Helios: A Mathematical Modeling Language for Newton

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-32
October 1995

`Helios`: A Mathematical Modeling Language for `Newton`

Laurent Michel and Pascal Van Hentenryck
Brown University, Box 1910
Providence, RI 02912
{ldm,pvh}@cs.brown.edu

Abstract

Numerous applications in science and engineering requires nonlinear constraint solving and optimization over real numbers. `Helios` is a mathematical modeling language designed to express these applications in a form close to traditional statements displayed in scientific papers and textbooks. `Helios` is compiled into `Newton`, a constraint logic programming language over nonlinear constraints which is one of the most efficient tools for the global solution of these problems. The paper illustrates the main functionalities of `Helios`, describes the compilation process, and reports experimental results on traditional benchmarks in this area. The results indicate that `Helios` should be a valuable addition to the set of tools available in this area.

1 Introduction

Many applications in science and engineering (e.g., chemistry, robotics, economics, mechanics) require finding all isolated solutions to a system of nonlinear real constraints or finding the minimum value of a nonlinear function subject to nonlinear constraints. These problems are difficult due to their inherent computational complexity (i.e., they are NP-hard) and due to the numerical issues involved to guarantee correctness (i.e., finding all solutions or the global optimum) and to ensure termination.

`Newton` [27] is a new constraint logic programming language designed to support this class of applications. It combines interval analysis from numerical analysis [2, 3, 4, 5, 7, 9, 10, 11, 12, 17, 22, 24]) with consistency techniques from artificial intelligence [14, 16] to produce one of most efficient constraint solvers in this area [26]. As is traditional with constraint programming languages, `Newton` allows a short development time of its applications. However, `Newton` programs, although short and generally easy to understand, are still far from the statements traditionally used by scientists and engineers to describe these applications. This restricts its accessibility to a limited class of users who are familiar with constraint programming technology. This limitation motivated the research described in this article and led to the development of the language `Helios`.

`Helios` is a mathematical modeling language for describing nonlinear continuous problems. `Helios` statements are almost identical to mathematical descriptions of nonlinear applications, making `Helios` an attractive tool for scientists and engineers. In particular, `Helios` offers a rich set of constructs, including arrays, constants, user-defined functions, aggregate operators, and generic constraints. `Helios` statements are compiled into `Newton` programs, which use advanced logic programming features such as difference-lists. The generated `Newton` programs are then executed to solve the original applications.

It is worth mentioning that `Helios` shares the same motivation and philosophy as `AMPL` [1]. In fact, `Helios` intends to be for `Newton`, and more generally for global optimization, what `AMPL` is to linear programming and its extensions. The two languages and their implementations are of course fundamentally different because of the nature of their driving application areas.

The goal of this article is to illustrate the main functionalities of Helios, to describe its implementation, and to report its experimental results. Section 2 presents a tour of Helios which should give readers an understanding of the expressiveness and performance of Helios. Section 3 describes the implementation of Helios, focusing on code generation, since it is unconventional to generate code in a very high-level language such as Newton. Section 4 reports some experimental results of Helios, while Section 5 concludes the paper.

2 A Tour of Helios

This section gives a gentle introduction to Helios through a number of examples. Its purpose is to describe informally the syntax of Helios, as well as its functionality and performance on a number of well-known problems. We start with some constraint-solving examples, continue with unconstrained and constrained optimization problems and conclude with a discussion of some additional facilities.

2.1 Constraint Solving in Helios

We start with a simple univariate problem which consists of finding the roots of the function

$$x^4 - 12x^3 + 47x^2 - 60x$$

in the interval $[-10^8, 10^8]$. The Helios statement to solve this problem is as follows:

```
Variable:
   x in [-10^8..10^8];
Body:
   solve system
      EQ: x^4 - 12 * x^3 + 47 * x^2 - 60 * x = 0;
```

The variable section declares a single variable which ranges over $[-10^8, 10^8]$. The body section specifies the system of constraints to be solved by Helios, i.e., the single constraint EQ. On this problem, Helios returns the four intervals

```
x in [0.0000000,0.00000000]
x in [2.9999999,3.00000001]
x in [4.0000000,4.00000000]
x in [4.9999999,5.00000001]
```

with the default precision of the system. Helios guarantees that all solutions to the equation are inside these intervals, i.e., a solution cannot be located outside these intervals. The compilation time for this problem is about 0.2 second (on a SUN SPARC 10), while the execution is less than 0.1 second. Note also that for the function

$$x^4 - 12x^3 + 47x^2 - 60x + 24$$

`Helios` returns the intervals

```
x in [0.88830577,0.88830578]
x in [0.99999999,1.00000000]
```

while `Helios` does not return any solution for the function

$$x^4 - 12x^3 + 47x^2 - 60x + 24.1$$

indicating the absence of solutions. Note that it is also possible to ask `Helios` to prove the existence and unicity of solutions. The statement simply becomes

```
Variable:
   x in [-10^8..10^8];
Body:
   safe solve system
      EQ: x^4 - 12 * x^3 + 47 * x^2 - 60 * x = 0;
```

where the keyword `safe` has been added. With this new statement, `Helios` guarantees that any interval returned as a solution contains a unique solution. Note that the proof of existence and unicity is obtained numerically and that `Helios` produces the same results for the above root-finding problems. For all the examples in this section, `Helios` is able to prove the existence and unicity of the solutions. However, in general, `Helios` may find some intervals that are safe, i.e., intervals which contain a unique solution, and some other intervals for which no conclusion can be drawn, i.e., intervals of small size which are not pruned away by the constraints but for which `Helios` cannot prove the existence or absence of a solution because of the precision of the floating-point systems.

We now consider a simple multivariate problem: the intersection of a circle and a parabola. The problem can be stated as follows:

```
Variable:
   x :  array[1..2];
Body:
   safe solve system
      Circle:    x[1]^2 + x[2]^2 = 1;
      Parabola:  x[1]^2 - x[2] = 0;
```

The variable section declares an array of two variables and the body section specifies the two constraints. `Helios` returns the two boxes (i.e., two tuples of intervals)

```
x in [-0.78615138,-0.78615137]
y in [+0.61803398,+0.61803399]
```

```
Set:
   idx = [1..6];

Variable:
   s :  array[idx] in [-10^8..10^8];
   c :  array[idx] in [-10^8..10^8];

Body:  safe solve system
   trigo(i in idx) :  s[i]^2 + c[i]^2 = 1;
   C1 :  s[2]*c[5]*s[6] - s[3]*c[5]*s[6] - s[4]*c[5]*s[6] +
         c[2]*c[6] + c[3]*c[6] + c[4]*c[6] = 0.4077;
   C2 :  c[1]*c[2]*s[5] + c[1]*c[3]*s[5] + c[1]*c[4]*s[5] + s[1]*c[5] = 1.9115;
   C3 :  s[2]*s[5] + s[3]*s[5] + s[4]*s[5] = 1.9791;
   C4 :  c[1]*c[2] + c[1]*c[3] + c[1]*c[4] + c[1]*c[2] + c[1]*c[3] + c[1]*c[2] = 4.0616;
   C5 :  s[1]*c[2] + s[1]*c[3] + s[1]*c[4] + s[1]*c[2] + s[1]*c[3] + s[1]*c[2] = 1.7172;
   C6 :  s[2] + s[3] + s[4] + s[2] + s[3] + s[2] = 3.9701;
```

Figure 1: A Robot Kinematics Application in `Helios`.

```
x in [+0.78615138,+0.78615137]
y in [+0.61803398,+0.61803399]
```

in about 0.1 second.

Some more interesting multivariate problems come from the area of robot kinematics, where the goal is to find the angles for the joints of a robot arm so that the robot hand ends up in a specified position. The `Helios` statement to solve a problem given in [7] is depicted in Figure 1. There are several novel features in the statement. First, the set section declares a set `idx` that is used subsequently to define arrays and to specify constraints. Second, the example illustrates how constraints can be stated generically. The `Helios` statement generates trigonometric constraints of the form

$$\mathtt{s[i]}^2 + \mathtt{c[i]}^2 = 1$$

for $1 \leq \mathtt{i} \leq 6$. The compilation of this program takes about 0.7 second and the running time to generate all solutions with the above statement is about 30 seconds. If only one solution is needed, the keyword `once` can be inserted after the `safe solve system` and only a single solution is produced (in about 2 seconds).

Chemistry is the source of many interesting nonlinear problems and our next application of `Helios` is a chemical equilibrium problem for the propulsion of propane into the air. The problem is taken from [15] and the `Helios` statement is depicted in Figure 2. It illustrates the use of constants in `Helios`. A constant gives a symbolic name to an expression; the name can then be used in specifying constraints. Constants are assumed to denote real numbers unless specified otherwise by an integer declaration. `Helios` guarantees that a constant is evaluated only once. The compilation time for this example is about 0.6 second and the running time to obtain the unique

```
Variable:
   y :  array[1..5] in [0..10^8];

Constant:
   r = 10;
   r5 = 0.193;
   r6 = 0.002597/sqrt(40);
   r7 = 0.003448/sqrt(40);
   r8 = 0.00001799/40;
   r9 = 0.0002155/sqrt(40);
   r10 = 0.00003846/40;

Body:  safe solve system
   C1:  0 = 3*y[5] - y[1]*(y[2] + 1);
   C2:  0 = y[2]*(2*y[1] + y[3]^2 + r8 + 2*r10*y[2] + r7*y[3] + r9*y[4]) + y[1] - r*y[5];
   C3:  0 = y[3]*(2*y[2]*y[3] + 2*r5*y[3] + r6 + r7*y[2]) - 8*y[5];
   C4:  0 = y[4]*(r9*y[2] + 2*y[4]) - 4*r*y[5];
   C5:  0 = y[2]*(y[1] + r10*y[2] + y[3]^2 + r8 + r7*y[3] + r9*y[4]) +
            y[1] + r5*y[3]^2 + y[4]^2 - 1 + r6*y[3];
```

Figure 2: A Chemical Equilibrium Application in Helios.

solution

$$\begin{array}{lll} \texttt{y[1]} & \texttt{in} & [0.00311409, 0.00311411] \\ \texttt{y[2]} & \texttt{in} & [34.59792453, 34.59792454] \\ \texttt{y[3]} & \texttt{in} & [0.06504177, 0.06504178] \\ \texttt{y[4]} & \texttt{in} & [0.85937805, 0.85937806] \\ \texttt{y[5]} & \texttt{in} & [0.03695185, 0.03695186] \end{array}$$

is about 5 seconds.

Our next example is the well-known Broyden Banded function problem (e.g., [5]) which amounts to finding the zeros of the system of n equations

$$f_i(x_1,\ldots,x_n) = x_i(2+5x_i^2)+1-\sum_{j\in J_i} x_j(1+x_j) \quad (1 \le i \le n)$$

where $J_i = \{j \mid \max(1, i-5) \le j \le \min(n, i+1) \ \& \ j \ne i\}$. The `Helios` statement depicted in Figure 3 follows closely the mathematical statement. There are several novelties that are worth discussing. The first novelty in this statement is the input section, which allows the user to specify constants at runtime. When `Helios` executes the above statement, it queries the user for the value of `n` to obtain the number of variables. This facility is particularly useful for problems where the dimension is not fixed. The second novelty is the use of a parametric set which defines the set J_i in the above mathematical statement. The set `sidx` is parametrized by an argument `i` and it defines a set for each `i` in `idx`. Note that the set is defined by first restricting `j` to lie in the interval `[max(1,i-5)..min(n,i+1)]` and then by using the filter `j <> i` to restrict the values in the interval. The last novelty is the use of the `Sum` operator in the constraint statement. The

```
Input:
   int n :  "Number of variables:  ";

Set:
   idx = [1..n];
   sidx(i in idx) = { j in [max(1,i-5)..min(n,i+1)] | j <> i };

Variable:
   x :  array[idx] in [-10^8..10^8];

Body:  safe solve system
   f(i in idx):
      0 = x[i] * (2 + 5 * x[i]^2) + 1 - Sum(j in sidx(i)) x[j] * (1 +x[j]);
```

Figure 3: The Broyden Banded Function in Helios

n	Compilation Time (ms)	Running Time (ms)	Growth Factor
5	290	350	
10	290	1860	5.31
20	290	6150	3.30
40	290	15360	2.49
80	290	38730	2.52
160	290	105610	2.72

Figure 4: Performance Results of Helios on the Broyden Banded function.

```
Input:
   int m :  "Give the number of variables:  ";

Set:
   idx = [1..m];

Variable:
   x :  array[idx] in [-4..5];

Constant:
   h = 1/(m+1);
   t[j in idx] = j * h;

Function:
   p(j in idx) = (x[j]+t[j]+1)^3;

Body:  safe solve system
   f(k in idx):
      0 = 2 * (m+1) * x[k] +
          (1 - t[k]) * Sum(j in [1..k]) t[j]*p(j) +
          t[k]       * Sum(j in [k+1..m]) (1-t[j])*p(j);
```

Figure 5: The Moré-Cosnard Benchmark in `Helios`.

operator takes a subscript that must range inside a set which is either defined in the set section (as it is the case here) or which is given inline. The example illustrates nicely that `Helios` statements are very close to the statements usually published in scientific papers. Another interesting fact is that `Helios` is essentially linear in the number of variables on this benchmark. The performance results are given in Figure 4.

Our last example of equation solving is another traditional benchmark from numerical analysis: the discretization of a nonlinear integral equation [19]. The objective is to find the zeros of the functions $f_k(x_1, \ldots, x_m)(1 \le k \le m)$ defined as follows:

$$x_k + \frac{1}{2(m+1)}[(1-t_k)\sum_{j=1}^{k} t_j(x_j+t_j+1)^3 + t_k \sum_{j=k+1}^{m}(1-t_j)(x_j+t_j+1)^3]$$

with $t_j = jh$ and $h = 1/(m+1)$. The `Helios` statement is depicted in Figure 5 and illustrates some new constructs as well. First, the statement shows the use of an array of constants (i.e., `t`) defined by a generic expression (i.e., `j * h`). These are called generic constants. Second, the statement illustrates the use of a function `p` which is an abbreviation for a complex expression. Note that functions can be defined in terms of variables contrary to constants. As a result, the `Helios` statement is once again very close to traditional published description of the problem. The

n	n^2	Compilation Time (ms)	Running Time (ms)	Growth Factor
5	25	480	1190	
7	49	480	3010	2.52
10	100	480	8350	2.77
14	196	480	20730	2.48
20	400	480	59490	2.86
28	784	480	176960	2.97
40	1600	480	535700	3.02

Figure 6: Performance Results of `Helios` on Moré-Cosnard Problem.

```
Set:
   idx = [1..2];
Variable:
   x :  array[idx] in [-10^8..10^8];
Body:  minimize
   100 * (x[2] - x[1]^2)^2 + (x[1] - 1)^2;
```

Figure 7: The Rosenbrock Function in `Helios`.

performance results are given in Figure 6 and indicates that `Helios` is between linear and quadratic in the size of the constraint system (i.e., between quadratic and cubic in the number of variables).

2.2 Unconstrained Optimization in `Helios`

We now turn to optimization problems and consider first unconstrained optimization. Our first example is the Rosenbrock function [13] which consists of minimizing the function

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (x_1 - 1)^2.$$

The `Helios` statement is depicted in Figure 7. It contains the keyword `minimize` followed by the objective function to be minimized (instead of the keyword `solve system` followed by the constraint system to be solved). On unconstrained optimization problems, `Helios` returns an interval bounding the value of the global optimum as well as boxes enclosing each of the optima. For instance, in the `Rosenbrock` problem, `Helios` returns

```
optimum  in  [0.00000000, 0.00000001]
x[1]     in  [0.99999999, 1.00000001]
x[2]     in  [0.99999999, 1.00000001]
```

The compilation and running times of `Helios` are 0.15 and 0.4 second respectively. It is important to stress that `Helios` bounds the global optimum value, not a local minimum value. In addition, `Helios` returns all the global optima. A nice example from [13] illustrating this fact is the

```
Input:
   int n :  "Number of variables";

Set:
   idx = [1..n];

Variable:
   x :  array[idx] in [-10..10];

Function:
   y(i in idx) = 1 + 0.25 * (x[i]-1);

Body:
   minimize
      10 * sin(pi*y(1))^2 + (y(n) - 1)^2 +
      Sum(i in [1..n-1])
          (y(i) -1)^2 * (1 + 10 * sin(pi*y(i+1))^2);
```

Figure 8: Unconstrained Optimization in Helios: Problem Levy 5.

n	Compilation Time (ms)	Running Time (s)	Growth Factor
5	230	1.04	
10	230	3.34	3.21
20	230	11.82	3.53
40	230	45.89	3.88

Figure 9: Performance Results of Helios on Problem Levy 5.

```
Set:
   idx = [1..2];

Variable:
   x :  array[idx] in [-10..10];

Body:
   minimize
      Prod(k in [1..2]) (Sum(i in [1..5]) i * cos((i+1)*x[k] + i));
```

Figure 10: Unconstrained Optimization in Helios: Problem Levy 3.

minimization of the function

$$f(x_1,\ldots,x_n) = 10\sin(\pi y_1)^2 + (y_n - 1)^2 + \sum_{i=1}^{n-1}(y_i - 1)^2(1 + 10\sin(\pi y_{i+1})^2).$$

For $n = 10$, the function has 10^{10} local minima but a single global optimum. Figure 8 depicts the `Helios` statement which involves several of the features of the languages: input constant, minimization, function, and summation. In addition, it uses a trigonometic function (i.e., `sin`) and a predefined constant `pi`. `Helios` seems to be essentially quadratic in the number of variables on this problem as shown in Figure 9. Another example is the minimization of the function

$$f(x_1, x_2) = \prod_{k=1}^{2}\sum_{i=1}^{5} i\cos((i+1)x_k + i).$$

which has 760 local minima and 18 global minima in [-10,10]. The `Helios` statement is shown in Figure 10 and it illustrates the product operator. Note that `Helios` returns boxes for all 18 global minima in about 20 seconds.

2.3 Constrained Optimization in `Helios`

Constrained optimization problems can also be stated in `Helios`. Figure 11 depicts the `Helios` statement of a constrained optimization problem taken from [6]. The main difference with unconstrained optimization is the keyword `subject to` which is followed by the description of the constraints to be satisfied. Once again, `Helios` is guaranteed to return an interval enclosing the value of the global optimum as well as boxes enclosing all the global optima. To obtain this functionality, it is only necessary to include the keyword `safe` before the keyword `minimize`. When the keyword `safe` is omitted, `Helios` is optimistic and simply assumes that each canonical box that is not pruned away by the constraints contains a solution.[1]

2.4 Additional Functionalities

`Helios` has a number of additional functionalities besides those described previously. For instance, `Helios` has a number of pragmas to influence the constraint solver (e.g., to specify the width of the intervals). It also enables users to state soft constraints which can be used to prune the search space but do not affect `Helios`' ability to prove existence and unicity of solutions. Conceptually, soft constraints should be seen as filters on the solutions which returns those solutions not falsifying them. Soft constraints are useful, for instance, to prune symmetric solutions in problems such as `Levy 3`. A new `Helios` statement for `Levy 3`, including soft constraints is depicted in Figure 12 and it reduces the execution of time by a factor of 2. There is however one additional functionality that is worth mentioning: the display section. The display section makes it possible to specify the information to be produced. Figure 13 describes the `Helios` statement for the kinematics example with a display section. The display section requests to display the variables as well as the constraints `C1,...,C6`. The output is depicted in Figure 14. As should be clear, an interval is associated with each variable. In addition, three intervals are associated with each constraint, bounding the left-hand side, the right-hand side, and their difference. Note that displaying the

[1] Note that, in unconstrained optimization, no such issue arises since there are no constraints.

```
Variable:
   x1 in [0..0.31];
   x2 in [0..0.046];
   x3 in [0..0.068];
   x4 in [0..0.042];
   x5 in [0..0.028];
   x6 in [0..0.0134];
Constant:
   b1 = 4.97;
   b2 = -1.88;
   b3 = -29.08;
   b4 = -78.02;
Body:
   safe minimize
      4.3 * x1 + 31.8 * x2 + 63.3 * x3 + 15.8 * x4 + 68.5 * x5 + 4.7 * x6
   subject to
      C1 :  17.1 * x1 + 38.2 * x2 + 204.2 * x3 + 212.3 * x4 + 623.4 * x5 +
            1495.5 * x6 - 169 * x1 * x3 - 3580 * x3 * x5 - 3810 * x4 * x5 -
            18500 * x4 * x6 - 24300 * x5 * x6 >= b1;
      C2 :  17.9 * x1 + 36.8 * x2 + 113.9 * x3 + 169.7 * x4 + 337.8 * x5 +
            1385.2 * x6 - 139 * x1 * x3 - 2450 * x4 * x5 - 16600 * x4 * x6 - 17200
            * x5 * x6 >= b2;
      C3 :  -273 * x2 - 70 * x4 - 819 * x5 + 26000 * x4 * x5 >= b3;
      C4 :  159.9 * x1 - 311 * x2 + 587 * x4 + 391 * x5 + 2198 * x6
            - 14000 * x1 * x6 >= b4;
```

Figure 11: Constrained Optimization in Helios.

```
Set:
   idx = [1..2];

Variable:
   x :  array[idx] in [-10..10];

Body:
   minimize
      Prod(k in [1..2]) (Sum(i in [1..5]) i * cos((i+1)*x[k] + i))
   with soft constraints
      x[1] >= x[2];
```

Figure 12: Unconstrained Optimization in Helios: Problem Levy 3 with Soft Constraints.

```
Set:
   idx = [1..6];

Variable:
   s :  array[idx] in [-10^8..10^8];
   c :  array[idx] in [-10^8..10^8];

Body:  safe solve system once
   trigo(i in idx) :  s[i]^2 + c[i]^2 = 1;
   C1 :  s[2]*c[5]*s[6] - s[3]*c[5]*s[6] - s[4]*c[5]*s[6] +
         c[2]*c[6] + c[3]*c[6] + c[4]*c[6] = 0.4077;
   C2 :  c[1]*c[2]*s[5] + c[1]*c[3]*s[5] + c[1]*c[4]*s[5] + s[1]*c[5] = 1.9115;
   C3 :  s[2]*s[5] + s[3]*s[5] + s[4]*s[5] = 1.9791;
   C4 :  c[1]*c[2] + c[1]*c[3] + c[1]*c[4] + c[1]*c[2] + c[1]*c[3] + c[1]*c[2] = 4.0616;
   C5 :  s[1]*c[2] + s[1]*c[3] + s[1]*c[4] + s[1]*c[2] + s[1]*c[3] + s[1]*c[2] = 1.7172;
   C6 :  s[2] + s[3] + s[4] + s[2] + s[3] + s[2] = 3.9701;
Display:
   variables:  c,s;
   constraints :  C1,C2,C3,C4,C5,C6;
```

Figure 13: A Robot Kinematics Application in `Helios` with Display Section.

constraints provides users with information on numerical accuracy and makes it possible to detect badly conditioned or badly formulated problems. It suffices to examine the size of the intervals associated with the constraints. Figure 15 describes the output of the constrained optimization problem `h95`. In optimization problems, the information displayed also allows users to find which inequalities are tight at optimality. In `h95`, for instance, the first constraint is tight, while the remaining ones are not.

3 The Implementation of `Helios`

We now turn to the implementation of `Helios`. The first subsection presents an overview of the code generation while the subsequent subsections describe the code generation for the input, constant, variable, and body sections. The whole language is not fully covered but the material should give readers a precise idea on the overall compilation process.

3.1 Overview of Code Generation

`Helios` is compiled down to `Newton`, a constraint logic programming language over nonlinear constraints based on interval analysis. More precisely, an `Helios` statement is compiled into a `Newton` program that can then be executed to solve the initial problem. A complete description of `Newton` is available in [27]. However, for the purpose of this paper, it is almost always sufficient to view `Newton` as `Prolog` enhanced with a number of predefined predicates for solving nonlinear constraints and for optimizing an objective function subject to a system of constraints. In other words, the constraint-solving aspects of `Newton` are encapsulated in predicates of the form `solveSystem(Constraints)`

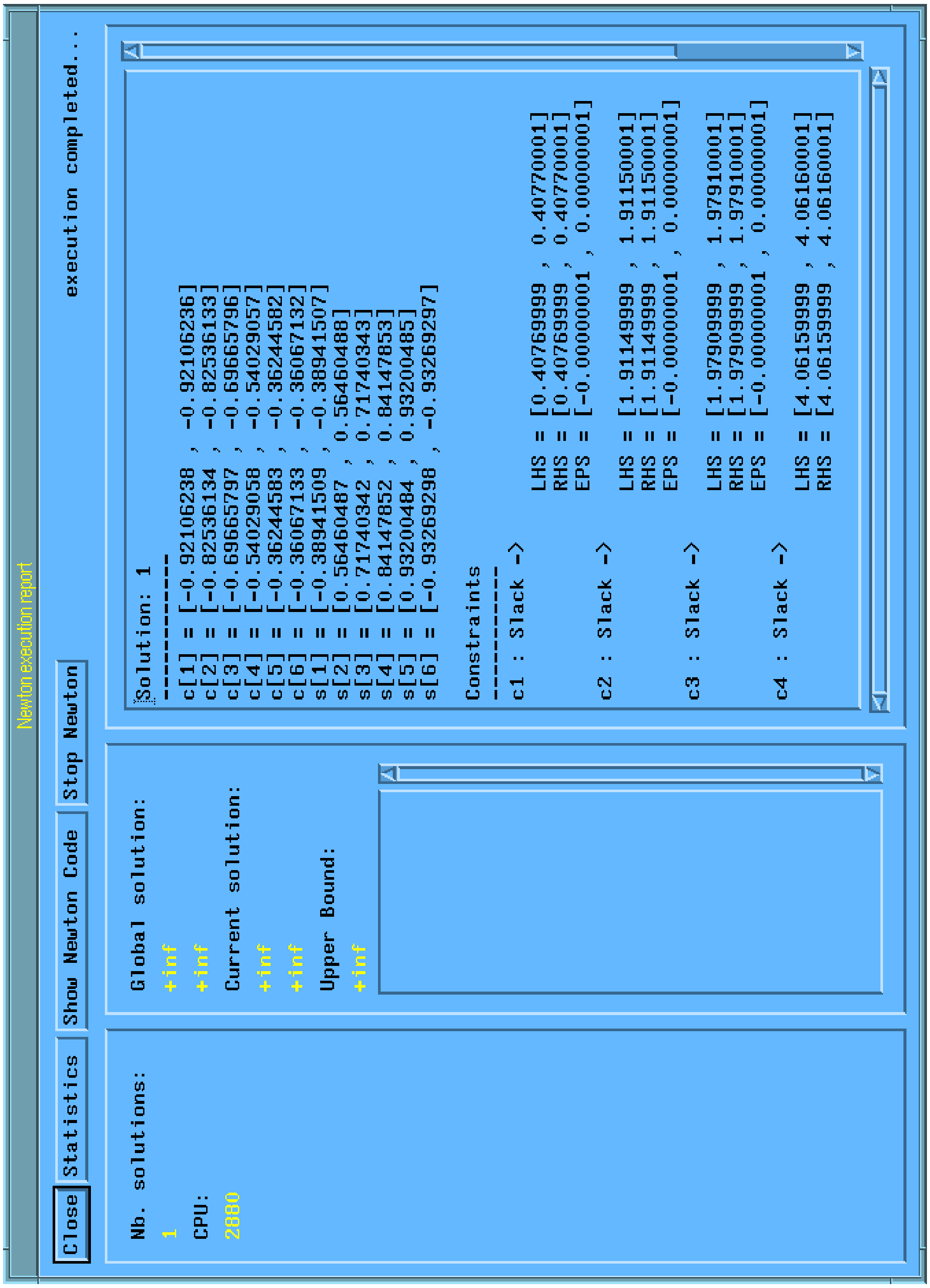

Figure 14: The Display of the Solution to the Kinematics Problem.

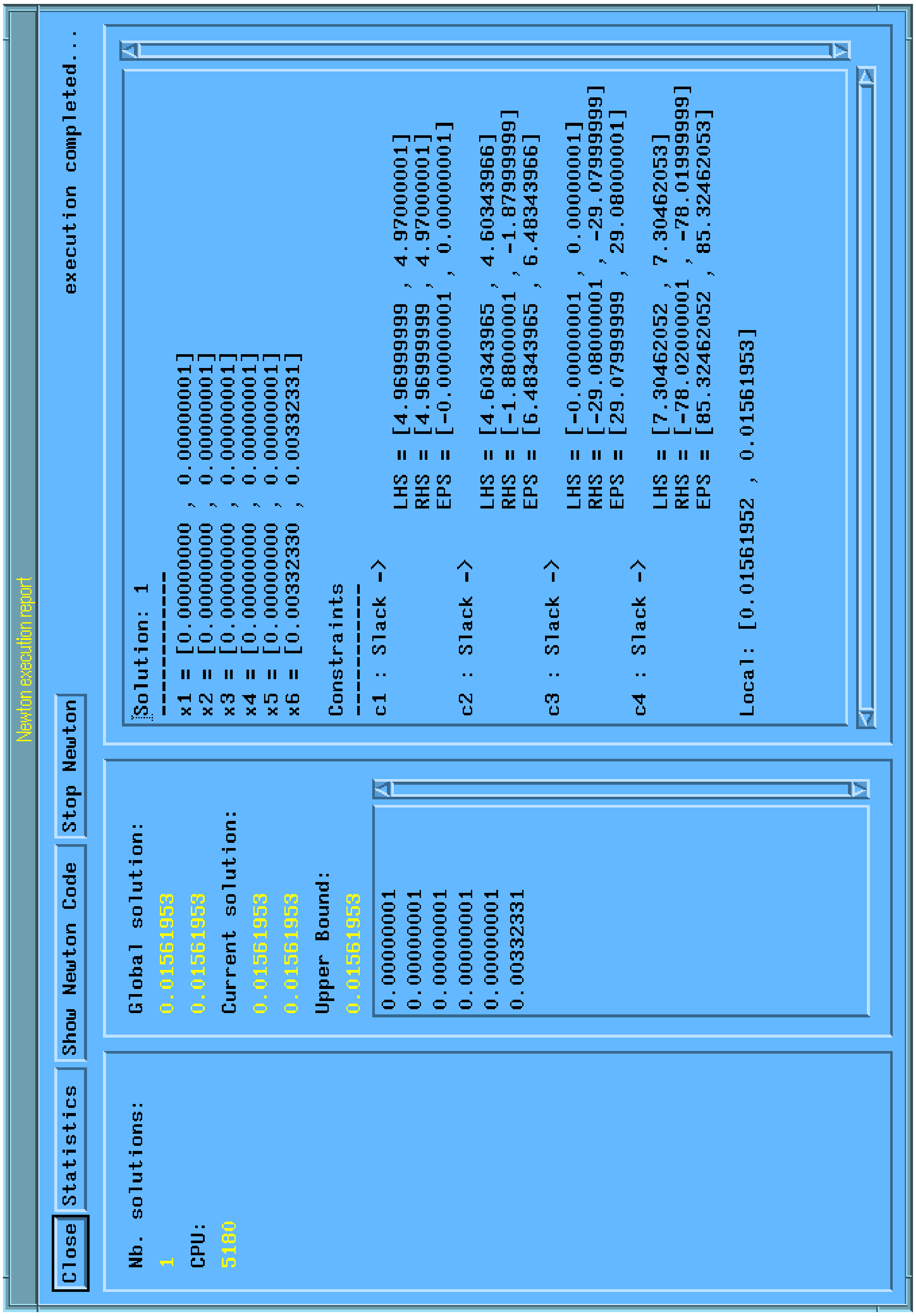

Figure 15: The Display of the Solution to Problem `h95`.

```
⟨Constraint ⟩ ::=  ⟨Expr⟩ >= ⟨Expr⟩     |
                   ⟨Expr⟩ <= ⟨Expr⟩     |
                   ⟨Expr⟩ =  ⟨Expr⟩

⟨Expr⟩        ::=  ⟨Var⟩                |
                   ⟨Float⟩              |
                   ⟨Expr⟩ * ⟨Expr⟩      |
                   ⟨Expr⟩ + ⟨Expr⟩      |
                   ⟨Expr⟩ - ⟨Expr⟩      |
                   ⟨Expr⟩ ^ ⟨Nat⟩       |
                   - ⟨Expr⟩             |
                   sin(⟨Expr⟩)          |
                   cos(⟨Expr⟩)          |
                   log(⟨Expr⟩)          |
                   exp(⟨Expr⟩)          |
                   ...
```

Figure 16: The Syntax of Constraints in `Newton`.

```
top :-
     createArray(Constant,nbConstant),
     createArray(Function,nbFunction),
     createArray(Variable,nbVariable),
     createArray(Scope,nbIndex),
     Global = environment(Constant,Function,Variable,Scope),
     generateInput(Global),
     generateConstant(Global),
     generateVariable(Global),
     generateFunction(Global),
     generateConstraints(Global,Constraints),
     solveSystem(Constraints).
```

Figure 17: Code Generation: The Top-level Predicate.

and `minimize(Function,Constraints)`.[2] As a consequence, the main task of the implementation is to translate an `Helios` statement into a `Newton` program which generates a set of `Newton` constraints. The syntax of `Newton` constraints is summarized in Figure 16. It should be clear that `Newton` constraints are relatively simple compared to `Helios` statements, since, for instance, they do not use arrays, functions, summations, and products.

Figure 17 presents the top-level part of the code generated by the `Helios` compiler. Executing predicate `top` solves the problem described by the corresponding `Helios` statement. The body of the clause creates a number of arrays to store the constants, functions, and variables of the `Helios` statement. There is also an array `Scope` that is used to implement generic constants, functions, and constraints as well as the product and sum operators. All these arrays are collected in a single data-structure, the environment, which is used by all other parts of the generated program. The remaining goals in the body of `top` (except the last one which calls the constraint solver) corresponds to the code generated for the various `Helios` sections: the input, constant, variable, function, and body sections. The rest of this section studies these goals in detail.

A Note on Notation The `Newton` code generated by the compiler often contains compiler constants which depends on the particular `Helios` statement being compiled. For instance, each name in `Helios` is associated with a natural number which represents an index in an array. We take the convention of depicting these compiler constants in italics to improve readibility. The top-level of the code depicted in Figure 17 illustrates this already. For instance, *nbVariable* is a compiler constant representing the number of variables in the `Helios` statement. We also assume the existence of a number of library functions on arrays. In particular, `createArray(A,Size)` creates an array `A` with `Size` elements, `get(A,I,V)` returns in `V` the element `A[I]`, and `put(A,I,V)` sets the value of `A[I]` to `V`. These functions can easily be implemented by open-ended lists in Prolog.

3.2 The Input Section

The compilation of the input section generates code that queries the user for the values of the input constants and that stores these values in the array `Constant`. As mentioned previously, the compiler (in the parsing phase) associates a natural number with each name in an `Helios` statement: this name is used as the index in the array. Consider, for instance, the following `Helios` code:

```
Input:
   int n :  "Number of rows:  ";
   int m :  "Number of columns:  ";
```

and assume that `0` and `1` are the natural numbers associated with `n` and `m`. The code generated is as follows:

```
generateInput(Global) :-
       input_0(Global).
input_0(Global) :-
       Global = environment(Constant,Function,Variable,Scope),
       heliosReadInteger('Number of rows:',Local),
```

[2] The actual predicates are more complex, since they take more parameters and return some output values. For readibility, we use the simplified forms in this section.

```
        put(Constant,0,Local),
        input₁(Global).

input₁(Global) :-
        Global = environment(Constant,Function,Variable,Scope),
        heliosReadInteger('Number of columns:',Local),
        put(Constant,1,Local),
        input₂(Global).
input₂(Global).
```

Informally speaking, predicate $\texttt{input}_0$ reads the value of the constant `n` and stores it in the array `Constant` at index 0, while predicate $\texttt{input}_1$ reads the value of the constant `m` and stores it in the array `Constant` at index 1. The two predicates are chained together to read the two constants. The general pattern for the input section (for n input constants) is thus as follows:

```
generateInput(Global) :-
        input₀(Global).
        ⋮
inputᵢ(Global) :-
        Global = environment(Constant,Function,Variable,Scope),
        heliosRead('Prompt',Local),
        put(Constant,Offset_object,Local),
        input_i+1(Global).
        ⋮
input_n(Global).
```

3.3 The Constant Section

The compilation of the constant section follows the same basic idea as the code for the input section. The general pattern is depicted in Figure 18. The main difference is of course that the value of the constant is not read from the user; rather it is defined by an expression that must be evaluated. In addition, the constant itself can be an array or it can be generic, which complicates the compilation substantially. These details are abstracted inside predicates of the form

$\texttt{generateConstant}_j\texttt{(Global,Value)}$

which generates the value for the jth constant. We now consider the implementation of this predicate for each of the three kinds of constants: individual, array, and generic. We assume that the constants are of type `real`; the same compilation process applies to integer constants.

Individual Constants: The value of an individual constant is a floating-point number resulting of the evaluation of its defining expression. The definition of $\texttt{generateConstant}_j$ for individual constant is as follows:

```
generateConstant_j(Global,Result) :-
        generateExpression_k(Global,Local),
        evalReal(Local,Result).
```

```
generateConstant(Global) :-
        constant_0(Global).
        ⋮
constant_i(Global) :-
        Global = environment(Constant,Function,Variable,Scope),
        generateConstant_j(Global,Local),
        put(Constant,Offset_object,Local),
        constant_{i+1}(Global).
        ⋮
constant_n(Global).
```

Figure 18: Constant Section: Top-Level Code Generation

Predicate `generateExpression`$_k$`/2` generates an expression that is then evaluated by `evalReal/2` to produce the result. The basic idea to generate the expression `Local` is to use the syntactic tree of the `Helios` expression. More precisely, a predicate is associated with each node in the syntactic tree and the predicate combines the expressions of its subtrees to produce its result. The basic cases are of course numbers, which are returned directly, and constants, which are handled by looking up their values in the array `Constant`. Note that `Helios` takes the same convention as `Pascal` in that an object can only be used after its declaration. Consider the `Helios` code

```
Constant:
   b = 2 + 3 * n;
```

The code generated for the fragment is as follows:

```
generateConstant_1(Global,Result) :-
        generateExpression_0(Global,Local),
        evalReal(Local,Result).
generateExpression_0(Global,Result) :-
        generateExpression_1(Global,Local1),
        generateExpression_2(Global,Local2),
        Result = Local1 + Local2.
generateExpression_1(Global,2).
generateExpression_2(Global,Result) :-
        generateExpression_3(Global,Local1),
        generateExpression_4(Global,Local2),
        Result = Local1 * Local2.
generateExpression_3(Global,3).
generateExpression_4(Global,Result) :-
        Global = environment(Constant,Function,Variable),
        get(Constant,Offset_n,Result).
```

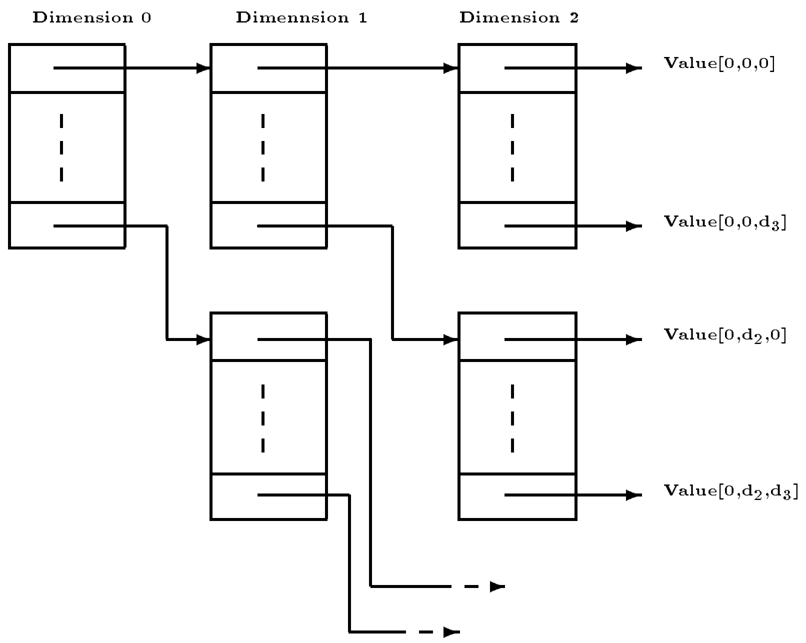

Figure 19: Constant Array: An Example of a 3-dimensional Array.

Predicate `generateExpression`$_1$`/2` shows the case of a number, predicate `generateExpression`$_4$`/2` illustrates the case of a constant, while `generateExpression`$_0$`/2` illustrates the case of an internal node.

Arrays: An array of constants is also assigned a single entry in the array `Constant`. Of course, the value of this entry does not represent a number but rather an array. In addition, multi-dimensional arrays are represented as arrays of one-dimensional arrays as in Pascal compilers. Figure 19 illustrates these principles on a 3-dimensional array. The purpose of the generated code is thus to create and initialize this array. Consider the following `Helios` code:

```
Constant:
       unit :  array[1..2,1..2] = [[1,0],[0,1]];
```

The code generated is as follows:

```
generateConstant(Global) :-
       constant0(Global).
constant0(Global) :-
       Global = environment(Constant,Function,Variable,Scope),
       generateConstant0(Global,Local),
       put(Constant,Offseta,Local),
       constant1(Global).
constant1(Global).
```

```
generateConstant0(Global,Result),
       Local1 = [ dim(1,2) , dim(1,2)],
       libBuildArray(Result,Local1),
       Local2 = [ 1 , 0 ],
       Local3 = [ 0 , 1 ],
       Local4 = [ Local2 , Local3 ],
       libFillArray(Result,Local4),
```

The interesting part is of course predicate $\texttt{generateConstant}_0$ which constructs a list of the dimensions and then calls the library function `libBuildArray` to construct the array. The rest of the body creates the list of values in a hierarchical way and calls the library function `libFillArray` to initialize the array.

Generic Constants: The compilation of generic constants follows essentially the same pattern as array of constants. There are however some important differences which complicate code generation. Consider, for instance, the `Helios` code

```
Constant:
       foo[i in [1..n],j in [1..10]] = i^j;
```

Two main differences with an array of constants can be observed. First, the dimension of the array is not necessarily fixed at compile time (e.g., `n` may be an input parameter). Second, and most important, the value for each position in the array is not given, but rather it must be computed from the expression on the right-hand side. The general pattern for predicate $\texttt{generateConstant}_i$ in the case of generic constants is as follows:

```
generateConstant_i(Global,Result) :-
       collect_j(Global,Dim),
       libBuildArray(Result,Dim),
       fillArray_k(Global,Result).
```

Predicate $\texttt{collect}_j$ is responsible for collecting the dimension of the array. Predicate $\texttt{fillArray}_k$ is responsible for filling the array and is the most difficult part in this case. The general pattern for collecting n dimensions is as follows:

```
collect_j(Global,Result) :-
       generateExpression_l(Global,LeftE),
       generateExpression_m(Global,RightE),
       evalInteger(LeftE,Left),
       evalInteger(RightE,Right),
       Result = [dim(Left,Right)|Tail],
       collect_{j+1}(Global,Tail).
       ⋮
collect_{j+n}(Global,[]).
```

Expressions for the lower and upper bound of the first dimension are generated and evaluated to obtain integers which are then stored as the first element of a list. The tail of the list is obtained by a predicate which collect the remaining dimensions.

Filling the array is the most difficult part as mentioned, since the expression must be evaluated for all tuples of indices. The code generated for this step makes use of the array `Scope`. An index into this array is associated with each of the formal parameters and the corresponding entry holds the current value of the parameter at any given time. The generated code associates a predicate with each dimension. Let us illustrate these principles on the simple example:

```
Constant:
        pow[i in [1..n]] = i^4;
```

The code is depicted in Figure 20. Predicate $\texttt{fillArray}_0\texttt{/2}$ evaluates the dimension for the parameter and calls $\texttt{fillArray}_0\texttt{/3}$ which considers each element in turn. For each element, the predicate $\texttt{fillArray}_1$ evaluates the expression given the current value of the parameter `i`, which is stored in array `Scope`.

For generic constants with several parameters, there are as many predicates $\texttt{fillArray}_i$ as the number of parameters and these predicates are once again chained together. The evaluation of the expression takes place at the end of the chain. For instance, the previously shown `Helios` code

```
Constant:
        foo[i in [1..n],j in [1..10]] = i^j;
```

uses two entries in the array `Scope` and requires the generation of three predicates $\texttt{fillArray}_i$. The evaluation of the expression takes place in $\texttt{fillArray}_2$.

3.4 Variable Section

The code generation for variable is essentially the same as the code for constants. The main difference is that entries in the array `Variable` do not hold values but rather free variables (that may have been constrained to take their values in some ranges).

3.5 Function Section

The code generation for the function section once again follows closely the code generation for constants. The main difference is that entries in array `Function` do not contain numbers but rather expressions, since functions can contain variables. From a code generation standpoint, this does not introduce any complication, since it suffices to omit the evaluation code and to return the expression itself as the result. Consider for instance the `Helios` code

```
Function:
        foo(i in [1..n],j in [1..4]) = x[i] + j;
```

The entry corresponding to `foo` in the array `Function` simply contains the expression depicted in Figure 21.

3.6 The Body Section

The compilation of the body section is responsible for generating the code to create the constraints. Recall that constraints can be individual or generic as was the case for constants and functions. The overall pattern for code generation is based on similar ideas as the code for constants and functions, except that all constraints must be collected. The collecting process is performed efficiently using difference-lists by using the following general pattern to generate n constraints:

```
fillArray0(Global,Array) :-
       Global = environment(Constant,Function,Variable,Scope),
       generateExpression1(Global,Local1),
       generateExpression2(Global,Local2),
       evalInteger(Local1,Left),
       evalInteger(Local2,Right),
       put(Scope,1,Left),
       fillArray0(Global,Array,Right).

fillArray0(Global,Array,Max) :-
       Global = environment(Constant,Function,Variable,Scope),
       get(Scope,1,Current),
       Current > Max.
fillArray0(Global,Array,Max) :-
       Global = environment(Constant,Function,Variable,Scope),
       get(Scope,1,Current),
       Current =< Max,
       Next is Current + 1,
       get(Array,Current,Slot),
       fillArray1(Global,Slot),
       put(Scope,1,Next),
       fillArray0(Global,Array,Max).

fillArray1(Global,Array) :-
       generateExpression3(Global,Local),
       evalInteger(Local,Array).

generateExpression3(Global,Result) :-
       generateExpression4(Global,Local1),
       generateExpression5(Global,Local2),
       Result = Local1^Local2.
generateExpression4(Global,Result) :-
       Global = environment(Constant,Function,Variable,Scope),
       get(Scope,1,Local),
       Result = Local.
generateExpression5(Global,4).
```

Figure 20: Constant Section: Code Generation of the Generic Constant pow.

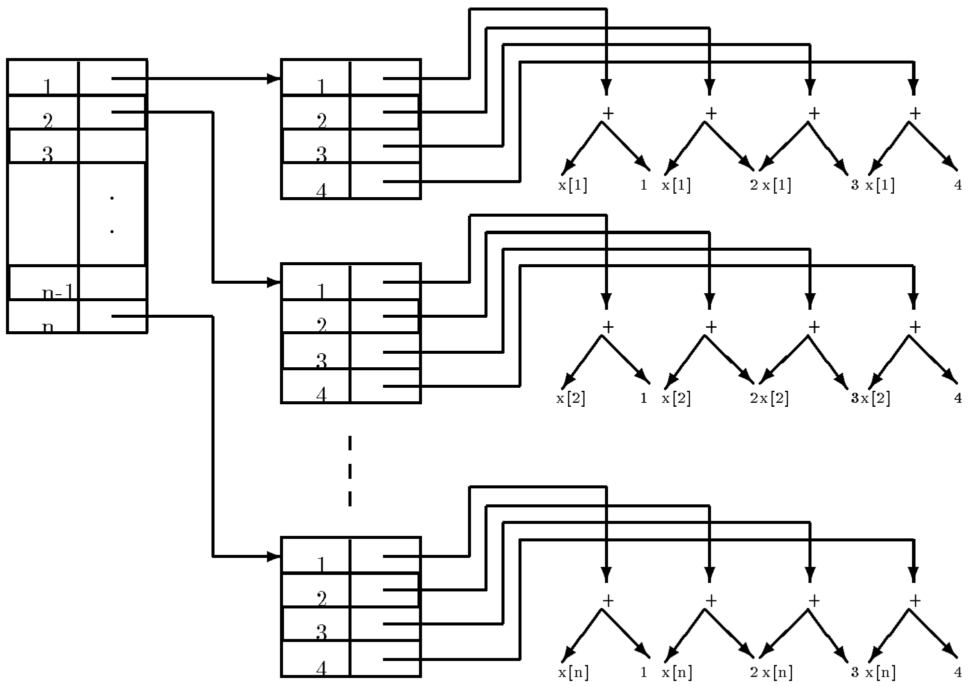

Figure 21: Function: The Entry in Array `Function` for Function `foo`.

```
generateConstraints(Global,Result) :-
        collect_i(Global,Result).
collect_i(Global,Result) :-
        Global = environment(Constant,Function,Variable,Scope),
        genThisConstraint_k(Global,Result,Tail),
        collect_{i+1}(Global,Tail).
        ⋮
collect_{i+n}(Global,[]).
```

The main difficulty is thus the generation of a constraint which possibly contains references to arrays, constants, and variables, may call user-defined functions, and may use product and sum operators. In the rest of this section, we consider the compilation of these various components which can be encountered during code generation of a constraint.

Individual References: Individual references include both individual constants and variables. The code consists of accessing the suitable array:

```
generateExpression_i(Global,Result) :-
        Global = environment(Constant,Function,Variable,Name),
        get(objectClass,Offset_object,Result).
```

where **objectClass** denotes the array `Constant` or the array `Variable` depending on the type of reference encountered.

```
generateExpression_i(Global,Result) :-
        generateExpression_p(Global,Exp_1),
        ⋮
        generateExpression_{p+n}(Global,Exp_n),
        evalInteger(Exp_1,Local_1),
        ⋮
        evalInteger(Exp_n,Local_n),
        Local_{n+1} = [Local_1,...,Local_n],
        Global = environment(Constant,Function,Variable),
        get(objectClass,Offset_object,Object),
        libDeref(Object,Local_{n+1},Result).
```

Figure 22: Code Pattern for Complex Reference of Arity n

Complex References: The code for accessing a complex reference contains two steps. First, its expressions for the indices (or parameters) are evaluated to produce a list of indices. For instance, a call to `foo(m+p,n)` generates code to evaluate `m+p` and `n` to produce a list `[i1,i2]`. Second, the list of indices is used to access the appropriate array. For instance, if `[i1,i2]` is the list `[5,4]` and the function `foo` is defined as previously, i.e.,

```
Function:
        foo(i in [1..n],j in [1..4]) = x[i] + j;
```

this second step fetches the expression `x[5] + 4` from array `Function`. This second step is performed by a predicate `listDeref/3` defined as follows:

```
listDeref([],Value,Value).
listDeref([H|T],Array,Value) :-
        get(Array,H,Element),
        listDeref(T,Element,Value).
```

The code pattern for complex references is depicted in Figure 22.

Aggregate Operators: It remains to specify how to generate code for the aggregate operators `sum` and `product`. The code generation strategy is based on the unfolding of these operators. Consider, for instance, an expression

$$\sum_{i=1,i\neq j}^{n} x[i] * j.$$

This expression can be unfolded into

$$x[1] * j + \sum_{i=2,i\neq j}^{n} x[i] * j$$

if $j \neq 1$ and into

$$\sum_{i=2, i \neq j}^{n} x[i] * j$$

otherwise. This suggests the generation of a recursive predicate that makes a case analysis to filter the value of the index variable. The code pattern for summation is depicted in Figure 23. Predicate $\texttt{sum}_k$ has three clauses. The first clause corresponds to the basic case and returns 0. The second case handles the case where the current value for the summation variable satisfies the filter (i.e., `i` $\neq$ `j` in the above example), while the third clause handles the case where the value does not satisfy the filter. When there is no filter, the third clause and the filter can be omitted. The pattern for the product operator follows the same principles.

4 Experimental Results

We now turn to the experimental results of `Helios` on traditional benchmarks. We consider successively equation solving, unconstrained optimization, and constrained optimization.

4.1 Equation Solving

We start with experimental results of `Helios` on a variety of standard benchmarks for equation solving. The benchmarks were taken from papers on numerical analysis [19], interval analysis [5, 7, 18], and continuation methods [28, 21, 20, 15]. Complete details on the benchmarks can be found in [26, 27]. We also compare `Helios` with a traditional interval method using the Hansen-Segupta's operator, range testing, and branching. This method uses the same implementation technology as `Helios` and is denoted by `HRB` in the following.[3] Finally, we compare `Helios` with a state-of-the-art continuation method [28], denoted by `CONT` in the following. Note that all results given in this section were obtained by running `Helios` on a `Sun Sparc 10` workstation to obtain all solutions. In addition, the final intervals must have widths smaller than 10^{-8}. The results are summarized in Table 1. For each benchmark, we give the number of variables (n), the total degree of the system (d), the initial range for the variables, and the results of each method in seconds. Note that the times for the continuation method are on a `DEC 5000/200`. A space in a column means that the result is not available for the method. A question mark means that the method does not terminate in a reasonable time (> 1 hour). Note that `Helios` solves `Broyden`, `Moré-Cosnard`, and interval benchmarks `i1, i2, i3` and `i5` without backtracking (contrary to most interval methods we know of).

4.2 Unconstrained Optimization

Table 2 describes the results of `Helios` on unconstrained optimization. The benchmarks were taken mainly from [13, 8, 23, 25] and, for each of them, we give the number of variables, the range of the variables, the `CPU` time, and the number of splits. Full details on the benchmarks can be found in [27]. The experimental results once again exhibit a number of interesting facts. `Helios` is able to

[3] Some interval methods such as [4] are more sophisticated than `HRB` but the sophistication aims at speeding up the computation near a solution. Our main contribution is completely orthogonal and aims at speeding up the computation when far from a solution and hence comparing it to `HRB` is meaningful.

```
generateExpression_i(Global,Result) :-
        Global = environment(Constant,Function,Variable,Scope),
        generateExpression_j(Global,Local_1),
        generateExpression_{j+1}(Global,Local_2),
        evalInteger(Local_1,Local_3),
        evalInteger(Local_2,Local_4,
        put(Scope,Offset_index,Local_3),
        sum_k(Global,Local_4,Result).

sum_k(Global,Upper,0) :-
        Global = environment(Constant,Function,Variable,Scope),
        get(Scope,Offset_index,Local_1),
        Local_1 > Upper.
sum_k(Global,Upper,Result) :-
        Global = environment(Constant,Function,Variable,Scope),
        get(Scope,Offset_index,Local_1),
        Local_1 =< Upper,
        filter_l(Global),
        generateExpression_m(Global,Local_2),
        Result = Local_2 + Local_3,
        Next is Local_1 + 1,
        put(Scope,Offset_index,Next),
        sum_k(Global,Upper,Local_3).
sum_k(Global,Upper,Result) :-
        Global = environment(Constant,Function,Variable,Scope),
        get(Scope,Offset_index,Local_1),
        Local_1 =< Upper,
        not filter_l(Global),
        Next is Local_1 + 1,
        put(Scope,Offset_index,Next),
        sum_k(Global,Upper,Result).
filter_l(Global) :-
        ... /* Some code that evaluates a predicate, succeeds
         if the predicate is true and fails otherwise.  */
```

Figure 23: Code Pattern for the Summation Operator.

Benchmarks	v	d	range	Helios	HRB	CONT
Broyden	10	3^{10}	[-1,1]	1.78	18.23	
Broyden	20	3^{20}	[-1,1]	5.36	?	
Broyden	160	3^{160}	[-1,1]	95.09	?	
Broyden	160	3^{160}	$[-10^8, 10^8]$	105.610	?	
Moré-Cosnard	20	3^{20}	$[-4, 5]$	59.49	968.25	
Moré-Cosnard	40	3^{40}	$[-4, 5]$	535.70	?	
Moré-Cosnard	40	3^{40}	$[-10^8, 0]$	538.41	?	
i1	10	3^{10}	[-2,2]	0.06	14.28	
i2	20	3^{20}	[-1,2]	0.30	1821.23	
i3	20	3^{20}	[-2,2]	0.31	5640.80	
i4	10	6^{10}	[-1,1]	73.94	445.28	
i5	10	11^{10}	[-1,1]	0.08	33.58	
kin1	12	4608	$[-10^8, 10^8]$	14.24	1630.08	
kin2	8	256	$[-10^8, 10^8]$	353.06	4730.34	35.61
eco	4	18	$[-10^8, 10^8]$	0.60	2.44	1.13
eco	5	54	$[-10^8, 10^8]$	3.35	29.88	5.87
eco	6	162	$[-10^8, 10^8]$	22.53	?	50.18
eco	7	486	$[-10^8, 10^8]$	127.65	?	991.45
eco	8	1458	$[-10^8, 10^8]$	915.24	?	
eco	9	4374	$[-10^8, 10^8]$	8600.28	?	
combustion	10	96	$[-10^8, 10^8]$	9.94	?	57.40
chemistry	5	108	$[0, 10^8]$	6.32	?	56.55
neuro	6	1024	$[-10, 10]$	0.91	28.84	5.02
neuro	6	1024	$[-1000, 1000]$	172.71	?	5.02

Table 1: Summary of the Experimental Results on Equation Solving.

Benchmarks	v	Range	Time	Splits
Hump	2	$[-10^7, 10^8]$	0.17	3
Levy1	1	$[-10^7, 10^7]$	0.09	2
Levy2	1	$[-10, 10]$	0.61	4
Levy3	2	$[-10, 10]$	16.14	30
Levy4	2	$[-10, 10]$	2.13	7
Levy5	3	$[-10, 10]$	0.75	1
Levy5	5	$[-10, 10]$	1.04	1
Levy5	10	$[-10, 10]$	3.34	1
Levy5	20	$[-10, 10]$	11.82	1
Levy5	40	$[-10, 10]$	45.89	1
Levy5	80	$[-10, 10]$	235.22	1
Levy6	3	$[-10, 10]$	0.96	1
Levy6	5	$[-10, 10]$	1.57	1
Levy6	10	$[-10, 10]$	4.29	1
Levy6	20	$[-10, 10]$	14.86	1
Levy6	40	$[-10, 10]$	64.11	1
Levy6	80	$[-10, 10]$	372.39	1
Beale	2	$[-4.5, 4.5]$	2.50	3
Beale	2	$[-10^2, 10^2]$	3.31	12
Beale	2	$[-10^4, 10^4]$	5.52	31
Beale	2	$[-10^7, 10^7]$	23.29	61
Schwefel1	3	$[-10^7, 10^7]$	0.24	0
Booth	2	$[-10^7, 10^7]$	0.11	0
Powell	4	$[-10, 20]$	6.69	267
Schwefel3	2	$[-10^7, 10^7]$	0.03	0
Rosenbrock	2	$[-10^7, 10^7]$	0.33	10
Ratz1	5	$[-500, 600]$	1.19	0
Ratz25	4	$[0, 10]$	2.86	0
Ratz27	4	$[0, 10]$	4.44	0
Ratz210	4	$[0, 10]$	7.42	0
Ratz3	6	$[0, 1]$	9.13	2
More1	3	$[-4, 4]$	10.04	5
More2	4	$[-25, 25]$	189.56	32

Table 2: Summary of the Experimental Results on Unconstrained Optimization.

Benchmarks	v	c	Time	Splits
h95	6	16	2.59	10
h96	6	16	2.64	11
h97	6	16	103.16	230
h98	6	16	51.59	226
h100	7	18	53.71	131
h106	8	22	926.72	149
h113	10	28	4410.26	7296

Table 3: Summary of the Experimental Results on Constrained Optimization.

solve problems such as `Levy5` and `Levy6` in essentially linear time in the number of variables. `Helios` solves the problems `Ratz25`, `Ratz27`, and `Ratz210` without splitting. These problems were used in [23] to study splitting strategies. Finally, `Helios` does not exhibit the behaviour of traditional interval methods on problems such as the Rosenbrock function. The performance of the traditional interval degrades substantially when the initial intervals are large, while `Helios` can be used with arbitrarily large intervals in these cases (without degrading the performance).

4.3 Constrained Optimization

We now turn to constrained optimization problems which are, in general, very difficult to solve. Table 3 summarizes some of our computation results on some of the toughest problems from [6]. We give the number of variables in the initial statement (v), the number of constraints (c), the `CPU` time, and the number of splits. Note that, for a problem with n variables and m constraints, the system generates a constraint problem involving $n + m$ variables when using the Fritz-John conditions.

5 Conclusion

This paper has presented `Helios`, a mathematical modeling language for nonlinear constraint solving and global optimization. `Helios` makes it possible to state nonlinear problems in a form close to traditional statements published in scientific papers. `Helios` is compiled into `Newton`, a constraint logic programming over nonlinear constraints. `Newton` is based on interval analysis and is one of the most efficient tools for the global solutions of nonlinear problems. The compilation of `Helios` into `Newton` has been described in detail and uses some advanced logic programming techniques. Experimental results on a variety of standard benchmarks indicate that `Helios` should be a valuable addition to the available tools in this application area.

Acknowledgments

This work was supported in part by the Office of Naval Research under grant ONR Grant N00014-94-1-1153 and a NSF National Young Investigator Award with matching funds of Hewlett-Packard.

References

[1] R. Fourer, D. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[2] R. Hammer, M. Hocks, M. Kulisch, and D. Ratz. *Numerical Toolbox for Verified Computing I – Basic Numerical Problems, Theory, Algorithms, and PASCAL-XSC Programs*. Springer-Verlag, Heidelberg, 1993.

[3] E. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, New York, 1992.

[4] E.R. Hansen and R.I. Greenberg. An Interval Newton Method. *Appl. Math. Comput.*, 12:89–98, 1983.

[5] E.R. Hansen and S. Sengupta. Bounding Solutions of Systems of Equations Using Interval Analysis. *BIT*, 21:203–211, 1981.

[6] W. Hock and K. Schittkowski. *Test Examples for Nonlinear Programming Codes.* Lecture Notes in Economics and Mathematical Systems. Springer Verlag, 1981.

[7] H. Hong and V. Stahl. Safe Starting Regions by Fixed Points and Tightening. *Computing*, 53(3-4):323–335, 1994.

[8] Moré. J., B. Garbow, and K. Hillstrom. Testing Unconstrained Optimization Software. *ACM Transactions on Mathematical Software*, 7(1):17–41, 1981.

[9] R.B. Kearfott. Preconditioners for the Interval Gauss-Seidel Method. *SIAM Journal of Numerical Analysis*, 27, 1990.

[10] R.B. Kearfott. A Review of Preconditioners for the Interval Gauss-Seidel Method. *Interval Computations 1*, 1:59–85, 1991.

[11] R.B. Kearfott. A Review of Techniques in the Verified Solution of Constrained Global Optimization Problems. (To Appear), 1994.

[12] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[13] A.V. Levy and A. Montalvo. The Tunnelling Algorithm for the Global Minimization of Functions. *SIAM J. Sci. Stat. Comput.*, 6(1):15–29, 1985.

[14] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[15] K. Meintjes and A.P. Morgan. Chemical Equilibrium Systems as Numerical test Problems. *ACM Transactions on Mathematical Software*, 16:143–151, 1990.

[16] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[17] R.E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[18] R.E. Moore and S.T. Jones. Safe Starting Regions for Iterative Methods. *SIAM Journal on Numerical Analysis*, 14:1051–1065, 1977.

[19] J.J. More and M.Y. Cosnard. Numerical Solution of Nonlinear Equations. *ACM Transactions on Mathematical Software*, 5:64–85, 1979.

[20] A.P. Morgan. Computing All Solutions To Polynomial Systems Using Homotopy Continuation. *Appl. Math. Comput.*, 24:115–138, 1987.

[21] A.P. Morgan. *Solving Polynomial Systems Using Continuation for Scientific and Engineering Problems.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[22] A. Neumaier. *Interval Methods for Systems of Equations.* PHI Series in Computer Science. Cambridge University Press, Cambridge, 1990.

[23] D. Ratz. Box-Splitting Strategies for the Interval Gauss-Seidel Step in a Global Optimization Method. *Computing*, 53(3-4):337–353, 1994.

[24] S.M. Rump. Verification Methods for Dense and Sparse Systems of Equations. In J. (Ed.) Herzberger, editor, *Topics in Validated Computations*, pages 217–231. Elsevier, 1988.

[25] H. Schwefel. *Numerical Optimization of Computer Models.* Wiley, New York, 1981.

[26] P. Van Hentenryck, D. McAllister, and D. Kapur. Solving Polynomial Systems Using a Branch and Prune Approach. *SIAM Journal on Numerical Analysis*, 1995. (to appear).

[27] P. Van Hentenryck and L. Michel. Newton: Constraint Programming over Nonlinear Constraints. *Science of Computer Programming.* To appear, 1996.

[28] J Verschelde, P. Verlinden, and R. Cools. Homotopies Exploiting Newton Polytopes For Solving Sparse Polynomial Systems. *SIAM Journal on Numerical Analysis*, 31(3):915–930, 1994.