the list becomes empty.

Sometimes, applying a method on a set of constraints, we get multiple solutions, that is, a set of sets of constraints, which represents a disjunction of conjunctions of constraints. However, it does not cause much trouble; we only need to keep a stack of sets of constraints, and deal with one set of constraints at a time.

The algorithm **Solver** in Figure **??** systematically incorporates these ideas. One important remark. The list M^* is an ordered set of all methods (identifiers). In case there are two methods that still need to applied, then the one appearing first in M^* will be chosen. The correctness of the algorithm does not depend on the order of the methods in M^*. But the efficiency of the algorithm heavily depends on the order. Thus, one needs to be careful in ordering the methods in M^*.

Finally we turn our attention to individual constraint solving methods. See Figure **??** where we give a quick summary of the methods that are currently used by the solver. Recall that the solver is designed in such a way that we can always plug in a new method without changing any other parts.

Below we give simple (trivial) examples in order to illustrate the functionalities of the methods.

Numerical Substitution:

In: `{ T = 3,  y = T * t^2  }`

```
Out:   {{ T = 3,   y = 3 * t^2  }}
```

Functional Substitution:

```
In:     { y = K * t^2,  y(1) = 2 }
Out:   {{ y = K * t^2,  K * 1^2 = 2 }}
```

Numerical Equations:

```
In:     { X^2 + Y^2  = 2,  X = Y}
Out:   {{ X = 1, Y = 1},  { X = -1, Y = -1 }}
```

Functional Equations:

```
In:     { t^2 * sin(y) = sin(t) }
Out:   {{ y = arcsin(sin(t)/t^2) }}
```

Numerical Equalities:

```
In:     { exp(Pi * I) = -1 }
Out:   {{ }}
```

Functional Equalities:

```
In:     { K * t = 0,  sin(L + t) = K + cos(t) }
Out:   {{ K = 0,  L = 1/2 * Pi }}
```

Differential Equations:

```
In:     { y' = 2 * y }
Out:   {{ y = C * exp(2 * t) }}
```

The substitution can be handled easily by careful syntactic operations. The other problems look also easy for the simple examples above. But they are actually quite non-trivial for real life examples which usually involve systems of several equations in many variables. These problems require sophisticated algorithms from computer algebra and numerical analysis. In fact, all these problems are currently hotly pursued research topics in those communities. We will not attempt to give any descriptions of these here since it is clearly beyond the scope of this paper and the conference, and also it will require too much space.

We only mention that the current implementation solves numerical equations by a combination of Gröbner basis method [?] and an interval method [?, ?], functional equations and numerical equalities by a certain set of rewrite rules [?], functional equalities by carrying out Laplace transform and applying quantifier elimation [?, ?], differential equations by Laplace transform and several other techniques such as integral factors [?].

5 Examples

A prototype system has been implemented in the C language using two computer algebra libraries SACLIB [?] and MAPLE [?]. In this section, we illustrate the system on a few examples taken from several application areas [?]. Each of the examples took less than 1 second on a SGI workstation.

5.1 Newton's Law of Cooling

Put a hot object (such as heated copper ball) into a cool medium (such as water). The hot object slowly cools down. Newton's law of cooling states that the time rate of change of the temperature T of the object is proportional to the difference between T and the temperature of the surrounding medium. The following clause captures this fact.

```
NewtonCoolingLaw [temp_obj, temp_med] :-
        temp_obj' = - K * (temp_obj - temp_med).
```

where

`temp_obj`	is the temperature of the object as a function of time.
`temp_med`	is the temperature of the medium as a function of time.
`K`	is an unknown physical constant.

Suppose that a copper ball is heated to a temperature of 100 degrees (C). Then at time $t = 0$, it is placed in water that is maintained at a temperature of 30 degrees. At the end of 3 minutes the temperature of the ball is reduced to 70 degrees. Find the time at which the temperature of the ball is reduced to 31 degrees.

One could directly compose a query for this problem. But assuming that similar questions will be asked often, it will be worthwhile to write an "access" clause that simplifies such kind of querying.

```
Cooling [Temp_med,Temp_obj, Time1,Temp1_obj, Time2,Temp2_obj] :-
        temp_med(0)     = Temp_med,
        temp_obj(0)     = Temp_obj,
        temp_obj(Time1) = Temp1_obj,
        temp_obj(Time2) = Temp2_obj,
        temp_med' = 0,
        NewtonCoolingLaw [temp_obj, temp_med].
```

where

`Temp_med`	is the initial temperature of the medium.
`Temp_obj`	is the initial temperature of the object.
`Time1`	is a time.
`Temp1_obj`	is the temperature of the object at time `Time1`.
`Time2`	is a time.
`Temp2_obj`	is the temperature of the object at time `Time2`.

Using this program, now we ask the question posed above to the RISC-CLP(CF) system.

```
?- Cooling [30, 100, 3, 70, Time, 31].
```

Then the system responds:

```
        Time = -3*ln(70)/ln(4/7)
             = 22.77542199
```

Now one might be interested in finding the relationship between the temperature of the copper ball and the time. Here is a query for this.

```
?- Cooling [30, 100, 3, 70, Time, Temp].
```

Then the system responds:

```
        Temp = 30+70*exp(1/3*Time*ln(4/7))
             = 30.+70.*exp(-.1865385960*Time)
```

Now we ask what should be the initial temperature of the copper ball so that its temperature drops by 40 degrees at the first minute and 10 more degrees at the second minute.

```
?- Cooling [30, Temp, 1, Temp - 40, 2, Temp - 50].
```

Then the system responds:

```
        Temp = 250/3
               83.33333333
```

One could go on asking various different questions such as these.

5.2 Radioactive Carbon Dating

In the atmosphere, the ratio of radioactive carbon ${}_6C^{14}$ and ordinary carbon ${}_6C^{12}$ does not change, and it is the same as the ratio of the two in living organisms. But when an organism dies, the absorption of ${}_6C^{14}$ by breathing and eating terminates. Then the ${}_6C^{14}$ in the dead organism decays at the rate proportional to its amount. The half life of ${}_6C^{14}$ is 5730 years. But the ordinary carbon does not decay.

Utilizing these facts, one can estimate the age of a fossil. This is W. Libby's idea of radiocarbon dating (Nobel Prize for chemistry 1960). The following program translates the facts described above in almost one-to-one manner.

```
CarbonDating [ Ratio_Air, Ratio_Fossil, Age_Fossil ] :-
   ratio_air'    = 0,
   ratio_air(0)  = c14_fossil(0) / c12_fossil(0),
   c14_fossil'   = K * c14_fossil,
   c14_fossil(5730) / c14_fossil(0) = 1/2,
   c12_fossil'  = 0,
   ratio_air(Age_Fossil) = Ratio_Air,
   c14_fossil(Age_Fossil) / c12_fossil(Age_Fossil) = Ratio_Fossil.
```

where

`Ratio_Air`	is the present ratio of ${}_6C^{14}$ and ${}_6C^{12}$ in the air.
`Ratio_Fossil`	is the present ratio of ${}_6C^{14}$ and ${}_6C^{12}$ in the fossil..
`Age_Fossil`	is the age of the fossil.
`ratio_air`	is the ratio of ${}_6C^{14}$ and ${}_6C^{12}$ in the air as a function of time.
`c14_fossil`	is the amount of ${}_6C^{14}$ in the fossil as a function of time.
`c12_fossil`	is the amount of ${}_6C^{12}$ in the fossil as a function of time.
`K`	is a physical constant of ${}_6C^{14}$.

Suppose that an experimentalist tells me the ratio of the two carbons in a fossil is 1/4 of that in the air, but without telling me the values of the two ratios. Now I ask: What is the age of the fossil?

```
?- CarbonDating[ Ratio_Air, 1/4 * Ratio_Air, Age_Fossil ].
```

Then the RISC-CLP(CF) system responds as follows:

```
        Age_Fossil = 5730*ln(4)/ln(2),
                   = 11460.00000,
        Ratio_Air = _C2
OR
        Ratio_Air = 0
```

Note that it found two mathematically correct solutions, though the second one is a degenerate case which is meaningless from the physical point of view. Note also that in the first solution, the value of `Ratio_Air` is set to some new variable called `_C2` generated by the system. This basically says that it can take any value. In this particular case, it would have been better not to produce this line at all. But currently we do not know how to do this in general.

Next, suppose that we want to obtain a general formula which relates the age of the fossil with the ratios.

```
?- CarbonDating [ Ratio_Air, Ratio * Ratio_Air, Age_Fossil].
```

The systems gives you:

```
        Ratio = exp(-1/5730*ln(2)*Age_Fossil),
              = exp(-1209680943.0E-13*Age_Fossil),
        Ratio_Air = _C2
OR
        Ratio_Air = 0
```

So it tells me, if I ignore the degenerate case, that the `Ratio` decreases exponentially in `Age_Fossil` with the decay constant $1/5730 \ln(2)$.

5.3 RLC circuit

Finally, we consider an example from electrical circuits involving resistors, inductors, and capacitors. We will just give a program without explaining the underlying electronic laws since it should be easy to read them off from the program directly.

```
Resistor [ R, v, i ] :-
        v = R * i.
Inductor [ L, v, i ] :-
        v = L * i'.
Capacitor [ C, v, i ] :-
        i = C * v'.
```

```
Serial_RLC_circuit [R, L, C, v, i] :-
        Resistor  [ R, v1, i ],
        Inductor  [ L, v2, i ],
        Capacitor [ C, v3, i ],
        v = v1 + v2 + v3.
```

Consider a serial RLC circuit with $R = 2$, $L = 1$, $C = 1/2$ with a 50 volt battery and a switch. Suppose the current was 0 at time 0. We turn on the switch at time 0. What will happen to the current?

```
?- Serial_RLC_circuit [2, 1, 1/2, 50, i ], i(0) = 0.
```

The system answers:

```
        i(t) = -_C2*exp(-t)*sin(t)
```

Note that a new constant `_C2` has been introduced, which indicates that we did not supply enough information pato uniquely determine the current as a function of time. So we supply one more condition.

```
?- Serial_RLC_circuit [2, 1, 1/2, 50, i ], i(0) = 0, i(1) = 1/2.
```

Now the system responds:

```
        i(t) = 1/2/exp(-1)/sin(1)*exp(-t)*sin(t)
             = 1.615196411*exp(-1.*t)*sin(t)
```

We see that the current is a damped oscillation.

6 Conclusion

We have described the constraint logic programming system RISC-CLP(CF) where constraints over functions are solved by iterating several specialized solving methods. Various experiments indicate that this approach is quite powerful and useful, in particular when the constraints are intricately related with each other. But it also raises new questions such as: How do we ensure termination/confluency of such a solver in general? Some work has been done in this direction [?], but it is still a wide open area of research.

I would like to thank Andreas Neubacher and Kurt Siegl for implementing some parts of the constraint solver and the communication routines between Maple and the RISC-CLP(CF) system.

References

1. F. Behhamou and W. Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 1993. Submitted.

2. B. Buchberger. Applications of Gröbner bases in non-linear computational geometry. In *Proc. Workshop on Scientific Software (invited lecture)*, pages 59–88. Springer Verlag, 1987.
3. B. Buchberger, G. Collins, M. Encarnación, H. Hong, J. Johnson, W. Krandick, R. Loos, and A. Neubacher. *A SACLIB Primer*. Tech. Rep. 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria.
4. Olga Caprotti. Extending risc-clp(real) to handle symbolic functions. In A. Miola, editor, *DISCO '93: International Symposium on Design and Implementation of Symbolic Computation Systems*. Springer Verlag, September 1993.
5. B. W. Char, K. O. Geddes, G. H. Gonnet, and S. M. Watt. *Maple User's Guide*. WATCOM Publications Limited, 4th edition, 1985.
6. G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, September 1991.
7. A. Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.
8. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.
9. H. Hong. Non-linear real constraints in constraint logic programming. In *International Conference on Algebraic and Logic Programming*, pages 201–212, 1992.
10. H. Hong, editor. *Computational Quantifier Elimination*. Oxford University press, 1993. Special issue of the Computer Journal: Volume 36, number 5.
11. H. Hong. RISC-CLP(Real): Constraint logic programming over real numbers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.
12. H. Hong. Confluency of Cooperative Constraint Solvers. Technical Report 94-08, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1994.
13. H. Hong and V. Stahl. Safe start region by fixed points and tightening. *Journal of Computing, (Archives for Informatics and Numerical Computation)*. Accepted, to appear in 1994.
14. H. Hong and V. Stahl. Safe start region by fixed points and tightening. In *The proceedings of Scientific Computing, Computer Arithmetic and Validated Numerics*, September 1993.
15. J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, Proceedings 4^{th} ICLP, pages 196–218, Cambridge, MA, May 1987. The MIT Press.
16. Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.
17. E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 1993.
18. W. Older and A. Vellino. Constraint arithmetic on real intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.
19. K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and its Applications. *Journal of Symbolic Computation*, 8:589–603, 1989.